Chapter 2
Computational Complexity

I usually start my class on computational complexity with the hypothetical story described in the book of Garey and Johnson as a strategy to protect one's job. Your boss assigned you a problem which you could not solve in spite of your tireless effort. If you report your boss that you could not solve the problem with your level of intelligence, then your boss would fire you and would recruit another person who is more intelligent than you. Since you are working hard on the problem and you believe that you are intelligent enough, you have the confidence that there is no solution for the problem. Then if you mention your boss confidently that nobody could solve this problem because the solution for this problem does not exist. Then your boss would not believe you and fire you. In this scenario how do you protect your job? One strategy might be to convince your boss by showing that none of the famous experts of the field could solve the problem. Unfortunately, it is very unlikely that any of the experts has tried to solve your problem. In this case you should try to show that your problem is at least as hard as their problems.

2.1 Comparing Hardness of Two Problems

Assume that your problem is X. You know that some expert tried to solve Problem Y and could not solve it. If you can prove that if there is a solution of your problem then this solution can be used to solve Problem Y. Then you can claim that your problem X is at least as hard as Problem Y.

Since in algorithm theory our desired solution is a polynomial time algorithm, to solve Problem Y, your solution is allowed to use polynomial (in the input size) times and if additional computation is needed that must be of polynomial time. Then if your problem has a polynomial-time solution, then Problem Y will have a polynomial-time solution. But the reverse may not be true. Thus your problem is at least as hard as Problem Y.

2.1.1 Polynomial-time reduction

We call this proof procedure *polynomial-time reduction.* Assume that we had a blackbox that could solve instances of a problem X. That is if we write down the input of an instance of X then the blackbox will return the correct answer in a single step. Now if we can solve any arbitrary instance of a problem Y using the blackbox that solves X polynomial time with a polynomial number of additional computation steps then we say that the problem Y is reducible to the problem X in polynomial time. We denote this by notation $Y \leq_P X$. Then the following claim holds.

Lemma 2.1 *Assume that $Y \leq_P X$. Then Y can be solved in polynomial time if X can be solved in polynomial time.*

Proof Assume that X can be solved in polynomial time. That is, the blackbox takes polynomial time to solve an instance of the problem X. If the blackbox is used polynomial times to solve problem Y and additional computation needs polynomial number of steps, then the total time to solve problem Y is also polynomial. □

If we take contrapositive of the statements in Lemma 2.1, we get the following corollary.

Corollary 2.1 *Assume that $Y \leq_P X$. Then X cannot be solved in polynomial time if Y cannot be solved in polynomial time.*

By Lemma 2.1 and Corollary 2.1 Problem X is at least as hard as Problem Y. That is, Problem X might be harder than Problem Y, but Problem Y cannot be harder than Problem X.

Since the blackbox is a solver of the problem X, its input is an instance of the problem X and its output is a solution of the problem X for the given input instance. So if we wish to solve the problem Y using the blackbox, we need to transform the input instance of problem Y to an input instance of problem X and also we need to transform the output of problem X to the output of problem Y.

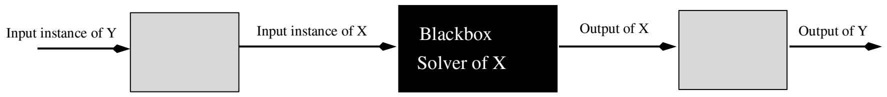

Fig. 2.1 Reducing Problem Y to Problem X.

In comparing computational complexity of problems, it is convenient to work with decision version of the problems. In the decision version of a problem, the output is always “yes” or “no”. Thus to reduce the decision version of Problem Y to the decision version of Problem X, we need to construct an instance of Problem X from the given instance of problem Y such that the answer of Problem Y for the given input instance is “Yes” if and only if the blackbox produces an “Yes” answer for the constructed input instance of Problem X.

2.1.2 Independent Set Problem and Vertex Cover Problem

A set S of vertices of a graph G is *independent* if no two of its vertices are adjacent in G. The subgraph of G induced by the vertices in S is a null graph. Figures 2.2(a) and (b) illustrate independent sets of 3 and 5 vertices, respectively, where vertices in the independent sets are drawn by white circles. A set containing a single vertex trivially an independent set. Finding a larger independent set needs careful checking of the adjacency of vertices. An independent set S of G is *maximal* if S is not a proper subset of any other independent set of G. An independent set S of G is *maximum* if no other independent set of G has more vertices than S. That is, a maximum independent set of G contains the maximum number of vertices among all independent sets of G. The independent set shown in Figure 2.2(a) is a maximal independent set whereas the independent set shown in Figure 2.2(b) is a maximum independent set. The independent set problem asks to find a maximum independent set of G.

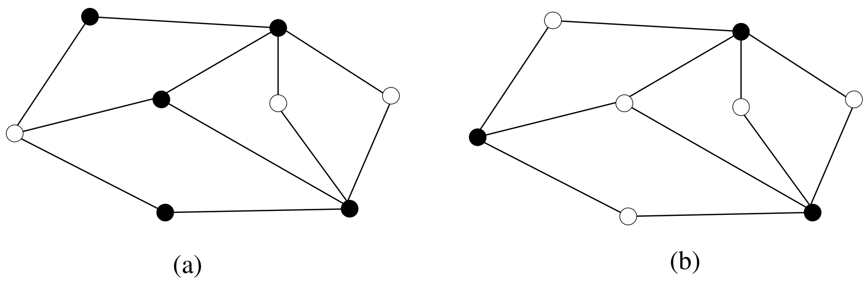

Fig. 2.2 Illustration for independent sets.

> Decision Version of Independent Set Problem

Given a graph G and an integer k, does G have an independent set of size at least k?

A *vertex cover* of a graph $G = (V, E)$ is a set $Q \subseteq V$ that contains at least one endpoint of every edge. A vertex cover of a graph G is a *minimum vertex cover* if it contains the minimum number of vertices among all vertex covers of G. The vertex cover problem asks to find a minimum vertex cover of a graph.

> Decision Version of Vertex Cover Problem

Given a graph G and an integer k, does G have an independent set of size at most k?

To compare these two problems, we need to establish a relationship between two problems which enables us to transform the input instance of one problem into the input instance of another problem. The following lemma states such a relationship.

Lemma 2.2 *Let $G = (V, E)$ be a graph. Then a set $S \subseteq V$ is an independent set of G if and only if $V - S$ is a vertex cover of G.*

Proof *Necessity.* Let S be an independent set of G. We show that $V - S$ is a vertex cover. Let (u, v) be an arbitrary edge of G. Since S is an independent set, S cannot contain both u and v; one of u and v will be contained in $V - S$. This implies that every edge has at least one end vertex in $V - S$, and hence $V - S$ is a vertex cover.

Sufficiency. Assume that $V-S$ is a vertex cover. We show that S is an independent set. Assume for a contradiction that S is not an independent set. Then there is a pair of adjacent vertices u and v in S. That means there is an edge (u, v) such that none of its end vertices belongs to $V - S$, and hence $V - S$ is not a vertex cover, a contradiction. □

Using Lemma 2.2 we are now ready to prove the following lemma.

Lemma 2.3 *Independent Set Problem $\leq_P$ Vertex Cover Problem*

Proof Assume that we have a blackbox which can solve the vertex cover problem. We wish to solve the independent set problem using the blackbox. We take any input instance of the independent set problem which consists of a graph G and an integer k. Our question is whether G has an independent set of size at least k. From Lemma 2.2 we know G has an independent set of size k if and only if G has a vertex cover of size $n - k$ where n is the number of vertices in G. We thus ask the blackbox whether G has a vertex cover of size $n - k$. If the answer is "yes", that is, G has a vertex cover of size $n - k$, then G has an independent set of size k. That means that the answer of the independent set problem is "yes". □

Observe that we have used the balckbox once and performed a subtraction operation as additional computation.

Similarly, we can also prove the following lemma.

Lemma 2.4 *Vertex Cover Problem $\leq_P$ Independent Set Problem*

Proof Assume that we have a blackbox which can solve the independent set problem. We wish to solve the vertex cover problem using the blackbox. We take any input instance of the vertex cover problem which consists of a graph G and an integer k. Our question is whether G has a vertex cover of size at least k. From Lemma 2.2 we know G has a vertex cover of size at least k if and only if G has an independent set of size at most $n - k$ where n is the number of vertices in G. We thus ask the blackbox whether G has an independent set of size $n - k$. If the answer is "yes", that is, G has an independent set of size $n - k$, then G has an independent set of size k. That means that the answer of the vertex cover problem is "yes". □

By Lemma 2.3 the independent set problem is at least as hard as the vertex cover problem. Again by Lemma 2.4 the vertex cover problem is at least as hard as the independent set problem. Thus the two problems are equally hard.

2.1.3 Clique Problem and Independent Set Problem

A *clique* in a graph G is a subset C of vertices such that, for each u and v in C, with $u \neq v$, (u, v) is an edge. The clique problem asks to find a clique in a graph of maximum size. The decision version of the clique problem can be stated as follows.

> Decision Version of Clique Problem

Given a graph G and an integer k, does G have a clique of size at least k?

We now reduce the independent set problem to the clique problem. Before proceeding, we need a definition. The *complement* of a graph $G = (V, E)$ is another graph $\overline{G} = (V, \overline{E})$ with the same vertex set such that for any pair of distinct vertices $u, v \in V$, $(u, v) \in \overline{E}$ if and only if $(u, v) \notin E$. We often denote the complement of a graph G by $\overline{G}$. The graph in Fig 2.3(b) is the complement $\overline{G}$ of the graph G in Fig 2.3(a). Observe that $\{c, d, e, f\}$ is an independent set in the graph G whereas the set $\{c, d, e, f\}$ induces a clique in $\overline{G}$. The following lemma holds regarding independent sets in G and cliques in $\overline{G}$.

Lemma 2.5 *A graph G has an independent set of size k if and only if $\overline{G}$ has a clique of size k.*

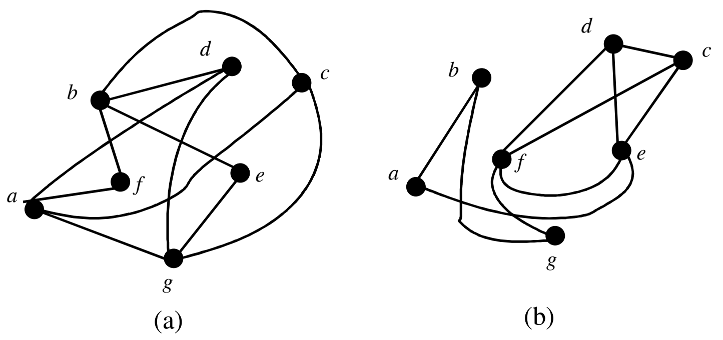

Fig. 2.3 (a) G and (b) $\overline{G}$.

We now show that the independent set problem is reducible to clique problem in polynomial time. Assume that we have a blackbox which is capable to solve the clique problem. We will solve the independent set problem using the blackbox. We take any input instance of the independent set problem. This instance consists of a graph G and an integer k. We construct the complement graph $\overline{G}$ of G. Then by Lemma 2.6 G has an independent set of size k if and only if $\overline{G}$ has a clique of size k. We thus give $\overline{G}$ and k as the input of the blackbox. $\overline{G}$ can be constructed from G in polynomial time, and hence the following lemma holds.

Lemma 2.6 *Independent Set Problem $\leq_P$ Clique Problem*

2.1.4 Set Cover Problem and Vertex Cover Problem

Let U be a set of n elements and let $S_1, S_2, \ldots, S_m$ be the m subsets U. The set cover problem asks to choose the minimum number of subsets whose union contains all elements of U. The decision version of the problem is stated as follows.

> Decision Version of Set Cover Problem

Given a set U, a collection of subsets $S_1, S_2, \ldots, S_m$ of U and an integer $k \leq m$, does there exist a collection of at most k of these subsets whose union contains all elements in U?

We wish to reduce the vertex cover problem to set cover problem. Assume that we have a blackbox which is capable of solving the set cover problem. But we wish to solve the vertex cover problem using the blackbox. The problem here is that the vertex cover problem is on graphs whereas the set cover problems is on sets. We need to transform any input instance of vertex cover to an input instance of set cover problem. Let a graph $G = (V, E)$ and an integer k is the input instance of the vertex cover problem. The problem asks to choose set S at most k vertices which covers all edges in E. The set S covers an edge e means at least one end vertex of e is in S.

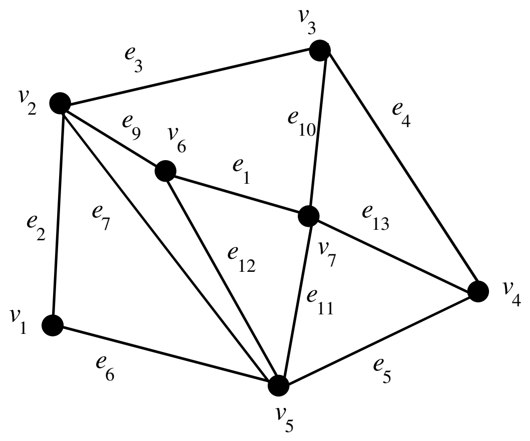

Fig. 2.4 A graph.

From the input instance of the vertex cover problem we construct an input instance of the set cover problem as follows. We set $U = E$. We now construct n

subsets of U, where n is the number of vertices. Let $V = \{v_1, v_2, \cdots, v_n\}$ be the vertex set of G. Then for $1 \leq i \leq n$, the subset S_i of U consists of all the edges in E which are incident to vertex v_i.

Consider the graph in Figure 2.4 and an integer $k \leq 7$ as an input instance of the vertex cover problem. Here $V = \{v_1, v_2, \cdots, v_7\}$ and $E = \{e_1, e_2, \cdots, e_{13}\}$. The corresponding input instance of the set cover problem will be as follows.

$$\begin{aligned}
U &= \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9, e_{10}, e_{11} e_{12}, e_{13}\} \\
S_1 &= \{e_2, e_6\} \\
S_2 &= \{e_2, e_3, e_7, e_9\} \\
S_3 &= \{e_3, e_4, e_{10}\} \\
S_4 &= \{e_4, e_5, e_{13}\} \\
S_5 &= \{e_5, e_6, e_7, e_{11}, e_{12}\} \\
S_6 &= \{e_1, e_9, e_{12}\} \\
S_7 &= \{e_1, e_{10}, e_{11}, e_{13}\}
\end{aligned}$$

Now we need to ensure that an "yes" output produced by the blackbox for this input instance of the set cover problem implies an "yes" output for the given instance of the vertex cover problem. Also if the vertex cover instance has an "yes" output the blackbox must produce a "yes" output. So we prove the following lemma.

Lemma 2.7 *G has a vertex cover of size at most k if and only if U can be covered with at most k of the sets $S_1, S_2, \cdots, S_n$.*

Proof *Necessity.* Assume that G has a vertex cover S of size at most k. Then S contains at least one end vertex of each edge and hence the union of the subsets of U corresponding to the vertices in S contains all edges of G. Thus U can be covered with at most k of the sets $S_1, S_2, \cdots, S_n$.

Sufficiency. Assume that U can be covered with at most k of the sets $S_1, S_2, \cdots, S_n$. Since U contains all the egdes of G, the set containing the vertices corresponding to the at most k subsets of U contains at least one vertex of each edge of G. Thus S is a vertex cover of size at most k. □

Now, given an input instance of the vertex cover problem, we construct an input instance of the set cover problem as described above and pass through the blackbox. The answer of the vertex cover instance will be "yes" if and only if the blackbox produces a "yes" answer. Hence the following lemma holds.

Lemma 2.8 *Vertex Cover Problem $\leq_P$ Set cover Problem*

2.1.5 3-SAT and Independent Set problem

In the 3-SAT problem, input is a Boolean formula X in conjunctive normal form (CNF) with each clause in X having exactly three literals and output is "yes" or "no" depending on X is satisfiable or not. An instance of the 3-SAT problem is given below.

Given

$$X = (x_1 + \overline{x_2} + x_4)(x_2 + x_3 + \overline{x_5})(\overline{x_2} + \overline{x_4} + x_5),$$

is there any assignment of $x_i \in \{0, 1\}$ such that the value X becomes 1?

We now show that 3-SAT Problem $\leq_P$ Independent Set Problem. Interestingly, here the two problems are from two different domains. 3-SAT problem is a Boolean logic problem and independent set problem is a graph problem. We have a blackbox which is capable of solving independent set problem. We thus need to construct an input instance of the independent set problem from any given input instance of the 3-SAT problem.

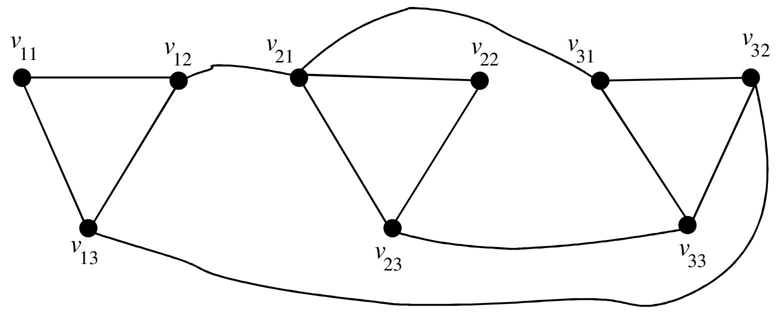

Fig. 2.5 A graph.

We construct a triangle of three vertices for each clause and label the vertices of the triangle v_{ij} which corresponds to the jth term of the ith clause. We add an edge between two vertices in different triangles if their corresponding terms are complement to each other. This ensures that any independent set in the constructed graph cannot contain the two vertices corresponding to a term and its complement. Let G be the graph constructed above. Let k be the number of clauses in the Boolean formula. Then number of vertices in G is exactly $3k$ and clearly this construction takes polynomial time.

One can observe that any independent set in G can contain at most one vertex from each triangle. Thus an independent set of G can have at most k vertices. On the other hand to make X satisfiable at least one term in each clause must be "1". If we take G and k as the input instance of the blackbox, we need to prove the following claim.

Lemma 2.9 *X is satisfiable if and only if G has an independent set of size at least k.*

Proof *Necessity.* Assume that X is satisfiable. Then each clause of the formula has a term which has value "1'. We take a set S of k vertices by taking exactly one vertex from each triangle whose corresponding term has value "1". Observe that S contains exactly one vertex from each triangle and two vertices in S from separate triangles cannot be adjacent since a term and its complement cannot have values "1" simultaneously. Then S is an independent set of size k.

Sufficiency. Assume that G has an independent set S of size at least k. Then G has an independent set exactly k since an independent set of G can have at most k vertices. Since S is an independent set of size k, S contains exactly one vertex from each triangle. We set each term corresponding to each vertex to 1 which set the value of X to "1". □

Now from a give 3-SAT formula of k clauses we construct a graph G as described above and give G and k as the input instance of the blackbox. Then if blackbox produces a "yes" output which implies "yes" output of of the 3-SAT instance. Thus 3-SAT Problem $\leq_P$ Independent Set Problem.

> Circuit-SAT, SAT and 3-SAT

Circuit-SAT is the problem that takes as input a Boolean circuit with a single output node, and asks whether there is an assignment of values to the circuit's inputs so that its output vale is "1".

The problem CNF-SAT takes a Boolean formula in conjunctive normal form (CNF) (product of sum form) and asks if there is an assignment of Boolean values to its variables so that the formula evaluates to 1. In a CNF formula each subexpression that are formed using OR (+) operations of variables or their negation is called a *clause*. Clauses are combined using AND operations. Variables or their negations are called *terms* or *literals*.

In the 3-SAT problem, input is a Boolean formula X in conjunctive normal form (CNF) with each clause in X having exactly three literals and output is "yes" or "no" depending on X is satisfiable or not.

2.2 Complexity Classes P, NP, NP-Hard, NP-Complete

In the last section we have learn how to compare the hardness of two problems. In this section we define the classes of problems in respect to their hardness.

Let Π be any problem and A be an algorithm to solve Π. Assume that the time complexity of A is $O(n^k)$, where n is the input size and k is a non-negative integer. Then we say that Algorithm A is a *polynomial time algorithm* for Problem Π. We call a polynomial time algorithm an *efficient algorithm.* There are many interesting real world problems whose efficient algorithms are not known. It would be nice if we could prove that finding an efficient algorithm is impossible in such cases.

Unfortunately, such proofs are even harder. It has been agreed upon in the computer science community to refer those problems for which there exist polynomial time algorithms as *tractable* and those for which it is unlikely that there exist polynomial time algorithms as *intractable*.

An algorithm for a problem is *deterministic* if it has only one choice in each step throughout its execution on a given instance of the problem. Thus if a deterministic algorithm is run again and again on the same input instance, its output never changes. A *nondeterministic* algorithm has several choices in step throughout its execution.

Assume that in a summer morning you are standing in front of a mango tree. You can see all the mangoes in the tree in your bare eye. Now I ask you a question: is there a ripe mango in the tree? Note that the color of a ripe mango is yellow whereas other mangoes have green color. Looking at the tree at a glance you can answer the question instantly. In answering the question you are following a nondeterministic approach. However, if you follow a deterministic approach, you would use a systematic tree traversal method to determine whether there is a ripe mango in the tree or not. In the systematic approach you follow a fixed rule of traversing the branches of the trees one after another which is deterministic and would take longer time.

Choice(j) arbitrarily choose one of the elements of set S. Whenever there is a set of choices that leads to a successful completion, then one such choice is always made and the algorithm terminates successfully.

A nondeterministic algorithm terminates unsuccessfully if and only if there exists no set of choices leading to a success signal.

> Class P

P is the set of all decision problems solvable by deterministic algorithms in polynomial time.

> Class NP

NP is the set of all decision problems solvable by nondeterministic algorithms in polynomial time.

An informal definition of nondeterministic algorithm is as follows. On input x, a nonderterministic algorithm executes in two phases: the guessing phase and the verification phase. In the guessing phase, an arbitrary string of characters y is generated. It may correspond to the solution to the input instance or not. In fact, it may not be in the proper format of the desired solution. It may differ from one run to another of the nondeterministic algorithm. It is only required that this string be generated in a polynomial number of steps.

In the verification phase, a deterministic algorithm verifies two things. First, it checks whether the generated solution string y is in the proper format. If it is not, then the algorithm halts with the answer "no". If y is in the proper format, then the algorithm continues to check whether it is a solution to the instance x of the problem.If it is indeed a solution to the instance x, then it halts and answers "yes"; otherwise it halts and answers "no". It is also required that this phase be completed in a polynomial number of steps.

Let A be a nondeterministic algorithm for a problem Π. We say that A *accepts* an instance I of Π if and only if on input I there is a guess that leads to a "yes" answer. In other words, A *accepts* I if and only if it is possible one some run the algorithm that its verification phase will answer "yes". Observe that, if the algorithm answers "no", then it does not mean that A does not accept its input, as the algorithm might have guessed an incorrect solution.

If a problem can be solved in polynomial time by a deterministic algorithm, obviously it can be solved in polynomial time by a nondeterministic algorithm. Thus P $\subset$ NP and the relation can be shown in Figure 2.6. However, it is a longstanding open problem whether P = NP or not. Cook proved that satisfiability problem is in P if and only if P = NP. Thus Cook established the satisfiabilty problem as a hard problem.

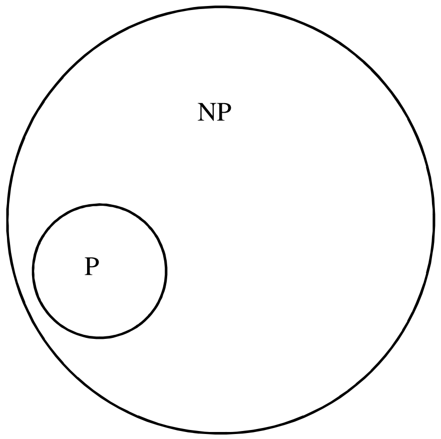

Fig. 2.6 Commonly believed relationship between P and NP complexity classes.

> Cook's Theorem

Satisfiability problem is in P if and only if P = NP.

Based on Cook's theorem the class NP-hard is defined as follows.

> NP-hard Problem

A problem X is NP-hard if and only if Satisfiability $\leq_P$ X.

> NP-Complete Problem

A problem X is NP-Complete if X is NP-hard and X $\in$ NP.

A commonly believed relationship among P, NP, NP-hard and NP-complete complexity classes are shown in Fig 2.7. NP-complete is a class of decision problem in NP such that if one of them is proven to be solvable in polynomial time by deterministic algorithm then all problem in NP are solvable by a polynomial time by deterministic algorithm, i.e. P=NP.

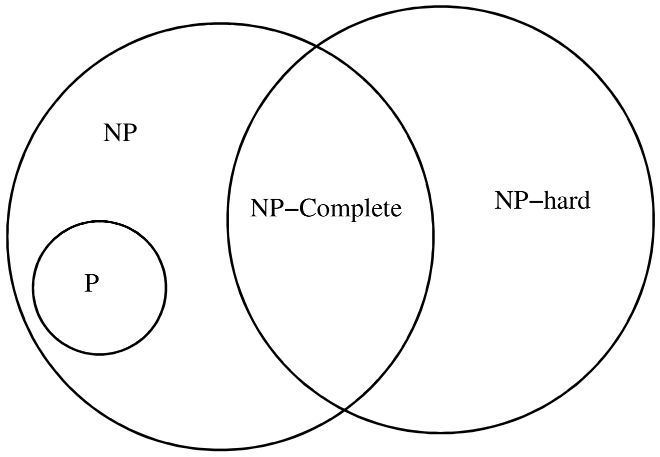

Fig. 2.7 Commonly believed relationship among P, NP, NP-hard and NP-complete complexity classes.

2.3 Proof of *NP*-Completeness

From Fig 2.7 it is clear that to show a problem X is NP-complete we need to show that (a) X is in NP and (b) X is NP-hard.

If we can verify a solution instance of X in polynomial time using deterministic algorithm then the problem is in NP. To prove X is NP-hard, we pick a problem Y which is known to be NP-hard and then we show Y $\leq_P$ X. In the rest of this section we will present NP-completeness proof of some well-known problems.

2.3.1 Vertex Cover

To prove the vertex cover problem is NP-complete, we first show that the vertex cover problem is in NP. Assume that S be a solution instance of the vertex cover problem. We can verify whether S is a vertex cover by deleting the vertices in S from the input graph G and check whether there is any edge which is not deleted. Clearly this can be done in polynomial time, and hence vertex cover problem is in NP.

We now prove 3-SAT $\leq_P$ vertex cover to prove vertex cover problem is NP-hard. We take any instance of the 3-SAT problem. Let $Y = (x_1 + \overline{x_2} + x_4)(x_2 + x_3 + \overline{x_4})(\overline{x_1} + \overline{x_3} + x_4)$ be the instance of the 3-SAT problem.

We construct a graph G as follows. We take one vertex for each variable x_i used in the Boolean expression of Y and one vertex for the complement $\overline{x_i}$ of the vertex x_i. We connect the two vertices corresponding to a variable and its complement by an edge which we call a *complement-connecting edge*. We represent each clause $C_i = (p + q + r)$ of the Boolean expression by a triangle of three vertices v_{i1}, v_{i2} and v_{i3}, where v_{i1} corresponds to the first term p of clause C_i, v_{i2} corresponds to the second term q of clause C_i, and v_{i3} corresponds to the third term r of clause C_i. We call each edge on a triangle a *triangle edge*. We then connect the edges (v_{i1}, p) (v_{i2}, q) and (v_{i3}, r) for each clause $C_i = (p + q + r)$. We call these edges *link edges*. We thus construct the graph G and clearly the construction is done in polynomial time.

Let s be the number of variables in the Boolean expression of Y and t be the number of clauses. We set $k = s + 2t$. If G has a vertex cover of size at most k then G has a vertex cover of size exactly k. This is because any vertex cover must contain at least one of the two vertices corresponding to a variable and its complement and at least two vertices from the triangle corresponding to each clause.

To complete the proof we need to show that the following lemma holds.

Lemma 2.10 *Y is satisfiable if and only if G has a vertex cover of size at most k.*

Proof *Necessity.* We first assume that Y is satisfiable, that is, there is an assignment of Boolean values to variables in Y so that Y is satisfied. We now construct a vertex subset S as follows. If the value of x_i is 1 in the assignment we take the vertex corresponding to x_i in S, otherwise $\overline{x_i}$ is 1 and we take the vertex corresponding to $\overline{x_i}$ in S. Thus we take one end vertex of each complement-connecting edge in S. Since Y is satisfied by the assignment, the value of at least one term in easch clause $C_i = (p + q + r)$ is 1. Fixing a term in each clause having value 1 in the assignment, take two vertices of the triangle corresponding to the other two terms in the clause

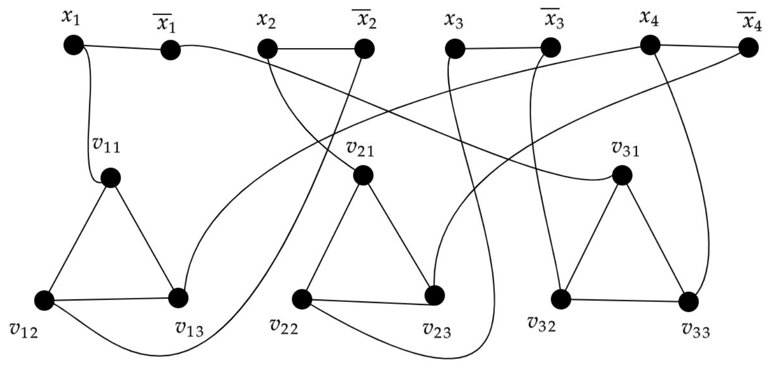

Fig. 2.8 Illustration for reducing a 3-SAT instance to a vertex cover instance.

in S. The two vertex of each triangle in S also the end vertex of two link edges. For the other link edge of the triangle, the other end vertex is taken since the value of its its corresponding literal is 1. Thus S is a vertex cover of G. S contains exactly $s + 2t = k$ verties.

Sufficiency. We now assume that G has a vertex cover S of size at most k. We show that the variables have an assignment by which Y is satisfied. By construction S has exactly $k = s + 2t$ vertices since it must contain one vertex from each complement-connecting edge and two vertex from each triangle. For each triangle one vertex is not in S. We set the value of the corresponding term 1. Thus each clause will have a term whose value is 1 and Y will be satisfied. □

2.3.2 *k*-Coloring

A *vertex coloring* of a graph is an assignment of colors to the vertices so that adjacent vertices have distinct colors. A *k-coloring* of a graph uses at most k colors for vertex coloring. The k-coloring problem asks to color the vertices of a given graph using minimum number of colors. The decision version of the k-coloring problem can be stated as follows.

> Decision Version of *k*-Coloring Problem

Given a graph G and an integer k, does G have a k-coloring?

A graph G has a 1-coloring if and only if G is a null graph, i.e., G has no edge. Thus one can easily check whether G is 1-colorable or not in $O(n)$ time. Again, G

has a 2-coloring if and only if G is bipartite, and it can also be checked in $O(n + m)$ time. Thus k-coloring problem is in P for $k \leq 2$.

We now consider the 3-coloring problem. If a graph has a triangle then it needs at least 3 colors, and if a graph has K_4 as an induced subgraph then it needs at least 4 colors. One may think that a graph is 3-colorable if it has no clique of size four or more. However it is not true. One can easily construct a graph with no clique of size four or more which is not 3-colorable. In fact we can prove the following theorem.

Theorem 2.1 *3-coloring problem is NP-Complete.*

Proof We first prove that the 3-coloring problem is in NP. An output instance, i.e., a coloring of a graph is given. One can easily verify that the coloring is a valid coloring and uses only three colors. Thus the problem is in NP.

We now show that 3-coloring problem is NP-hard. To do this we will show that 3-SAT $\leq_P$ 3-coloring. Take any instance Φ of the 3-SAT problem. Let Φ has m clauses over n variables. From Φ we construct a graph G as an instance of the 3-coloring problem.

We first construct a base graph G_b considering the n variables in Φ. For each variable x_i, $1 \leq i \leq n$, we take two vertices one for x_i and the other for $\overline{x_i}$. we add the edge $(x_i, \overline{x_i})$ for each i, $1 \leq i \leq n$. We now add a triangle of three vertices labeled by T, F, N, where T stands for "True", F stands for "False" and N stands for "Neutral". Finally add edge (x_i, N) for for each i, $1 \leq i \leq n$, as illustrated in Figure 2.9. In any three coloring of G_b the vertices T, F and N get three different color. We refer "true color", "false color" and "neutral color" corresponding to the colors in vertices T, F and N, respectively. Then in the 3-coloring one of x and $\overline{x_i}$ gets "true color" and the other gets "false color" for each i, $1 \leq i \leq n$. Thus we get a truth assignment for the variables in the 3-SAT instance from any 3-coloring of G_b. However, we can not assure that at least one literal of each clause of the 3-SAT assignment will get a "True" assignment.

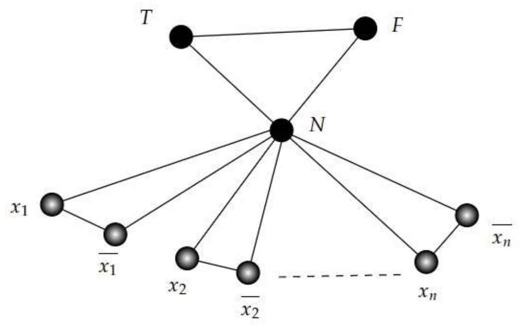

Fig. 2.9 Construction of the base graph from a 3-SAT instance.

To ensure that each clause will have at least one literal of "True" value by the assignment obtained from a 3-coloring, we add a gadget for each clause with G_b as follows. We add a gadget of six vertices $\{a_i, b_i, c_i, d_i, e_i, f_i\}$ and edges (a_i, b_i), (a_i, c_i), (a_i, d_i), $(b_i, e_i)(d_i, f_i)$ for each clause C_i. We add an edge from T of G_b to each of b_i, c_i, e_i, f_i and an edge from F of G_b to d_i. We also add an edge between the first literal of C_i and e_i, between the second literal of C_i and c_i and between the 3rd literal literal of C_i and d_i. Thus we complete the construction of the graph G.

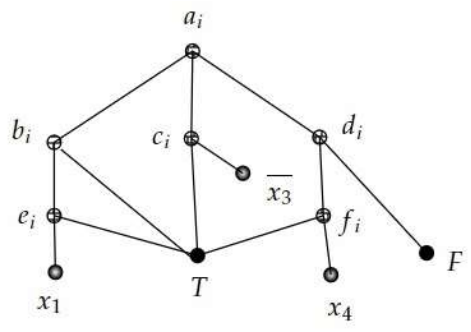

Fig. 2.10 A gadget $\{a_i, b_i, c_i, d_i, e_i, f_i\}$ is connected to clause $C_i = x_1 + \overline{x_3} + x_4$ where $x_1, \overline{x_3}, x_4, T$ and F are vertices of the base graph G_d .

Figure 2.10 illustrate a gadget corresponding to clause $C_i = x_1 + \overline{x_3} + x_4$. Assume that all of the three literals $x_1, \overline{x_3}, x_4$ of clause C_i are assigned “F”. Then e_i, c_i and f$_i$ get “N”, b_i gets “F”, c_i gets “N”, d_i gets “T”, and hence a_i cannot be assigned any color in a 3-coloring of G. That is, a 3-coloring in which all of the three literals of a clause have “false” value cannot be extended to a 3-coloring of the graph G. On the other hand one can check that a 3-coloring can be extended if at least one of the three literals of a clause gets a “true” value.

We now prove that a 3-SAT instance Y is satisfiable if and only if G has a 3-coloring. We first prove the necessity. Assume that there is a satisfying assignment for Y. For each i, $1 \leq i \leq n$, we assign “T” to the vertex x_i of G_b and “F” to $\overline{x_i}$ if $x_i = 1$ in the satisfying assignment of Y. Otherwise, we assign “F” to the vertex x_i of G_b and “T” to $\overline{x_i}$ if $x_i = 0$ in the satisfying assignment of Y. We assign “N” to the vertex of the base triangle which is the common neighbors of x_i and $\overline{x_i}$. We assign “F” to the vertex of the base triangle which has exactly one neighbor in a gagdet corresponding to a clause and assign “T” to the other vertex of the base triangle. Finally, as we have explained earlier, it is now possible to extend this 3-coloring to the six-vertex gadget for each clause. Thus we obtain a 3-coloring of G.

For the proof of sufficiency assume that G has a 3-coloring. Consider the triangle of G containing the vertices corresponding to x_i and $\overline{x_i}$. We choose one of the colors of x_i and $\overline{x_i}$ as “T” color and the other as “F” color. We choose the color of the remaining vertex of the triangle as ”N” color. Then the vertices corresponding to a variable and its complement will have colors from the set $\{T, F\}$. Taking “T”colors as 1 and “F” color as 0, we get an assignment of variables in Y. Clearly this assignment will be a satisfying assignment of Y where each clause has at least a literal with value 1, otherwise the coloring of G would not be a 3-coloring, a contradiction. □

We have seen that k-coloring problem is NP-complete for $k = 3$. What will happen if $k > 3$? In fact it is not difficult to reduce 3-coloring problem to k-coloring problem for $k > 3$. Let G be any instance of the 3-coloring problem. We construct an instance G' of k-coloring problem from G by adding a cliqueK_{k-3} of $k - 3$ new vertices and joining each vertex of the clique with all vertices of G with new edges. We now show that G has a 3-coloring if and only if G' has a k-coloring. We first assume that G has a 3-coloring. Then we can color the vertices of K_{k-3} with the remaining $k - 3$ colors. Thus G' has a k-coloring. We now assume that G' has a k-coloring. Then the vertices of the clique K_{k-3} use $k - 3$ colors. Since every vertex of G is joined to all vertices of K_{k-3} by edges, none of the $k - 3$ colors are used to color the vertices of G in the k-coloring of G'. Hence the vertices of G are colored by three colors, that is, G is 3-colorable.

2.4 co-NP

We first define the complement of a decision problem. The *complement* of a decision problem is the decision problem resulting from reversing the yes and no answers. For example, one important problem is whether a number is a prime number. Its complement is to determine whether a number is a composite number (a number which is not prime). Given a graph G, does G have a 2-coloring? Complement of this decision problem is: Given a graph G, does G have no 2-coloring? If we define decision problems as sets of finite strings, then the complement of this set over some fixed domain is its complement problem.

If we have a polynomial time algorithm to solve a problem X, then clearly we can also solve $\overline{X}$ in polynomial time. Thus if $X \in P$, then $\overline{X} \in P$. However, it is not true for the class NP.

Given a graph G and an integer k, does G have an independent set of size k? If we find an instance of k vertices which is an independent set then the answer of the question is yes. So a single "yes" instance is sufficient to give "yes" answer. But if we consider the complement question: Given a graph G and an integer k, does G have no independent set of size k? We need to check all instances of k vertices to give a "yes" answer of this question. In the former case we say a *short proof exists* but in the latter case we say a *short proof does not exist.*

co-NP is a complexity class. A decision problem X is a member of co-NP if and only if its complement $\overline{X}$ is in the complexity class NP.

Does NP = co-NP ?

Lemma 2.11 *If $NP \neq co\text{-}NP$, then $P \neq NP$.*

Proof We prove the contrapositive of the above statement. If $P = NP$ then $NP = co\text{-}NP$. Assume that $P = NP$. Then

$$X \in NP \implies X \in P \implies \overline{X} \in P \implies \overline{X} \in NP \implies X \in co\text{-}NP$$

$$X \in co\text{-}NP \implies \overline{X} \in NP \implies \overline{X} \in P \implies X \in P \implies X \in NP$$

Therefore $NP \subseteq co\text{-}NP$ and $co\text{-}NP \subseteq NP$ and hence $NP = co\text{-}NP$.

If a problem has short proofs for both "yes" and "no" answer then the problem belongs to both NP and co-NP. Consider the network flow problem: Given a flow network N, does N contain a flow of value at least v? We can verify a "yes" instance of this problem in polynomial time by showing a flow that achieves this value. On the other hand a "no" instance can also be verified by finding the value of a cut whose value is less than v. Thus the problem is in $NP \cap co\text{-}NP$.

As mentioned earlier, if a problem is in P, then its complement is also in NP. Since P is in NP, a problem in P belongs to the classes both NP and co-NP. Thus the relationship shown in Fig 2.11 holds.

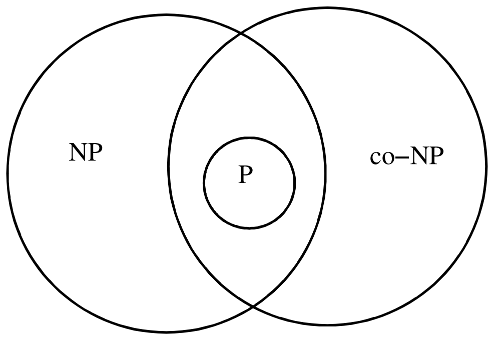

Fig. 2.11 Commonly believed relationship among P, NP and co-NP complexity classes.

2.5 PSPACE

We now study the space complexity of algorithms. The class PSPACE contains the set of all problems that can be solved by an algorithm with polynomial space complexity. That is, an algorithm that uses an amount of space that is polynomial in the size of the input.

The class P takes polynomial time and the class PSPACE takes polynomial space. Since any algorithm can consume only a polynomial space in polynomial time the following claim holds.

Lemma 2.12 $P \subseteq PSPACE$

However PSPACE is much wider than P. An exponential time algorithm may consume polynomial space. For example, consider the operation of a counter. It can count 0 to $2^n - 1$ using only n-bit space by reusing the space.

Lemma 2.13 *3-SAT problem can be solved using only a polynomial amount of space.*

Proof We give a bruteforce algorithm which takes polynomial amount of space. Let n be the number of variables in the 3-SAT expression. We take an n-bit counter whose counting output is given as the input of the 3-SAT expression. The values in the counter correspond to an assignment as follows: when the counter holds a value q, we consider it as an assignment that sets x_i to be the value of the ith bit of q. Clearly it enumerate all possible assignment using polynomial amount of space. $\square$

Since 3-SAT is a NP-complete problem, the following relationship holds.

Lemma 2.14 $NP \subseteq PSPACE$

Thus the commonly believed relationship is shown in Figure 2.12.

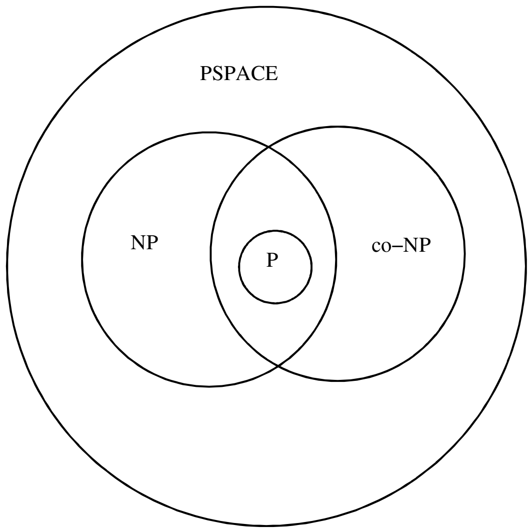

Fig. 2.12 Commonly believed relationship among P, NP and co-NP complexity classes.