Chapter 9

The Theory of NP-Completeness

The birth of the theory of **NP**-Completeness is usually identified with the publication of an important theorem by Stephen Cook in 1971 [4]. In this section we will examine some of the fundamental ideas of the theory of **NP**-Completeness, including Cook's fundamental theorem. Cook's theorem has been re-expressed and re-proven in many forms since 1971; the proof provided in these notes is most similar to that given by Aho, Hopcroft, and Ullman [2].

9.1 Introduction

Let's begin by posing a question that we already know the answer to; it will be a short step from there to posing a much more interesting question:

Question 1 *If there exists a nondeterministic Turing machine that can decide a language L, is there also a deterministic Turing machine that can decide L?*

The answer to question 1 is "yes," since we have already determined that a deterministic Turing machine can compute anything that a nondeterministic machine can compute. However, this says nothing about the *time* it takes to carry out the computation. When time is taken into consideration the question becomes much more difficult to answer. As we shall see, by placing time constraints on Turing machine computations the class of Turing decidable languages splits into numerous subsets with important properties; one of the most important being the set know as **NP**-Complete.

The theory of **NP**-Completeness deals with a particular class of problems that are seemingly intractable; that is, a class of problems that clearly can be solved algorithmically, yet for which there are no known "reasonable" (or "efficient") algorithms. By "reasonable" in this context we mean algorithms that run in polynomial time[1]; algorithms that *do not* fall into this category rapidly become

[1]That is, problems (such as language recognition) that can be solved by a Turing machine in at most $p(n)$ steps, where p is *some* polynomial, and n is the length of the input string. Here, the polynomial is fixed for all input strings, but may depend on the language.

useless as the size of the problem instances grow to even modest size.

While there are many problems that can be proven to require exponential time, and therefore do not fit our definition of "reasonable," there are others for which no such proof has ever been constructed, nor have any polynomial-time algorithms been discovered. The class **NP** contains many such decision problems.

Nondeterminism does *not* make a Turing machine any more powerful in one respect; it does not allow it to *decide* any languages that would be otherwise undecidable. However, the situation appears to be quite different with respect to time-complexity; that is, nondeterminism may indeed have a profound effect on the power of a Turing machine when the number of steps to perform a computation is limited in certain ways. As yet, however, this is not a proven fact. The open question is this:

Question 2 *If there exists a nondeterministic Turing machine that can decide a language L in polynomial time, is there also a deterministic Turing machine that can decide L in polynomial time?*

Note that question 2 differs from question 1 only by the inclusion of the underlined text. This question can be expressed more succinctly as follows:

Question 3 *Is it the case that* $\mathbf{P} = \mathbf{NP}$*?*

This is the most tantalizing open question in computer science. While there is almost universal agreement that the answer is very likely "no," until it is proven there will be a shadow of doubt. The widespread feeling that $\mathbf{P} \neq \mathbf{NP}$ is the result of an ever-growing catalog of problems in **NP** for which nobody has been able to invent a polynomial-time algorithm, despite massive effort and immense economic incentives. Moreover, these problems are all connected in a very intriguing way, first noticed by Stephen Cook. Informally, Cook discovered that

Theorem 43 (Informal Statement of Cook's Theorem): *Among all the languages in the class* **NP***, the language* CNF-SAT *is the "hardest" to decide.*

By "hardest" we mean the highest asymptotic time complexity, to within composition with a polynomial. In particular, Cook showed that if we could come up with a polynomial-time algorithm for determining the satisfiability of boolean formulas in CNF form (i.e. if CNF-SAT $\in \mathbf{P}$), then we could solve *all* decision problems in **NP** in polynomial time! In other words, $\mathbf{P} = \mathbf{NP}$. We will now develop the machinery necessary to prove this formally, and investigate a few of the many profound consequences of this fact.

9.2 Polynomial-Time Reductions

Definition 27 Let L_1 and L_2 be languages over the alphabets Σ_1 and Σ_2, respectively. We define the relation $L_1 \leq_m^P L_2$ to mean that there exists a polynomial-time algorithm for computing a function $f : \Sigma_1^* \rightarrow \Sigma_2^*$, where $w \in L_1$ if and only if $f(w) \in L_2$.

If $L_1 \leq_m^P L_2$ holds, we say that L_1 is *polynomial-time many-to-one reducible*[2] (or simply *reducible* when the rest is clear from context) to L_2. Intuitively, this means that L_2 is *at least as difficult* to decide as L_1, or that L_1 is *no more difficult* to decide than L_2, modulo a polynomial.[3] The symbol $\leq$ is very appropriate for this relation, as it connotes an ordering in terms of difficulty. Moreover, the relation $\leq_m^P$ is both reflexive and transitive, and is exactly analogous to the $\leq_m$ relation we studied in the context of undecidable languages. The importance of this relation is made evident by the following theorem, which is trivial to prove.

Theorem 44 *Let L_1 and L_2 be languages. If $L_2 \in \mathbf{P}$, then $L_1 \leq_m^P L_2$ implies that $L_1 \in \mathbf{P}$.*

Stated informally, if L_2 is "easy," and L_1 is no more difficult than L_2, then L_1 is also "easy." The proof of theorem 44 follows immediately from the fact that the composition of any two polynomials is again a polynomial; that is, polynomials are closed under composition.

9.3 NP-Completeness

We begin by formally defining the class of languages known as **NP**-Complete and observing some important properties of the languages in this set. We shall also prove that this set is *not empty*, which will require a rather elaborate proof.

Definition 28 Let class **NP**-complete consists of those languages in **NP** to which *every* language in **NP** is reducible in polynomial time. That is, $L' \in$ **NP**-complete if and only if $L \leq_m^P L'$ for all $L \in$ **NP**.

Thus, **NP**-complete consists of the very "hardest" problems in **NP**, in the sense that if a polynomial-time algorithm could be found for *any* language in **NP**-complete, then *every* language in **NP** would be decidable in polynomial time. We will frequently say that "language L is **NP**-complete," or that "decision problem P is **NP**-complete" rather than saying that they are *elements* of this set.

Theorem 45 *Let $L' \in$ **NP**-complete and $L \in$ **NP** be two languages such that $L' \leq_m^P L$. Then $L \in$ **NP**-complete.*

[2]The modifier *many-to-one* simply emphasizes the fact that the function needn't be (and is generally not) injective (i.e. one-to-one). In theory, the function f need only have a range consisting of two distinct strings: one that is in L_2 and one that is not. Then f could map *all* the strings that are in L_1 to the single string that is in L_2, and *all* the strings that are not in L_1 to the single string that is not in L_2. This would fulfill the requirement of the function f. In practice, however, we generally make use of infinitely many problem instances in L_2; whether or not the function f ends up being injective is immaterial.

[3]In the context of **NP**-completeness, polynomials are routinely ignored. For example, an algorithm that runs in $\mathcal{O}(n^5)$ time is not distinguished from an algorithm that runs in $\mathcal{O}(n)$ time. While this difference is generally very significant in practice, we shall be concerned here with a much more drastic distinction; polynomial vs. non-polynomial. As composition with an arbitrary polynomial does not remove this distinction, we are justified in completely ignoring any such composition (i.e. one polynomial being composed with, or fed into, another). This is what we mean by "modulo a polynomial."

Proof: This follows from the transitivity of the $\leq_m^P$ relation. Let $L' \in$ **NP**-complete and suppose that $L' \leq_m^P L$, where $L \in$ **NP**. Then for all $L'' \in$ **NP**

$$L'' \ \leq_m^P \ L' \ \leq_m^P \ L,$$

which implies that $L'' \leq_m^P L$, by transitivity. Therefore, $L \in$ **NP**-complete. □

Thus, given a single element of **NP**-complete, we can establish that other languages are also **NP**-complete using polynomial-time reductions. However, at this point, we have not yet shown that anything at all is in **NP**-complete. The first such result was established by Stephen Cook in 1971. We can now give a precise and succinct statement of Cooks's theorem.

Theorem 46 (Cook 1971): CNF-SAT $\in$ **NP**-complete.

Proof: We will show that if there is a nondeterministic Turing machine M that decides a language L, then for each $w \in \Sigma^*$ we can construct a boolean expression in CNF form that is satisfiable if and only if $w \in L$. Moreover, we shall show that this construction can be carried out in polynomial time (as a function of the length of the input string). The upshot of this construction is that determining membership in L is no harder than determining satisfiability of CNF formulas, modulo a polynomial; that is, ignoring the polynomial cost of creating the CNF formula from the non-deterministic Turing machine, and any polynomial increase in the complexity of determining the satisfiability of the CNF formula versus running the Turing machine. In general, we shall consider any such polynomial costs to be inconsequential even though, in practice, such costs could be quite substantial.

Let M be a non-deterministic Turing machine that decides the language $L \subset \Sigma^*$ in polynomial time. We will show that $L \leq_m^P$ CNF-SAT. It will then follow that CNF-SAT $\in$ **NP**-complete, since there exists such a machine for every language in **NP**, and hence every language in **NP** reduces to CNF-SAT. The essence of the proof is to show that a boolean formula can be constructed in polynomial time directly from the definition of the Turing machine M and the input string w that is satisfiable if and only if $w \in \mathcal{L}(M)$. Moreover, we will show that such a formula can be constructed in CNF form.

Let p denote the worst-case time-complexity function of M; that is, p is a polynomial such that M will always halt within $p(n)$ steps on input strings of length n. The boolean formula $\mathcal{F}$ that we will construct must admit a satisfying truth assignment if and only if there is a sequence of valid configurations that the nondeterministic machine M can follow, given input w, that leads to an accept state after at most $p(n)$ steps, where $n = |w|$. We first examine how a satisfying truth assignment of $\mathcal{F}$ can encode a legal sequence of configurations of the machine M run on w. Let t denote "time," by which we mean a count of the discrete steps of M, with the initial configuration corresponding to $t = 0$. Then we must enforce the following properties to faithfully simulate the behavior of M as it is run on input w:

1. The read-write head of M is over exactly one cell at any time t.
2. The machine M is in exactly one state q at any time t.
3. Each cell of the tape has exactly one symbol in it at every time t.

4. The machine starts in the initial state, with read/write head over the left-most cell, which is cell 1, and with w written at the start of an otherwise blank-filled tape.

5. Only the symbol under the read/write head at time t can change at time $t+1$.

6. The state, position of the head, and symbol being scanned at time t determine the state, position of the head, and symbol last written at time $t+1$, according to the transition relation Δ of M.

An essential ingredient that enables us to model all of these properties with a *finite* (in fact, polynomially bounded) boolean formula is the fact that the machine must halt after $p(n)$ steps; therefore, the machine can access at most $p(n)$ cells of its tape. To simplify the construction, we first define a function $f : \{\top, \bot\}^n \rightarrow \{\top, \bot\}$ such that $f(x_1, \ldots, x_n)$ is $\top$ if and only if exactly one of its arguments is $\top$. The function f is easy to construct:

$$f(x_1, \ldots, x_k) \stackrel{\text{def}}{=} \left[\bigvee_{i=1}^{k} x_i\right] \wedge \left[\bigwedge_{i \neq j} (\neg x_i \vee \neg x_j)\right]. \tag{9.1}$$

The first bracketed expression on the right of Equation (9.1) ensures that *at least one* of the arguments is $\top$, while the second bracketed expression ensures that *at most one* argument is $\top$ (since *no two* arguments can be true). Observe that f is in CNF form, and that its size (i.e. the number of literals and clauses) is $\mathcal{O}(k^2)$, where k is the number of arguments.

Without loss of generality, we shall assume that M has tape alphabet $\Gamma = \{0, 1, \ldots, \gamma\}$, with 0 being the blank symbol, and states $Q = \{1, 2, \ldots, q\}$, with 1 being the initial state and q being the single accept state. Furthermore, we shall modify M so that it remains in its accept state forever once it accepts; that is, we will add transitions that keep M in state q once it has reached that state.

We now introduce a large number of boolean variables that will allow us to enforce all of the properties enumerated above during the entire operation of the machine, which will entail up to $p(n)$ steps. Let $m = p(n)$ and define the $m^2(1+\gamma) + mq$ boolean variables

$$H_i^t \quad \text{for} \quad t = 0, \ldots, m, \;\; i = 1, \ldots, m \tag{9.2}$$

$$S_j^t \quad \text{for} \quad t = 0, \ldots, m, \;\; j = 1, \ldots, q \tag{9.3}$$

$$C_{i,s}^t \quad \text{for} \quad t = 0, \ldots, m, \;\; i = 1, \ldots, m, \;\; s = 1, \ldots, \gamma, \tag{9.4}$$

which we shall interpret as follows: $H_i^t = \top$ if and only if the read-write head of M is over cell i at time t, $S_j^t = \top$ if and only if M is in state j at time t, and $C_{i,s}^t = \top$ if and only if cell i contains the

symbol s at time t. We now define the following boolean formulas using only these variables:

$$E_{\text{H}} \stackrel{\text{def}}{=} f(H_1^t, H_2^t, \ldots, H_m^t) \tag{9.5}$$

$$E_{\text{S}} \stackrel{\text{def}}{=} \bigwedge_{t=0}^{m} f(S_1^t, S_2^t, \ldots, S_q^t) \tag{9.6}$$

$$E_{\text{C}} \stackrel{\text{def}}{=} \bigwedge_{t=0}^{m} \bigwedge_{i=1}^{m} f(C_{i,1}^t, C_{i,2}^t, \ldots, C_{i,\gamma}^t) \tag{9.7}$$

$$E_{\text{I}} \stackrel{\text{def}}{=} S_1^0 \wedge H_1^0 \wedge \left[\bigwedge_{j=1}^{n} C_{j,w_j}^0\right] \wedge \left[\bigwedge_{j=n+1}^{m} C_{j,0}^0\right] \tag{9.8}$$

$$E_{\text{W}} \stackrel{\text{def}}{=} \bigwedge_{t=0}^{m-1} \bigwedge_{i=1}^{m} \left\{\bigvee_{k=0}^{\gamma} \left[(H_i^t \vee C_{i,k}^t \vee \neg C_{i,k}^{t+1}) \wedge (H_i^t \vee \neg C_{i,k}^t \vee C_{i,k}^{t+1})\right]\right\} \tag{9.9}$$

$$E_{\text{T}} \stackrel{\text{def}}{=} \bigwedge_{t=0}^{m} \bigwedge_{i=1}^{m} \left\{\bigvee_{(p,a,q,b,\delta)\in\Delta} \left[H_i^t \wedge C_{i,a}^t \wedge S_p^t \wedge H_{i+\delta}^{t+1} \wedge C_{i+\delta,b}^{t+1} \wedge S_q^{t+1}\right]\right\} \tag{9.10}$$

$$E_{\text{A}} \stackrel{\text{def}}{=} S_q^m \tag{9.11}$$

where Δ is the transition relation of M. Expression (9.5) ensures that the read-write head is positioned over exactly one cell at each time t, expression (9.6) ensures that the machine is in exactly one state at each time t, expression (9.7) ensures that each cell on the tape contains exactly one symbol, expression (9.8) ensures that the machine starts in the initial configuration, with w on its tape, expression (9.9) ensures that only the symbol under the read head can change at each time step, expression (9.10) ensures that each time step corresponds to some transition in the transition relation of M, where "left" and "right" are encoded by $\delta = -1$ and $\delta = 1$, respectively, and finally expression (9.11) ensures that the accept state is eventually reached. Observe that the inner-most bracketed expression in Equation (9.9) is equivalent to

$$\forall\, 1 \le i \le m,\ \forall\, 0 \le t \le m-1,\ \exists\, 1 \le k \le \gamma,\ H_i^t \vee (C_{i,k}^t = C_{i,k}^{t+1}). \tag{9.12}$$

That is, at each cell i and time t, *either* the read/write head is over that cell, *or* the symbol in the cell remains unchanged from time t to time $t+1$.

Thus, the boolean formula

$$\mathcal{F} \stackrel{\text{def}}{=} E_{\text{H}} \wedge E_{\text{S}} \wedge E_{\text{C}} \wedge E_{\text{W}} \wedge E_{\text{T}} \wedge E_{\text{I}} \wedge E_{\text{A}} \tag{9.13}$$

is satisfiable if and only if the machine M has a sequence of valid configurations leading to an accept state given input w; that is, if and only if $w \in \mathcal{L}(M)$. Note that nondeterministic branches of M pose no problem here; they simply lead to multiple truth assignments that model valid computations of the machine.

All the sub-expressions are already in CNF form, except for equations (9.9) and (9.10); for the latter cases, it is only the expressions in the braces that remain to be put into CNF form. We needn't convert these explicitly, however, since both of these expressions depend only on the specifics of M (i.e. the tape alphabet and transition relation), and therefore do not depend on the size of the input w. Consequently, we need only observe that any boolean formula can be converted into CNF form, and that doing so here will result in CNF formulas of *fixed size*, determined by M.

All that remains is to determine the total size of the expression (9.13), since the time required to construct this formula is clearly proportional to its size, as there is no searching of any kind involved in its construction from M. Expressions (9.6) and (9.8) are linear in m, while expressions (9.5), (9.7), (9.9), and (9.10) are quadratic in m. Thus, the entire formula is $\mathcal{O}\big(p^2(|w|)\big)$, which is clearly polynomial in the size of the input w.

We have demonstrated that $L \leq_m^P$ CNF-SAT. Since we have previously shown that CNF-SAT is in fact in **NP**, we have established that CNF-SAT is **NP**-complete. □

Many important languages with practical applications are **NP**-complete. We will typically talk about these languages in the guise of their equivalent *decision problems*, since the decision problems seem more natural, and are closer to actual applications. It should be understood, however, that each class of "decision problems" that we describe is technically a language.

1. SATISFIABILITY: Given a boolean formula $\mathcal{F}$ determine whether $\mathcal{F}$ is satisfiable.
2. 3CNF-SAT: Given a boolean formula $\mathcal{F}$ in conjunctive normal form (CNF) that has exactly three literals per clause, determine whether $\mathcal{F}$ is satisfiable.
3. 3-COLOR: Given a graph G, determine whether G can be 3-colored. That is, given an undirected graph G, determine whether there is a way to assign one of three colors to each vertex in such a way that no two adjacent vertices have the same color.
4. PLANAR-3-COLOR: Given a planar graph G, determine whether G can be 3-colored. This is the same as the previous problem, but with the extra constraint that it must be possible to draw the graph in such a way that no edges cross.
5. VERTEX-COVER: Given a graph $G = (V, E)$ and $k \in \mathbb{N}$, determine whether there exists a subset of vertices, $V' \subset V$, such that $|V'| \leq k$ and each edge in E is incident upon some vertex in V'. We think of the subset V' as "covering" the edges of G.
6. CLIQUE: Given a graph G and $k \in \mathbb{N}$, determine whether G has a subgraph of size k or larger that is a clique.
7. HAMILTONIAN-CIRCUIT: Given a graph G, determine whether G has a Hamiltonian circuit; that is, a circuit (cycle) that passes through each vertex exactly once.
8. PARTITION: Given a list of natural numbers, $(a_1, a_2, \ldots, a_n)$, determine whether the list can be split into two sublists that have the same sum.
9. SUBSET-SUM: Given a list of natural numbers, $(a_1, a_2, \ldots, a_n)$, and $k \in K$, determine whether there is a sublist that sums to k.
10. SUBGRAPH-ISOMORPHISM: Given two graphs, G_1 and G_2, and $k \in \mathbb{N}$, determine whether there exist subgraphs $G_1' \subseteq G$ and $G_2' \subseteq G$ such that G_1' and G_2' are isomorphic and have at least k vertices.
11. TRAVELING-SALESMAN: Given a function $d : [1, 2, \ldots n]^2 \rightarrow N$ and $k \in \mathbb{N}$, determine whether there is a permutation $(p_1, p_2, \ldots, p_n)$ of $(1, 2, \ldots n)$ such that $\sum_{i=1}^{n} d(p_i, p_{i+1}) \leq k$, where we define $p_{n+1} \stackrel{\text{def}}{=} p_1$. We think of the numbers $1, 2, \ldots n$ as "cities" and $d(i, j)$ as the "distance" from city i to city j. We call the permutation a "tour" of the cities.

12. EUCLIDEAN-TRAVELING-SALESMAN: Given a finite set of 2D integer coordinates, $C \subset \mathbb{N} \times \mathbb{N}$, and $k \in \mathbb{N}$, determine whether there is a simple circuit connecting all the points whose total Euclidean length is less than or equal to k. This is simply TRAVELING-SALESMAN where the distance function $d(i, j)$ is constrained to be the standard Euclidean distance between two points:

$$d(P, Q) = \left\lfloor \sqrt{(P_x - Q_x)^2 + (P_y - Q_y)^2} \right\rfloor,$$

where $P = (P_x, P_y)$ and $Q = (Q_x, Q_y)$ are elements of $\mathbb{N}^2$.

13. NESFRE (Non-Equivalence of Star-Free Regular Expressions): Given two regular expressions, R_1 and R_2, neither containing the Kleene star symbol, determine whether there exists a string that is generated by one but not the other. In other words, determine whether R_1 and R_2 are non-equivalent.

14. NERE (Non-Equivalence of Regular Expressions): Given two regular expressions, R_1 and R_2, determine whether there exists a string that is generated by one but not the other. In other words, determine whether R_1 and R_2 are non-equivalent.

15. EXACT-3-COVER: Given a finite set S and a finite collection $C \subset 2^S$ of subsets of S, each consisting of exactly three elements, determine whether C contains an exact cover of S: that is, a collection of disjoint sets whose union is S.

16. KNAPSACK: Given $S = \{(w_1, v_1), (w_2, v_2), \ldots, (w_n, v_n)\} \subset \mathbb{N}^2$, and $W, V \in \mathbb{N}$, determine whether there is a subset $S' \subset S$ such that $\sum_{(w,v)\in S'} w \leq W$ and $\sum_{(w,v)\in S'} v \geq V$. We think of the set S as a collection of items with given "weights" and "values." We wish to stuff items whose combined value is at least V into a "knapsack" that can hold at most a weight of W.

17. BIN-PACKING: Given $S = \{s_1, s_2, \ldots, s_n\} \in \mathbb{N}$, and $k, b \in \mathbb{N}$, determine whether there is a partition of S into k disjoint subsets such that the sum of the numbers in each subset is less than or equal to b.

18. GEOMETRIC-SEPARATION: Given a collection of simple non-overlapping orthogonal polygons in the plane, $P_1, P_2, \ldots, P_n$, and an integer k, determine whether polygon P_1 can be "separated" from the others by a sequence of k or fewer vertical or horizontal translations (applied to any of the polygons) in such a way that the shapes never overlap.

19. ART-GALLERY: Given a simple polygon P in the plane, with n vertices, and $k \in \mathbb{N}$, determine whether the edges of P can be "guarded" by k or fewer of the vertices. We think of the polygon as the floor plan of an art gallery. The "guards" must be able to "see" every inch of every wall to watch over the paintings. Each guard is assumed to be able to view 360 degrees.

20. INTEGER-PROGRAMMING: Given $\mathbf{b}, \mathbf{c} \in \mathbb{N}^n$, and $\mathbf{A} \in \mathbb{N}^{n\times k}$ (that is, $\mathbf{A}$ is an $n \times k$ matrix of integers), determine whether there exists an $\mathbf{x} \in \mathbb{N}^n$ such that $\mathbf{c} \cdot \mathbf{x} \geq 0$, and $\mathbf{Ax} \leq \mathbf{b}$.

According to definition 28, a language is **NP**-complete if it is in **NP**, and every other language in **NP** can be reduced to it in polynomial time. Although Cook's theorem (theorem 46) requires a very direct and tedious proof that CNF-SAT possesses these properties, thanks to theorem 45 we needn't go to all that trouble every time we wish to show that some other language is **NP**-complete. To prove that some language L is **NP**-complete, we need only establish two things: that $L \in \mathbf{NP}$,

and that $L' \leq_m^P L$ for *some* language L' that is already known to be **NP**-complete. In practice, we typically proceed in *three* distinct steps to prove that a given language L is **NP**-complete:

1. Show that $L \in \mathbf{NP}$.
2. Show that L' can be reduced to L, for some $L' \in \mathbf{NP}$-complete.
3. Show that the reduction can be performed in polynomial time.

Very frequently, steps 1 and 3 are easy (in fact, usually trivial), while step 2 is extremely challenging, requiring some creativity. Finding an actual reduction requires us to "rephrase" one language (the one that is *already known* to be **NP**-complete) using only the "vocabulary" of another language (the one we wish to *prove* is **NP**-complete). Figure 9.1 shows how we will establish the **NP**-Completeness of a few important problems. Once a language is shown to be **NP**-complete, we can then use *it* as a tool for showing that other languages are **NP**-complete. You can see how we will use this strategy in Figure 9.1.

To show that a problem is in **NP** we must show that *some* nondeterministic Turing machine can solve it in polynomial time. By "solve" we mean "decide" (i.e. answer "yes" or "no"), so this definition of **NP** can only apply directly to decision problems. (As we will see, the implications extend well beyond decision problems, however.) We typically proceed in three steps, using the concept of an easily verifiable *certificate*, as described earlier.

1. Describe how to phrase a succinct "proof" that a given "yes" answer is correct.
2. Show that no such "proof" can exist for a "no" answer.
3. Show that such a "proof" can be verified (or rejected) in polynomial time.

For example, given a boolean formula in CNF-SAT, a succinct proof that it belongs to this language is simply a satisfying truth assignment. No such truth assignment can exist for an unsatisfiable boolean formula, and the validity of the truth assignment is very easy to check – we simply plug in the values and verify that they satisfy the formula, which can be done in linear time.

We now provide detailed **NP**-Completeness proofs for an assortment of well-known decision problems. Each proof will be structured according to the three steps outlined above, although steps 1 and 3 will often be treated in a very cursory fashion if they are very obvious.

Things to keep in mind: reductions do not *solve* the original problem; they merely translate the question from one vocabulary into another; that is, they map one *problem instance* into another *problem instance*, without regard to whether the answer to the original instance is "yes" or "no." If, in your reduction, you appeal in any way to the solution of the original problem (e.g. the satisfying truth assignment of a 3CNF-SAT instance, or the subsets in a PARTITION instance), then you have made a fundamental error. Your reduction is invalid since that information is unavailable within the polynomial time budget that the reduction must adhere to.

The set of all problem instances *generated* by the reduction need not be "typical" in any obvious sense within the language being reduced to. For example, the reduction of 3CNF-SAT to

INTEGER-PROGRAMMING (See section 10.9) generates instances with numerical entries consisting of only 0, 1, and -1, despite the fact that an integer program can accommodate arbitrary integers within the matrices and vectors. What this tells us is that even problem instances with very restricted numerical entries are already sufficiently hard to solve that we needn't use the full power of integer programming to solve satisfiability problems.

9.4 Straightforward Polynomial-Time Reductions

We will assume that the following languages have already been shown to be in **NP**-complete: PARTITION, 3-COLOR, CLIQUE, NESFRE, HAMILTONIAN-CIRCUIT.

9.4.1 Subset Sum

Proof by reduction from PARTITION.

Given an instance of PARTITION, which is simply a finite subset $S \subset \mathbb{N}$, we can easily determine what the equal partitions must sum to; it is simply half of the sum of all the numbers in S, which we shall denote by $\frac{1}{2}\Sigma S$. If the sum ΣS is odd, then we know immediately that such a partition is impossible. This leads to the following reduction:

$$f(S) = \begin{cases} (S, \frac{1}{2}\Sigma S) & \text{when } \Sigma S \text{ is even} \\ (\emptyset, 1) & \text{otherwise} \end{cases} \tag{9.14}$$

Note that when ΣS is odd, the mapping must still produce an instance of SUBSET-SUM, so we simply pick one for which we know *a priori* that the answer is "no," such as $(\emptyset, 1)$, since the empty set has no subset that sums to 1. Another possibility would be $(S, 1 + \Sigma S)$, since the requires sum is larger than can be attained. In the former case, all instances in which ΣS is odd get mapped to the same problem instance, clearly making the mapping many-to-one. In the latter case, the mapping is one-to-one.

9.4.2 Bin Packing

Proof by reduction from PARTITION. Need to map the set S to a 3-tuple, (S', k, b), where $k \in \mathbb{N}$ is the number of bins, and $b \in \mathbb{N}$ is their capacity.

$$f(S) = \begin{cases} (S, 2, \frac{1}{2}\Sigma S) & \text{when } \Sigma S \text{ is even} \\ (\{1\}, 2, 1) & \text{otherwise} \end{cases} \tag{9.15}$$

9.4.3 Knapsack

Proof by reduction from SUBSET-SUM. Let (S, k) be an instance of SUBSET-SUM. Suppose $S = \{s_1, s_2, \ldots, s_k\}$. We define an instance of KNAPSACK as...

$$\left(\{(s_1, s_1), \ldots, (s_k, s_k)\}, m, m\right) \tag{9.16}$$

where $m = \frac{1}{2}\Sigma S$. Since the sum of the weights is to be bounded above by m, and the sum of the values is to be bounded below by m, they must sum to exactly m.

9.4.4 4-Color

Proof by reduction from 3-COLOR. We define a mapping from instances of 3-COLOR to instances of 4-COLOR such that

$$f(G) \;=\; G', \tag{9.17}$$

where G is 3-colorable if and only if G' is 4-colorable. Notice that if $G' = G$, then $G' \in$ 4-COLOR whenever $G \in$ 3-COLOR, but that the latter may be true while the former is false. That is, "yes" answers are preserved, but "no" answers may not be. To remedy this, we must ensure that if G is *not* 3-colorable, then G' will *not* be 4-colorable. We accomplish this by ensuring that $\chi(G') = \chi(G) + 1$. Thus, we define $G' = (V', E')$ such that $V' = V \cup \{v\}$, where v is a new vertex, and $E' = E \cup \{(u, v) \mid u \in V\}$. That is, the new vertex v is adjacent to each of the original vertices in G. Consequently, whatever the chromatic number of G, G' will require exactly one more color to color the vertex v.

9.4.5 Subgraph Isomorphism

The language SUBGRAPH-ISOMORPHISM consists of three-tuples (G_1, G_2, k) such that graphs G_1 and G_2 have subgraphs $G_1' \subseteq G_1$ and $G_2' \subseteq G_2$ where G_1' and G_2' are isomorphic and have k or more vertices. First, SUBGRAPH-ISOMORPHISM $\in$ **NP** because each element of SUBGRAPH-ISOMORPHISM has a concise certificate of membership, which consists of the two subgraphs, plus the correspondence between the vertices. Such a certificate is trivial to verify.

Next, we show that SUBGRAPH-ISOMORPHISM is **NP**-complete by reduction from CLIQUE. We define a mapping from instances of CLIQUE, (G, n), to instance of SUBGRAPH-ISOMORPHISM, (G_1, G_2, k), such that G has a clique of size n or greater as a subgraph if and only if G_1 and G_2 have an isomorphic subgraph of size k or greater. We simply let

$$f(\ (G, n)\ ) \;=\; (G, C_n, n), \tag{9.18}$$

where C_n is the clique with n vertices. Thus, we have very directly rephrased the question of membership in CLIQUE as a special case of SUBGRAPH-ISOMORPHISM. This mapping clearly requires only linear time. Therefore, SUBGRAPH-ISOMORPHISM is **NP**-complete.

9.4.6 Non-Equivalent Regular Expressions

The language of non-equivalent regular expressions, or NERE, is the language consisting of pairs of regular expressions that generate different regular languages. The only difference between NERE and NESFRE is that we allow the use of the Kleene star operator in the former but not the latter. Thus, NERE is simply a generalization of NESFRE; that is, any algorithm that decides NERE also decides NESFRE. As a consequence, the reduction NESFRE $\leq_m^P$ NERE is immediate, being accomplished by the identity function. It follows that NERE is **NP**-complete.

9.4.7 Traveling Salesman

The Traveling Salesman decision problem, or TRAVELING-SALESMAN, can be expressed as follows. Given a complete graph[4] $G = (V, E)$, a *distance function*, $d : V \times V \rightarrow \mathbb{N}$, that assigns each edge in E a natural number, and a number $k \in \mathbb{N}$, determine whether there exists "tour" of all the vertices (that is, a *circuit*, or *loop*) whose total distance is k or less. TRAVELING-SALESMAN is clearly in **NP**, as it is possible to guess a solution, which is simply a permutation of the vertices, then verify that its length is less than k in polynomial time.

A Hamiltonian circuit of a graph is a circuit, or loop, that passes through each vertex of the graph exactly once. Given that the Hamiltonian circuit decision problem, or HAMILTONIAN-CIRCUIT, is **NP**-complete, we can show that TRAVELING-SALESMAN is also **NP**-complete by polynomial-time reduction from HAMILTONIAN-CIRCUIT. This is in fact quite easy to do, given the great similarity between the two problems.

Given a graph $G = (V, E)$, with $|V| = n$, we can determine whether it has a Hamiltonian circuit by transforming it into an instance of the traveling salesman problem, (G', d, k), as follows. Let $G' = (V, E')$ be the graph G with each missing edge added; that is, G' is the smallest clique containing G as a subgraph. We then define the distance function $d : V \times V \rightarrow \mathbb{N}$ as follows:

$$d(u, v) = \begin{cases} 1 & \text{if } (u, v) \in E \\ n + 1 & \text{if } (u, v) \in E' - E. \end{cases} \tag{9.19}$$

Finally, we let the target length of the tour, k, be the number of vertices, n. In other words, there exists a Hamiltonian Circuit in G if and only if G' has a tour of length n or less. The mapping $G \rightarrow (G', d, n)$ is therefore a many-to-one reduction of HAMILTONIAN-CIRCUIT to TRAVELING-SALESMAN. Since this mapping can clearly be computed in polynomial time, this shows that TRAVELING-SALESMAN is **NP**-complete.

9.5 Further Reading

See Gary and Johnson [7] for a more detailed discussion of the classes **P**, **NP**, **NP**-complete, etc., along with the history of their development. The book by Gary and Johnson [7] is also an

[4] A *complete graph*, also known as a *clique*, is a graph in which every pair of vertices has an edge connecting them.

extraordinarily useful reference as it contains a catalog of hundreds of **NP**-Complete problems. Having such a catalog generally simplifies the process of determining whether some unknown problem is **NP**-complete or not, as one can generally choose a problem that is somewhat similar to perform the reduction from.

9.6 Exercises

1. Each element of the language SUBSET-SUM consists of a finite multiset $S = (a_1, a_2, \ldots, a_n)$ of natural numbers and a constant $k \in \mathbb{N}$ such that some subset (possibly a multiset) of S sums to k. Let us define a variant of SUBSET-SUM, which we will call INTERVAL-SUBSET-SUM, each of whose elements is a finite multiset S of natural numbers and a constant $k \in \mathbb{N}$ such that some subset (possibly a multiset) of S sums to any value between k and $2k$, inclusive.

   (a) Show that the decision problem SUBSET-SUM is **NP**-complete using the **NP**-complete problem PARTITION.

   (b) Is INTERVAL-SUBSET-SUM **NP**-complete? Justify your answer by giving a polynomial-time algorithm for deciding it or by reducing some known **NP**-complete problem to it.

2. The **NP**-complete problem known as PARTITION is as follows: Given a finite multiset of natural numbers $S = \{x_1, x_2, \ldots, x_n\}$, determine whether there is a subset (possibly a multiset) $S' \subset S$ such that

$$\sum_{x \in S'} x = \sum_{x \in S - S'} x.$$

   (a) Explain why the obvious brute-force method of solving the PARTITION problem requires exponential time *in the worst case*. That is, show that there is a sequence of problem instances for which the time-complexity is exponential.

   (b) Let K be a fixed positive integer. Show that PARTITION can be solved in *polynomial* time if the numbers in the multiset are bounded above by K. Describe the algorithm with pesudo-code, and explain why it runs in polynomial time. You need not find an efficient algorithm; any polynomial algorithm will do.

   (c) Why must we require that K be a fixed constant? That is, why is your algorithm no longer polynomial-time if we simply set K to be the maximum of all the numbers in a given list?

3. Show that the relation $\leq_m$ is transitive.

4. Let HAMILTONIAN-CIRCUIT-CONSTRUCTION denote the problem of finding a Hamiltonian circuit in a graph, or determining that no such circuit exists. Prove that this problem is **NP**-equivalent, but not **NP**-complete. Clearly label and explain all the steps.

5. Let us define a variant of SUBSET-SUM, called SUBSET-SUM-PLUS-2, as follows. Each instance of SUBSET-SUM-PLUS-2 consists of a finite set S of natural numbers and a constant $k \in \mathbb{N}$ such that some subset of S sums to k or $k+1$ or $k+2$. Thus, we are no longer looking for a specific sum, but any of three possible sums. Show that SUBSET-SUM-PLUS-2 is **NP**-complete using a polynomial-time many-to-one reduction from SUBSET-SUM.

6. Let us define a variant of SUBSET-SUM, which we will call SUBSET-SUM-TIMES-2, each instance of which is a finite set S of natural numbers and a constant $k \in \mathbb{N}$ such that some subset of S sums to any value between k and $2k$, inclusive. Is SUBSET-SUM-TIMES-2 **NP**-complete? Justify your answer by giving a polynomial-time algorithm for deciding it or by reduction from some known **NP**-complete problem.

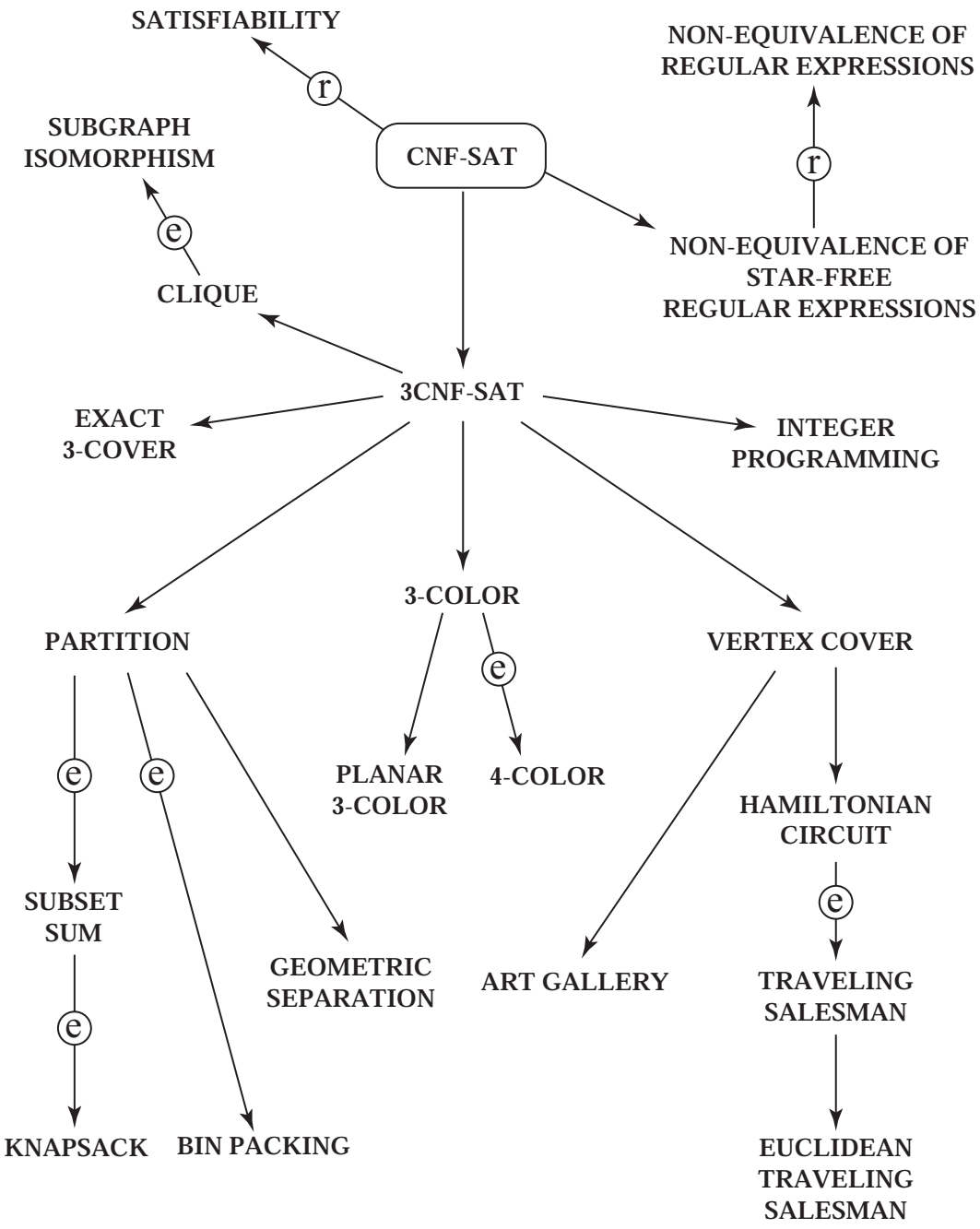


Figure 9.1: *This figure shows the strategy we will follow in proving a wide variety of problems to be* **NP**-complete. *Starting from* CNF-SAT *at the top, the arrows indicate the polynomial-time reductions that we will study. For example, we will show how to reduce* 3CNF-SAT *to* 3-COLOR *in polynomial time, thus proving* 3-COLOR *to be* **NP**-complete. *(The asterisks indicate relatively straightforward reductions.)*

Chapter 10

Challenging Polynomial-Time Reductions

This chapter supplements the previous chapter on the theory of **NP**-Completeness. Provided here are fairly detailed reductions showing that 3CNF-SAT, 3-COLOR, PLANAR-3-COLOR, NESFRE, NERE, and TRAVELING-SALESMAN are all **NP**-complete. Moreover, we show that the *Chromatic Number* problem is **NP**-hard; that is, it is just as hard as anything in **NP**-complete, but without being in **NP** itself, as it is an optimization problem rather than a decision problem.

These problems all involve the invention of *gadgets* that allow us to capture the essence of one problem in the vocabulary of another. For example, a recurring idea is that of mimicking the characteristics of a truth assignment in the context of something altogether different from boolean logic, such as a graph, or a set of natural numbers. In particular, any legitimate truth assignment will assign one of two values to each boolean variable, and must assign *different* values to the variable and its negation. In the context of 3-COLOR we achieve this behavior by creating a set of vertices in which any valid 3-coloring will assign them one of two colors and ensure that a node representing a given variable will have a different color than the one representing its negation.

The gadgets used in these reductions are often very clever. While the asserted behavior and ultimate role of a gadget may be easy to grasp, supplying the details of such a thing can be a substantial puzzle in itself, and there is no general strategy by which such puzzles can be solved. Each reduction seems to be a unique invention, crafted for the express purpose of proving the **NP**-completeness of a specific language.

Many of these problems were first shown to be **NP**-complete by Richard Karp in 1972[11].

10.1 3CNF-SAT

We shall show that 3CNF-SAT is **NP**-complete by reduction from CNF-SAT. Clearly 3CNF-SAT $\in$ **NP**, since it is a proper subset of CNF-SAT, which is in **NP**. Given a formula $F \in$ CNF-SAT, we must show how to construct another formula $F' \in$ 3CNF-SAT, which has exactly three literals per clause, that is satisfiable if and only if F is. This transformation must be accomplished in polynomial-time, which is insufficient time to actually test the satisfiability of F. Thus, we must perform the transformation without knowing whether F is satisfiable or not. What we shall do is to successively reduce the size of clauses in F that have more than three literals, and also increase the size of clauses in F that have fewer than three literals[1], all the while retaining the satisfiability or unsatisfiability of the original formula.

Let $\mathcal{F}$ be any CNF formula. Then consider the formulas

$$\mathcal{F}_1 = \mathcal{F} \wedge (\ell_1 \vee \ell_2 \vee \ell_3 \vee \cdots \vee \ell_n) \tag{10.1}$$

$$\widehat{\mathcal{F}}_1 = \mathcal{F} \wedge (x \vee \ell_1 \vee \ell_2) \wedge (\neg x \vee \ell_3 \vee \cdots \vee \ell_n) \tag{10.2}$$

where $n > 3$, the symbols $\ell_1, \ell_2, \ldots, \ell_n$ denote literals (i.e. either Boolean variables, or the negation of Boolean variables), and x denotes a Boolean variable that does not appear anywhere in formula $\mathcal{F}_1$. We claim that $\mathcal{F}_1$ is satisfiable if and only if $\widehat{\mathcal{F}}_1$ is satisfiable. To see this, suppose that a truth assignment $\mathcal{A}$ satisfies $\mathcal{F}_1$, which we denote by $\mathcal{A} \models \mathcal{F}_1$. Then $\mathcal{A} \models \ell_i$ for some $1 \leq i \leq n$, since $\mathcal{A}$ must satisfy at least one literal in the last clause. If $i < 3$, then $\mathcal{A} \cup \{x = \bot\}$ is a truth assignment that satisfies $\widehat{\mathcal{F}}_1$. On the other hand, if $i \geq 3$, then $\mathcal{A} \cup \{x = \top\}$ satisfies $\widehat{\mathcal{F}}_1$. Therefore, $\widehat{\mathcal{F}}_1$ is satisfiable. Now suppose that $\mathcal{A} \models \widehat{\mathcal{F}}_1$. Then $\mathcal{A} \models \mathcal{F}_1$ since $\mathcal{A} \models \ell_i$ for some $1 \leq i \leq n$. Therefore, $\mathcal{F}_1$ is satisfiable. While the above procedure introduces more clauses, it reduces the size of the last clause by one literal.

In a similar fashion, we can add new variables to clauses that have fewer than three literals either by simply repeating literals, or by introducing new variables as we have done above. Consider the following two formulas

$$\mathcal{F}_2 = \mathcal{F} \wedge (\ell_1 \vee \ell_2) \tag{10.3}$$

$$\widehat{\mathcal{F}}_2 = \mathcal{F} \wedge (x \vee \ell_1 \vee \ell_2) \wedge (\neg x \vee \ell_1 \vee \ell_2) \tag{10.4}$$

where, again, x denotes a Boolean variable that does not appear in formula $\mathcal{F}_2$. It is easy to see that $\mathcal{F}_2$ is satisfiable if and only if $\widehat{\mathcal{F}}_2$ is satisfiable. These two operations can be used repeatedly to create an equivalent Boolean formula of the desired form. The algorithm is summarized in Figure 10.1.

The first loop replaces a clause with $n > 3$ literals with a clause of length 3 and a clause of length $n - 1$. Similarly, the second while loop replaces a clause of length $n < 3$ with two clauses of length $n + 1$. Therefore, both while loops are guaranteed to terminate eventually, as each iteration makes progress toward reducing or increasing the size of clauses that have more or fewer than three literals. The function returns a new CNF formula that is satisfiable if and only if the original formula $\mathcal{F}$

[1] We wish to end up with *exactly* three literals per clause, rather than three or fewer, because this is often convenient in constructing reductions from 3CNF-SAT. That is, it often simplifies the bookkeeping if we can assume that all clauses have the same number of literals. The reduction of 3CNF-SAT to 3-COLOR, for example, requires only one type of gadget because each clause has exactly the same form.

```
    function CNFSAT-to-3CNFSAT( F )
1   while F has a clause C containing more than three literals do
2       x ← a new variable name
3       replace C with (x ∨ ℓ1 ∨ ℓ2)  ∧  (¬x ∨ ℓ3 ∨ ··· ∨ ℓn)
4   endwhile
5   while F has a clause C containing fewer than three literals do
6       x ← a new variable name
7       replace C with (x ∨ ℓ1 ∨ ···)  ∧  (¬x ∨ ℓ1 ∨ ···)
8   endwhile
9   return F
```

Figure 10.1: *An polynomial-time algorithm that converts a formula in* CNF-SAT *into an "equivalent" formula in* 3CNF-SAT*; that is, the new formula is satisfiable if and only if the original formula is.*

is, but in which each clause has exactly three distinct literals. This function computes a many-to-one reduction from CNF-SAT to 3CNF-SAT. Since the loops each iterate no more than kn times, where k is the number of clauses, and n is the number of distinct variable names, this reduction is clearly computable in polynomial time. Therefore, CNF-SAT $\leq_m^P$ 3CNF-SAT, which establishes that 3CNF-SAT is **NP**-complete.

10.2 3-Color

The language known as 3-COLOR consists of graphs that can be colored using three or fewer colors; that is, graphs in which each vertex can be assigned one of three colors in such a way that no two adjacent vertices are assigned the same color. The corresponding decision problem can be stated thus: "Given a graph G, can it be colored using three of fewer colors?" We shall show that 3-COLOR is **NP**-Complete by reduction from 3CNF-SAT. In this example we will spell out all the steps in detail.

Step 1: First, we verify that 3-COLOR is in **NP**. Observe that if we are given an assignment of colors to the vertices of a given graph G, then it is an easy matter to check that this assignment constitutes a valid 3-coloring; we need only verify that the endpoints of each edge are assigned different colors, and that only three colors are used altogether. This can clearly be accomplished in polynomial (in fact, linear) time. We also observe that no such assignment of colors can exist for graphs that are not in 3-COLOR.

Step 2: Next, we show that we can reduce a problem that is already known to be **NP**-complete to 3-COLOR. In particular, we will show that 3CNF-SAT $\leq_m^P$ 3-COLOR. That is, we will show how to convert an arbitrary instance of 3CNF-SAT into an instance of 3-COLOR in such a way that the yes/no answer is preserved, and requiring only polynomial time. Thus, the resulting graph G will be three-colorable if and only if the given 3-CNF formula is satisfiable.

Let F be any propositional formula in 3CNF form with n distinct propositional variables and k clauses. Let $X = \{x_1, x_2, \ldots, x_n\}$ denote the set of variables appearing in F, let $\overline{X} = \{\overline{x}_1, \overline{x}_2, \ldots, \overline{x}_n\}$ denote the set of their negations, and let $C = \{c_1, c_2, \ldots, c_k\}$ denote the set of clauses in F, each containing exactly three literals from $X \cup \overline{X}$.

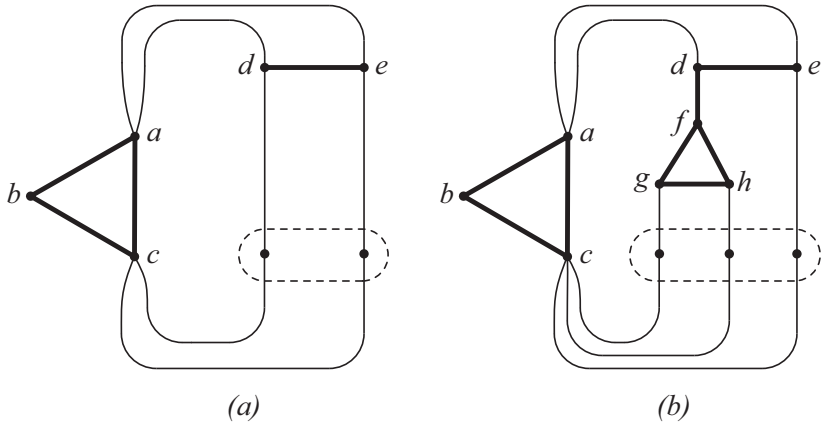


Figure 10.2: *These graphs illustrate the reduction from* 3CNF-SAT *to* 3-COLOR. *(a) This graph is 3-colorable if and only if at least one of the two vertices in the dashed circle is the same color as vertex "a." (b) By similar reasoning, this graph is 3-colorable if and only if at least one of the three vertices in the circle is the same color as vertex "a." Once we interpret the color of vertex a as "true," this idea can be used to ensure that the graph can be 3-colored if and only if at least one literal in each clause can be satisfied.*

Consider the undirected graph $G = (V, E)$ that is constructed from F as follows. Let the vertex set V consist of the union of these sets: $\{a, b, c\}$, and $\{x_i, \overline{x}_i\}$ for $i = 1, 2, \ldots, n$, and $\{d_j, e_j, f_j, g_j, h_j\}$, for $j = 1, 2, \ldots k$. The resulting vertices are to be connected as shown in Figure 10.3, which shows a portion of the construction corresponding to the 3-CNF formula

$$(x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_3 \vee \neg x_4) \wedge \cdots \wedge (x_2 \vee \neg x_4 \vee x_n). \tag{10.5}$$

Figure 10.2 shows several simpler graphs illustrating how the construction in Figure 10.3 works.

We claim that G will have the desired property; that is, G will be 3-colorable if and only if F is satisfiable. Suppose that G is 3-colorable. Then all three colors must appear in the triangle formed by vertices a, b, and c. Let us refer to these colors $\top$, $\bot$, and $\Diamond$, respectively, as this will make clear the interpretation that we will be giving the coloring. Since each of the vertices representing the literals is connected to the vertex with color $\Diamond$, and each literal is connected to its negation, this means that each literal is colored either $\top$ or $\bot$, with its negation colored appropriately; that is, one is $\top$ and the other is $\bot$. The gadget shown on the right of Figure 10.2 is designed in such a way that in any valid 3-coloring, at least one of the vertices in the dashed circle must be colored $\top$; this fact is used to ensure that each clause will have a minimum of one true literal. Thus, a valid 3-coloring indicates the existence of a valid truth-assignment such that at least one literal per clause is true, which implies that the formula F is satisfiable. A similar argument shows that if F is satisfiable, there is a valid 3-coloring of the graph G.

Step 3: Finally, we verify that the many-to-one reduction from 3CNF-SAT to 3-COLOR can be computed in polynomial time. But this is almost immediate. The vertex set V of the graph G that we have constructed will have

1. three "special" vertices: a, b, and c,

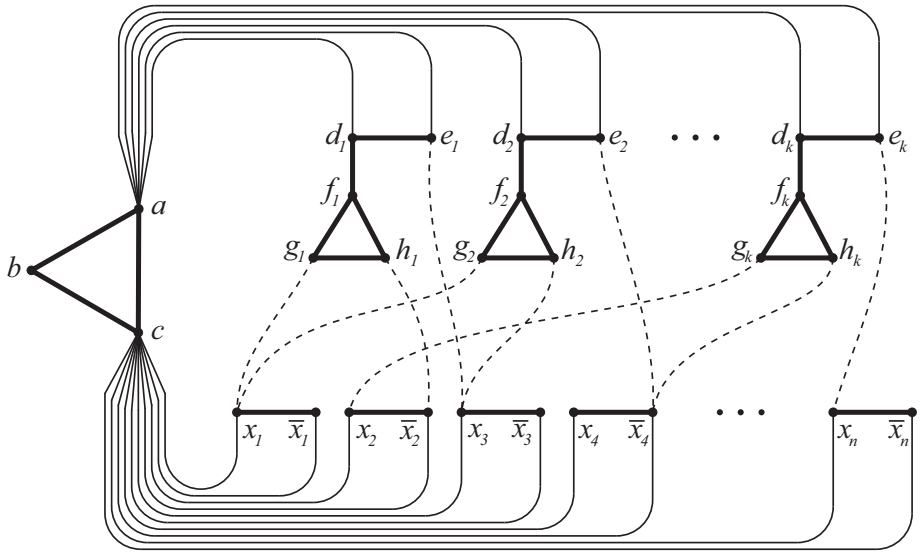


Figure 10.3: *This is the complete construction corresponding to a formula in 3CNF with n distinct variables and k clauses. The graph is 3-colorable if and only if the original formula is satisfiable. Shown is a portion of the construction for the formula* $(x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_3 \vee \neg x_4) \wedge \cdots \wedge (x_2 \vee \neg x_4 \vee x_n)$.

2. a vertex for each variable in F,
3. a vertex for the negation of each variable F,
4. vertices, d_j, e_j, f_j, g_j, and h_i, for $j = 1, 2, \ldots, k$,

for a total of $2n + 5k + 3$ vertices. If E is the edge set of G, then E consists of

1. (a, b), (b, c), (c, a),
2. $(x_1, c), (x_2, c), \ldots, (x_n, c)$,
3. $(\overline{x}_1, c), (\overline{x}_2, c), \ldots, (\overline{x}_n, c)$,
4. $(x_1, \overline{x}_1), (x_2, \overline{x}_2), \ldots, (x_n, \overline{x}_n)$,
5. the five edges for each gadget, as shown by the dark lines in Figure 10.3, for each of the k clauses in F,
6. (d_j, a) and (e_j, a) for $j = 1, 2, \ldots, k$.
7. The three edges connecting each of e_j, g_j, and h_j to a single literal, for $j = 1, 2, \ldots, k$. These are shown as dashed lines in Figure 10.2.

for a total of $3n + 10k + 3$ edges in E. Typically, this level of accounting is unnecessary, as we are only obliged to demonstrate that the construction can take place in polynomial time. Thus, all that

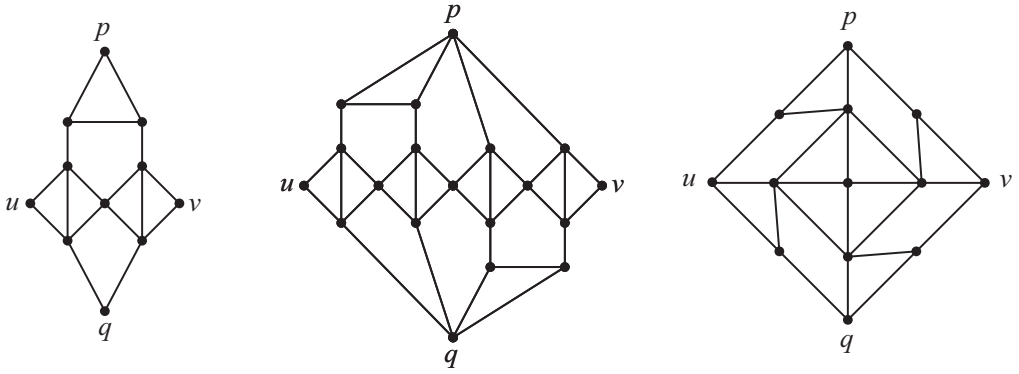


Figure 10.4: *Planar gadgets used in the reduction of* 3-COLOR *to* PLANAR-3-COLOR. *(Left) If this graph is 3-colored, vertices u and v must have the same color. If the color of q is different from that of u and v, then p must be the same color as q. However, if q is the same color as u and v, then p can be assigned any of the three colors. (Center) This graph is built by combining two copies of graph on the left, which completes the constraints. That is, if it is 3-colored, then the colors of u and v must match, and the colors of p and q must match. However, the colors of p and u are independent. (Right) This is a somewhat simpler gadget, due to Michael Fischer, that has the same properties as the one in the center.*

is relevant is that we are constructing a copy of a particular gadget for each clause, another for each variable, and adding another fixed collection of vertices and edges (the triangle *abc*).

We have therefore shown that 3-COLOR is **NP**-complete by reduction from 3CNF-SAT. The word "from" is crucial here, as it emphasizes that instances of 3CNF-SAT were rephrased as instances of 3-COLOR, not the other way around. Performing the reduction in the other direction would not have shown anything useful[2], yet this is a common mistake.

10.3 Planar 3-Color

A planar graph is a graph that can be drawn in such a way that no edges cross. The famous "four-color theorem" assures us four colors always suffice to color a planar graph. However, three colors do not always suffice, and determining whether a given planar graph is 3-colorable is intractable.

When one places additional restrictions on the form of a problem, it often makes the problem easier. For example, if we place the additional restriction on instances of 3CNF-SAT that each clause is to have at most *two* literals, rather than three, the problem becomes solvable in polynomial time. Thus, when we restrict the problem instances, it is generally necessary to either re-establish **NP**-Completeness, or conclude that the restricted problem is now solvable in polynomial time. In the case of restricting the 3-coloring problem to planar graphs, it so happens that the problem remains just as hard.

[2]In fact, we already know that 3-COLOR $\leq_m^P$ CNF-SAT by virtue of the fact that 3-COLOR $\in$ **NP** and CNF-SAT is **NP**-complete; hence, exhibiting such a reduction would not demonstrate anything new.

Consider the three graphs shown in Figure 10.4, which are all 3-colorable. Copies of the "gadget" in the center or on the right can be used to convert an arbitrary graph into an "equivalent" planar graph; that is, a planar graph that is 3-colorable if and only if the original graph is 3-colorable. We now provide a brief sketch of how this is accomplished.

The trick to the conversion is to notice that if the gadget in Figure 10.4 is 3-colored, then vertices u and v must be assigned the same color, and vertices p and q must be assigned the same color. However, the colors assigned to the two pairs are independent. This gadget allows for one edge to"cross through" another, while preserving 3-colorability and remaining planar. Thus, the reduction from 3-COLOR to PLANAR-3-COLOR proceeds by replacing each crossing pair of edges with a copy of this gadget. When no more crossing edges remain, the resulting graph is planar, and is 3-colorable if and only if the original graph is.

10.4 Vertex Cover

The decision problem known as VERTEX-COVER is as follows: Given a graph $G = (V, E)$ and $k \in \mathbb{N}$, determine whether there exists a subset of vertices, $V' \subset V$, such that $|V'| \leq k$ and each edge in E is incident upon some vertex in V'. We think of the subset V' as "covering" the edges of G. It is well known that VERTEX-COVER is **NP**-complete; in fact, it is one of the most fundamental **NP**-complete problems in that it is frequently used to show that other problems are **NP**-complete. The proof that VERTEX-COVER is **NP**-complete is a classic example of how such proofs are constructed.

Step 1: It is easy to see that VERTEX-COVER is in **NP**. The certificate of membership is simply the set of vertices in the cover. The size of this set is clearly no larger than the graph G, and the fact that it is a cover of the required size can be easily checked in $\mathcal{O}(|E|)$ time; we simply check each edge to ensure that one of its endpoints is in the cover.

Step 2: We shall show that 3CNF-SAT $\leq_m^P$ VERTEX-COVER. Let $\mathcal{F}$ be a boolean formula in 3-CNF form, and let $X = \{x_1, \overline{x}_1, x_2, \overline{x}_2, \ldots, x_n, \overline{x}_n\}$ be the set of all variables in $\mathcal{F}$ along with their negations, and let $C = \{\{c_{11}, c_{12}, c_{13}\}, \ldots, \{c_{k1}, c_{k2}, c_{k3}\}\}$ denote the set of clauses, where each $c_{i,j} \in X$. We shall construct a graph $G = (V, E)$, starting with some basic elements:

$$V = X \cup \{c_{i1}, c_{i2}, c_{i3} \mid i = 1, 2, \ldots, k\}, \tag{10.6}$$

$$E = \{(c_{i1}, c_{i2}), (c_{i2}, c_{i3}), (c_{i3}, c_{i1}) \mid i = 1, 2, \ldots, k\} \cup \{(x_i, \overline{x}_i) \mid i = 1, 2, \ldots, n\}, \tag{10.7}$$

where n is the number of distinct variables, and k is the number of clauses in $\mathcal{F}$.[3] These vertices and edges define a line for each distinct variable, and a triangle for each clause, as shown in Figure 10.5. The minimal vertex cover for this basic graph consists of $n + 2k$ vertices; one for each line and two for each triangle. We next add one additional edge for each literal in each clause. The edge will connect each vertex of the jth triangle to a different vertex representing a literal in the jth clause. This is depicted in Figure 10.6.

Claim: The resulting graph has a vertex cover of size $n + 2k$ if and only if the boolean formula $\mathcal{F}$ is satisfiable. To see this note that the two vertices in the cover that are in each triangle cover the

[3]Here we are using the variable names x_i (as well as their negations) and the symbols c_{ij}, which denote the literals of each clause, merely as vertex labels in a graph. In their role as vertex labels, they lose their former meanings as variables and such.

three edges of the triangle, plus exactly two of the edges connecting the triangle to the corresponding literals. The third edge connecting the triangle to a literal must be covered in another way; the only other option is to cover it with the vertex that is covering the line it connects to.

If a cover of size $n + 2k$ exists, then the vertices covering the literals must also succeed in covering at least one edge that leads to *each* of the clauses. If we identify the covered literals as representing $\top$, this means that each clause will have at least one $\top$ literal, which means that the formula is satisfiable. On the other hand, if the formula is satisfiable, then it is possible to assign truth values to each of its variables in such a way that each clause will have at least one $\top$ literal. The existence of a satisfying truth assignment ensures that the graph G can be covered with exactly $n+2k$ vertices, since the literals can be covered in such a way as to cover at least one edge leading to each triangle.

Step 3: The construction of the graph from the boolean formula can clearly be accomplished in $\mathcal{O}(k + n)$ time, as each clause and each literal requires only a constant amount of work. This is trivially polynomial in the size of the boolean formula.

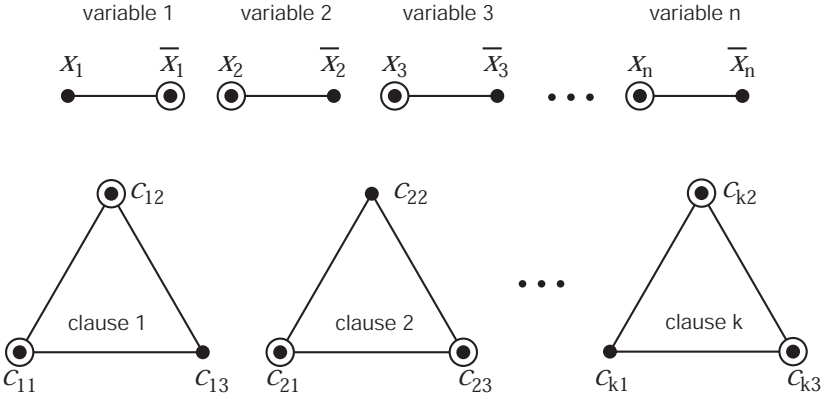


Figure 10.5: *This figure shows part of the construction used to prove that* VERTEX-COVER *is* **NP**-complete *by reduction from* 3CNF-SAT. *The construction begins with* $3k + 2n$ *vertices and* $3k + n$ *edges, organized as* k *triangles and* n *lines, where* k *is the number of clauses and* n *is the number of distinct variables in the boolean formula. This graph requires exactly* $2k + n$ *vertices to cover all the edges. One such selection of vertices is shown here; the vertices in the cover are circled.*

10.5 Clique

The decision problem known as CLIQUE is as follows: Given a graph G and $k \in \mathbb{N}$, determine whether G has a subgraph of size k or larger that is a clique (i.e. completely connected). We will show that CLIQUE is **NP**-complete by reduction from 3CNF-SAT.

Step 1: CLIQUE is clearly in **NP**, since it is a decision problem with an obvious certificate of membership, namely, the list of vertices in the clique. Given such a list, it is trivial to check that they form a clique simply by checking that each is adjacent to all the others in the graph G.

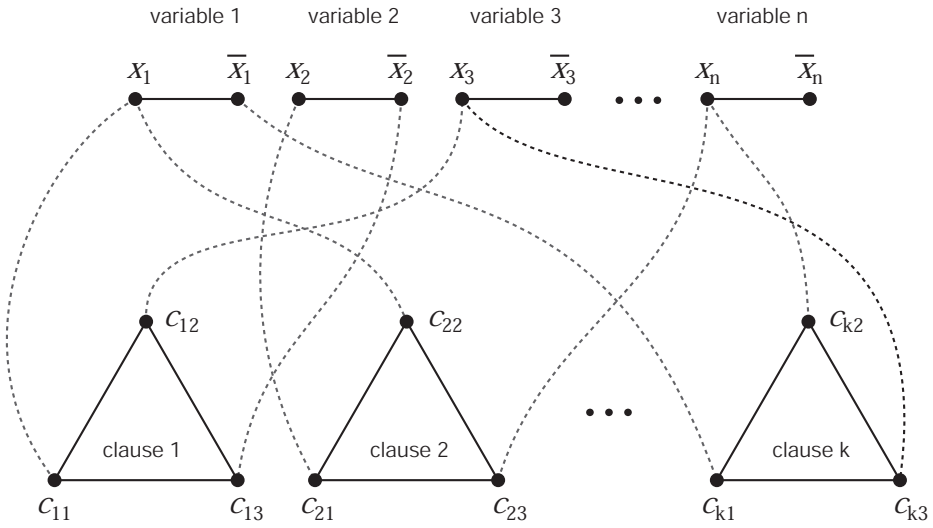


Figure 10.6: *This figure shows the complete construction that is used to transform an instance of* 3CNF-SAT *to an instance of* VERTEX-COVER. *The portion of the graph shown here corresponds to the 3-CNF boolean formula* $(x_1 \vee x_3 \vee \overline{x}_2) \wedge (x_2 \vee x_1 \vee x_n) \wedge \cdots \wedge (\overline{x}_1 \vee x_n \vee x_3)$. *The dashed lines connect the triangles, which represent the clauses, to the corresponding literals found in that clause. The complete graph consists of* $2n+3k$ *vertices and* $n+6k$ *edges, where* k *is the number of clauses and* n *is the number of distinct variables in the boolean formula. The graph has a vertex cover consisting of* $n+2k$ *vertices if and only if the boolean formula is satisfiable.*

Step 2: Given a boolean formula in 3-CNF form with k clauses, we construct a graph $G = (V, E)$ consisting of $3k$ vertices as follows. Each vertex will correspond to one literal in the formula. We add edges connecting each vertex to *all* other vertices that represent non-contradictory literals in *other* clauses. For instance, the (vertex representing the) literal x_1 will be connected to every literal in every other clause, except for $\overline{x}_1$. See Figure 10.7. In other words, each literal x will be adjacent to every literal in every other clause that it is "compatible" with – i.e. to every other literal that could be made $\top$ simultaneously with x. If there exists a clique of size k in this graph, it means that there is at least one literal that can be set to $\top$ in each clause simultaneously, since each of the literals is compatible with all the others. Thus, the formula is satisfiable. Clearly the converse is also true; if the formula is satisfiable, the graph G will have a clique of size k.

Step 3: The construction of the graph G can be done in $\mathcal{O}(k^2)$ time, which is polynomial in the size of the boolean formula.

10.6 Hamiltonian Circuit

The language HAMILTONIAN-CIRCUIT consists of graphs that contain Hamiltonian circuits; that is, graphs that contain a circuit (cycle) that passes through each vertex exactly once. HAMILTONIAN-CIRCUIT is clearly in **NP**, since the ordered list of vertices in the circuit suffices for a certificate of membership

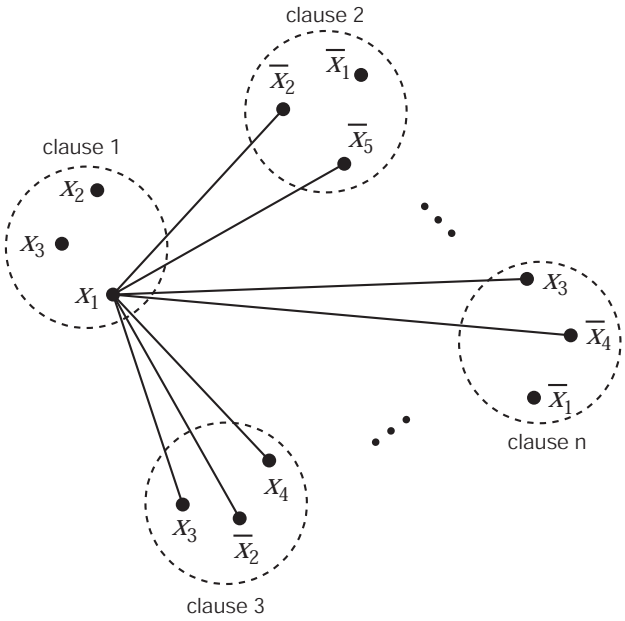


Figure 10.7: *This figure shows the construction that is used to transform an instance of* 3CNF-SAT *to an instance of* CLIQUE. *The complete graph consists of* $3k$ *vertices and up to* $9k(k-1)$ *edges, where* k *is the number of clauses and* n *is the number of distinct variables in the boolean formula. An edge is placed between every pair of vertices that represent non-contradictory variables in different clauses. The edges adjacent to one vertex, labeled* x_1*, within one clause are shown here. This vertex can be connected to every other vertex associated with every other clause except for* $\overline{x_1}$*, which it is incompatible with. The graph has a clique of size* k *if and only if the boolean formula is satisfiable.*

in HAMILTONIAN-CIRCUIT; such a certificate can be checked in linear time to verify that it contains all the vertices of the graph and defines a simple circuit.

We will show that HAMILTONIAN-CIRCUIT is **NP**-complete by reduction from VERTEX-COVER. That is, given any graph $G = (V, E)$ and an integer k, we will show how to construct another graph $G' = (V', E')$ such that G has a vertex cover of size k if and only if G' has a Hamiltonian circuit.

The reduction hinges on the use of the clever "gadget" shown in Figure 10.8a. This 12-vertex, 14-edge, graph will be replicated many times as a sub-graph of the graph G' that we will construct; each copy will be connected to the larger graph only at its four corners, as shown in the figure. This gadget was devised to have a very special property: If the graph to which it is a sub-graph has a Hamiltonian circuit, then the circuit must "pass through"[4] the gadget in one of two ways:

1. By entering at one corner, following a sideways "Ω" pattern, then exiting the gadget on the *same side*, as shown in figures 10.8b and 10.8c.

[4]Although we are considering only undirected graphs and undirected circuits, it is useful to talk about a circuit as if it were directed; e.g. by speaking of the circuit "passing through" vertices. This will aid in visualizing how the construction works.

2. By passing through the gadget twice, in "parallel" paths, as shown in Figure 10.8d.

This gadget will allow us to build a graph that mimics the essential properties of a vertex cover using a Hamiltonian circuit. Each gadget will represent an edge e in the original graph G (the one that we are testing for a vertex cover), and a circuit passing though it will correspond to a vertex covering e. Since one or both of the endpoints of the edge e may be included in a vertex cover of G, we must allow the circuit to pass through the gadget either once or twice, but to pass though all vertices exactly once in either case. This is precisely what the gadget allows us to do.

Here are the details of the construction. For ease of notation, we'll denote some of the vertices of the graph by arrays with three subscripts, $x[u, v, i]$, where u and v are vertices in the original graph G, and i is a positive integer. First, define the vertex sets

$$V_1 \stackrel{\text{def}}{=} \{x[a, b, i], x[b, a, i] \mid (a, b) \in E, 1 \leq i \leq 6\} \tag{10.8}$$

$$V_2 \stackrel{\text{def}}{=} \{y_1, y_2, \ldots, y_K\} \tag{10.9}$$

and the edge sets

$$E_1 \stackrel{\text{def}}{=} \{(x[a, b, 1], x[a, b, 2]), (x[a, b, 2], x[a, b, 3]), \ldots \mid (a, b) \in E\} \tag{10.10}$$

$$E_2 \stackrel{\text{def}}{=} \{(x[a, N_i(a), 6], x[a, N_{i+1}(a), 1]) \mid 1 \leq i < \deg(a)\} \tag{10.11}$$

$$E_3 \stackrel{\text{def}}{=} \{(x[a, N_1(a), 1], y_j) \mid a \in V, 1 \leq j \leq K\} \tag{10.12}$$

$$E_4 \stackrel{\text{def}}{=} \{(x[a, N_d(a), 6], y_j) \mid a \in V, 1 \leq j \leq K, d = \deg(a)\} \tag{10.13}$$

Here $\deg(v)$ is the degree of vertex v, and $N_i(v)$ denotes the ith neighbor of vertex v according to some fixed but arbitrary ordering. The sets V_1 and E_1 consist of many copies of the 12 vertices and the 14 edges, respectively, that are shown in Figure 10.8a; one copy for each edge in the original graph G. Thus, the subgraph $G_1 = (V_1, E_1)$ consists of $|E|$ copies of the gadget. The edges E_2 create chains of gadgets that correspond to the edges adjacent to each vertex in G. The edges E_3 and E_4 connect the start and end of each chain of gadgets to *all* of the vertices in V_2. Finally, we define $G' = (V', E')$ by

$$V' \stackrel{\text{def}}{=} V_1 \cup V_2 \tag{10.14}$$

$$E' \stackrel{\text{def}}{=} E_1 \cup E_2 \cup E_3 \cup E_4. \tag{10.15}$$

An example of this construction is shown in Figure 10.9, where the gray boxes are schematic representations of the gadgets. To keep the figure legible, the edges in E_3 and E_4 are depicted in abbreviated form for all but three vertices. We claim that G' has a Hamiltonian circuit if and only if G has a vertex cover of size K.

Observe that if G has a vertex cover of size K, then we can find K simple paths in G' that begin and end in V_2 and pass through *each* of the gadgets either once or twice (according to whether one or both of the endpoints of a given edge are in the vertex cover). Thus, we can construct a Hamiltonian circuit through G' by joining these paths together, forming one large circuit. Conversely, if G' has a Hamiltonian circuit, then such a circuit must pass through the collection of gadgets exactly K times, since it must pass through all the vertices of V_2. The K vertices of G that correspond to these K loops through the gadgets of G' will suffice as a vertex cover of G, since the circuit must pass through each gadget at least once, and this implies that each edge of G is adjacent to at least one of these vertices.

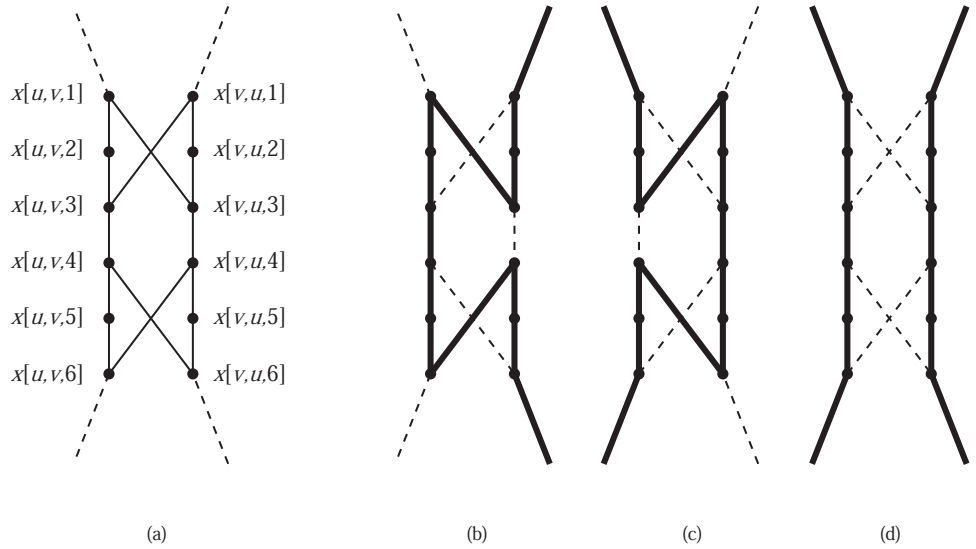


Figure 10.8: *(a) The "gadget" that is used to reduce* VERTEX-COVER *to* HAMILTONIAN-CIRCUIT. *Each instance of this gadget corresponds to an edge (u, v) of the graph G, and consists of fourteen edges, and twelve vertices, which we denote by $x[u, v, 1], \ldots, x[u, v, 6]$, and $x[v, u, 1], \ldots, x[v, u, 6]$, as shown. Each gadget is connected via its four corner vertices. All twelve vertices can be included in a Hamiltonian circuit in one of exactly three ways, as shown on the right: (b) a path that passes through on the right, (c) a path that passes through on the left, or (d) two parallel paths.*

10.7 Partition

Each member of the language PARTITION consists of a list (multiset) of natural numbers, $(a_1, a_2, \ldots, a_n)$, that can be split into two sublists with the same sum. We will show that PARTITION is **NP**-complete by reduction from 3CNF-SAT.

Let $\mathcal{F} = \{C_1, \ldots, C_K\}$ be a boolean formula in CNF form with the K clauses $C_1, \ldots, C_K$. Let n be the number of distinct variables, $x_1, \ldots, x_n$ in $\mathcal{F}$ and let $B \in \mathbb{N}$ be a constant such that $B \geq 7$; we will use B as the base of a number system to construct an instance of PARTITION. Define the function $\delta : N \times N \rightarrow \{0, 1\}$, based on the clauses of $\mathcal{F}$, such that

$$\delta(i, j) \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } x_i \in C_j \\ 0 & \text{otherwise} \end{cases} . \tag{10.16}$$

Let us also define $\overline{\delta}(i, j)$ analogously using the negated variables:

$$\overline{\delta}(i, j) \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } \overline{x}_i \in C_j \\ 0 & \text{otherwise} \end{cases} . \tag{10.17}$$

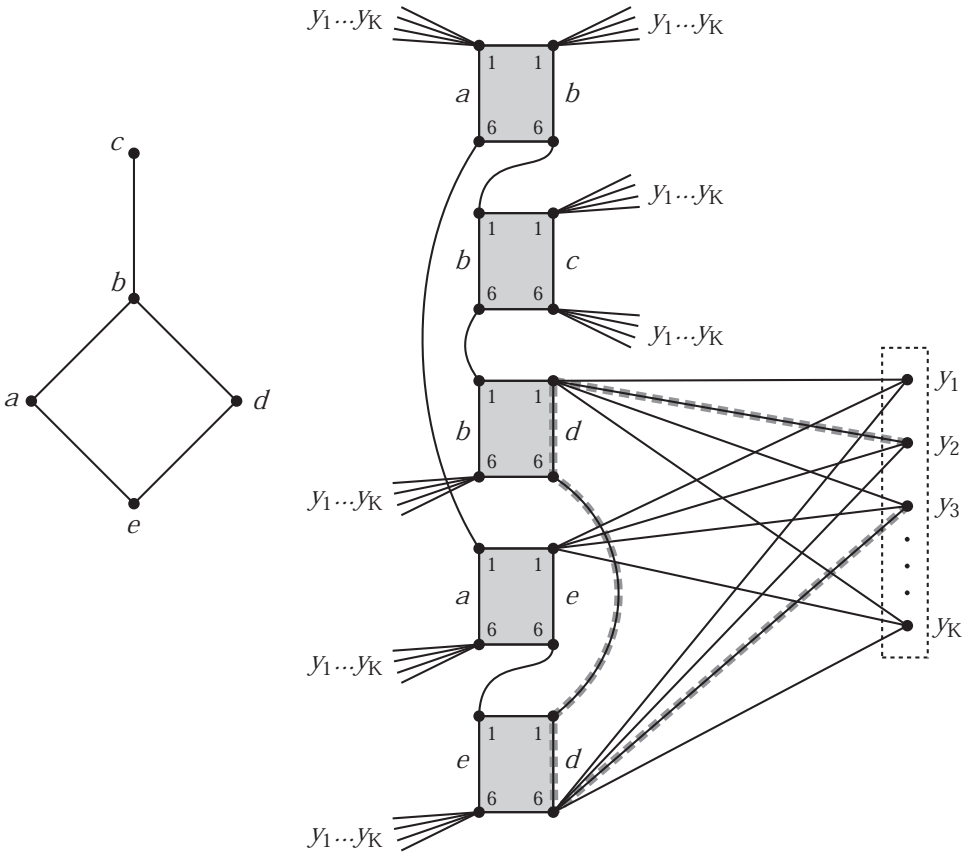


Figure 10.9: *The construction used for reducing* VERTEX-COVER *to* HAMILTONIAN-CIRCUIT. *The graph on the left leads to the five gadgets on the right, one for each edge. Each path through the gadgets represents the edges adjacent to one vertex of the original graph. The vertices* $y_1, \ldots, y_k$ *on the far right allow any Hamiltonian path to make exactly* k *loops through the collection of gadgets.*

We now define $2n$ natural numbers, $U_1, \ldots, U_n$ and $\overline{U}_1, \ldots, \overline{U}_n$, by

$$U_i \stackrel{\text{def}}{=} B^{K+i-1} + \sum_{1 \le j \le K} \delta(i,j) B^{j-1} \tag{10.18}$$

$$\overline{U}_i \stackrel{\text{def}}{=} B^{K+i-1} + \sum_{1 \le j \le K} \overline{\delta}(i,j) B^{j-1} \tag{10.19}$$

for $i = 1, 2, \ldots, n$. Thus, U_i encodes the clauses that the literal x_i appears in and $\overline{U}_i$ encodes the clauses that the literal $\overline{x}_i$ appears in. Next we define the numbers $V_1, \ldots, V_K$ by

$$V_j \stackrel{\text{def}}{=} B^{j-1}, \tag{10.20}$$

for $j = 1, 2, \ldots, K$, and the natural numbers W by

$$W \stackrel{\text{def}}{=} \sum_{1 \le j \le K} B^{j-1}. \tag{10.21}$$

Finally, we define the list (multiset) of $2(n+K)+1$ natural numbers

$$S \stackrel{\text{def}}{=} \left\{U_1, \ldots, U_n, \overline{U}_1, \ldots, \overline{U}_n, V_1, \ldots, V_K, V_1, \ldots, V_K, W\right\}. \tag{10.22}$$

We claim that S can be partitioned into two lists (multi-sets) with equal sums if and only if the original boolean formula is satisfiable. Moreover, the satisfying truth assignment can be easily read from the solution to the partition problem; all literals in the set containing W are set to $\bot$, and all literals in the set not containing W are set to $\top$.

To see this, imagine the collection of numbers S organized as shown in Figure 10.10. The constant B was chosen so that the sum of any subset of these numbers cannot result in a carry from one field to the next, as no column sums to more than six. The sum of the fields in the gray column is always exactly two. This ensures that the numbers corresponding to a variable and its negation (i.e. U_i and $\overline{U}_i$) cannot both be on the same side of the equation. In this context, the "equation" is that in which the sum of a subset of the numbers is on one side, and the sum of the rest is on the other. The sum of the fields in a white column is always six: three from group (a) that represent the variables in a clause, two from group (b), since there are two copies of these numbers for each clause, and one from group (c), which contains a single number with a one in each white column.

Consider the column corresponding to clause C_j. Observe that not all of the numbers representing the literals in this clause can be on the same side of the equation as W, since the fields in this column would then sum to at least four on one side, which cannot be matched on the other side (since the total for this field is only six). Consequently, at least one of the literals in clause j must be on the side *opposite* W. This is the *only* constraint, however, since the V numbers assure that all other combinations can result in an even partition; that is, if one, two, or three literals are on the side opposite W, the numbers within field j can be evenly partitioned by placing zero, one, or both of the V's on the side of W, respectively. This is true within each white field independently. Since the gray fields prevent both U_i and $\overline{U}_i$ from being on the same side of the equation, for $1 \leq i \leq n$, the side that *does not* contain W defines a valid truth assignment that places at least one true literal in each clause; that is, it is possible to simultaneously set all these literals to $\top$, and doing so satisfies the original boolean formula.

This construction shows how to reduce 3CNF-SAT to PARTITION. Each row represents a natural number that is partitioned into a number of smaller fields. The fields are sufficiently wide so that adding these numbers will not result in in a carry from one field to the next. There is an number in block (a) for each variable as well as its negation. The gray fields prevent both a variable and its negation from being in the same partition (i.e. on the same side of the equation). There are two identical numbers in block (b) for each clause in the boolean formula. There is only one number in block (c); this number will indicate which half of the partition contains the truth assignment; specifically, setting all the literals in the sub-list that does not contain W to $\top$ will satisfy the original boolean formula.

10.8 Non-Equivalent Star-Free Regular Expressions

The language of *non-equivalent star-free regular expressions*, or NESFRE, is the language consisting of pairs of regular expressions, (e_1, e_2), such that neither e_1 nor e_2 makes use of the Kleene star

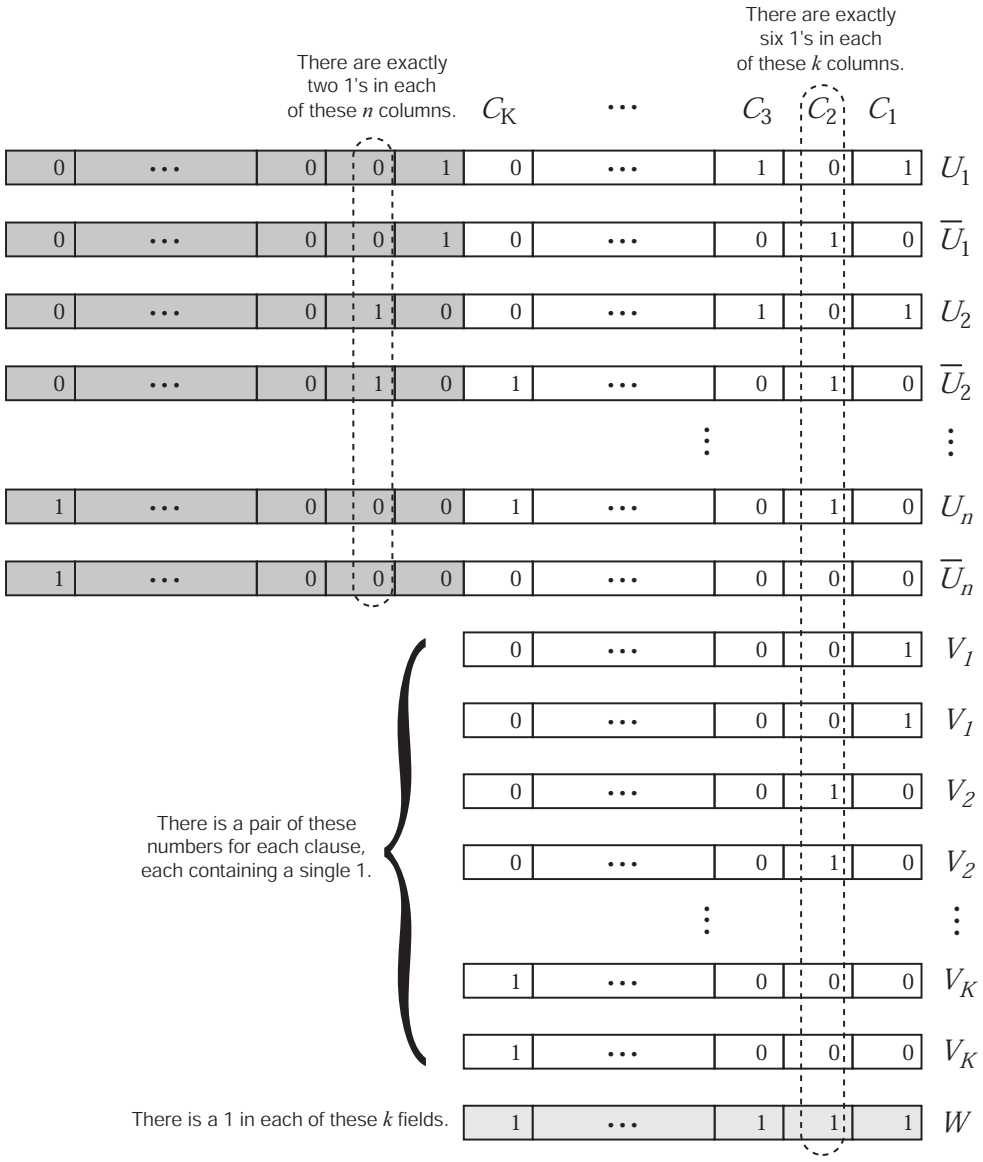


Figure 10.10: *The construction used to reduce* 3CNF-SAT *to* PARTITION. *A boolean formula $\mathcal{F}$ with n variables and k clauses is mapped to a set of $2n + 2k + 1$ natural numbers. Each number is formed using fields wide enough to prevent a carry from one field to the next when added. There is one number for each variable, and one for each negation of a variable. In any valid partition, the high-order bits (in gray) force the numbers representing negated and un-negated variables to be in different subsets. The low-order bits encode the clauses that the literal appears in. If a partition exists, falsifying the literals in the subset containing W will satisfy $\mathcal{F}$. If no partition exists $\mathcal{F}$ cannot be satisfied.*

operator, and the languages generated by e_1 and e_2 are different, making them *non-equivalent*. This language corresponds to the following decision problem: "Given two star-free regular expressions,

e_1 and e_2, are they non-equivalent?" We shall show that this problem is **NP**-complete by reduction from CNF-SAT.

Let F be a Boolean formula in CNF form, and let C_1, $C_2, \ldots, C_k$ denote the clauses of F, and let $x_1, x_2, \ldots, x_n$ denote the Boolean variables appearing in F; Thus, F has k clauses and n distinct variables. Let us encode truth assignments for F as a string of length n consisting of the symbols $\top$ and $\bot$; the ith symbol in such a string simply indicates the value assigned to variable x_i.

We shall create a regular expression, R_1, over the alphabet $\Sigma = \{\top, \bot\}$ that generates all *falsifying* truth assignments of F, and another regular expression, R_2, over the same alphabet that generates all 2^n possible truth assignments for F. Observe that F is satisfiable if and only if it is *not* falsified by all truth assignments. Thus, F is satisfiable if and only if R_1 is *not* equivalent to R_2. This is equivalent to the statement that F is in CNF-SAT if and only if (R_1, R_2) is in NESFRE.

We first define a family of simple regular expressions, one for each literal of each clause.

$$X_{ij} \;=\; \begin{cases} \bot & \text{if } x_i \text{ is a literal in clause } C_j \\ \top & \text{if } \neg x_i \text{ is a literal in clause } C_j \\ (\top \mid \bot) & \text{otherwise} \end{cases} \tag{10.23}$$

Then X_{ij} indicates the possible truth assignments for variable i that are compatible with clause C_j being falsified. If we juxtapose all n of these for a given clause C_j,

$$E_j \;=\; X_{1j}\, X_{2j} \cdots X_{nj}, \tag{10.24}$$

then E_j is a regular expression that generates *all* possible truth assignments that falsify clause C_j. To falsify the original formula we need only falsify a single clause. Taking the union of the truth assignments that falsify the individual clauses, we arrive at the regular expression that produces all falsifying truth assignments of F:

$$R_1 = E_1 \mid E_2 \mid \cdots \mid E_k. \tag{10.25}$$

The regular expression that generates *all* possible truth assignments for the n variables is simply

$$R_2 = (\top \mid \bot)\,(\top \mid \bot) \cdots (\top \mid \bot), \tag{10.26}$$

where the expression in parentheses is concatenated n times. Then $\mathcal{L}(R_1) = \mathcal{L}(R_2)$ if and only if *every* truth assignment falsifies the Boolean formula F, which is to say that F is unsatisfiable. Equivalently, $\mathcal{L}(R_1) \neq \mathcal{L}(R_2)$ if and only if F is satisfiable.

The language NESFRE is in **NP** since we need only non-deterministically guess a string that is generated by one expression and not the other, then check that this is the case. We next observe that CNF-SAT $\leq_m^P$ NESFRE via the construction above, which clearly can be carried out in polynomial time.

Observe that the situation is entirely different if we were to be testing for the *equivalence* of star-free regular expressions rather than *non-equivalence*. In this case, the corresponding decision problem is not even in **NP**, since elements of the language do not seem to admit short certificates of membership; that is, there is no "guess" that we can make that can be easily checked.

10.9 Integer Programming

Proof by reduction from 3CNF-SAT. That is, we will show that 3CNF-SAT $\leq_m^P$ INTEGER-PROGRAMMING.

Let $\mathcal{F}$ be a boolean formula in 3CNF. Let $v_1, v_2, \ldots, v_n$ denote the variables appearing in $\mathcal{F}$, and let $\overline{v}_1, \overline{v}_2, \ldots, \overline{v}_n$ denote their negations. We will encode a truth assignment for $\mathcal{F}$ as a vector $\mathbf{x}$ of $2n$ elements, consisting of 0's and 1's. The first n elements will encode the truth values of $v_1, v_2, \ldots, v_n$, while the last n elements encode the truth values of $\overline{v}_1, \overline{v}_2, \ldots, \overline{v}_n$. Specifically, a 1 will encode $\top$, and a 0 will encode $\bot$. We wish to set up an integer program such that any valid solution will represent a valid truth assignment that satisfies $\mathcal{F}$; that is, a truth assignment that will simultaneously satisfy at least one literal in each clause of $\mathcal{F}$. We can accomplish this by imposing the following constraints on the elements of $\mathbf{x}$:

1. $x_i \leq 1$, for $i = 1, 2, \ldots, 2n$.
2. $x_i + x_{i+n} \leq 1$, for $i = 1, 2, \ldots, n$.
3. $x_i + x_j + x_k \geq 1$, where (i, j, k) are the indices of the literals appearing in each of the clauses, $C_1, C_2, \ldots, C_k$, of $\mathcal{F}$.

The first two sets of constraints ensure that any integer solution will correspond to a valid truth assignment; moreover, any valid truth assignment will automatically satisfy these constraints. The third set of constraints ensures that any valid solution will assign at least one variable in each clause a positive value. These constraints can be easily expressed as an integer program by defining the $m \times k$ integer matrix, $\mathbf{A}$, where $m = 3n + k$, and the vector $\mathbf{b} \in \mathbb{N}^m$, as follows:

$$\mathbf{A} = \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{I} \\ \mathbf{I} & \mathbf{I} \\ -\mathbf{B} & -\overline{\mathbf{B}} \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} \mathbf{1} \\ \mathbf{1} \\ \mathbf{1} \\ -\mathbf{1} \end{bmatrix} \tag{10.27}$$

where $\mathbf{I}$ is the $n \times n$ identity matrix, and $\mathbf{B}$ and $\overline{\mathbf{B}}$ are both $k \times n$ matrices, defined as follows:

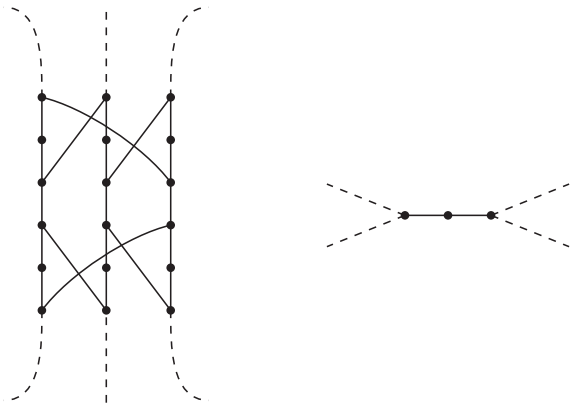
$$\mathbf{B}_{ij} = \begin{cases} 1 & \text{if } v_j \in C_i \\ 0 & \text{otherwise} \end{cases} \tag{10.28}$$

and

$$\overline{\mathbf{B}}_{ij} = \begin{cases} 1 & \text{if } \overline{v}_j \in C_i \\ 0 & \text{otherwise} \end{cases} \tag{10.29}$$

10.10 Exercises

1. Describe how to preform the reduction CNF-SAT $\leq_m^P$ 3-COLOR by generalizing the gadget used in the reduction of 3CNF-SAT to 3-COLOR.
2. Describe how to preform the reduction 3CNF-SAT $\leq_m^P$ HAMILTONIAN-CIRCUIT by using the two types of gadgets shown below.



3. Describe how to preform the reduction CNF-SAT $\leq_m^P$ HAMILTONIAN-CIRCUIT by generalizing the gadget used in the reduction of 3CNF-SAT to HAMILTONIAN-CIRCUIT.

4. Let DIRECTED-HAMILTONIAN-CIRCUIT be the problem of determining whether there exists a Hamiltonian circuit in a directed graph. Show that DIRECTED-HAMILTONIAN-CIRCUIT is **NP**-complete. Hint: Modify the gadget used to show that 3CNF-SAT $\leq_m^P$ HAMILTONIAN-CIRCUIT.

5. Show that EXACT-2-COVER is in **P**.

6. Assuming real number distances, show that the optimal solution to EUCLIDEAN-TRAVELING-SALESMAN is a simple polygon (i.e. the path does not cross itself).