Chapter 10

NP-hard Problems and Approximation Algorithms

10.1 What is the class NP?

The class P consists of all polynomial-time solvable decision problems. What is the class NP? There are two misunderstandings:

(1) NP is the class of problems which cannot be solved in polynomial-time.

(2) A decision problem belongs to the class NP if its answer can checked in polynomial-time.

The misunderstanding (1) comes from misexplanation of the brief name NP for "Not Polynomial-time sovable". Actually, it is polynomial-time solvable, but in a wide sense of computation, nodeterministic computation, that is, **NP is th class of all nondeterministic polynomial-time solvable decision problems.** Thus, NP is the brief name of "Nondeterministic Polynomial-time".

In nondeterministic computation, there may contains some guess steps. Let us look at an example. Consider the HAMILTONIAN CYCLE problem: *Given a graph* $G = (V, E)$*, does* G *contain a Hamiltonian cycle?* Here, a Hamiltonian cycle is a cycle passing through each vertex exactly once. The following is a nonterministic algorithm for the HAMILTONIAN CYCLE problem.

input a graph $G = (V, E)$.
step 1 guess a permutation of all vertices.
step 2 check if guessed permutation gives a Hamitonian cycle.

if yes, **then** accept input.

Note that in step 2, if the outcome of checking is no, then we cannot give any conclusion and hence nondeterministic computation stucks. However, a nondeterministic algorithm is considered to solve a decision problem correctly if there exists a guessed result leading to correct yes-answer. For example, in above algorithm, if input graph contains a Hamiltonian cycle, then there exists a guessed permutation which gives a Hamiltonian cycle and hence gives yes-answer. Therefore, it is a nondeterministic algorithm which solves the HAMITONIAN CYCLE problem.

Why (2) is wrong? This is because not only checking step is required to be polynomial-time computable, but also guessing step is required to be polynomial-time computable. How do we estimate guessing time? Let us explain this starting from what is a legal guess. **A legal guess is a guess from a pool with size independent from input size.** For example, in above algorithm, guess in step 1 is not legal because the number of permutation of n vertices is $n!$ which depends on input size.

What is the running time of step 1? It is the number of legal guesses spent in implementation of the guess in step 1. To implement the guess in step 1, we may encode each vertex into a binary code of length $\lceil \log_2 n \rceil$. Then each permutation of n vertices is encoded into a binary code of length $O(n \log n)$. Now, guessing a permutation can be implemented by $O(n \log n)$ legal guesses each of which chooses either 0 or 1. Therefore, the runnig time of step 1 is $O(n \log n)$.

Why the pool size for a legal guess is restricted to be bounded by a constant independent from input size? To give the answer, we need first to explain what is the model of computation.

The computation model for the study of computational complexity is the Turing machine (TM) as shown in Fig. 10.1, which consists of three parts, a tape, a head, and a finite control. Each TM can be described by the following parameters, an alphabet Σ of input symbols, an alphabet Γ of tape symbols, a finite set Q of states in finite control, a transition function δ, and an initial state. There exist two types of Turing machines, *deterministic* TM and *nondeterministic* TM. They differ with respect to the transition function of the finite control. In the deterministic TM, the transition function δ is a mapping from $Q \times \Gamma$ to $Q \times \Gamma \times \{R, L\}$. A transition $\delta(q, a) = (p, b, R)$ ($\delta(q, a) = (p, b, L)$) means that when the machine in state q reads symbol a, it would change state to p, the symbol to b and move the head to right (left) as shown in Fig. 10.2. However, in the nondeterministic TM, the

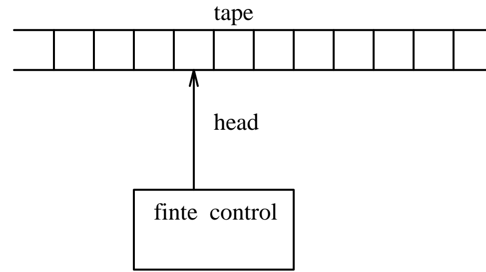

Figure 10.1: A Turing machine.

Figure 10.2: A Turing machine.

transition function δ is a mapping from $Q \times \Gamma$ to $2^{Q\times\Gamma\times\{R,L\}}$. That is, for any $(q, a) \in Q \times \Gamma$, $\delta(q, a)$ is a subset of $Q \times \Gamma \times \{R, L\}$. In this subset, every member can be used for the rule to guide the transition. When this subset contain at least two members, the transition corresponds to a legal guess. The size of the subset is at most $|Q| \cdot |\Gamma| \cdot 2$ which is a constant independent to input size.

In many cases, the guessinf step is easily implemented by a polynomial number of legal guesses. However, there are some exceptions; one of them is the following.

> Integer Programming: Given an $m \times n$ integer matrix A and an n-dimensional integer vector b, determine whether there exists a m-dimensional integer vector x such that $Ax \geq b$.

In order to prove Integer Programming in NP, we may guess an n-dimensional integer vector x and check whether x satisfies $Ax \geq b$. However, we need to make sure that guessing can be done in nondeterministic polynomial-time. That is, we need to show that if the problem has a solution, then there is a solution of polynomial size. Otherwise, our guess cannot find it. This is not an easy job. We include the proof into the following three lemmas.

Let α denote the maximum absolute value of elements in A and b. Denote $q = \max(m, n)$.

Lemma 10.1.1 *If B is a square submatrix of A, then $|\det B| \leq (\alpha q)^q$.*

Proof. Let k be the order of B. Then $|\det B| \leq k!\alpha^k \leq k^k\alpha^k \leq q^q\alpha^q = (q\alpha)^q$. □

Lemma 10.1.2 *If $rank(A) = r < n$, then there exists a nonzero vector z such that $Az = 0$ and every component of z is at most $(\alpha q)^q$.*

Proof. Without loss of generality, assume that the left-upper $r \times r$ submatrix B is nonsingular. Set $x_{r+1} = \cdots = x_{n-1} = 0$ and $x_n = -1$. Apply Cramer's rule to system of equations

$$B(x_1, \cdots, x_r)^T = (a_{1n}, \cdots, a_{rn})^T$$

where a_{ij} is the element of A on the ith row and the jth column. Then we can obtain $x_i = \det B_i / \det B$ where B_i is a submatrix of A. By Lemma 3.1, $|\det B_i| \leq (\alpha q)^q$. Now, set $z_1 = \det B_1, \cdots, z_r = \det B_r, z_{r+1} = \cdots = z_{n-1} = 0$, and $z_n = \det B$. Then $Az = 0$. □

Lemma 10.1.3 *If $Ax \geq b$ has an integer solution, then it must have an integer solution whose components of absolute value not exceed $2(\alpha q)^{2q+1}$.*

Proof. Let a_i denote the ith row of A and b_i the ith component of b. Suppose that $Ax \geq b$ has an integer solution. Then we choose a solution x such that the following set gets the maximum number of elements.

$$\mathcal{A}_x = \{a_i \mid b_i \leq a_i x \leq b_i + (\alpha q)^{q+1}\} \cup \{e_i \mid |x_i| \leq (\alpha q)^q\},$$

where $e_i = (\underbrace{0, \cdots, 0, 1}_{i}, 0, \cdots, 0)$. We first prove that the rank of $\mathcal{A}_x$ is n. For otherwise, suppose that the rank of $\mathcal{A}_x$ is less than n. Then we can find nonzero integer vector z such that for any $d \in \mathcal{A}_x$, $dz = 0$ and each component of z does not exceed $(\alpha q)^q$. Note that $e_k \in \mathcal{A}_x$ implies that kth component z_k of z is zero since $0 = e_k z = z_k$. If $z_k \neq 0$, then $e_k \notin \mathcal{A}_x$, so, $|x_k| > (\alpha q)^q$. Set $y = x + z$ or $x - z$ such that $|y_k| < |x_k|$. Then for every $e_i \in \mathcal{A}_x$, $y_i = x_i$, so, $e_i \in \mathcal{A}_y$, and for $a_i \in \mathcal{A}_x$, $a_i y = a_i x$, so, $a_i \in \mathcal{A}_y$. Thus, $\mathcal{A}_y$ contains $\mathcal{A}_x$. Moreover, for $a_i \notin \mathcal{A}_x$, $a_i y \geq a_i x - |a_i z| \geq b_i + (\alpha q)^{q+1} - n\alpha(\alpha q)^q \geq b_i$. Thus, y is an integer solution of $Ax \geq b$. By the maximality of $\mathcal{A}_x$, $\mathcal{A}_y = \mathcal{A}_x$. This means that we can decrease the value of the kth component again. However, it cannot be decreased forever. Finally, a contradiction would appear. Thus, $\mathcal{A}_x$ must have rank n.

Now, choose n linearly independent vectors $d_1, \cdots, d_n$ from $\mathcal{A}_x$. Denote $c_i = d_i x$. Then $|c_i| \leq \alpha + (\alpha q)^{q+1}$. Applying Cramer's rule to the system of

equations $d_i x = c_i$, $i = 1, 2, \cdots, n$, we obtain a representation of x through c_i's: $x_i = \det D_i / \det D$ where D is a square submatrix of $(A^T, I)^T$ and D_i is a square matrix obtained from D by replacing the ith column by vector $(c_1, \cdots, c_n)^T$. Note that the determinant of any submatrix of $(A^T, I)^T$ equals to the determinant of a submatrix of A. By Laplac expansion, it is easy to see that $|x_i| \leq |\det D_i| \leq (\alpha q)^q (|c_1| + \cdots + |c_n|) \leq (\alpha q)^q n(\alpha + (\alpha q)^{q+1}) \leq 2(\alpha q)^{2q+1}$. □

By Lemma 10.1.3, it is enough to guess a solution x whose total size is at most $n \log_2(2(\alpha q)^{2q+1})) = O(q^2(\log_2 q + \log_2 \alpha)$. Note that the input A and b have total length at least $\beta = \sum_{i=1}^m \sum_{j=1}^n \log_2 |a_{ij}| + \sum_{j=1}^n \log_2 |b_j| \geq mn + \log_2 \alpha \geq q+$ IP is in NP. □

Theorem 10.1.4 INTEGER PROGRAMMING *is in NP.*

Proof. It follows immediately from Lemma 10.1.3. □

The defintion of the class NP involves three concepts, nondeterministic computation, polynomial-time, and decision problems. The first two concepts have been explained as above. Next, we explain what is the decision problem.

A problem is called a *decision* problem if its answer is "Yes" or "No". For example, the HAMILTONIAN CYCLE problem is a decision problem and all combinatorial optimization problems are not decision problems. However, every combinatorial optimization problem can be transformed into a decision version. For example, consider the TRAVELING SALESMAN problem as follows: *Given n cities and a distance table between n cities, find the shortest Hamiltonian tour where a Hamitonian tour is a Hamitonian cycle in the complete graph on the n cities.*

Its decision version is as follows: *Given n cities, a distance table between n cities, and an integer $K > 0$, is there a Hamiltonian tour with total distance at most K?*

Clearly, if the HAMILTONIAN CYCLE problem can be solved in polynomial-time, so is its decision version. Conversely, if its decision version can be solved in polynomial-time, then we may solve the HAMILTONIAN CYCLE problem in polynomial-time in the following way.

Let d_{min} and d_{max} be the smallest distance and the maximum distance between two cities. Let $a = nd_{min}$ and $b = nd_{max}$. Set $K = \lceil (a+b)/2 \rceil$.

Determine whether there is a tour with total distance at most K by solving the decision version of the HAMILTONIAN CYCLE problem. If answer is Yes, then set $b \leftarrow K$; else set $a \leftarrow K$. Repeat this process until $|b - a| \leq 1$. Then, compute the exact optimal objective function value of the HAMILTONIAN CYCLE problem by solving its decision version twice with $K = a$ and $K = b$, respectively. In this way, suppose the decision version of the HAMILTONIAN CYCLE problem can be solved in polynomial-time $p(n)$. Then the HAMILTONIAN CYCLE problem can be solved in polynomial-time $O(\log(nd_{max}))p(n)$.

10.2 What is NP-completeness?

A problem is NP-hard if the existence of polynomial-time solution for it implies the existence of polynomial-time solution for every problem in NP. An NP-hard problem is NP-complete if it also belongs to the class NP. The first NP-complete problem was discovered by S. Cook in 1971. To introduce this problem, let us introduce some knowledge on Boolean algebra.

A Boolean function is a function whose variable values and function value all are in $\{true, false\}$. Here, we would like to denote true by 1 and false by 0. In the following table, there are two boolean functions of two variables, *conjunction* $\wedge$ and *disjunction* $\vee$, and a Boolean function of a variable, *negation* $\neg$.

x	y	$x \wedge y$	$x \vee y$	$\neg x$
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

For simplicity, we also write $x \wedge y = xy$, $x \vee y = x + y$ and $\neg x = \bar{x}$. The conjunction and disjunction follow the commutative, associative, and distributive laws. An interesting and important law about negation is De Morgan's law, i.e. $\overline{xy} = \bar{x} + \bar{y}$ and $\overline{x + y} = \bar{x}\bar{y}$. The SATISFIABILITY (SAT) problem is defined as follows: *Given a Boolean formula F, is there a satisfied assigment for F?* An assignment to variables of F is *satisfied* if the assignment makes F equal to 1. A Boolean formula F is *satisfiable* if there exists a satisfied assignment for F.

The SAT problem has many applications. For example, the following puzzle can be formulated into an instance od SAT.

Example 10.2.1 *After three men interviewed, the department Chair said:"We need Brown and if we need John then we need David, if and only if we need either Brown or John and don't need David." If this department actually need more than one new faculty, which ones were they?*

Solution. Let B, J, and D denote respectively Brown, John, and David. What Chair said can be written as a Boolean formula

$$\begin{aligned} & [[B(\bar{J}+D)][(B+J)\bar{D}] + \overline{B(\bar{J}+D)} \cdot \overline{(B+J)\bar{D}} \\ = \ & B\bar{D}\bar{J} + (\bar{B}+J\bar{D})(\bar{B}\bar{J}+D) \\ = \ & B\bar{D}\bar{J} + \bar{B}\bar{J} + \bar{B}D \end{aligned}$$

Since this department actually need more than one new faculty, there is only one way to satisfy this Boolean formula, that is, $B = 0$, $D = J = 1$. Thus, John and David will be hired. □

Theorem 10.2.2 (Cook Theorem) *The* SAT *problem is NP-complete.*

After the first NP-complete problem is descovered, there are a large number of problems have been found to be NP-hard or NP-complete. Indeed, there are many tools passing the NP-hardness from one problem to another problem. We introduce one opf them as follows.

Consider two decision problems A and B. A is said to be polynomial-time many-one reducible to B, denoted by $A \leq_m^p B$, if there exists a polynomial-time computable mapping f from all inputs of A to inputs of B such that A receives Yes-answer on input x if and only if B receives Yes-answer on input $f(x)$.

For example, we have

Example 10.2.3 *The* Hamiltonian Cycle *problem is polynomial-time many-one reducible to the decision version of the* Traveling Salesman *problem.*

Proof. To construct this reduction, for each input graph $G = (V, E)$ of the Hamiltonian Cycle problem, we consider V as the set of cities and define a distance table D by setting

$$d(u, v) = \begin{cases} 1 & \text{if } (u, v) \in E \\ |V| + 1 & \text{otherwise.} \end{cases}$$

Moreover, set $K = |V|$. If G contains a Hamiltonian cycle, this this Hamiltonian cycle would give a tour with total distance $|V| = K$ for the TRAVELING SALESMAN problem on defined instance. Conversely, if the TRAVELING SALESMAN problem on defined instance has a Hamiltonian tour with total distance at most K, then this tour cannot contain an edge $(u, v) \notin E$ and hence it induces a Hamiltonian cycle in G. Since the reduction can be constructed in polynomial-time, it is a polynomial-time many-one reduction from the HAMILTONIAN CYCLE problem to the decision version of the TRAVELING SALESMAN problem. □

There are two important properties of the polynomial-time many-one reduction.

Proposition 10.2.4 *(a) If $A \leq_m^p B$ and $B \leq_m^p C$, then $A \leq_m^p C$.*
(b) If $A \leq_m^p B$ and $B \in P$, then $A \in P$.

Proof. (a) Let $A \leq_m^p B$ via f and $B \leq_m^p C$ via g. Then $A \leq_m^p C$ via h where $h(x) = g(f(x))$. Let f and g be computable in polynomial-times $p(n)$ and $q(n)$, respectively. Then for any x with $|x| = n$, $|f(x)| \leq p(n)$. Hence, h can be computed in time $p(n) + q(p(n))$

(b) Let $A \leq_m^p B$ via f. f is computable in polynomial-time $p(n)$ and B can be solved in polynomial-time $q(n)$. Then A can be solved in polynomia-time $p(n) + q(p(n))$. □

Property (a) indicates that $\leq_m^p$ is a partial ordering. Property (b) gives us a simple way to establish the NP-hardness of a decision problem. To show the NP-hardness of a decision problem B, it suffices to find an NP-complete problem A and prove $A \leq_m^p B$. In fact, if $B \in P$, then $A \in P$. Since A is NP-complete, every problem in NP is polynomial-time solvable. Therefore, B is NP-hard.

The SAT problem is the root to establish the NP-hardness of almost all other peoblems. However, is is hard to use the SAT problem directly to construct reduction. Often, we use an NP-complete special case of the SAT problem. To introduce this special case, let us first explain a special Boolean formula, 3CNF.

A *literal* is either a Boolean variable or a negation of a Boolean variable. An *elementary sum* is a sum of several literals. Consider an elementary sum c and a Boolean function f. If $c = 0$ implies $f = 0$, then c is called a *clause* of f. A CNF (conjunctive normal form) is a product of its clauses. A CNF is called a 3-CNF if each clause of the CNF contains exactly three distinct literals about three variables.

3SAT: Given a 3CNF F, determine whether the F is satisfiable.

Theorem 10.2.5 *The 3SAT problem is NP-complete.*

Proof. It is easy to see that the 3SAT problem belongs to NP. Next, we show $SAT \leq_m^p 3SAT$.

First, we show two facts.

(a) $w = x + y$ if and only if $p(w, x, y)$ is satisfiable where

$$p(w, x, y) = (\bar{w} + x + y)(w + \bar{x} + y)(w + x + \bar{y})(w + \bar{x} + \bar{y}).$$

(b) $w = xy$ if and only if $q(w, x, y)$ is satisfiable where

$$q(w, x, y) = p(\bar{w}, \bar{x}, \bar{y}).$$

To show (a), we note that $w = x + y$ if and only if $\bar{w}\bar{x}\bar{y} + w(x + y) = 1$. Moreover, we have

$$\begin{aligned} & \bar{w}\bar{x}\bar{y} + w(x + y) \\ = \; & (\bar{w} + x + y)(\bar{x}\bar{y} + w) \\ = \; & (\bar{w} + x + y)(\bar{x} + w)(\bar{y} + w) \\ = \; & (\bar{w} + x + y)(w + \bar{x} + y)(w + x + \bar{y})(w + \bar{x} + \bar{y}) \\ = \; & q(w, x, y). \end{aligned}$$

Therefore, (a) holds.

(b) can be derivated from (a) by noting that $w = xy$ if and only if $\bar{w} = \bar{x} + \bar{y}$.

Now, for any Boolean formula F, $F' \leftarrow 1$. Note that F must contain a term xy or $x + y$ where x and y are two literals. In the former case, replace xy by a new variable w in F and set $F' \leftarrow q(w, x, y)F'$. In the latter case, replace $x + y$ by a new variable w in F and set $F' \leftarrow p(w, x, y)F'$. Repeat this operation until F becomes a literal z. Let u and v be two new variables. Finally, set $F' \leftarrow F'(z + u + v)(z + \bar{u} + v)(z + u + \bar{v})(z + \bar{u} + \bar{v})$. Then the original F is satisfiable if and only if F' is satisfiable. □

Many combinatorial optimization problems have their NP-hardness established through reduction from the 3SAT problem. This means that it is unlikely for them to be polynomial-time solvable. Therefore, their approximation solutions have attracted a lot of attentions. In later sections, we study some classic ones in the literature.

10.3 Hamiltonian Cycle

Consider an NP-complete decision problem A and a possible NP-hard decision problem B. How do we construct a polynomial-time many-one reduction? Every reader who has no experience would like to know the answer of this question. Of course, we would not have an efficient algorithm to produce such a reduction. Indeed, we do not know if such an algorithm exists. However, we may give some idea to follow.

Let us recall how to show a polynomial-time many-one reduction from A to B.

(1) Construct a polynomial-time computable mapping f from all inputs of problem A to inputs of problem B.

(2) Prove that problem A on input x receives Yes-answer if and only if problem B on input $f(x)$ receives Yes-answer.

Since the mapping f has to satisfy (2), the idea is to find the relationship of output of problem A and output of problem B, that is, **find the mappring from inputs to inputs through the relationship between outputs of two problems.** Let us explain this idea through an example.

Theorem 10.3.1 *The* HAMILTONIAN CYCLE *problem is NP-complete.*

Proof. We already proved previously that the HAMILTONIAN CYCLE problem belongs to NP. Next, we are going to construct a polynomial-time many-one reduction from the NP-complete 3SAT problem to the HAMILTONIAN CYCLE problem.

The input of 3SAT is a 3CNF F and the input of HAMILTONIAN-CYCLE is a graph G. We need to find a mapping f such that for any 3CNF F, $f(F)$ is a graph such that F is satisfiable if and only if $f(G)$ contains a Hamiltonian cycle. What can make F satisfiable? It is a satified assignment. Therefore, our construction should give a relationship between assignments and Hamiltonian cycles. Suppose F contains n variables $x_1, x_2, ..., x_n$ and m clauses $C_1, C_2, ..., C_m$. To do so, we first build a ladder H_i corresponding to a variable x_i as shown in Fig. 10.3. In this ladder, there are exactly

Figure 10.3: A ledder H_i.

two Hamiltonian paths corresponding two values 0 and 1 for x_i. Connect n ledders into a cycle as shown in Fig. 10.4. Then we obtained a graph H with exactly 2^n Hamiltonian cycles corresponding to 2^n assignments of F.

Figure 10.4: Graph H.

Now, we need to find a way to involve clauses. An idea is to represent each clause C_j by a point and represent the fact "clause C_j is satisfied under an assignment" by the fact "point C_j is included in the Hamiltonian cycle corresponding to the assignment". To realize this idea, for each variable x_i in clause C_j, we connected point C_j to two endpoints of an edge on the path corresponding to $x_i = 1$ (Fig. 10.5), and for each $\bar{x}_i$ in clause C_j, we connected point C_j to two endpoints of an edge on the path corresponding to $x_i = 0$. This completes our construction for graph $f(F) = G$.

Figure 10.5: A point C_j is included.

To see this construction meeting our requirement, we first assume F has a satisfied assignment σ and show that G has a Hamiltonian cycle. To this end, we find the Hamiltonian cycle C in H corresponding to the satisfied assignment. Note that each clause C_j contains a literal $y = 1$ under assignment σ. Thus, C_j is connected to endpoints of an edge (u, v) on the path corresponding to $y = 1$. Replacing this edge (u, v) by two edges (C_j, u) and (C_j, v) would include point C_j into the cycle, which would becomes a Hamiltonian cycle of G when all points C_j are included.

Conversely, suppose G has a Hamiltonian cycle C. We claim that in C, each point C_j must connect to two endpoints of an edge (u, v) in H. If our claim holds, then replace two edges (C_j, u) and (C_j, v) by edge (u, v). We would obtain a Hamiltonian cycle of graph H, corresponding an assignment of F, which makes every clause C_j satisfied.

Now, we show the claim. For contradiction, suppose that cycle C contains its two edges (C_j, u) and (C_j, v) for some clause C_j such that u and v are located in different H_i and $H_{i'}$, respectively. To find a contradiction, we look at closely the local structure of vertex u as shown in Fig. 10.6. We would

Figure 10.6: Local structure near vertex u.

construct each ledder with a Hamiltonian path of length $4m+6$ so that every clause C_j has a special location in ledder H_i and locations for different clauses with at least distance three away each other (see Fig. 10.3). This makes that at vertex u, edges possible in C form a structure as shown in Fig. 10.6(a).

In this local structure, since C contains vertex w, C must contain edges (u, w) and (w, z), which imply that (u, u') and (u, u'') are not in C. Note that either (z, z') or (z, z'') is not in C. Without loss of generality assume that (z, z') is not in C. Then edges possible in C form a structure as shown in 10.6(b). Since C contains vertices u'', w'', and z'', cycle C must contain edges (u''', u''), (u'', w''), (w'', z''), (z'', z'''). Since C contains vertex w''', C must contain edges (u''', w''') and (w''', z'''). This means that C must contain the small cycle $(u'', w'', z'', z''', w''', u''')$. However, a Hamiltonian cycle is a simple cycle which cannot properly contain a small cycle, a contradiction. □

Next, we give another example, the HAMILTONIAN PATH problem as follows: *Given a graph $G = (V, E)$, does G contains a Hamiltonian path?* A *Hamiltonian path* of a graph G is a simple path on which every vertex appears exactly once.

Theorem 10.3.2 *The* HAMILTONIAN PATH *problem is NP-complete.*

Proof. The HAMILTONIAN PATH problem belongs to NP because we can guess a permutation of all vertices in $O(n \log n)$ time and then check, in $O(n)$ time, if guessed permutation gives a Hamiltonian path. To show the NP-hardness of the HAMILTONIAN PATH problem, we may follow the proof of Theorem 10.3.1 to construct a reduction from the 3SAT problem to the HAMILTONIAN PATH problem by making a little change on graph H, which is obtained from connecting all H_i into a path instead of a cycle. However, in the following we would like to give a simple proof by reducing the HAMILTONIAN CYCLE problem to the HAMILTONIAN PATH problem.

We are going to find a polynomial-time computable mapping f from graphs to graphs such that G contains a Hamiltonian cycle if and only if $f(G)$ contains a Hamiltonian path. Our analysis sarts from how to build a relationship between a Hamiltonian cycle of G and a Hamiltonian path of $f(G)$. If $f(G) = G$, then from a Hamiltonian cycle of G we can find a Hamiltonian path of $f(G)$; however, from a Hamiltonian path of $f(G)$ we may not be able to find a Hamiltonian cycle of G. To have "if and only if" relation, we first consider a simple case that there is an edge (u, v) such that if G contains a Hamiltonian cycle C, then C must contains edge (u, v). In this special case, we may put two new edges (u, u') and (v, v') at u and v, respecyively (Fig. 10.7). For simplicity of speaking, we may call these two edges as two *horns*. Now, if G has the Hamiltonian cycle C, then $f(G)$ has a Hamiltonian path between endpoints of two horns, u' and v', Conversely,

Figure 10.7: Install two horn in a specal case.

if $f(G)$ has a Hamiltonian path, then this Hamiltonian path must have two endpoints u' and v'; hence we can get back C by deleting two horns and putting back edge (u, v).

Now, we consider the genreral case that such an edge (u, v) may not exists. Note that for any vertex u of G, suppose u have k neighbors $v_1, v_2, ..., v_k$. Then a Hamiltonian cycle of G must contain one of edges (u, v_1), (u, v_2), ..., (u, v_k). Thus, we may first connect all $v_1, v_2, ..., v_k$ to a vertex w and put a horn (w, w') at w (Fig. 10.8). This construction would work similarly as

Figure 10.8: Install two horns in general case.

above. □

As a corollary of Theorem 10.3.1, we have

Corollary 10.3.3 *The* Traveling Salesman *problem is NP-hard.*

Proof. A polynomial-time many-one reduction has been constructed from the Hamiltonian Cycle problem to the Traveling Salesman problem. □

The Longest Path problem is a maximization problem as follows: *Given a graph $G = (V, E)$ with positive edge length $c : E \to R^+$, and two vertices s and t, find a longest simple path between s and t.*

As another corollary of Theorem 10.3.1, we have

Corollary 10.3.4 *The* Longest Path *problem is NP-hard.*

Proof. We will construct a polynomial-time many-one reduction from the Hamiltonian Cycle problem to the decision version of the Longest Path problem as follows: *Given a graph $G = (V, E)$ with positive edge length $c : E \to R^+$, two vertices s and t, and an integer $K > 0$, is there a simple path between s and t with length at least K?.*

Let graph $G = (V, E)$ be an input of the Hamiltonian Cycle problem. Choose a vertex $u \in V$. We make a copy of u by adding a new vertex u' and connecting u' to all neighbors of u. Add two new edges (u, s) and (u', t). Obtained graph is denoted by $f(G)$. Let $K = |V| + 2$. We show that G contains a Hamiltonian cycle if and only if $f(G)$ contains a simple path between s and t with length at most K.

First, assume that G contains a Hamiltonian cycle C. Break C at vertex u by replacing an edge (u, v) with (u', v). We would obtain a simple path between u and u' with length $|V|$. Extend this path to s and t. We would obtain a simple path between s and t with length $|V| + 2 = K$.

Conversely, assume that $f(G)$ contains a simple path between s and t with length at most K. Then this path contains a simple subpath between u and u' with length $|V|$. Merge u and u' by replacing edge (u', v), on the subpath, with edge (u, v). Then we would obtain a Hamiltonian cycle of G. $\square$

For NP-hard optimization problems like the TRAVELING SALESMAN problem and the LONGEST PATH problem, it is unlikely to have an efficient algorithm to compute their exact optimal solution. Therefore, one usually study algorithms which produce approximation solutions for them. Such algorithms are simply called *approximations*.

For example, let us study the TRAVELING SALESMAN problem. When the given distance table satisfies the triangular inequality, that is,

$$d(a, b) + d(b, c) \geq d(a, c)$$

for any three vertices a, b and c where $d(a, b)$ is the distance between a and b, there is an easy way to obtain a tour (i.e, a Hamiltonian cycle) with total distance within twice from the optimal.

To do so, at the first compute a minimum spanning tree in the input graph and then travel around the minimum spanning tree (see Fig. 10.9). During this trip, a vertex which appearing at the second time can be skipped without increasing the total distance of the trip due to the triangular inequality. Note that the length of a minimum spanning tree is smaller than the minimum length of a tour. Moreover, this trip uses each edge of the minimum spanning tree exactly twice. Thus, the length of the Hamiltonian cycle obtained from this trip is within twice from the optimal.

Christofids in 1976 introduced an idea to improve above approximation. After computing the minimum spanning tree, he consider all vertices of odd degree in the tree and compute a minimum perfect matching among these odd vertices. Because in the union of the minimum spanning tree and the minimum perfect matching, every vertex has even degree, one can travel along edges in this union using each edge exactly once. This trip, called *Euler tour*, produces an approximation of length bounded by the length of minimum spanning tree plus the length of the minimum perfect matching on the set of vertices with odd degree. We claim that each Hamiltonian

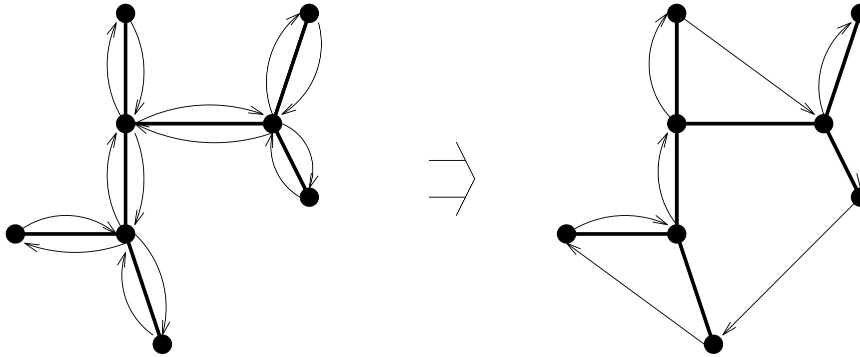

Figure 10.9: Travel around the minimum spanning tree.

cycle (namely, a traveling salesman tour) can be decomposed into a disjoint union of two parts that each is not smaller than the minimum perfect matchings for vertices with odd degree. To see this, we first note that the number of vertices with odd degree is even since the sum of degrees over all vertices in a graph is even. Now, let $x_1, x_2, \cdots, x_{2k}$ denote all vertices with odd degree in clockwise ordering of the considered Hamiltonian cycle. Then $(x_1, x_2), (x_3, x_4), \cdots, (x_{2k-1}, x_{2k})$ form a perfect matching for vertices with odd degree and $(x_2, x_3), (x_4, x_5), \cdots, (x_{2k}, x_1)$ form the other perfect matching. The claim then follows immediately from the triangular inequality. Thus, the length of the minimum matching is at most half of the length of the minimum Hamiltonian cycle. Therefore, Christofids gave an approximation within one and a half from the optimal.

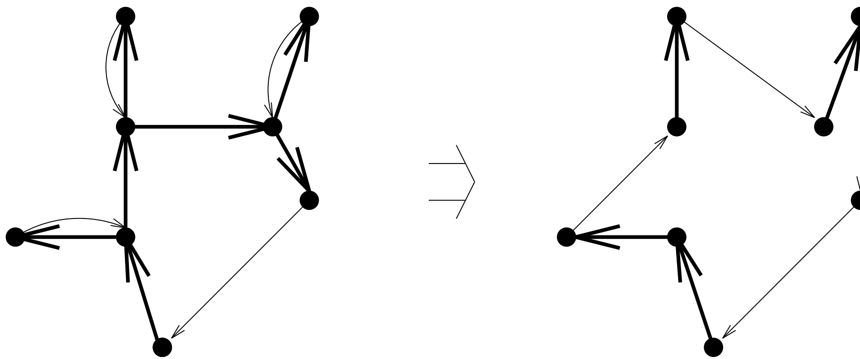

Figure 10.10: Christofids approximation.

From the above example, we see that the ratio of objective function values between approximation solution and optimal solution is a measure for the performance of an approximation.

For a minimization problem, the *performance ratio* of an approximation algorithm A is defined as follows:

$$r(A) = \sup_I \frac{A(I)}{OPT(I)}$$

where I is over all possible instances and $A(I)$ and $OPT(I)$ are respectively the objective function values of the approximation produced by algorithm A and the optimal solution with respect to instance I.

For a maximization problem, the performance ratio of an approximation algorithm A is defined by

$$r(A) = \sup_I \frac{OPT(I)}{A(I)}.$$

For example, the performance ratio of Christofids approximation is at most 3/2 as we showed in the above. Actually, the performance ratio of Christofids approximation is exactly 3/2. To see this, we consider $2n+1$ points (vertices) with distances as shown in Figure 10.11. The minimum spanning tree of these $2n+1$ points has distance $2n$. It has only two odd vertices with distance $n(1+\varepsilon)$. Hence, the length of Christofids approximation is $2n+n(1+\varepsilon)$. Moreover, the minimum tour has length $(2n-1)(1+\varepsilon)+2$. Thus, in this example, $A(I)/OPT(I) = (3n+n\varepsilon)/(2n+1+(2n-1)\varepsilon))$ which is appoarch to 3/2 as ε goes to 0 and n goes to infinity.

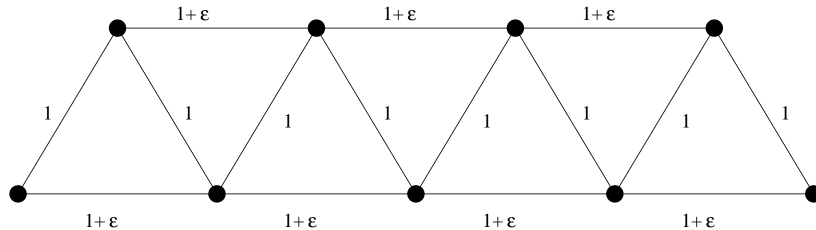

Figure 10.11: Extremal case for Christofids approximation.

Theorem 10.3.5 *For the* TRAVELING SALESMAN *problem in metric space, the Christofids approximation A has the performance ratio $r(A) = 3/2$.*

For simplicity, an approximation A is said to be α-approximation if $r(A) \leq \alpha$. For example, Christofids approximation is a 1.5-approximation, but not α-approximation of the TRAVELING SALESMAN problem in metric space for any constant $\alpha < 1.5$.

Not every problem has a polynomial-time approximation with constant performance ratio. TSP without request for triangular inequality is an example. In fact, for contradition, performance ratio $r(A) < K$ for a constant K. Then we show that HAMILTONIAN CYCLE can be solved in polynomial time. For any graph $G = (V, E)$, define that for any pair of vertice u and v,

$$d(u,v) = \begin{cases} 1 & \text{if } \{u,v\} \in E \\ K \cdot |V| & \text{otherwise} \end{cases}$$

This gives an instance I for TSP. Then, G has a Hamiltonian cycle if and only if for I, the travel salesman has a tour with length less than $K|V|$. Th optimal tour has length $|V|$. Applying approximation algorithm A to I, we obtain a tour of length less than $K|V|$. Thus, G has a Hamiltonian cycle if and only if approximation algorithm A produces a tour of length less than $K|V|$. This means that Hamiltonian Ccycle can be solved in polynomial time. Because the HAMILTONIAN CYCLE problem is NP-complete, the above argument proved the following.

Theorem 10.3.6 *If $P \neq NP$, then no polynomial-time approximation algorithm for the* TRAVELING SALESMAN *problem in general case has a constant performance ratio.*

10.4 Partition

Given n positive integers $a_1, a_2, ..., a_n$, is there a partition (N_1, N_2) of $[n]$ such that $\sum_{i \in N_1} a_i = \sum_{i \in N_2} a_i$? This problem is called the PARTITION problem. In this section, we show the NP-completeness of this problem and some consequences.

The PARTITION problem is a special case of the SUBSUM problem as follows: Given $n+1$ positive integers $a_1, a_2, ..., a_n$ and L where $1 \leq p \leq S = \sum_{i=1}^{n} a_i$, is there a subset N_1 of $[n]$ such that $\sum_{i \in N_1} a_i = L$? In fact, the PARTITION problem is equivalent to the SUBSUM problem with $L = S/2$.

Let us first show the following.

Theorem 10.4.1 *The* SUBSUM *problem is NP-complete.*

Proof. The SUBSUM problem belongs to NP because it can be done in polynomial-time to guess a subset N_1 of $[n]$ in $O(n)$ time and check whether $\sum_{i \in N_1} a_i = L$.

Next, we show $3SAT \leq_m^p$ SUBSUM. Let F be a 3CNF with n variables $x_1, x_2, ..., x_n$ and m clauses $C_1, C_2, ..., C_m$. For each variable x_i, we construct two positive decimal integers b_{x_i} and $b_{\bar{x}_i}$, representing two literals x_i and $\bar{x}_i$, respectively. Each b_{x_i} ($b_{\bar{x}_i}$) contains $m + n$ digits. Let $b_{x_i}[k]$ ($b_{\bar{x}_i}[k]$) be the kth rightmost digit of b_{x_i} ($b_{\bar{x}_i}$). Set

$$b_{x_i}[k] = b_{\bar{x}_i}[k] = \begin{cases} 1 & \text{if } k = i, \\ 0 & \text{otherwise} \end{cases}$$

for recording the ID of variable x_i. To record information on relationship between literals and clauses, set

$$b_{x_i}[n + j] = \begin{cases} 1 & \text{if } x_i \text{ appears in clause } C_j, \\ 0 & \text{otherwise,} \end{cases}$$

and

$$b_{\bar{x}_i}[n + j] = \begin{cases} 1 & \text{if } \bar{x}_i \text{ appears in clause } C_j, \\ 0 & \text{otherwise.} \end{cases}$$

Finally, define $2m + 1$ addtional positive integers c_j, c'_j for $1 \leq j \leq m$ and L as follows:

$$c_j[k] = c'_j[k] = \begin{cases} 1 & \text{if } k = n + j, \\ 0 & \text{otherwise.} \end{cases}$$

$$L = \overbrace{3...3}^{m} \overbrace{1...1}^{n}.$$

For example, if $F = (x_1 + x_2 + \bar{x}_3)(\bar{x}_2 + \bar{x}_3 + x_4)$, then we would construct the following $2(m + n) + 1 = 11$ positive integers.

$$\begin{aligned} b_{x_1} &= 010001, & b_{\bar{x}_1} &= 000001, \\ b_{x_2} &= 010010, & b_{\bar{x}_2} &= 100010, \\ b_{x_3} &= 000100, & b_{\bar{x}_3} &= 110100, \\ b_{x_4} &= 101000, & b_{\bar{x}_4} &= 001000, \\ c_1 = c'_1 &= 010000, & c_2 = c'_2 &= 100000, \\ L &= 331111. \end{aligned}$$

Now, we show that F has a satisfied assignment if and only if $A = \{b_{i,j} \mid 1 \leq n, j = 0, 1\} \cup \{c_j, c'_j \mid 1 \leq j \leq m\}$ has a subset A' such that the sum of all integers in A' is equal to p.

First, suppose F has a satified assignment σ. For each variable x_i, put b_{x_i} into A' if $x_i = 1$ under assignment σ and put $b_{\bar{x}_i}$ into A' if $x_i = 0$ under assignment σ. For each clause C_j, put both c_j and c'_j into A' if C_j contains exactly one satisfied literal under assignment σ, put c_j into A' if C_j contains exactly two satisfied literal under assignment σ, and put neither c_j nor c'_j into A' if all three literals in C_j are satisfied under assignment σ. Clearly, obtained A' meets the condition that the sum of all numbers in A isequal to L.

Conversely, suppose that there exists a subset A' of A such that the sum of all numbers in A is equal to L. Since $L[i] = 1$ for $1 \le i \le n$, A' contains exactly one of b_{x_i} and $b_{\bar{x}_i}$. Define an assignment σ by setting

$$x_i = \begin{cases} 1 & \text{if } b_{x_i} \in A', \\ 0 & \text{if } b_{\bar{x}_i} \in A'. \end{cases}$$

We claim that σ is a satisfied assignment for F. In fact, for any clause C_j, since $p[n+j] = 3$, there must be a b_{x_i} or $b_{\bar{x}_i}$ in A' whose the $(n+j)$th leftmost digit is 1. This means that there is a liertal with assignment 1, appearing C_j, i.e., making C_j satisfied. □

Now, we show the NP-completeness of the PARTITION problem.

Theorem 10.4.2 *The* PARTITION *problem is NP-complete.*

Proof. The PARTITION problem belongs to NP because it can be done in polynomial-time to guess a partition and to check if guessed partition meets the requirement. Next, we show SUBSUM $\le_m^p$ PARTITION.

Consider an instance of the SUBSUM problem, $n+1$ positive integers $a_1, a_2, ..., a_n$ and L where $0 < L \le S = a_1 + a_2 + \cdots + a_n$. Since the PARTITION problem is equivalent to the SUBSUM problem with $2L = S$, we may assume without of generality that $2L \neq S$. Now, consider an input for the PARTITION problem, consisting of $n+1$ positive integers $a_1, a_2, ..., a_n$ and $|2L - S|$. Next, we show that there exists a subset N_1 of $[n]$ such that $\sum_{i \in N_1} a_i = L$ if and only if $A = \{a_1, a_2, ..., a_n, |2L - S|\}$ has a partition (A_1, A_2) such the sum of all numbers in A_1 equals the sume of all numbers in A_2. Consider two cases.

Case 1. $2L > S$. First, suppose there exists a subset N_1 of $[n]$ such that $\sum_{i \in N_1} a_i = L$. Let $A_1 = \{a_i \mid i \in N_1\}$ and $A_2 = A - A_1$. Then, the sum of all numbers in A_2 is equal to

$$\sum_{i \in [n] - N_1} a_i + 2L - S = S - L + 2L - S = L = \sum_{i \in N_1} a_i.$$

Conversely, suppose A has a partition (A_1, A_2) such the sum of all numbers in A_1 equals the sume of all numbers in A_2. Without loss of generality, assume $2L - S \in A_2$. Note that the sum of all numbers in A equals $S + 2L - S = 2L$. Therefore, the sum of all numbers in A_1 equals L.

Case 2. $2L < S$. Let $L' = S - L$ and $N_2 = [n] - N_1$. Then $2L' - S > 0$ and $\sum_{i \in N_1} a_i = L$ if and only if $\sum_{i \in N_2} a_i = L'$. Therefore, this case can be done in a way similar to Case 1 by replacing L and N_1 with L' and N_2, respectively. □

We next study an optimization problem, the KNAPSACK problem as follows: Suppose you get in a cave and find n items. However, you have only a knapsack to carry them and this knapsack cannot carry all of them. The knapsack has a space limit S and the ith item takes space a_i and has value c_i. Therefore, you would face a problem of choosing a subset of items, which can be put in the knapsack, to maximize the total value of chosen items. This problem can be formulated into the following linear 0-1 programming.

$$\begin{array}{rl} \max & c_1x_1 + c_2x_2 + \cdots + c_nx_n \\ \text{subject to} & a_1x_1 + a_2x_2 + \cdots + a_nx_n \le S \\ & x_1, x_2, ..., x_n \in \{0, 1\} \end{array}$$

In this linear 0-1 programming, variable x_i is an indicator that $x_i = 1$ if the ith item is chosen, and $x_i = 0$ if the ith item is not chosen.

Theorem 10.4.3 *The* KNAPSACK *problem is NP-hard.*

Proof. For each instance of the SUBSUM problem, which consists of $n + 1$ positive integers $a_1, a_2, ..., a_n$ and L, consider the following KNAPSACK problem:

$$\begin{array}{rl} \max & a_1x_1 + a_2x_2 + \cdots + a_nx_n \\ \text{subject to} & a_1x_1 + a_2x_2 + \cdots + a_nx_n \le L. \end{array}$$

Clearly, there exista a subset N_1 of $[n]$ such that $\sum_{i \in N_1} a_i = L$ if and only if above corresponding KNAPSACK problem has an optimal solution with objective function value L. Therefore, if the KNAPSACK problem is polynomial-time solvable, so is the SUBSUM problem. □

Let $opt(k,S)$ be the objective function value of an optimal solution of the following problem:

$$\begin{aligned} \max \quad & c_1x_1 + c_2x_2 + \cdots + c_kx_k \\ \text{subject to} \quad & a_1x_1 + a_2x_2 + \cdots + a_kx_k \\ & x_1, x_2, ..., x_k \in \{0,1\}. \end{aligned}$$

Then

$$opt(k,S) = \max(opt(k-1,S), c_k + opt(k-1, S-a_k)).$$

This recursive formula gives a dynamic programming to solve the KNAPSACK problem within $O(nS)$ time. This is a pseudopolynomial-time algorithm, not a polynomial-time algorithm because the input size of S is $\lceil \log_2 S \rceil$, not S.

An optimization problem is said to have PTAS (polynomial-time approximation scheme) if for any $\varepsilon > 0$, there is a polynomial-time $(1+\varepsilon)$-approximation for the problem. The KNAPSACK problem has a PTAS. To construct a PTAS, we need to design another pseudopolynomial-time algorithm for the KNAPSACK problem.

Let $c(i,j)$ denote a subset of index set $\{1,\ldots,i\}$ such that

(a) $\sum_{k \in c(i,j)} c_k = j$ and

(b) $\sum_{k \in c(i,j)} s_k = \min\{\sum_{k\in I} s_k \mid \sum_{k \in I} c_k = j, I \subseteq \{1,...,i\}\}$.

If no index subset satisfies (a), then we say that $c(i,j)$ is undefined, or write $c(i,j) = nil$. Clearly, $opt = \max\{j \mid c(n,j) \neq nil \text{ and } \sum_{k \in c(i,j)} s_k \leq S\}$. Therefore, it suffices to compute all $c(i,j)$. The following algorithm is designed with this idea.

The 2nd Exact Algorithm for Knapsack

Initially, compute $c(1,j)$ for $j = 0,\ldots,c_{sum}$ by setting

$$c(1,j) := \begin{cases} \emptyset & \text{if } j = 0, \\ \{1\} & \text{if } j = c_1, \\ nil & \text{otherwise,} \end{cases}$$

where $c_{sum} = \sum_{i=1}^{n} c_i$.

Next, compute $c(i,j)$ for $i \geq 2$ and $j = 0,\ldots,c_{sum}$.

for $i = 2$ **to** n **do**

  for $j = 0$ **to** c_{sum} **do**

    case 1 $[c(i-1, j-c_i) = nil]$

      set $c(i,j) = c(i-1,j)$

case 2 $[c(i-1, j-c_i) \neq nil]$
and $[c(i-1, j) = nil]$
set $c(i, j) = c(i-1, j-c_i) \cup \{i\}$
case 3 $[c(i-1, j-c_i) \neq nil]$
and $[c(i-1, j) \neq nil]$
if $[\sum_{k \in c(i-1,j)} s_k > \sum_{k \in c(i-1,j-c_i)} s_k + s_i]$
then $c(i, j) := c(i-1, j-c_i) \cup \{i\}$
else $c(i, j) := c(i-1, j)$;
Finally, set $opt = \max\{j \mid c(n, j) \neq nil$ and $\sum_{k \in c(i,j)} s_k \leq S\}$.

This algorithm computes the exact optimal solution for KNAPSACK with running time $O(n^3 M \log(MS))$ where $M = \max_{1 \leq k \leq n} c_k$, because the algorithm contains two loops, the outside loop runs in $O(n)$ time, the inside loop runs in $O(nM)$ time, and the central part runs in $O(n \log(MS))$ time. This is a pseudopolynomial-time algorithm because the input size of M is $\log_2 M$, the running time is not a polynomial with respect to input size.

Now, we use the second pseudopolynomial-time algorithm to design a PTAS.

For any $\varepsilon > 0$, choose integer $h > 1/\varepsilon$. Denote $c'_k = \lfloor c_k n(h+1)/M \rfloor$ for $1 \leq k \leq n$ and consider a new KNAPSACK problem as follows:

$$\begin{array}{ll} \max & c'_1 x_1 + c'_2 x_2 + \cdots + c'_n x_n \\ \text{subject to} & s_1 x_1 + s_2 x_2 + \cdots + s_n x_n \leq S \\ & x_1, x_2, \ldots, x_n \in \{0, 1\}. \end{array}$$

Apply the second pseudopolynomial-time algorithm to this new problem. The running time will be $O(n^4 h \log(nhS))$, a polynomial-time with respect to n, h, and $\log S$. Suppose x^h is an optimal solution of this new problem. Set $c^h = c_1 x^* h1 + \cdots + c_n x_n^h$. We claim that

$$\frac{c^*}{c^h} \leq 1 + \frac{1}{h},$$

that is, x^h is a $(1 + 1/h)$-approximation.

To show our claim, let $I^h = \{k \mid x_k^h = 1\}$ and $c^* = \sum_{k \in I^*} c_k$. Then, we have

$$c^h = \sum_{k \in I^h} \frac{c_k n(h+1)}{M} \cdot \frac{M}{n(h+1)}$$

$$
\begin{aligned}
&\geq \sum_{k\in I^h} \lfloor \frac{c_k n(h+1)}{M} \rfloor \cdot \frac{M}{n(h+1)} \\
&= \frac{M}{n(h+1)} \sum_{k\in I^h} c'_k \\
&\geq \frac{M}{n(h+1)} \sum_{k\in I^*} c'_k \\
&\geq \frac{M}{n(h+1)} \sum_{k\in I^*} (\frac{c_k n(h+1)}{M} - 1) \\
&\geq opt - \frac{M}{h+1} \\
&\geq opt(1 - \frac{1}{h+1}).
\end{aligned}
$$

A PTAS is called a FPAS (fully polynomial-time approximation scheme) if for any $\varepsilon > 0$, there exists a $(1+\varepsilon)$-approximation with running time which is a polynomial with respect to $1/\varepsilon$ and the input size. For example, above PTAS is actually a FPTAS for the KNAPSACK problem.

Theorem 10.4.4 *The* KNAPSACK *problem has FPTAS.*

For an application of this result, we study a scheduling problem.

Suppose there are two identical machines and n jobs $J_1, ..., J_n$. Each job J_i has a processing time a_i, which does not allow preemption, i.e., the processing cannot be cut. All jobs are available at the beginning. The problem is to find a scheduling to minimize the complete time, called *makespan*. This problem is equivalent to find a partition (N_1, N_2) for $[n]$ to minimize $\max(\sum_{i\in N_1} a_i, \sum_{i\in N_2} a_i)$. This problem is NP-hard since it is easy to reduce the PARTITION problem to the decision version of this problem. We claim that this problem has also a FPTAS.

To this end, we consider the following KNAPSACK problem:

$$
\begin{aligned}
\max \quad & a_1x_1 + a_2x_2 + \cdots + a_nx_n \\
\text{subject to} \quad & a_1x_1 + a_2 + \cdots + a_nx_n \leq S/2 \\
& x_1, x_2, ..., x_n \in \{0, 1\}
\end{aligned}
$$

where $S = a_1 + a_2 + \cdots + a_n$. It is easy to see that if opt_k is the objective function value of an optimal solution for this KNAPSACK problem, then $opt_s = S - opt_k$ is the objective function value of an optimal solution of above scheduling problem.

Applying the FPTAS to above kapsack problem, we may obtain a $(1+\varepsilon)$-approximation solution $\hat{x}$. Let $N_1 = \{i \mid \hat{x}_i = 1\}$ and $N_2 = \{i \mid \bar{x}_i = 0\}$. Then (N_1, N_2) is a partition of $[n]$ and moreover, we have

$$\max(\sum_{i \in N_1} a_i, \sum_{i \in N_2} a_i) = \sum_{i \in N_2} a_i = S - \sum_{i \in N_1} a_i$$

and

$$\frac{opt_k}{\sum_{i \in N_1} a_i} \leq 1 + \varepsilon.$$

Therefore,

$$\frac{S - opt_s}{S - \sum_{i \in N_2} a_i} \leq 1 + varepsilon,$$

that is,

$$S - \sum_{i \in N_2} a_i \geq (S - opt_s)/(1 + varepsilon).$$

Thus,

$$\sum_{i \in N_2} a_i \leq \frac{\varepsilon S + opt_s}{1 + varepsilon} \leq \frac{\varepsilon \cdot 2opt_s + opt_s}{1 + varepsilon} \leq opt_s(1 + \varepsilon).$$

Therefore, (N_1, N_2) is a $(1+\varepsilon)$-approximation solution for the scheduling problem.

10.5 Vertex Cover

Given a graph $G = (V, E)$ and a positive integer K, is there a vertex cover of size at most K?" This is called the VERTEX-COVER *problem. A* vertex cover *C is a subset of vertices such that every edge has at least one endpoint in C. The* VERTEX-COVER *problem is the decision version of the* MIN VERTEX-COVER *problem as follows:* Given a graph $G = (V, E)$, compute a vertex cover with minimum cardinality.

Theorem 10.5.1 *The* VERTEX-COVER *problem is NP-complete.*

Proof. *Let F be a 3-CNF of m clauses and n variables. We construct a graph $G(F)$ of $2n + 3m$ vertices as follows: For each variable x_i, we give an edge with two endpoints labeled by two literals x_i and $\bar{x}_i$. For each clause $C_j = x + y + z$, we give a triangle $j_1 j_2 j_3$ and connect j_1 to literal x, j_2 to*

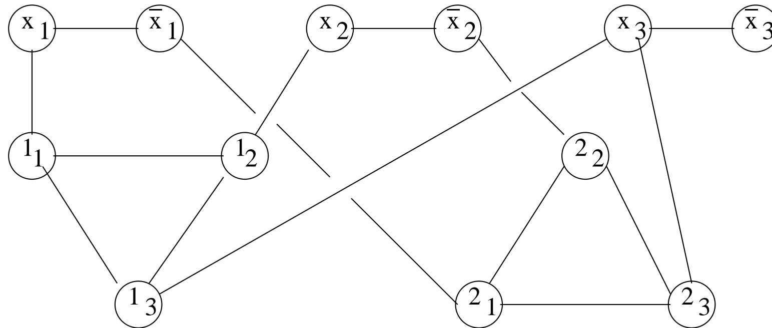

Figure 10.12: G(F)

literal y and j_3 to literal z. (Fig.1) Next, we prove that F is satisfiable if and only if $G(F)$ has a vertex-covering of size at most $n + 2m$.

First, suppose that F is satisfiable. Consider a truth-assignment. Let us construct a vertex-covering S as follows: (1) S contains all truth literals; (2) for each triangle $j_1j_2j_3$, put two vertices into S such that the remainder j_k is adjacent to a truth literal. Then S is a vertex-covering of size exactly $n + 2m$.

Conversely, suppose that $G(F)$ has a vertex-covering S of size at most $n + 2m$. Since each triangle $j_1j_2j_3$ must have at least two vertices in S and each edge $(x_i, \bar{x}_i)$ has at least one vertex in S, S is of size exactly $n + 2m$. Furthermore, each triangle $j_1j_2j_3$ must have exactly two vertices in S and each edge $(x_i, \bar{x}_i)$ must have exactly one vertex in S. Set $x_i = 1$ if $x_i \in S$ and $x_i = 0$ if $\bar{x}_i \in S$. Then each clause C_j must have a truth literal which is the one adjacent to the j_k not in S. Thus, F is satisfiable.

The above construction is clearly polynomial-time computable. Hence, 3SAT $\leq_m^p$ VC. □

Corollary 10.5.2 *The* Min Vertex-Cover *problem is NP-hard.*

Proof. *It is NP-hard since its decision version is NP-complete.* □

There are two problems closely related to the Min Vertex-Cover *problem.*

The first one is the Max Independent Set *problem:* Given a graph $G = (V, E)$, find an independent set with maximum cardinality. *Here, an independent set is a subset of vertices such that no edge exists between two vertices in the subset. A subset of vertices is an independent set if and only if its complement is a vertex cover. In fact, from the definition, every edge*

has to have at least one endpoint in the complement of an independent set, which means that the complement of an independent set must be a vertex cover. Conversely, if the complement of a vertex subset I is a vertex cover, then every edge has an endpoint not in I and hence I is independent. Furthermore, it is easy to see that A vertex subset I is the maximum independent set if and only if the complement of I is the minimum vertex cover.

The second one is the Max Clique *problem:* Given a graph $G = (V, E)$, find a clique with maximum size. *Here, a clique is a complete subgraph and its size is the number of vertices in the clique. Let $\bar{G}$ be the complementary graph of G, that is, $\bar{G} = (V, \bar{E})$ where $\bar{E}$ is the complement of E. Then a subgraph on a vertex subset I is a clique in G if and only if I is an independent set in $\bar{G}$. Thus, a subgraph on a vertex subset I is a maximum clique if and only if I is a maximum independent set in $\bar{G}$.*

From their relationship, we already see that the Min Vertex Cover *problem, the* Max Independent Set *problem and the* Max Clique *problem have the same complexity in term of computing exact optimal solutions. However, it may be interesting to point out that they have different computational complexities in term of computing approximation solutions.*

Theorem 10.5.3 *The* Min Vertex Cover *problem has a polynomial-time 2-approximation.*

Proof. *Compute a maximal matching. The set of all endpoints of edges in this maximal matching form a vertex cover which is a 2-approximation for the* Min Vertex Cover *problem since each edge in the matching must have an endpoint in the minimum vertex cover.* □

Theorem 10.5.4 *For any $\varepsilon > 0$, the* Max Independent Set *problem (the* Max Clique *problem) has no polynomial-time $n^{1-\varepsilon}$-approximation unless $NP = P$.*

*The reader may find the proof of Theorem 10.5.4 in [**?**, **?**].*

The Vertex Cover *problem can be generalized to hypergraphs. This generalization is called the* Hitting Set *problem as follows:* Given a collection $\mathcal{C}$ of subsets of a finite set X, find a minimum subset S of X such that every subset in $\mathcal{C}$ contains an element in S.

The Hitting Set *problem is equivalent to the* Set Cover *problem as follows:* Given a collection $\mathcal{C}$ of subsets of a finite set X, find a minimum set cover $\mathcal{A}$ where a set cover $\mathcal{A}$ is a subcollection of $\mathcal{C}$ such that every element of X is contained in a subset in $\mathcal{A}$.

To see the equvalence between two problems, for each element $x \in X$, define $\mathcal{S}_x = \{C \in \mathcal{C} \mid x \in C\}$. Then the SET COVER *problem is equivalent to the* HITTING SET *problem on collection $\{\mathcal{S}_x \mid x \in X\}$ and finite set $\mathcal{C}$.*

For any subcollection $\mathcal{A} \subseteq \mathcal{C}$, define

$$f(\mathcal{A}) = | \cup_{A \in \mathcal{A}} A|.$$

The SET COVER *problem has a greedy approximation as follows:*

Greedy Algorithm SC
$\mathcal{A} \leftarrow \emptyset$;
while $f(\mathcal{A}) < |S|$ **do**
  choose $A \in \mathcal{C}$ to maximize $f(\mathcal{A} \cup \{A\})$
  and set $\mathcal{A} \leftarrow \mathcal{A} \cup \{A\}$;
Output $\mathcal{A}$.

This approximation can be analysed as follows:

Lemma 10.5.5 *For any two subcollections $\mathcal{A} \subset \mathcal{B}$ and any subset $A \subseteq X$,*

$$\Delta_A f(\mathcal{A}) \geq \Delta_A f(\mathcal{B}), \tag{10.1}$$

where $\Delta_A f(\mathcal{A}) = f(\mathcal{A} \cup \{A\}) - f(\mathcal{A})$.

Proof. *Since $\mathcal{A} \subset \mathcal{B}$, we have*

$$\Delta_A f(\mathcal{A}) = |A \setminus \cup_{S \in \mathcal{A}} S| \geq |A \setminus \cup_{S \in \mathcal{B}} S| = \Delta_A f(\mathcal{B}).$$

□

Theorem 10.5.6 *Greedy Algorithm SC is a polynomial-time $(1 + \ln \gamma)$-approximation for the* SET-COVER *problem, where γ is the maximum cardinality of a subset in input collection $\mathcal{C}$.*

Proof. *Let $A_1, ..., A_g$ be subsets selected in turn by Greedy Algorithm SC. Denote $\mathcal{A}_i = \{A_1, ..., A_i\}$. Let opt be the number of subsets in a minimum set-cover.*

Let $\{C_1, ..., C_{opt}\}$ be a minimum set-cover. Denote $\mathcal{C}_j = \{C_1, ..., C_j\}$.

By the greedy rule,

$$f(\mathcal{A}_{i+1}) - f(\mathcal{A}_i) = \Delta_{A_{i+1}} f(\mathcal{A}_i) \geq \Delta_{C_j} f(\mathcal{A}_i)$$

for $1 \le j \le opt$. *Therefore,*

$$f(\mathcal{A}_{i+1}) - f(\mathcal{A}_i) \ge \frac{\sum_{j=1}^{opt} \Delta_{C_j} f(\mathcal{A}_i)}{opt}.$$

On the other hand,

$$\begin{aligned}\frac{|S| - f(\mathcal{A}_i)}{opt} &= \frac{f(\mathcal{A}_i \cup \mathcal{C}_{opt}) - f(\mathcal{A}_i)}{opt} \\ &= \frac{\sum_{j=1}^{opt} \Delta_{C_j} f(\mathcal{A}_i \cup \mathcal{C}_{j-1})}{opt}.\end{aligned}$$

By Lemma 10.5.5,

$$\Delta_{C_j} f(\mathcal{A}_i) \ge \Delta_{C_j} f(\mathcal{A}_i \cup \mathcal{C}_{j-1}).$$

Therefore,

$$f(\mathcal{A}_{i+1}) - f(\mathcal{A}_i) \ge \frac{|S| - f(\mathcal{A}_i)}{opt}, \tag{10.2}$$

that is,

$$\begin{aligned}|S| - f(\mathcal{A}_{i+1}) &\le (|S| - f(\mathcal{A}_i))(1 - \frac{1}{opt}) \\ &\le |S|(1 - \frac{1}{opt})^{i+1} \\ &\le |S| e^{-(i+1)/opt}.\end{aligned}$$

Choose i *such that* $|S| - f(\mathcal{A}_{i+1}) < opt \le |S| - f(\mathcal{A}_i)$. *Then*

$$g \le i + opt$$

and

$$opt \le |S| e^{-i/opt}.$$

Therefore,

$$g \le opt(1 + \ln \frac{|S|}{opt}) \le opt(1 + \ln \gamma).$$

□

The following theorem indicates that above greedy approximation has the best possible performance for the SET COVER *problem.*

Theorem 10.5.7 *For* $\rho < 1$, *there is no polynomial-time* $(\rho \ln n)$*-approximation for* SET-COVER *unless* $NP \subseteq DTIME(n^{O(\log \log n)})$.

*The reader may find the proof of Theorem 10.5.7 in [**?**].*

10.6 Three-dimensional Matching

10.7 Planar 3SAT

10.8 Complexity of Approximation