Chapter 13

Some $\mathcal{NP}$-Complete Problems

13.1 Statements of the Problems

In this chapter we will show that certain classical algorithmic problems are $\mathcal{NP}$-complete.

This chapter is heavily inspired by Lewis and Papadimitriou's excellent treatment [**?**].

In order to study the complexity of these problems in terms of resource (time or space) bounded Turing machines (or RAM programs), it is crucial to be able to encode instances of a problem P as strings in a language L_P.

Then an instance of a problem P is solvable iff the corresponding string belongs to the language L_P.

This implies that our problems must have a yes–no answer, which is not always the usual formulation of optimization problems where what is required is to find some *optimal* solution, that is, a solution minimizing or maximizing so objective (cost) function F.

For example the standard formulation of the traveling salesman problem asks for a tour (of the cities) of minimal cost.

Fortunately, there is a trick to reformulate an optimization problem as a yes–no answer problem, which is to explicitly incorporate a *budget* (or *cost*) term B into the problem, and instead of asking whether some objective function F has a minimum or a maximum w, we ask whether there is a solution w such that $F(w) \leq B$ in the case of a minimum solution, or $F(w) \geq B$ in the case of a maximum solution.

We will see several examples of this technique in Problems 5–8 listed below.

The problems that will consider are

(1) Exact Cover

(2) Hamiltonian Cycle for directed graphs

(3) Hamiltonian Cycle for undirected graphs

(4) The Traveling Salesman Problem

(5) Independent Set

(6) Clique

(7) Node Cover

(8) Knapsack, also called subset sum

(9) Inequivalence of $*$-free Regular Expressions

(10) The 0-1-integer programming problem

We begin by describing each of these problems.

(1) Exact Cover

We are given a finite nonempty set $U = \{u_1, \ldots, u_n\}$ (the *universe*), and a family $\mathcal{F} = \{S_1, \ldots, S_m\}$ of $m \geq 1$ *nonempty subsets* of U.

The question is whether there is an *exact cover*, that is, a subfamily $\mathcal{C} \subseteq \mathcal{F}$ of subsets in $\mathcal{F}$ such that the sets in $\mathcal{C}$ are disjoint and their union is equal to U.

For example, let
$U = \{u_1, u_2, u_3, u_4, u_5, u_6\}$, and let $\mathcal{F}$ be the family

$$\mathcal{F} = \{\{u_1, u_3\}, \{u_2, u_3, u_6\}, \{u_1, u_5\}, \{u_2, u_3, u_4\}, \{u_5, u_6\}, \{u_2, u_4\}\}.$$

The subfamily

$$\mathcal{C} = \{\{u_1, u_3\}, \{u_5, u_6\}, \{u_2, u_4\}\}$$

is an exact cover.

It is easy to see that **Exact Cover** is in $\mathcal{NP}$.

To prove that it is $\mathcal{NP}$-complete, we will reduce the **Satisfiability Problem** to it.

This means that we provide a method running in polynomial time that converts every instance of the **Satisfiability Problem** to an instance of **Exact Cover**, such that the first problem has a solution iff the converted problem has a solution.

(2) **Hamiltonian Cycle (for Directed Graphs)**

Recall that a *directed graph* G is a pair $G = (V, E)$, where $E \subseteq V \times V$.

Elements of V are called *nodes* (or *vertices*). A pair $(u, v) \in E$ is called an *edge* of G.

We will restrict ourselves to *simple graphs*, that is, graphs without edges of the form (u, u);

equivalently, $G = (V, E)$ is a simple graph if whenever $(u, v) \in E$, then $u \neq v$.

Given any two nodes $u, v \in V$, a *path from u to v* is any sequence of $n + 1$ edges $(n \geq 0)$

$$(u, v_1), (v_1, v_2), \ldots, (v_n, v).$$

(If $n = 0$, a path from u to v is simply a single edge, (u, v).)

A directed graph G is *strongly connected* if for every pair $(u, v) \in V \times V$, there is a path from u to v. A *closed path, or cycle*, is a path from some node u to itself.

We will restrict out attention to finite graphs, i.e. graphs (V, E) where V is a finite set.

Definition 13.1. Given a directed graph G, a *Hamiltonian cycle* is a cycle that passes through all the nodes exactly once (note, some edges may not be traversed at all).

Hamiltonian Cycle Problem (for Directed Graphs): Given a directed graph G, is there an Hamiltonian cycle in G?

Is there is a Hamiltonian cycle in the directed graph D shown in Figure 13.1?

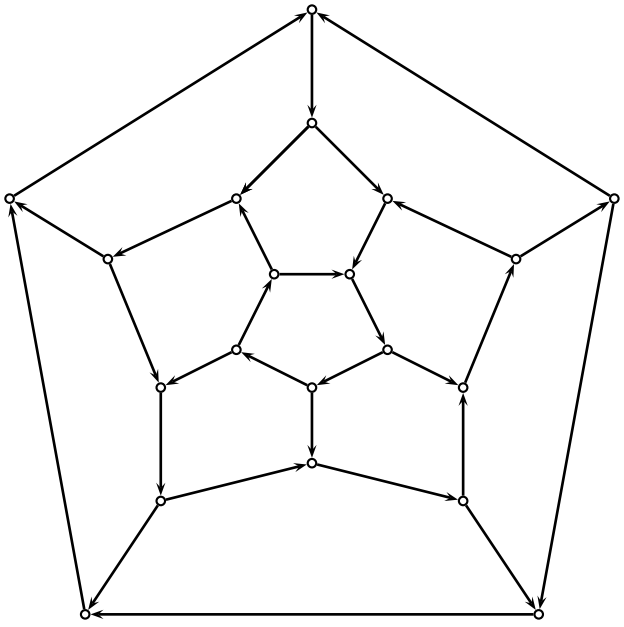

Figure 13.1: A tour "around the world."

Finding a Hamiltonian cycle in this graph does not appear to be so easy! A solution is shown in Figure 13.2 below.

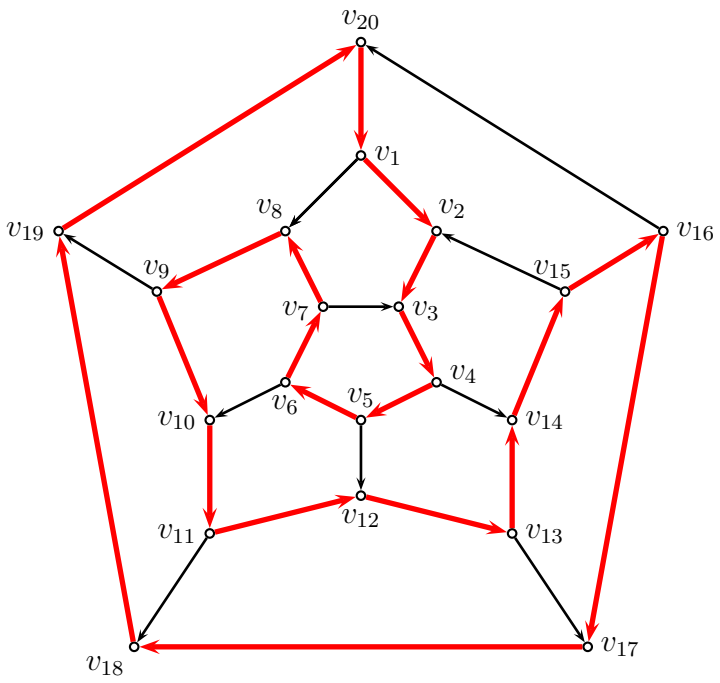

Figure 13.2: A Hamiltonian cycle in D.

It is easy to see that **Hamiltonian Cycle (for Directed Graphs)** is in $\mathcal{NP}$.

To prove that it is $\mathcal{NP}$-complete, we will reduce **Exact Cover** to it.

This means that we provide a method running in polynomial time that converts every instance of **Exact Cover** to an instance of **Hamiltonian Cycle (for Directed Graphs)** such that the first problem has a solution iff the converted problem has a solution. This is perphaps the hardest reduction.

(3) **Hamiltonian Cycle (for Undirected Graphs)**

Recall that an *undirected graph* G is a pair $G = (V, E)$, where E is a set of subsets $\{u, v\}$ of V consisting of exactly two distinct elements.

Elements of V are called *nodes* (or *vertices*). A pair $\{u, v\} \in E$ is called an *edge* of G.

Given any two nodes $u, v \in V$, a *path from u to v* is any sequence of n nodes ($n \geq 2$)

$$u = u_1, u_2, \ldots, u_n = v$$

such that $\{u_i, u_{i+1}\} \in E$ for $i = 1, \ldots, n-1$. (If $n = 2$, a path from u to v is simply a single edge, $\{u, v\}$.)

An undirected graph G is *connected* if for every pair $(u, v) \in V \times V$, there is a path from u to v.

A *closed path, or cycle*, is a path from some node u to itself.

Definition 13.2. Given an undirected graph G, a *Hamiltonian cycle* is a cycle that passes through all the nodes exactly once (note, some edges may not be traversed at all).

Hamiltonian Cycle Problem (for Undirected Graphs): Given an undirected graph G, is there an Hamiltonian cycle in G?

An instance of this problem is obtained by changing every directed edge in the directed graph of Figure 13.1 to an undirected edge.

The directed Hamiltonian cycle given in Figure 13.2 is also an undirected Hamiltonian cycle of the undirected graph of Figure 13.3.

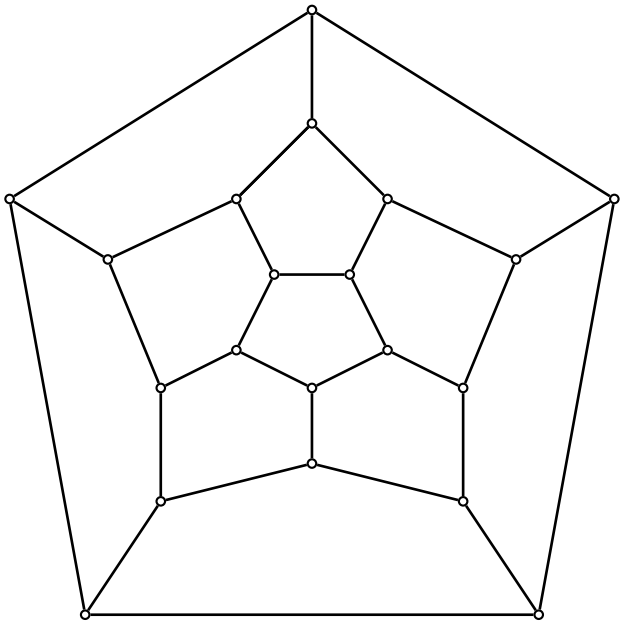

Figure 13.3: A tour "around the world," undirected version.

We see immediately that **Hamiltonian Cycle (for Undirected Graphs)** is in $\mathcal{NP}$.

To prove that it is $\mathcal{NP}$-complete, we will reduce **Hamiltonian Cycle (for Directed Graphs)** to it.

This means that we provide a method running in polynomial time that converts every instance of **Hamiltonian Cycle (for Directed Graphs)** to an instance of **Hamiltonian Cycle (for Undirected Graphs)** such that the first problem has a solution iff the converted problem has a solution. This is an easy reduction.

(4) **Traveling Salesman Problem**

We are given a set $\{c_1, c_2, \ldots, c_n\}$ of $n \geq 2$ cities, and an $n \times n$ matrix $D = (d_{ij})$ of nonnegative integers, where d_{ij} is the *distance* (or *cost*) of traveling from city c_i to city c_j.

We assume that $d_{ii} = 0$ and $d_{ij} = d_{ji}$ for all i, j, so that the matrix D is symmetric and has zero diagonal.

Traveling Salesman Problem: Given some $n \times n$ matrix $D = (d_{ij})$ as above and some integer $B \geq 0$ (the *budget* of the traveling salesman), find a permutation π of $\{1, 2, \ldots, n\}$ such that

$$c(\pi) = d_{\pi(1)\pi(2)} + d_{\pi(2)\pi(3)} + \cdots + d_{\pi(n-1)\pi(n)} + d_{\pi(n)\pi(1)} \leq B.$$

The quantity $c(\pi)$ is the *cost* of the trip specified by π.

The Traveling Salesman Problem has been stated in terms of a budget so that it has a yes or no answer, which allows us to convert it into a language. A minimal solution corresponds to the smallest feasible value of B.

Example 13.1. Consider the 4×4 symmetric matrix given by

$$D = \begin{pmatrix} 0 & 2 & 1 & 1 \\ 2 & 0 & 1 & 1 \\ 1 & 1 & 0 & 3 \\ 1 & 1 & 3 & 0 \end{pmatrix},$$

and the budget $B = 4$. The tour specified by the permutation

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 4 & 2 & 3 \end{pmatrix}$$

has cost 4, since

$$\begin{aligned} c(\pi) &= d_{\pi(1)\pi(2)} + d_{\pi(2)\pi(3)} + d_{\pi(3)\pi(4)} + d_{\pi(4)\pi(1)} \\ &= d_{14} + d_{42} + d_{23} + d_{31} \\ &= 1 + 1 + 1 + 1 = 4. \end{aligned}$$

The cities in this tour are traversed in the order

$$(1, 4, 2, 3, 1).$$

It is clear that the **Traveling Salesman Problem** is in $\mathcal{NP}$.

To show that it is $\mathcal{NP}$-complete, we reduce the **Hamiltonian Cycle Problem (Undirected Graphs)** to it.

This means that we provide a method running in polynomial time that converts every instance of **Hamiltonian Cycle Problem (Undirected Graphs)** to an instance of the **Traveling Salesman Problem** such that the first problem has a solution iff the converted problem has a solution.

(5) **Independent Set**

The problem is this: Given an undirected graph $G = (V, E)$ and an integer $K \geq 2$, is there a set C of nodes with $|C| \geq K$ such that for all $v_i, v_j \in C$, there is *no* edge $\{v_i, v_j\} \in E$?

A maximal independent set with 3 nodes is shown in Figure 13.4.

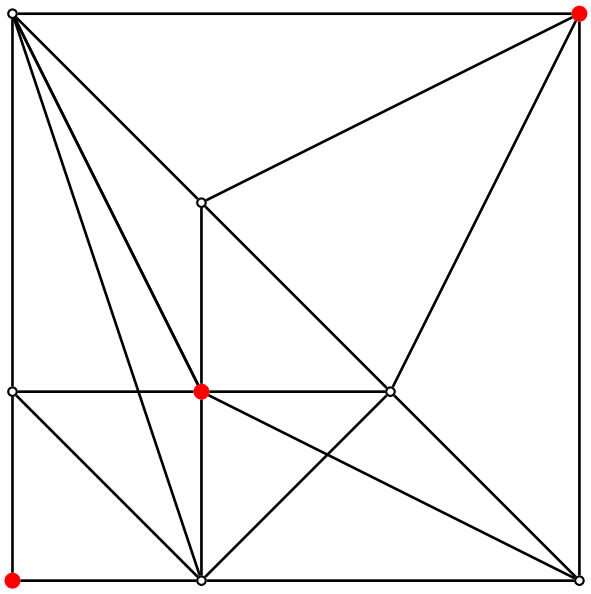

Figure 13.4: A maximal Independent Set in a graph

A maximal solution corresponds to the largest feasible value of K.

The problem **Independent Set** is obviously in $\mathcal{NP}$.

To show that it is $\mathcal{NP}$-complete, we reduce **Exact 3-Satisfiability** to it.

This means that we provide a method running in polynomial time that converts every instance of **Exact 3-Satisfiability** to an instance of **Independent Set** such that the first problem has a solution iff the converted problem has a solution.

(6) **Clique**

The problem is this: Given an undirected graph $G = (V, E)$ and an integer $K \geq 2$, is there a set C of nodes with $|C| \geq K$ such that for all $v_i, v_j \in C$, there is *some* edge $\{v_i, v_j\} \in E$?

Equivalently, does G contain a complete subgraph with at least K nodes?

A maximal clique with 4 nodes is shown in Figure 13.5.

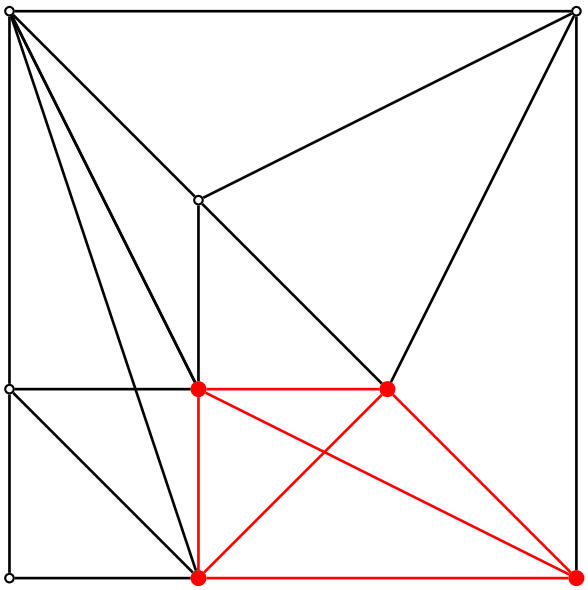

Figure 13.5: A maximal Clique in a graph

A maximal solution corresponds to the largest feasible value of K.

The problem **Clique** is obviously in $\mathcal{NP}$.

To show that it is $\mathcal{NP}$-complete, we reduce **Independent Set** to it.

This means that we provide a method running in polynomial time that converts every instance of **Independent Set** to an instance of **Clique** such that the first problem has a solution iff the converted problem has a solution.

(7) **Node Cover**

The problem is this: Given an undirected graph $G = (V, E)$ and an integer $B \geq 2$, is there a set C of nodes with $|C| \leq B$ such that *C covers all edges in G*, which means that for every edge $\{v_i, v_j\} \in E$, either $v_i \in C$ or $v_j \in C$?

A minimal node cover with 6 nodes is shown in Figure 13.6.

A minimal solution corresponds to the smallest feasible value of B.

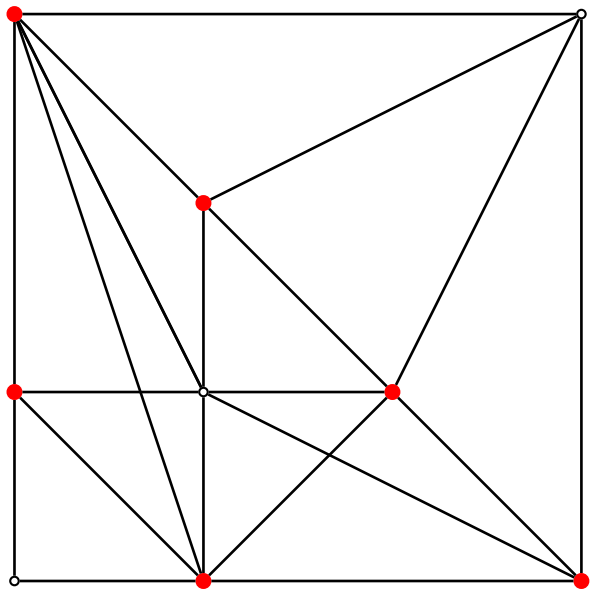

Figure 13.6: A minimal Node Cover in a graph

The problem **Node Cover** is obviously in $\mathcal{NP}$.

To show that it is $\mathcal{NP}$-complete, we reduce **Independent Set** to it.

This means that we provide a method running in polynomial time that converts every instance of **Independent Set** to an instance of **Node Cover** such that the first problem has a solution iff the converted problem has a solution.

The Node Cover problem has the following interesting interpretation:

think of the nodes of the graph as rooms of a museum (or art gallery *etc.*), and each edge as a straight corridor that joins two rooms.

Then Node Cover may be useful in assigning as few as possible guards to the rooms, so that all corridors can be seen by a guard.

(8) **Knapsack (also called Subset sum)**

The problem is this: Given a finite nonempty set $S = \{a_1, a_2, \ldots, a_n\}$ of nonnegative integers, and some integer $K \geq 0$, all represented in binary, is there a nonempty subset $I \subseteq \{1, 2, \ldots, n\}$ such that

$$\sum_{i \in I} a_i = K?$$

A "concrete" realization of this problem is that of a hiker who is trying to fill her/his backpack to its maximum capacity with items of varying weights or values.

It is easy to see that the **Knapsack** Problem is in $\mathcal{NP}$.

To show that it is $\mathcal{NP}$-complete, we reduce **Exact Cover** to it.

This means that we provide a method running in polynomial time that converts every instance of **Exact Cover** to an instance of **Knapsack** Problem such that the first problem has a solution iff the converted problem has a solution.

Remark: The **0-1 Knapsack Problem** is defined as the following problem.

Given a set of n items, numbered from 1 to n, each with a *weight* $w_i \in \mathbb{N}$ and a *value* $v_i \in \mathbb{N}$, given a maximum capacity $W \in \mathbb{N}$ and a budget $B \in \mathbb{N}$, is there a set of n variables $x_1, \ldots, x_n$ with $x_i \in \{0, 1\}$ such that

$$\sum_{i=1}^{n} x_i v_i \geq B,$$
$$\sum_{i=1}^{n} x_i w_i \leq W.$$

Informally, the problem is to pick items to include in the knapsack so that the sum of the values exceeds a given minimum B (the goal is to maximize this sum), and the sum of the weights is less than or equal to the capacity W of the knapsack.

A maximal solution corresponds to the largest feasible value of B.

The **Knapsack** Problem as we defined it (which is how Lewis and Papadimitriou define it) is the special case where $v_i = w_i = 1$ for $i = 1, \ldots, n$ and $W = B$.

For this reason, it is also called the **Subset Sum** Problem.

Clearly, the **Knapsack** (**Subset Sum**) Problem reduces to the **0-1 Knapsack** Problem, and thus the **0-1 Knapsack** Problem is also NP-complete.

(9) Inequivalence of $*$-free Regular Expressions

Recall that the problem of deciding the equivalence $R_1 \cong R_2$ of two regular expressions R_1 and R_2 is the problem of deciding whether R_1 and R_2 define the same language, that is, $\mathcal{L}[R_1] = \mathcal{L}[R_2]$.

Is this problem in $\mathcal{NP}$?

In order to show that the equivalence problem for regular expressions is in $\mathcal{NP}$ we would have to be able to somehow check in polynomial time that two expressions define the same language, but this is still an open problem.

What might be easier is to decide whether two regular expressions R_1 and R_2 are *inequivalent*.

For this, we just have to find a string w such that either $w \in \mathcal{L}[R_1] - \mathcal{L}[R_2]$ or $w \in \mathcal{L}[R_2] - \mathcal{L}[R_1]$.

The problem is that if we can guess such a string w, we still have to check in polynomial time that $w \in (\mathcal{L}[R_1] - \mathcal{L}[R_2]) \cup (\mathcal{L}[R_2] - \mathcal{L}[R_1])$, and this implies that there is a bound on the length of w which is polynomial in the sizes of R_1 and R_2.

Again, this is an open problem.

To obtain a problem in $\mathcal{NP}$ we have to consider a restricted type of regular expressions, and it turns out that $*$-free regular expressions are the right candidate.

A *$*$-free regular expression* is a regular expression which is built up from the atomic expressions using only $+$ and $\cdot$, but not $*$. For example,

$$R = ((a+b)aa(a+b) + aba(a+b)b)$$

is such an expression.

It is easy to see that if R is a $*$-free regular expression, then for every string $w \in \mathcal{L}[R]$ we have $|w| \leq |R|$. In particular, $\mathcal{L}[R]$ is finite.

The above observation shows that if R_1 and R_2 are $*$-free and if there is a string $w \in (\mathcal{L}[R_1] - \mathcal{L}[R_2]) \cup (\mathcal{L}[R_2] - \mathcal{L}[R_1])$, then $|w| \leq |R_1| + |R_2|$, so we can indeed check this in polynomial time.

It follows that the inequivalence problem for $*$ -free regular expressions is in $\mathcal{NP}$.

To show that it is $\mathcal{NP}$-complete, we reduce the **Satisfiability Problem** to it.

This means that we provide a method running in polynomial time that converts every instance of **Satisfiability Problem** to an instance of **Inequivalence of Regular Expressions** such that the first problem has a solution iff the converted problem has a solution.

(10) 0-1 **integer programming problem**

Let A be any $p \times q$ matrix with integer coefficients and let $b \in \mathbb{Z}^p$ be any vector with integer coefficients.

The 0-1 **integer programming problem** is to find whether a system of p linear equations in q variables

$$\begin{aligned} a_{11}x_1 + \cdots + a_{1q}x_q &= b_1 \\ &\vdots \qquad\qquad \vdots \\ a_{i1}x_1 + \cdots + a_{iq}x_q &= b_i \\ &\vdots \qquad\qquad \vdots \\ a_{p1}x_1 + \cdots + a_{pq}x_q &= b_p \end{aligned}$$

with $a_{ij}, b_i \in \mathbb{Z}$ has any solution $x \in \{0,1\}^q$, that is, with $x_i \in \{0,1\}$.

In matrix form, if we let

$$A = \begin{pmatrix} a_{11} & \cdots & a_{1q} \\ \vdots & \ddots & \vdots \\ a_{p1} & \cdots & a_{pq} \end{pmatrix}, \quad b = \begin{pmatrix} b_1 \\ \vdots \\ b_p \end{pmatrix}, \quad x = \begin{pmatrix} x_1 \\ \vdots \\ x_q \end{pmatrix},$$

then we write the above system as

$$Ax = b.$$

It is immediate that 0-1 **integer programming problem** is in $\mathcal{NP}$.

To prove that it is $\mathcal{NP}$-complete we reduce the **bounded tiling** problem to it.

This means that we provide a method running in polynomial time that converts every instance of the **bounded tiling** problem to an instance of the 0-1 **integer programming problem** such that the first problem has a solution iff the converted problem has a solution.

13.2 Proofs of $\mathcal{NP}$-Completeness

(1) Exact Cover

To prove that **Exact Cover** is $\mathcal{NP}$-complete, we reduce the **Satisfiability Problem** to it:

Satisfiability Problem $\leq_P$ Exact Cover

Given a set $F = \{C_1, \ldots, C_\ell\}$ of ℓ clauses constructed from n propositional variables $x_1, \ldots, x_n$, we must construct in polynomial time an instance $\tau(F) = (U, \mathcal{F})$ of **Exact Cover** such that F is satisfiable iff $\tau(F)$ has a solution.

Example 13.2. If

$$F = \{C_1 = (x_1 \vee \overline{x_2}),\ C_2 = (\overline{x_1} \vee x_2 \vee x_3),\ C_3 = (x_2),\\ C_4 = (\overline{x_2} \vee \overline{x_3})\},$$

then the universe U is given by

$$U = \{x_1, x_2, x_3, C_1, C_2, C_3, C_4, p_{11}, p_{12}, p_{21}, p_{22}, p_{23}, p_{31},\\ p_{41}, p_{42}\},$$

and the family $\mathcal{F}$ consists of the subsets

$\{p_{11}\}, \{p_{12}\}, \{p_{21}\}, \{p_{22}\}, \{p_{23}\}, \{p_{31}\}, \{p_{41}\}, \{p_{42}\}$
$T_{1,\mathbf{F}} = \{x_1, p_{11}\}$
$T_{1,\mathbf{T}} = \{x_1, p_{21}\}$
$T_{2,\mathbf{F}} = \{x_2, p_{22}, p_{31}\}$
$T_{2,\mathbf{T}} = \{x_2, p_{12}, p_{41}\}$
$T_{3,\mathbf{F}} = \{x_3, p_{23}\}$
$T_{3,\mathbf{T}} = \{x_3, p_{42}\}$
$\{C_1, p_{11}\}, \{C_1, p_{12}\}, \{C_2, p_{21}\}, \{C_2, p_{22}\}, \{C_2, p_{23}\},$
$\{C_3, p_{31}\}, \{C_4, p_{41}\}, \{C_4, p_{42}\}.$

It is easy to check that the set $\mathcal{C}$ consisting of the following subsets is an exact cover:

$T_{1,\mathbf{T}} = \{x_1, p_{21}\}, T_{2,\mathbf{T}} = \{x_2, p_{12}, p_{41}\}, T_{3,\mathbf{F}} = \{x_3, p_{23}\}$, $\{C_1, p_{11}\}, \{C_2, p_{22}\}, \{C_3, p_{31}\}, \{C_4, p_{42}\}$.

The general method to construct $(U, \mathcal{F})$ from $F = \{C_1, \ldots, C_\ell\}$ proceeds as follows. Say

$$C_j = (L_{j1} \vee \cdots \vee L_{jm_j})$$

is the jth clause in F, where L_{jk} denotes the kth literal in C_j and $m_j \geq 1$. The universe of $\tau(F)$ is the set

$$\begin{aligned} U = \{x_i \mid 1 \leq i \leq n\} &\cup \{C_j \mid 1 \leq j \leq \ell\} \\ &\cup \{p_{jk} \mid 1 \leq j \leq \ell,\ 1 \leq k \leq m_j\} \end{aligned}$$

where in the third set p_{jk} corresponds to the kth literal in C_j.

The following subsets are included in $\mathcal{F}$:

(a) There is a set $\{p_{jk}\}$ for every p_{jk}.

(b) For every boolean variable x_i, the following two sets are in $\mathcal{F}$:

$$T_{i,\mathbf{T}} = \{x_i\} \cup \{p_{jk} \mid L_{jk} = \overline{x_i}\}$$

which contains x_i and all negative occurrences of x_i, and

$$T_{i,\mathbf{F}} = \{x_i\} \cup \{p_{jk} \mid L_{jk} = x_i\}$$

which contains x_i and all its positive occurrences. Note carefully that $T_{i,\mathbf{T}}$ involves negative occurrences of x_i whereas $T_{i,\mathbf{F}}$ involves positive occurrences of x_i.

(c) For every clause C_j, the m_j sets $\{C_j, p_{jk}\}$ are in $\mathcal{F}$.

It remains to prove that F is satisfiable iff $\tau(F)$ has a solution.

We claim that if v is a truth assignement that satisfies F, then we can make an exact cover $\mathcal{C}$ as follows:

For each x_i, we put the subset $T_{i,\mathbf{T}}$ in $\mathcal{C}$ iff $v(x_i) = \mathbf{T}$, else we we put the subset $T_{i,\mathbf{F}}$ in $\mathcal{C}$ iff $v(x_i) = \mathbf{F}$.

Also, for every clause C_j, we put some subset $\{C_j, p_{jk}\}$ in $\mathcal{C}$ for a literal L_{jk} which is made true by v.

By construction of $T_{i,\mathbf{T}}$ and $T_{i,\mathbf{F}}$, this p_{jk} is not in any set in $\mathcal{C}$ selected so far. Since by hypothesis F is satisfiable, such a literal exists for every clause.

Having covered all x_i and C_j, we put a set $\{p_{jk}\}$ in $\mathcal{C}$ for every remaining p_{jk} which has not yet been covered by the sets already in $\mathcal{C}$.

Going back to Example 13.2, the truth assigment $v(x_1) = \mathbf{T}, v(x_2) = \mathbf{T}, v(x_3) = \mathbf{F}$ satisfies

$$F = \{C_1 = (x_1 \vee \overline{x_2}),\ C_2 = (\overline{x_1} \vee x_2 \vee x_3),\ C_3 = (x_2),\\ C_4 = (\overline{x_2} \vee \overline{x_3})\},$$

so we put

$$T_{1,\mathbf{T}} = \{x_1, p_{21}\}, T_{2,\mathbf{T}} = \{x_2, p_{12}, p_{41}\}, T_{3,\mathbf{F}} = \{x_3, p_{23}\},\\ \{C_1, p_{11}\}, \{C_2, p_{22}\}, \{C_3, p_{31}\}, \{C_4, p_{42}\}$$

in $\mathcal{C}$.

Conversely, if $\mathcal{C}$ is an exact cover of $\tau(F)$, we define a truth assigment as follows:

For every x_i, if $T_{i,\mathbf{T}}$ is in $\mathcal{C}$, then we set $v(x_i) = \mathbf{T}$, else if $T_{i,\mathbf{F}}$ is in $\mathcal{C}$, then we set $v(x_i) = \mathbf{F}$.

Example 13.3. Given the exact cover

$$T_{1,\mathbf{T}} = \{x_1, p_{21}\}, T_{2,\mathbf{T}} = \{x_2, p_{12}, p_{41}\}, T_{3,\mathbf{F}} = \{x_3, p_{23}\},$$
$$\{C_1, p_{11}\}, \{C_2, p_{22}\}, \{C_3, p_{31}\}, \{C_4, p_{42}\},$$

we get the satisfying assigment $v(x_1) = \mathbf{T}, v(x_2) = \mathbf{T}, v(x_3) = \mathbf{F}$.

If we now consider the proposition is CNF given by

$$F_2 = \{C_1 = (x_1 \vee \overline{x_2}),\ C_2 = (\overline{x_1} \vee x_2 \vee x_3),\ C_3 = (x_2),$$
$$C_4 = (\overline{x_2} \vee \overline{x_3} \vee x_4)\}$$

where we have added the boolean variable x_4 to clause C_4, then U also contains x_4 and p_{43} so we need to add the following subsets to $\mathcal{F}$:

$$T_{4,\mathbf{F}} = \{x_4, p_{43}\},\ T_{4,\mathbf{T}} = \{x_4\},\ \{C_4, p_{43}\},\ \{p_{43}\}.$$

The truth assigment $v(x_1) = \mathbf{T}, v(x_2) = \mathbf{T}, v(x_3) = \mathbf{F}, v(x_4) = \mathbf{T}$ satisfies F_2, so an exact cover $\mathcal{C}$ is

$$T_{1,\mathbf{T}} = \{x_1, p_{21}\}, T_{2,\mathbf{T}} = \{x_2, p_{12}, p_{41}\},$$
$$T_{3,\mathbf{F}} = \{x_3, p_{23}\}, T_{4,\mathbf{T}} = \{x_4\},$$
$$\{C_1, p_{11}\}, \{C_2, p_{22}\}, \{C_3, p_{31}\}, \{C_4, p_{42}\}, \{p_{43}\}.$$

Observe that this time, because the truth assignment v makes both literals corresponding to p_{42} and p_{43} true and since we picked p_{42} to form the subset $\{C_4, p_{42}\}$, we need to add the singleton $\{p_{43}\}$ to $\mathcal{C}$ to cover all elements of U.

(2) Hamiltonian Cycle (for Directed Graphs)

To prove that **Hamiltonian Cycle (for Directed Graphs)** is $\mathcal{NP}$-complete, we will reduce **Exact Cover** to it:

Exact Cover $\leq_P$ Hamiltonian Cycle (for Directed Graphs)

We need to find an algorithm working in polynomial time that converts an instance $(U, \mathcal{F})$ of **Exact Cover** to a directed graph $G = \tau(U, \mathcal{F})$ such that G has a Hamiltonian cycle iff $(U, \mathcal{F})$ has an exact cover.

The construction of the graph G uses a trick involving a small subgraph Gad with 7 (distinct) nodes known as a *gadget* shown in Figure 13.7.

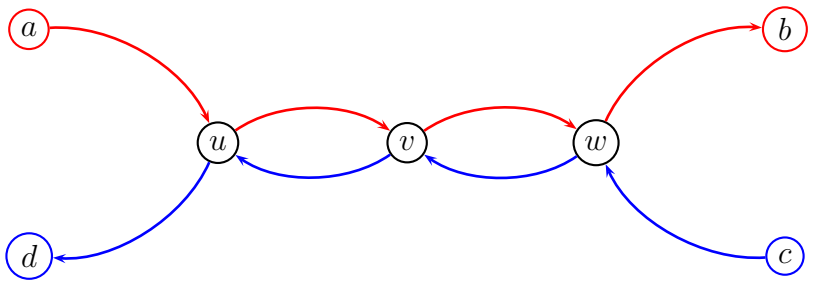

Figure 13.7: A gadget *Gad*

The crucial property of the graph *Gad* is that if *Gad* is a subgraph of a bigger graph G in such a way that no edge of G is incident to any of the nodes u, v, w unless it is one of the eight edges of *Gad* incident to the nodes u, v, w, then *for any Hamiltonian cycle in G, either the path $(a, u), (u, v), (v, w), (w, b)$ is traversed or the path $(c, w), (w, v), (v, u), (u, d)$ is traversed, but not both*.

It is convenient to use the simplified notation with a special type of edge labeled with the exclusive or sign $\oplus$ between the "edges" between a and b and between d and c, as shown in Figure 13.8.

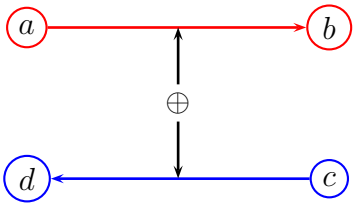

Figure 13.8: A shorthand notation for a gadget

This abbreviating device can be extended to the situation where we build gadgets between a given pair (a, b) and several other pairs $(c_1, d_1), \ldots, (c_m, d_m)$, all nodes beeing distinct, as illustrated in Figure 13.9.

Either all three edges $(c_1, d_1), (c_2, d_2), (c_3, d_3)$ are traversed or the edge (a, b) is traversed, and these possibilities are mutually exclusive.

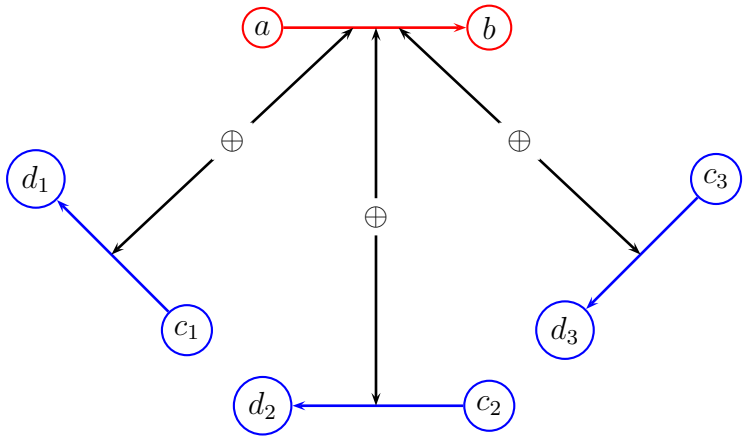

Figure 13.9: A shorthand notation for several gadgets

Example 13.4. The construction of the graph is illustrated in Figure 13.10 for the instance of the exact cover problem given by

$$U = \{u_1, u_2, u_3, u_4\},\ \mathcal{F} = \{S_1 = \{u_3, u_4\},\\ S_2 = \{u_2, u_3, u_4\},\ S_3 = \{u_1, u_2\}\}.$$

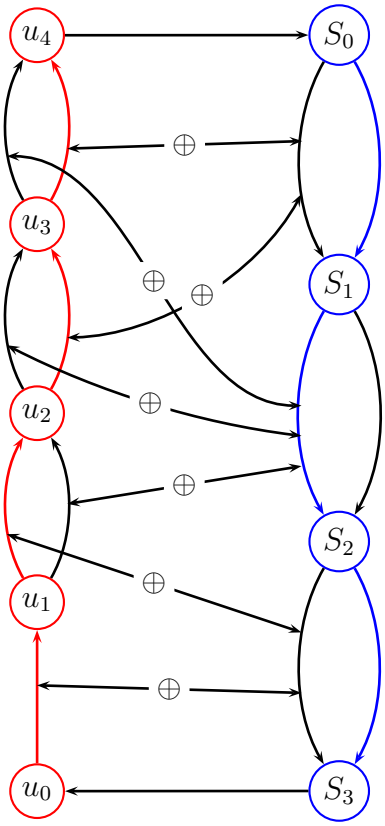

Figure 13.10: The directed graph constructed from the data $(U, \mathcal{F})$ of Example 13.4

In our example, there is a Hamiltonian where the *blue edges* are traversed between the S_i nodes, and the *red edges* are traversed between the u_j nodes.

An edge between S_i nodes which is not connected by another $\oplus$-edge is called a *short edge*, and otherwise a *long edge*.

The Hamiltonian is the following path:

$$\text{short } (S_0, S_1),\ \text{long } (S_1, S_2),\ \text{short } (S_2, S_3),\ (S_3, u_0),$$
$$(u_0, u_1)_3,\ (u_1, u_2)_3,\ (u_2, u_3)_1,\ (u_3, u_4)_1,\ (u_4, S_0).$$

Each edge between u_{j-1} and u_j corresponds to an occurrence of u_j in some uniquely determined set $S_i \in \mathcal{F}$ (that is, $u_j \in S_i$), and we put an exclusive-or edge between the edge (u_{j-1}, u_j) and the the long edge (S_{i-1}, S_i) between S_{i-1} and S_i,

The subsets corresponding to the short (S_{i-1}, S_i) edges are S_1 and S_3, and indeed $\mathcal{C} = \{S_1, S_3\}$ is an exact cover.

It can be proved that $(U, \mathcal{F})$ has an exact cover iff the graph $G = \tau(U, \mathcal{F})$ has a Hamiltonian cycle.

For example, if $\mathcal{C}$ is an exact cover for $(U, \mathcal{F})$, then consider the path in G obtained by traversing each *short edge* (S_{i-1}, S_i) for which $S_i \in \mathcal{C}$, each *edge* (u_{j-1}, u_j) such that $u_j \in S_i$, which means that this edge is connected by a $\oplus$-sign to the long edge (S_{i-1}, S_i) (by construction, for each u_j there is a unique such S_i), and the edges (u_n, S_0) and (S_m, u_0), then we obtain a Hamiltonian cycle.

In our example, the exact cover $\mathcal{C} = \{S_1, S_3\}$ yields the Hamiltonian

$$\text{short } (S_0, S_1),\ \text{long } (S_1, S_2),\ \text{short } (S_2, S_3),\ (S_3, u_0),$$
$$(u_0, u_1)_3,\ (u_1, u_2)_3,\ (u_2, u_3)_1,\ (u_3, u_4)_1,\ (u_4, S_0)$$

that we encountered earlier.

(3) Hamiltonian Cycle (for Undirected Graphs)

To show that **Hamiltonian Cycle (for Undirected Graphs)** is $\mathcal{NP}$-complete we reduce **Hamiltonian Cycle (for Directed Graphs)** to it:

Hamiltonian Cycle (for Directed Graphs) $\leq_P$ Hamiltonian Cycle (for Undirected Graphs)

Given any directed graph $G = (V, E)$ we need to construct in polynomial time an undirected graph $\tau(G) = G' = (V', E')$ such that G has a (directed) Hamiltonian cycle iff G' has a (undirected) Hamiltonian cycle.

We make three distinct copies v_0, v_1, v_2 of every node $v \in V$ which we put in V', and for every edge $(u, v) \in E$ we create five edges as illustrated in the diagram shown in Figure 13.11.

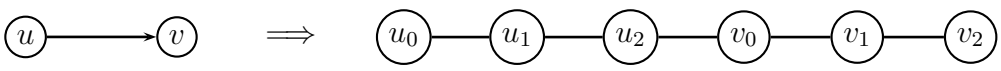

Figure 13.11: Conversion of a directed graph into an undirected graph

The crucial point about the graph G' is that although there may be several edges adjacent to a node u_0 or a node u_2, the only way to reach u_1 from u_0 is through the edge $\{u_0, u_1\}$ and the only way to reach u_1 from u_2 is through the edge $\{u_1, u_2\}$.

This implies that any Hamiltonian cycle in G' arriving to a node v_0 along an edge (u_2, v_0) must continue to node v_1 and then to v_2, which means that the edge (u, v) is traversed in G.

By considering a Hamiltonian cycle in G' or perhaps its reversal, it is not hard to show that a Hamiltonian cycle in G' determines a Hamiltonian cycle in G.

Conversely, a Hamiltonian cycle in G determines a Hamiltonian in G'.

(4) Traveling Salesman Problem

To show that the **Traveling Salesman Problem** is $\mathcal{NP}$-complete, we reduce the **Hamiltonian Cycle Problem (Undirected Graphs)** to it:

Hamiltonian Cycle Problem (Undirected Graphs) $\leq_P$ **Traveling Salesman Problem**

Given an undirected graph $G = (V, E)$, we construct an instance $\tau(G) = (D, B)$ of the traveling salesman problem so that G has a Hamiltonian cycle iff the traveling salesman problem has a solution.

If we let $n = |V|$, we have n cities and the matrix $D = (d_{ij})$ is defined as follows:

$$d_{ij} = \begin{cases} 0 & \text{if } i = j \\ 1 & \text{if } \{v_i, v_j\} \in E \\ 2 & \text{otherwise.} \end{cases}$$

We also set the budget B as $B = n$.

Any tour of the cities has cost equal to n plus the number of pairs (v_i, v_j) such that $i \neq j$ and $\{v_i, v_j\}$ is *not* an edge of G. It follows that a tour of cost n exists iff there are no pairs (v_i, v_j) of the second kind iff the tour is a Hamiltonian cycle.

(5) **Independent Set**

To show that **Independent Set** is $\mathcal{NP}$-complete, we reduce **Exact 3-Satisfiability** to it:

Exact 3-Satisfiability $\leq_P$ Independent Set

Recall that in **Exact 3-Satisfiability** every clause C_i has exactly three literals L_{i1}, L_{i2}, L_{i3}.

Given a set $F = \{C_1, \ldots, C_m\}$ of $m \geq 2$ such clauses, we construct in polynomial time an undirected graph $G = (V, E)$ such that F is satisfiable iff G has an independent set C with at least $K = m$ nodes.

For every i $(1 \leq i \leq m)$, we have three nodes c_{i1}, c_{i2}, c_{i3} corresponding to the three literals L_{i1}, L_{i2}, L_{i3} in clause C_i, so there are $3m$ nodes in V.

The "core" of G consists of m triangles, one for each set $\{c_{i1}, c_{i2}, c_{i3}\}$. We also have an edge $\{c_{ik}, c_{j\ell}\}$ iff L_{ik} and $L_{j\ell}$ are complementary literals.

Example 13.5. Let F be the set of clauses

$$F = \{C_1 = (x_1 \vee \overline{x_2} \vee x_3),\ C_2 = (\overline{x_1} \vee \overline{x_2} \vee x_3),$$
$$C_3 = (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3}),\ C_4 = (x_1 \vee x_2 \vee x_3)\}.$$

The graph G associated with F is shown in Figure 13.12.

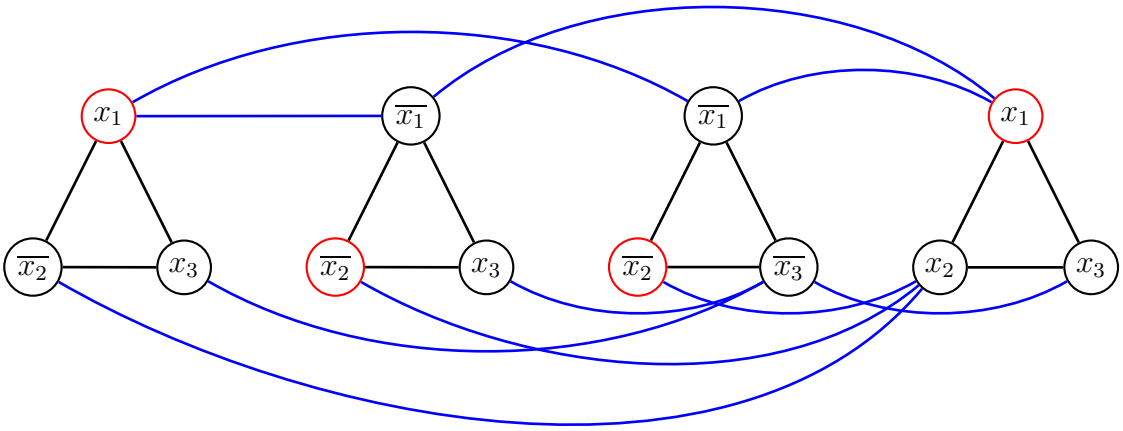

Figure 13.12: The graph constructed from the clauses of Example 13.5

Since any three nodes in a triangle are connected, an independent set C can have at most one node per triangle and thus has at most m nodes. Since the budget is $K = m$, we may assume that there is an independent with m nodes.

Define a (partial) truth assignment by

$$v(x_i) = \begin{cases} \mathbf{T} & \text{if } L_{jk} = x_i \text{ and } c_{jk} \in C \\ \mathbf{F} & \text{if } L_{jk} = \overline{x_i} \text{ and } c_{jk} \in C. \end{cases}$$

Since the non-triangle edges in G link nodes corresponding to complementary literals and nodes in C are not connected, our truth assigment does not assign clashing truth values to the variables x_i.

Not all variables may receive a truth value, in which case we assign an arbitrary truth value to the unassigned variables. This yields a satisfying assignment for F.

In Example 13.5, the set $C = \{c_{11}, c_{22}, c_{32}, c_{41}\}$ corresponding to the nodes shown in red in Figure 13.12 form an independent set, and they induce the partial truth assignment $v(x_1) = \mathbf{T}, v(x_2) = \mathbf{F}$.

The variable x_3 can be assigned an arbitrary value, say $v(x_3) = \mathbf{F}$, and v is indeed a satisfying truth assignment for F.

Conversely, if v is a truth assignment for F, then we obtain an independent set C of size m by picking for each clause C_i a node c_{ik} corresponding to a literal L_{ik} whose value under v is $\mathbf{T}$.

(6) Clique

To show that **Clique** is $\mathcal{NP}$-complete, we reduce **Independent Set** to it:

Independent Set $\leq_P$ Clique

The key the reduction is the notion of the complement of an undirected graph $G = (V, E)$.

The *complement* $G^c = (V, E^c)$ of the graph $G = (V, E)$ is the graph with the same set of nodes V as G but there is an edge $\{u, v\}$ (with $u \neq v$) in E^c iff $\{u, v\} \notin E$.

Then, it is not hard to check that there is a bijection between maximum independent sets in G and maximum cliques in G^c.

The reduction consists in constructing from a graph G its complement G^c, and then G has an independent set iff G^c has a clique.

This construction is illustrated in Figure 13.13, where a maximum independent set in the graph G is shown in blue and a maximum clique in the graph G^c is shown in red.

Figure 13.13: A graph (left) and its complement (right)

(7) Node Cover

To show that **Node Cover** is $\mathcal{NP}$-complete, we reduce **Independent Set** to it:

Independent Set $\leq_P$ Node Cover

This time the crucial observation is that if N is an independent set in G, then the complement $C = V - N$ of N in V is a node cover in G.

Thus there is an independent set of size at least K iff there is a node cover of size at most $n - K$ where $n = |V|$ is the number of nodes in V.

The reduction leaves the graph unchanged and replaces K by $n - K$.

An example is shown in Figure 13.14 where an independent set is shown in blue and a node cover is shown in red.

Figure 13.14: An inpendent set (left) and a node cover (right)

(8) Knapsack (also called Subset sum)

To show that **Knapsack** is $\mathcal{NP}$-complete, we reduce **Exact Cover** to it:

Exact Cover $\leq_P$ **Knapsack**

Given an instance $(U, \mathcal{F})$ of set cover with $U = \{u_1, \ldots, u_n\}$ and $\mathcal{F} = \{S_1, \ldots, S_m\}$, a family of subsets of U, we need to produce in polynomial time an instance $\tau(U, \mathcal{F})$ of the knapsack problem consisting of k nonnegative integers $a_1, \ldots, a_k$ and another integer $K > 0$ such that there is a subset $I \subseteq \{1, \ldots, k\}$ such that $\sum_{i \in I} a_i = K$ iff there is an exact cover of U using subsets in $\mathcal{F}$.

The trick here is the relationship between *set union* and *integer addition*.

Example 13.6. Consider the exact cover problem given by $U = \{u_1, u_2, u_3, u_4\}$ and

$$\mathcal{F} = \{S_1 = \{u_3, u_4\}, S_2 = \{u_2, u_3, u_4\}, S_3 = \{u_1, u_2\}\}.$$

We can represent each subset S_j by a binary string a_j of length 4, where the ith bit from the left is 1 iff $u_i \in S_j$, and 0 otherwise.

In our example

$$\begin{aligned} a_1 &= 0011 \\ a_2 &= 0111 \\ a_3 &= 1100. \end{aligned}$$

Then, the trick is that some family $\mathcal{C}$ of subsets S_j is an exact cover if the sum of the corresponding numbers a_j adds up to $1111 = 2^4 - 1 = K$.

For example,

$$\mathcal{C} = \{S_1 = \{u_3, u_4\},\ S_3 = \{u_1, u_2\}\}$$

is an exact cover and

$$a_1 + a_3 = 0011 + 1100 = 1111.$$

Unfortunately, there is a problem with this encoding which has to do with the fact that addition may involve carry. For example, assuming four subsets and the universe $U = \{u_1, \ldots, u_6\}$,

$$11 + 13 + 15 + 24 = 63,$$

in binary

$$001011 + 001101 + 001111 + 011000 = 111111,$$

but if we convert these binary strings to the corresponding subsets we get the subsets

$$\begin{aligned} S_1 &= \{u_3, u_5, u_6\} \\ S_2 &= \{u_3, u_4, u_6\} \\ S_3 &= \{u_3, u_4, u_5, u_6\} \\ S_4 &= \{u_2, u_3\}, \end{aligned}$$

which are not disjoint and do not cover U.

The fix is surprisingly simple: use *base m* (where m is the number of subsets in $\mathcal{F}$) instead of base 2.

Example 13.7. Consider the exact cover problem given by $U = \{u_1, u_2, u_3, u_4, u_5, u_6\}$ and $\mathcal{F}$ given by

$$\begin{aligned} S_1 &= \{u_3, u_5, u_6\} \\ S_2 &= \{u_3, u_4, u_6\} \\ S_3 &= \{u_3, u_4, u_5, u_6\} \\ S_4 &= \{u_2, u_3\}, \\ S_5 &= \{u_1, u_2, u_4\}. \end{aligned}$$

In base $m = 5$, the numbers corresponding to $S_1, \ldots, S_5$ are

$$\begin{aligned} a_1 &= 001011 \\ a_2 &= 001101 \\ a_3 &= 001111 \\ a_4 &= 011000 \\ a_5 &= 110100. \end{aligned}$$

This time,

$$\begin{aligned} a_1 + a_2 + a_3 + a_4 &= 001011 + 001101 + 001111 \\ &\quad + 011000 \\ &= 014223 \neq 111111, \end{aligned}$$

so $\{S_1, S_2, S_3, S_4\}$ is not a solution. However

$$a_1 + a_5 = 001011 + 110100 = 111111,$$

and $\mathcal{C} = \{S_1, S_5\}$ is an exact cover.

Thus, given an instance $(U, \mathcal{F})$ of **Exact Cover** where $U = \{u_1, \ldots, u_n\}$ and $\mathcal{F} = \{S_1, \ldots, S_m\}$ the reduction to **Knapsack** consists in forming the m numbers $a_1, \ldots, a_m$ (each of n bits) encoding the subsets S_j, namely $a_{ji} = 1$ iff $u_i \in S_j$, else 0, and to let $K = 1 + m^2 + \cdots + m^{n-1}$, which is represented in base m by the string $\underbrace{11 \cdots 11}_{n}$.

In testing whether $\sum_{i \in I} a_i = K$ for some subset $I \subseteq \{1, \ldots, m\}$, we use *arithmetic in base m*.

If a candidate solution $\mathcal{C}$ involves at most $m-1$ subsets, then since the corresponding numbers are added in base m, a carry can never happen.

If the candidate solution involves all m subsets, then $a_1 + \cdots + a_m = K$ iff $\mathcal{F}$ is a partition of U, since otherwise some bit in the result of adding up these m numbers in base m is not equal to 1, even if a carry occurs.

(9) Inequivalence of $*$-free Regular Expressions

To show that **Inequivalence of $*$-free Regular Expressions** is $\mathcal{NP}$-complete, we reduce the **Satisfiability Problem** to it:

Satisfiability Problem $\leq_P$ Inequivalence of $*$-free Regular Expressions

We already argued that **Inequivalence of $*$-free Regular Expressions** is in $\mathcal{NP}$ because if R is a $*$-free regular expression, then for every string $w \in \mathcal{L}[R]$ we have $|w| \leq |R|$.

We reduce the **Satisfiability Problem** to the **Inequivalence of $*$-free Regular Expressions** as follows.

For any set of clauses $P = C_1 \wedge \cdots \wedge C_p$, if the propositional variables occurring in P are $x_1, \ldots, x_n$, we produce two $*$-free regular expressions R, S over $\Sigma = \{0, 1\}$, such that P is satisfiable iff $L_R \neq L_S$.

The expression S is actually

$$S = \underbrace{(0+1)(0+1)\cdots(0+1)}_{n}.$$

The expression R is of the form

$$R = R_1 + \cdots + R_p,$$

where R_i is constructed from the clause C_i in such a way that L_{R_i} corresponds precisely to the set of truth assignments that falsify C_i; see below.

Given any clause C_i, let R_i be the $*$-free regular expression defined such that, if x_j and $\overline{x}_j$ both belong to C_i (for some j), then $R_i = \emptyset$, else

$$R_i = R_i^1 \cdot R_i^2 \cdots R_i^n,$$

where R_i^j is defined by

$$R_i^j = \begin{cases} 0 & \text{if } x_j \text{ is a literal of } C_i \\ 1 & \text{if } \overline{x}_j \text{ is a literal of } C_i \\ (0+1) & \text{if } x_j \text{ does not occur in } C_i. \end{cases}$$

(10) 0-1 integer programming problem

It is easy to check that the problem is in $\mathcal{NP}$.

To prove that the is $\mathcal{NP}$-complete we reduce the **bounded-tiling problem** to it:

bounded-tiling problem $\leq_P$ 0-1 integer programming problem

Given a tiling problem, $((\mathcal{T}, V, H), \widehat{s}, \sigma_0)$, we create a 0-1-valued variable x_{mnt}, such that $x_{mnt} = 1$ iff tile t occurs in position (m, n) in some tiling.

Write equations or inequalities expressing that a tiling exists and then use "slack variables" to convert inequalities to equations.

For example, to express the fact that every position is tiled by a single tile, use the equation

$$\sum_{t \in \mathcal{T}} x_{mnt} = 1,$$

for all m, n with $1 \leq m \leq 2s$ and $1 \leq n \leq s$. We leave the rest as as exercise.

13.3 Succinct Certificates, co$\mathcal{NP}$, and $\mathcal{EXP}$

All the problems considered in Section 13.1 share a common feature, which is that for each problem, *a solution is produced nondeterministically* (an exact cover, a directed Hamiltonian cycle, a tour of cities, an independent set, a node cover, a clique *etc.*), and then *this candidate solution is checked deterministically and in polynomial time*. The candidate solution is a string called a *certificate* (or *witness*).

It turns out that membership on $\mathcal{NP}$ can be defined in terms of certificates.

To be a certificate, a string must satisfy two conditions:

1. It must be *polynomially succinct*, which means that its length is at most a polynomial in the length of the input.
2. It must be *checkable* in polynomial time.

All "yes" inputs to a problem in $\mathcal{NP}$ must have at least one certificate, while all "no" inputs must have none.

The notion of certificate can be formalized using the notion of a polynomially balanced language.

Definition 13.3. Let Σ be an alphabet, and let ";" be a symbol not in Σ. A language $L' \subseteq \Sigma^*;\Sigma^*$ is said to be *polynomially balanced* if there exists a polynomial $p(X)$ such that for all $x, y \in \Sigma^*$, if $x;y \in L'$ then $|y| \leq p(|x|)$.

Suppose L' is a polynomially balanced language and that $L' \in \mathcal{P}$. Then we can consider the language

$$L = \{x \in \Sigma^* \mid (\exists y \in \Sigma^*)(x;y \in L')\}.$$

The intuition is that for each $x \in L$, the set

$$\{y \in \Sigma^* \mid x;y \in L'\}$$

is the set of certificates of x.

For every $x \in L$, a Turing machine can nondeterministically guess one of its certificates y, and then use the deterministic Turing machine for L' to check in polynomial time that $x; y \in L'$. It follows that $L \in \mathcal{NP}$.

Conversely, if $L \in \mathcal{NP}$ and the alphabet Σ has at least two symbols, we can encode the paths in the computation tree for every input $x \in L$, and we obtain a polynomially balanced language $L' \subseteq \Sigma^*; \Sigma^*$ in $\mathcal{P}$ such that

$$L = \{x \in \Sigma^* \mid (\exists y \in \Sigma^*)(x; y \in L')\}.$$

In summary, we obtain the following theorem.

Theorem 13.1. *Let $L \subseteq \Sigma^*$ be a language over an alphabet Σ with at least two symbols, and let ";" be a symbol not in Σ. Then $L \in \mathcal{NP}$ iff there is a polynomially balanced language $L' \subseteq \Sigma^*; \Sigma^*$ such that $L' \in \mathcal{P}$ and*

$$L = \{x \in \Sigma^* \mid (\exists y \in \Sigma^*)(x; y \in L')\}.$$

A striking illustration of the notion of succint certificate is illustrated by the set of *composite* integers, namely those natural numbers $n \in \mathbb{N}$ that can be written as the product pq of two numbers $p, q \geq 2$ with $p, q \in \mathbb{N}$.

For example, the number

$$4,294,967,297$$

is a composite!

This is far from obvious, but if an oracle gives us the certificate $\{6,700,417,\ \ 641\}$, it is easy to carry out in polynomial time the multiplication of these two numbers and check that it is equal to $4,294,967,297$.

Finding a certificate is usually (very) hard, but checking that it works is easy. This is the point of certificates.

We conclude this section with a brief discussion of the complexity classes co$\mathcal{NP}$ and $\mathcal{EXP}$.

By definition,

$$\text{co}\mathcal{NP} = \{\overline{L} \mid L \in \mathcal{NP}\},$$

that is, co$\mathcal{NP}$ consists of all complements of languages in $\mathcal{NP}$.

Since $\mathcal{P} \subseteq \mathcal{NP}$ and $\mathcal{P}$ is closed under complementation,

$$\mathcal{P} \subseteq \text{co}\mathcal{NP},$$

but nobody knows whether $\mathcal{NP}$ is closed under complementation, that is, nobody knows whether $\mathcal{NP} = \text{co}\mathcal{NP}$.

What can be shown is that if $\mathcal{NP} \neq \text{co}\mathcal{NP}$ then $\mathcal{P} \neq \mathcal{NP}$.

However it is possible that $\mathcal{P} \neq \mathcal{NP}$ and yet $\mathcal{NP} = \text{co}\mathcal{NP}$, although this is considered unlikely.

Of course, $\mathcal{P} \subseteq \mathcal{NP} \cap \text{co}\mathcal{NP}$.

There are problems in $\mathcal{NP} \cap \text{co}\mathcal{NP}$ not known to be in $\mathcal{P}$. One of the most famous in the following problem:

Integer factorization problem:

Given an integer $N \geq 3$, and another integer M (a budget) such that $1 < M < N$, does N have a factor d with $1 < d \leq M$?

That **Integer factorization** is in $\mathcal{NP}$ is clear.

To show that **Integer factorization** is in co$\mathcal{NP}$, we can guess a factorization of N into distinct factors all greater than M, check that they are prime using the results of Chapter **??** showing that testing primality is in $\mathcal{NP}$ (even in $\mathcal{P}$, but that's much harder to prove), and then check that the product of these factors is N.

It is widely believed that **Integer factorization** does not belong to $\mathcal{P}$, which is the technical justification for saying that this problem is hard.

Most cryptographic algorithms rely on this unproven fact.

If **Integer factorization** was either $\mathcal{NP}$-complete or co$\mathcal{NP}$-complete, then we would have $\mathcal{NP} = \text{co}\mathcal{NP}$, which is considered very unlikely.

Remark: If $\sqrt{N} \leq M < N$, the above problem is equivalent to asking whether N is prime.

A natural instance of a problem in co$\mathcal{NP}$ is the *unsatisfiability problem* for propositions, namely deciding that a proposition P has no satisfying assignmnent.

A proposition P (in CNF) is *falsifiable* if there is some truth assigment v such that $\widehat{v}(P) = \mathbf{F}$.

It is obvious that the set of falsifiable propositions is in $\mathcal{NP}$. Since a proposition P is valid iff P is not falsifiable, the *validity (or tautology) problem* TAUT for propositions is in co$\mathcal{NP}$.

In fact, TAUT is co$\mathcal{NP}$-complete.

Despite the fact that this problem has been extensively studied, not much is known about its exact complexity.

The reasoning used to show that TAUT is co$\mathcal{NP}$-complete can also be used to show the following interesting result.

Proposition 13.2. *If a language L is $\mathcal{NP}$-complete, then its complement $\overline{L}$ is co$\mathcal{NP}$-complete.*

The class $\mathcal{EXP}$ is defined as follows.

Definition 13.4. A deterministic Turing machine M is said to be *exponentially bounded* if there is a polynomial $p(X)$ such that for every input $x \in \Sigma^*$, there is no ID ID_n such that

$$ID_0 \vdash ID_1 \vdash^* ID_{n-1} \vdash ID_n, \quad \text{with} \quad n > 2^{p(|x|)}.$$

The class $\mathcal{EXP}$ is the class of all languages that are accepted by some exponentially bounded deterministic Turing machine.

Remark: We can also define the class $\mathcal{NEXP}$ as in Definition 13.4, except that we allow nondeterministic Turing machines.

One of the interesting features of $\mathcal{EXP}$ is that it contains $\mathcal{NP}$.

Theorem 13.3. *We have the inclusion* $\mathcal{NP} \subseteq \mathcal{EXP}$.

It is also immediate to see that $\mathcal{EXP}$ is closed under complementation. Furthermore the strict inclusion $\mathcal{P} \subset \mathcal{EXP}$ holds.

Theorem 13.4. *We have the strict inclusion* $\mathcal{P} \subset \mathcal{EXP}$.

The proof involves a diagonalization argument to produce a language E such that $E \notin \mathcal{P}$, yet $E \in \mathcal{EXP}$.

In summary, we have the chain of inclusions

$$\mathcal{P} \subseteq \mathcal{NP} \subseteq \mathcal{EXP},$$

where the left inclusion and the right inclusion are both open problems, but we know that at least one of these two inclusions is strict.

We also have the inclusions

$$\mathcal{P} \subseteq \mathcal{NP} \subseteq \mathcal{EXP} \subseteq \mathcal{NEXP}.$$

Nobody knows whether $\mathcal{EXP} = \mathcal{NEXP}$, but it can be shown that if $\mathcal{EXP} \neq \mathcal{NEXP}$, then $\mathcal{P} \neq \mathcal{NP}$; see Papadimitriou [**?**].