Chapter 14

Tractable and Intractable Problems

14.2. Suppose that the input string is of length n, and we wish to find all occurrences of a substring $\alpha = \$1^k\$$ of the input. Comparing a single character of α to another character of the input might take up to $2n$ moves, but no character in the input needs to be compared to a character of α more than once, so all occurrences of α can be found within $2n^2$ moves.

14.3. (c) The statement $L \in NP$ means that there exist a nondeterministic TM T and a polynomial p so that T accepts L and $\tau_T \leq p$. This implies that for any string of length n in L, there is a sequence of no more than $p(n)$ moves that causes T to accept the string; and for a string of length n not in L, there is a sequence of $p(n)$ or fewer moves causing T to reject the string. If we construct another TM T' that halts whenever T crashes, and vice versa, then $\tau_{T'}$ is still $\leq p$. However, T' accepts the strings for which there is at least one sequence of moves of T causing T to crash, and these are not in general the same as the strings not in L. There is no other obvious way to modify T so that the modified TM accepts L' in polynomial time.

14.4. (a) Any function reducing L_1 to L_2 also reduces L_1' to L_2'.

(b) Suppose L is NP-complete and $L' \in NP$. Let L_1 be any language in NP. Since $L' \in NP$, we can show $L_1' \in NP$ by showing that $L_1' \leq L'$. But since L is NP-complete, $L_1 \leq L$. Therefore, the result follows from part (a).

14.5. By assumption, there are two strings x_1 and x_2 so that $x_1 \in L_2$ and $x_2 \notin L_2$. Define the function $f : \Sigma^* \to \Sigma^*$ by the formula

$$f(x) = \begin{cases} x_1 & \text{if } x \in L_1 \\ x_2 & \text{otherwise} \end{cases}$$

Then for any x, $x \in L_1$ if and only if $f(x) \in L_2$, and the assumption that $L_1 \in P$ implies that f is computable in polynomial time.

14.6. (a) To say that "P_2 is hard" presumably means that there are instances of P_2 that are hard (not necessarily that all instances are hard). It may be that all the instances of P_1 are easy, so that P_1 is easy.

(b) We show that *3-Satisfiable* $\leq_p$ *Satisfiable*. Define $f : \Sigma^* \to \Sigma^*$ by $f(x) = x$ if x is the encoding of a CNF expression involving three or fewer conjuncts, and $f(x) = \Lambda$ otherwise. This function reduces *3-Satisfiable* to *Satisfiable*, and it is not hard to see that it is polynomial-time computable.

(c) *3-Satisfiable* is the intersection of two languages, *Satisfiable* and the language *3-CNF* of all encodings of CNF expressions in which there are three or fewer conjuncts. The second language is in P. One generalization is therefore the following: if $L, L_1 \subseteq \Sigma^*$, $L \neq \Sigma^*$, and $L_1 \in P$, then $L \cap L_1 \leq_p L$. To show this, use the reduction f, where $f(x)$ is x if $x \in L_1$ and x_0 otherwise (where x_0 is a fixed string not in L). Then if $x \in L \cap L_1$, $f(x) \in L$; if

$x \notin L$, then $f(x) \notin L$ (either because $f(x) = x$ or because $f(x) = x_0$); and if $x \notin L_1$, then $f(x) = x_0 \notin L$.

14.9. We give the solution for $k = 4$, and the more general case is similar. Given an instance

$$\bigwedge_{i=1}^{n} (x_{i,1} \vee x_{i,2} \vee x_{i,3})$$

of the 3-satisfiability problem, we may construct the instance

$$\bigwedge_{i=1}^{n} (a \vee x_{i,1} \vee x_{i,2} \vee x_{i,3}) \wedge \bigwedge_{i=1}^{n} (\overline{a} \vee x_{i,1} \vee x_{i,2} \vee x_{i,3})$$

of the 4-satisfiability problem (where a is a variable not appearing in the original expression), which is satisfiable if and only if the original is. This clearly implies that the 3-satisfiability problem is polynomial-time reducible to the 4-satisfiability problem. Since the former is NP-complete, so is the latter.

14.10. Consider any possible conjunct, say $x \wedge y \wedge z$. If this one is missing from the expression, then the expression must be satisfied by the assignment in which all three variables are false, because every other possible conjunct contains the negation of one variable. Therefore, the smallest unsatisfiable expression satisfying these conditions is the one with all 8 possible conjuncts.

14.11. (a) The entire expression is satisfiable if and only if at least one of the disjuncts is satisfiable, and this is true if and only if at least one of the disjuncts doesn't contain both a variable and its negation. Testing this condition can obviously be done in polynomial time.

(b) A CNF expression is a tautology if and only if each conjunct contains some variable and its negation, and this can be checked in polynomial time.

14.14. The language L is in *NP*; a nondeterministic procedure to accept L is to take the input string and, provided it is of the form $e(T)\$e(x)\$1^n$, choose a sequence of n moves of T on input x, accepting the input if the sequence causes x to be accepted by T.

To show that L is *NP*-hard, let L_1 be any language in *NP*, and let T be a nondeterministic TM that accepts L_1 and has nondeterministic time complexity bounded by some polynomial p. Then consider the function $f : \Sigma^* \to \{0,1\}^*$ defined by $f(x) = e(T)\$e(x)\$1^{p(|x|)}$. Any string x is in L_1 if and only if $f(x) \in L$, and it is easy to check that f is polynomial-time computable.

14.15. Consider the following procedure for coloring the vertices in each "connected component" of the graph. Choose one and color it white. Color all the vertices adjacent to it black. For each of those, color all the vertices adjacent to it (and not yet colored) white. Continue this until there are no colored vertices with uncolored adjacent vertices. Repeat for each component. This can be carried out in polynomial time, and it can be determined

in polynomial time whether the resulting assignment of colors is in fact a 2-coloring of the graph. Furthermore, the graph can be 2-colored if and only if this procedure produces a 2-coloring.

14.17. A TM can perform these two steps on the input string x in order to recognize $f^{-1}(A)$: first calculate $f(x)$, and then test $f(x)$ for membership in A. Since $A \in P$, and since the length of the string $f(x)$ is no more than a polynomial function of $|x|$, both steps can be performed in time no more than a polynomial function of $|x|$.

14.18. We already have a reduction from **CSG** to **VC**; exactly the same function defines a reduction in the other direction. If $G = (V, E)$, (G, k) is an instance of **VC**, and there is a vertex cover of G having the k vertices $\nu_1, \ldots, \nu_k$, then the remaining $|V| - k$ vertices determine a complete subgraph in the complement of G, since no two of them can be joined by an edge in V. Conversely, if there is a complete subgraph on $|V| - k$ vertices in the complement of G, then the remaining k vertices form a vertex cover for G.

Now consider the function that sends a pair (G, k) to the pair (G', k), where G' is the complement of G. This function is both a reduction from **CSG** to **IS** and a reduction from **IS** to **CSG**. The reason is that a set of vertices determines a complete subgraph in one graph if and only if the same set is independent in the complement.

Finally, we can get reductions from **IS** to **VC** and back by considering the composition of the two functions above. If (G, k) is an instance of **IS** , consider the instance $(G, |V| - k)$ of **VC**. Vertices $\nu_1, \ldots, \nu_k$ form an independent set in G if and only if the remaining $|V| - k$ vertices form a vertex cover of G. This function is therefore a reduction from either of the two problems to the other.

14.21. (a) The function f defined by $f(x) = x0$ reduces L_1 in polynomial time to $L_1 \oplus L_2$, because if $x \in L_1$, $f(x) \in L_1\{0\}$, and if $x \notin L_1$, $x0 \notin L_1\{0\} \cup L_2\{1\}$. Similarly, $g(x) = x1$ reduces L_1 to $L_1 \oplus L_2$.

(b) The problem also requires the hypothesis that $L \neq \Sigma^*$; let $z \notin L$. Suppose that f_1 and f_2 are polynomial-time reductions from L_1 to L and from L_2 to L, respectively. Define $f : \Sigma^* \to \Sigma^*$ by $f(\Lambda) = z$, $f(y0) = f_1(y)$, and $f(y1) = f_2(y)$ (for any $y \in \Sigma^*$). Then if $x \in L_1 \oplus L_2$, there are two cases: if $x = y0$ for some $y \in L_1$, then $f(x) = f_1(y) \in L$; and if $x = y1$ for some $y \in L_2$, $f(x) = f_2(y) \in L$. If $x \notin L_1 \oplus L_2$, then there are three cases. If $x = \Lambda$, $f(x) \notin L$ by the assumption on z. If $x = y0$ and $y \notin L_1$, then $f(x) = f_1(y) \notin L$; and if $x = y1$ and $y \notin L_2$, then $f(x) = f_2(y) \notin L$. Since f_1 and f_2 are both polynomial-time computable, f is also.

14.22. We shall describe a polynomial-time algorithm for determining whether a CNF expression in which each conjunct has exactly two literals is satisfiable. (It is taken from Salomaa, *Computation and Automata*, Cambridge University Press, 1985, pp. 162-163.)

1. Delete any conjuncts of the form $(x \vee \overline{x})$.

2. Replace any conjuncts of the form $(x \vee x)$ by the single term x (where x is either an a or an $\overline{a}$).

3. If the resulting expression contains a conjunct that is a single x (either an a or an $\overline{a}$), then

   (a) If $\overline{x}$ appears by itself as another conjunct, then stop, and conclude that the expression is unsatisfiable.

   (b) Remove all conjuncts containing x (either by itself or with another variable).

   (c) Replace any conjunct of the form $(\overline{x} \vee y)$ by y.

   Iterate step 3 until there are no remaining conjuncts of the form x.

4. At this point, all conjuncts look like $(x \vee y)$, where $x \neq y$ and $x \neq \overline{y}$. If some conjunct contains x and no other conjunct contains $\overline{x}$, then delete all conjuncts containing x. Iterate this step until, for every literal x in the expression, both x and $\overline{x}$ appear in the expression. If there are no literals remaining, stop and conclude that the expression is satisfiable.

5. Choose an arbitrary literal x in the expression, and let the conjuncts containing x and $\overline{x}$ be

$$(x \vee y_1), \ldots, (x \vee y_m), (\overline{x} \vee z_1), \ldots, (\overline{x} \vee z_n)$$

   Let the remaining portion of the expression be α. Then replace the expression by

$$\bigwedge_{i=1}^{m} \bigwedge_{j=1}^{n} (y_i \vee z_j) \wedge \alpha$$

   (This is correct because the expression is equivalent to

$$((\bigwedge_{i=1}^{m} y_i) \vee (\bigwedge_{j=1}^{n} z_j)) \wedge \alpha$$

   and the given expression is simply the CNF version of this.)

6. Go to Step 1.

It is straightforward to check that each step requires time that is only polynomial in the length of the current expression, and that the expression resulting from each step has length that is only polynomial in the length of the previous expression. Since each time the algorithm returns to step 1, the number of distinct variables has been reduced by at least 1, we may conclude that the entire algorithm takes only polynomial time.

14.23. The part that is least obvious is that P is closed under Kleene *. Suppose $L \in P$. We describe an algorithm for recognizing L^* in general terms, and it is not difficult to see that a TM can execute it in polynomial time. (There are considerably more efficient algorithms, whose runtimes would be polynomials of lower degree.)

Given a string $x = a_1a_2 \ldots a_n$ of length n, and integers i and j with $1 \leq i \leq j \leq n+1$, denote by $x[i,j]$ the substring $a_ia_{i+1} \ldots a_{j-1}$ of length $i-j$. There are $(n+1)(n+2)/2$ ways of choosing i and j (although the substrings $x[i,j]$ are not all distinct). In the algorithm, we keep a table in which every $x[i,j]$ is marked with a 1 if it has been found to be in L^* and 0 otherwise. We initialize the table by marking $x[i,j]$ with a 1 if it is in L, and 0 if not. Since $L \in P$, this initialization can be done in polynomial time. We then begin a sequence of iterations. Each iteration consists of examining every pair $x[i,j]$, $x[j,k]$, where $0 \leq i \leq j \leq k \leq n+1$; if $x[i,j]$ and $x[j,k]$ are both marked with 1, then their concatenation $x[i,k]$ is marked with 1. We continue until either we have performed n iterations or we have executed an iteration in which the table did not change. x is in L^* if and only if $x = x[1, n+1]$ is marked with a 1 at the termination of the algorithm.

14.24. Suppose L is *NP*-complete, T recognizes L, and $\tau_T \leq f$, where f is subexponential. Let L_1 be any language in *NP*. Since L is *NP*-complete, there is a function r reducing L_1 to L so that r can be computed in time bounded by some polynomial p. We construct a TM T_1 to recognize L_1 as follows: T_1 takes an input string, computes $y = r(x)$, and then tests y for membership in L. The time required for this is no more than $p(|x|) + f(|r(x)|)$. In order to show that this is a subexponential function of $|x|$, it is sufficient to show that the second term is.

We may assume that f is nondecreasing, since the function f_1 defined by $f_1(n) = \max\{f(i) \mid 0 \leq i \leq n\}$ is nondecreasing, at least as big as f, and still subexponential. Then $f(|r(x)|) \leq f(p(|x|))$, since $|r(x)| \leq p(|x|)$. Since p is a polynomial, there is a k so that $p(n) \leq n^k$ for sufficiently large n. Therefore, $f(p(n)) \leq f(n^k)$. Since f is subexponential, then for any $\delta > 0$, $f(n^k) \leq C2^{(n^k)^\delta}$ for some C and all sufficiently large n. Let us choose $\delta = \epsilon/k$. It follows that for sufficiently large n, if $|x| = n$,

$$f(|r(x)|) \leq f(p(|x|)) \leq f(p(n)) \leq f(n^k) \leq C2^{(n^k)^\delta} \leq C2^{n^\epsilon}$$

14.25. We reduce the vertex cover problem to this one as follows. Given G and k, an instance of the vertex cover problem, we may construct G_1 and k_1 as follows: add 3 new vertices v_1, v_2, and v_3 to G; add the three edges joining each pair of these; and add edges to connect v_1 to every vertex of G having odd degree; $k_1 = k + 2$. In G_1, every vertex has even degree. Now the claim is that G has a vertex cover with k vertices if and only if G_1 has a vertex cover with k_1 vertices. One direction is easy: if G has a vertex cover with k vertices, those vertices plus v_1 and v_2 clearly form a vertex cover for G_1. Now suppose there is a vertex cover for G_1 having $k+2$ vertices. Then clearly, the vertices in the vertex cover that are in G form a vertex cover for G. The vertex cover for G_1 must contain at least two of the three vertices v_1, v_2, and v_3, since it has edges joining any two of these. Therefore, there is a vertex cover for G having k vertices. Since it is fairly obvious that this is a polynomial-time reduction, and since the vertex cover problem is *NP*-complete, this problem is also *NP*-complete.

14.28. Let $G = (E, V)$ be a graph and k an integer. We wish to construct an instance

$(S_1, S_2, \ldots, S_m)$ of the exact cover problem corresponding to (G, k). The elements belonging to the subsets $S_1, \ldots, S_m$ will be of two types: elements of V, and pairs of the form (e, i), where $e \in E$ and $1 \leq i \leq k$. Specifically, for each $v \in V$ and each i with $1 \leq i \leq k$, let

$$S_{v,i} = \{v\} \cup \{(e, i) \mid v \text{ is an end point of } e\}$$

In addition, for each pair (e, i), let $T_{e,i} = \{(e, i)\}$. Then the union of all the sets $S_{v,i}$ and $T_{e,i}$ contains all the vertices of the graph, as well as all the possible pairs (e, i).

Suppose on the one hand that there is a k-coloring of G. Let us denote the colors by 1, 2, $\ldots$, k, and let us suppose that each vertex v is colored with the color i_v. Then we can construct an exact cover for our collection of sets as follows. We include every set S_{v,i_v}. The union of these contains all the vertices. We also include all the $T_{e,i}$'s for which the pair (e, i) has not already been included in some S_{v,i_v}. Clearly, these sets form a cover—that is, their union contains all the elements in the original union. To see that it is an exact cover, it is sufficient to check that no two sets S_{v,i_v} and S_{w,i_w} can intersect if $v \neq w$. This is obvious if v and w are not adjacent; if they are adjacent, it follows from the fact that the colors i_v and i_w are different.

On the other hand, suppose that there is an exact cover for the collection of $S_{v,i}$'s and $T_{e,i}$'s. Then for each vertex v, v can appear in only one set in the exact cover, and thus there can be only one i, say i_v, for which $S_{v,i}$ is included. Now we can check that if we color each vertex v with the color i_v, we do in fact get a k-coloring of G. If not, then some edge e joins two vertices v and w with $i_v = i_w = i$. Then, however, both S_{v,i_v} and S_{w,i_w} contain the pair (e, i) which is impossible if the cover is exact.

14.29. Suppose S_1, S_2, $\ldots$, S_k are finite sets, where $\cup_{j=1}^{k} S_j = X = \{x_0, x_1, \ldots, x_{n-1}\}$. We view the S_j's as an instance of the exact cover problem: it is a yes-instance if it is possible to choose some of the S_j's which are pairwise disjoint and still have union X. We wish to construct from the S_j's an instance of the sum-of-subsets problem.

For each j with $1 \leq j \leq k$, define

$$a_j = \sum_{i=0}^{n-1} \epsilon_{ji}(k+1)^i, \text{ where } \epsilon_{ji} = \begin{cases} 1 & \text{if } x_i \in S_j \\ 0 & \text{otherwise} \end{cases}$$

Let the integer M be $\sum_{i=0}^{n-1}(k+1)^i$. This is a yes-instance of the sum-of-subsets problem if it is possible to choose some of the a_j's whose sum is M.

Suppose on the one hand that there is an exact cover for the S_j's, say $\{S_j \mid j \in J\}$. Then for each i, there is exactly one $j \in J$ for which $x_i \in S_j$, so that $\sum_{j \in J} \epsilon_{ji} = 1$. It follows that

$$\sum_{j \in J} a_j = \sum_{j \in J} \sum_{i=0}^{n-1} \epsilon_{ji}(k+1)^i = \sum_{i=0}^{n-1} (\sum_{j \in J} \epsilon_{ji})(k+1)^i = \sum_{i=0}^{n-1} (k+1)^i = M$$

On the other hand, if there is a subset J of $\{1, 2, \ldots, k\}$ for which $\sum_{j \in J} a_j = M$, then rearranging the double sum the way we did above, we obtain

$$\sum_{i=0}^{n-1} b_i(k+1)^i = \sum_{i=0}^{n-1} 1(k+1)^i = M$$

where $b_i = \sum_{j \in J} \epsilon_{ji}$. The set J contains at most k elements, so that $0 \leq b_i \leq k$. Since expansions of an integer to base $k+1$ are unique, it follows that $b_i = 1$ for each i. This means that for each i, there is exactly one $j \in J$ with $\epsilon_{ji} = 1$, and this implies that the S_j's with $j \in J$ form an exact cover for the S_j's.

As usual, it is necessary to check that the instance of the sum-of-subsets problem can be constructed in polynomial time from the instance of the exact cover problem, but this is straightforward.

14.30. Let $\{a_1, a_2, \ldots, a_n\}$ and M represent an instance of the sum-of-subsets problem. We construct an instance b_1, b_2, $\ldots$, b_m of the partition problem by letting $m = n+2$ and letting the b_i's be defined by

$$b_i = a_i \text{ for } 1 \leq i \leq n \qquad b_{n+1} = M+1 \qquad b_{n+2} = \sum_{i=1}^{n} a_i + 1 - M$$

If there is a subset I of $\{1, 2, \ldots, n\}$ so that $\sum_{i \in I} a_i = M$, then we let $J = I \cup \{n+1\}$, and it is easy to check that

$$\sum_{j \in J} b_j = \sum_{j \notin J} b_j$$

On the other hand, if there is a subset J of $\{1, 2, \ldots, n+2\}$ with $\sum_{j \in J} b_j = \sum_{j \notin J} b_j$, then since $b_{n+1} + b_{n+2}$ is bigger than the sum of the remaining b_j's, exactly one of the two numbers $n+1$, $n+2$ is in J. If we suppose that it's $n+1$, then it's easy to check that $\sum_{i \in I} a_i = M$, where $I = J - \{n+1\}$.

14.31. Let a_1, a_2, $\ldots$, a_n represent an instance of the partition problem. We construct an instance of the 0-1 knapsack problem by letting $w_i = p_i = a_i$ for each i, and letting $W = P = (1/2)\sum_{i=1}^{n} a_i$. If there is a subset I of $\{1, 2, \ldots, n\}$ with $\sum_{i \in I} a_i = \sum_{i \notin I} a_i$, then $\sum_{i \in I} w_i = W$ and $\sum_{i \in I} p_i = P$. On the other hand, if there is an I so that $\sum_{i \in I} w_i \leq W$ and $\sum_{i \in I} p_i \geq P$, then both inequalities are in fact equalities, and I determines a solution to the partition problem.