17.3 CNF-SAT and 3SAT

The first reductions we present are for problems involving Boolean formulas. A Boolean formula is a parenthesized expression that is formed from Boolean variables using Boolean operations, such as OR ($+$), AND ($\cdot$), NOT (drawn as a bar over the negated subexpression), IMPLIES ($\rightarrow$), and IF-AND-ONLY-IF ($\leftrightarrow$). A Boolean formula is in ***conjunctive normal form*** (CNF) if it is formed as a collection of subexpressions, called ***clauses***, that are combined using AND, with each clause formed as the OR of Boolean variables or their negation, called ***literals***. For example, the following Boolean formula is in CNF:

$$(\overline{x_1} + x_2 + x_4 + \overline{x_7})(x_3 + \overline{x_5})(\overline{x_2} + x_4 + \overline{x_6} + x_8)(x_1 + x_3 + x_5 + \overline{x_8}).$$

This formula evaluates to 1 if x_2, x_3, and x_4 are 1, where we use 0 for **false** and 1 for **true**. CNF is called a "normal" form, because any Boolean formula can be converted into this form.

CNF-SAT

Problem CNF-SAT takes a Boolean formula in CNF form as input and asks whether there is an assignment of Boolean values to its variables so that the formula evaluates to 1. It is easy to show that CNF-SAT is in ***NP***, for, given a Boolean formula S, we can construct a simple nondeterministic algorithm that first "guesses" an assignment of Boolean values for the variables in S and then evaluates each clause of S in turn. If all the clauses of S evaluate to 1, then S is satisfied; otherwise, it is not.

To show that CNF-SAT is ***NP***-hard, we will reduce the Circuit-SAT problem to it in polynomial time. So, suppose we are given a Boolean circuit, C. Without loss of generality, we assume that each AND and OR gate has two inputs and each NOT gate has one input. To begin the construction of a formula S equivalent to C, we create a variable x_i for each input for the entire circuit C. One might be tempted to limit the set of variables to just these x_i's and immediately start constructing a formula for C by combining subexpressions for inputs, but it is not clear that this approach would take polynomial time. (See Exercise C-17.5.) Instead, we create a variable y_i for each output of a gate in C. Then we create a short formula B_g that corresponds to each gate g in C as follows:

- If g is an AND gate with inputs a and b (which could be either x_i's or y_i's) and output c, then $B_g = (c \leftrightarrow (a \cdot b))$.
- If g is an OR gate with inputs a and b and output c, then $B_g = (c \leftrightarrow (a+b))$.
- If g is a NOT gate with input a and output b, then $B_g = (b \leftrightarrow \overline{a})$.

We wish to create our formula S by taking the AND of all of these B_g's, but such a formula would not be in CNF. So our method is to first convert each B_g to be in

a	b	c	$B = (c \leftrightarrow (a \cdot b))$
1	1	1	1
1	1	0	0
1	0	1	0
1	0	0	1
0	1	1	0
0	1	1	1
0	0	1	0
0	0	0	1

DNF formula for $\overline{B} = a \cdot b \cdot \overline{c} + a \cdot \overline{b} \cdot c + \overline{a} \cdot b \cdot c + \overline{a} \cdot \overline{b} \cdot c$

CNF formula for $B = (\overline{a} + \overline{b} + c) \cdot (\overline{a} + b + \overline{c}) \cdot (a + \overline{b} + \overline{c}) \cdot (a + b + \overline{c})$.

Figure 17.8: A truth table for a Boolean formula B over variables a, b, and c. The equivalent formula for $\overline{B}$ in DNF, and equivalent formula for B in CNF.

CNF, and then combine all of these transformed B_g's by AND operations to define the CNF formula S.

To convert a Boolean formula B into CNF, we construct a truth table for B, as shown in Figure 17.8. We then construct a short formula D_i for each table row that evaluates to 0. Each D_i consists of the AND of the variables for the table, with the variable negated if and only if its value in that row is 0. We create a formula D by taking the OR of all the D_i's. Such a formula, which is the OR of formulas that are the AND of variables or their negation, is said to be in ***disjunctive normal form***, or ***DNF***. In this case, we have a DNF formula D that is equivalent to $\overline{B}$, since it evaluates to 1 if and only if B evaluates to 0. To convert D into a CNF formula for B, we apply, to each D_i, De Morgan's laws, which establish that

$$\overline{(a + b)} = \overline{a} \cdot \overline{b} \quad \text{and} \quad \overline{(a \cdot b)} = \overline{a} + \overline{b}.$$

From Figure 17.8, we can replace each B_g that is of the form $(c \leftrightarrow (a \cdot b))$, by

$$(\overline{a} + \overline{b} + c)(\overline{a} + b + \overline{c})(a + \overline{b} + \overline{c})(a + b + \overline{c}),$$

which is in CNF. Likewise, for each B_g that is of the form $(b \leftrightarrow \overline{a})$, we can replace B_g by the equivalent CNF formula

$$(\overline{a} + \overline{b})(a + b).$$

We leave the CNF substitution for a B_g of the form $(c \leftrightarrow (a + b))$ as an exercise (R-17.2). Substituting each B_g in this way results in a CNF formula S' that corresponds exactly to each input and logic gate of the circuit, C. To construct the final Boolean formula S, then, we define $S = S' \cdot y$, where y is the variable that is associated with the output of the gate that defines the value of C itself. Thus, C is satisfiable if and only if S is satisfiable. Moreover, the construction from C to S builds a constant-sized subexpression for each input and gate of C; hence, this construction runs in polynomial time. Therefore, this local-replacement reduction gives us the following.

Theorem 17.7: CNF-SAT *is* ***NP****-complete.*

3SAT

Consider the 3SAT problem, which takes a Boolean formula S that is in conjunctive normal form (CNF) with each clause in S having exactly three literals, and asks whether S is satisfiable. Recall that a Boolean formula is in CNF if it is formed by the AND of a collection of clauses, each of which is the OR of a set of literals. For example, the following formula could be an instance of 3SAT:

$$(\overline{x_1} + x_2 + \overline{x_7})(x_3 + \overline{x_5} + x_6)(\overline{x_2} + x_4 + \overline{x_6})(x_1 + x_5 + \overline{x_8}).$$

Thus, the 3SAT problem is a restricted version of the CNF-SAT problem. (Note that we cannot use the restriction form of ***NP***-hardness proof, however, for this proof form only works for reducing a restricted version to its more general form.) In this subsection, we show that 3SAT is ***NP***-complete, using the local-replacement form of proof. Interestingly, the 2SAT problem, in which every clause has exactly two literals, can be solved in polynomial time. (See Exercises C-17.6 and C-17.7.)

Note that 3SAT is in ***NP***, for we can construct a nondeterministic polynomial-time algorithm that takes a CNF formula S with 3-literals per clause, guesses an assignment of Boolean values for S, and then evaluates S to see if it is equal to 1.

To prove that 3SAT is ***NP***-hard, we reduce the CNF-SAT problem to it in polynomial time. Let C be a given Boolean formula in CNF. We perform the following local replacement for each clause C_i in C:

- If $C_i = (a)$, that is, it has one term, which may be a negated variable, then we replace C_i with $S_i = (a + b + c) \cdot (a + \overline{b} + c) \cdot (a + b + \overline{c}) \cdot (a + \overline{b} + \overline{c})$, where b and c are new variables not used anywhere else.
- If $C_i = (a + b)$, that is, it has two terms, then we replace C_i with the subformula $S_i = (a + b + c) \cdot (a + b + \overline{c})$, where c is a new variable not used anywhere else.
- If $C_i = (a + b + c)$, that is, it has three terms, then we set $S_i = C_i$.
- If $C_i = (a_1 + a_2 + a_3 + \cdots + a_k)$, that is, it has $k > 3$ terms, then we replace C_i with $S_i = (a_1 + a_2 + b_1) \cdot (\overline{b_1} + a_3 + b_2) \cdot (\overline{b_2} + a_4 + b_3) \cdots (\overline{b_{k-3}} + a_{k-1} + a_k)$, where $b_1, b_2, \ldots, b_{k-1}$ are new variables not used anywhere else.

Notice that the value assigned to the newly introduced variables is completely irrelevant. No matter what we assign them, the clause C_i is 1 if and only if the small formula S_i is also 1. Thus, the original clause C is 1 if and only if S is 1. Moreover, note that each clause increases in size by at most a constant factor and that the computations involved are simple substitutions. Therefore, we have shown how to reduce an instance of the CNF-SAT problem to an equivalent instance of the 3SAT problem in polynomial time. This, together with the earlier observation about 3SAT belonging to ***NP***, gives us the following theorem.

Theorem 17.8: 3SAT *is* ***NP****-complete.*

17.4 VERTEX-COVER, CLIQUE, and SET-COVER

In the VERTEX-COVER problem, we are given a graph G and an integer k and asked whether there is a vertex cover for G containing at most k vertices. That is, VERTEX-COVER asks whether there is a subset C of vertices of size at most k, such that for each edge (v, w), we have $v \in C$ or $w \in C$. We showed, in Lemma 17.4, that VERTEX-COVER is in ***NP***.

VERTEX-COVER is ***NP***-Complete

Given that VERTEX-COVER is in ***NP***, to show that VERTEX-COVER is ***NP***-complete, we will show that VERTEX-COVER is ***NP***-hard, by reducing the 3SAT problem to it in polynomial time. This reduction is interesting in two respects. First, it shows an example of reducing a logic problem to a graph problem. Second, it illustrates an application of the component-design proof technique.

Let S be a given instance of the 3SAT problem, that is, a CNF formula such that each clause has exactly three literals. We construct a graph G and an integer k such that G has a vertex cover of size at most k if and only if S is satisfiable. We begin our construction by adding the following:

- For each variable x_i used in the formula S, we add two vertices in G, one that we label with x_i and the other we label with $\overline{x_i}$. We also add the edge $(x_i, \overline{x_i})$ to G. (Note: These labels are for our own benefit; after we construct the graph G, we can always relabel vertices with integers if that is what an instance of the VERTEX-COVER problem should look like.)

Each edge $(x_i, \overline{x_i})$ is a "truth-setting" component, for, with this edge in G, a vertex cover must include at least one of x_i or $\overline{x_i}$. In addition, we add the following:

- For each clause $C_i = (a + b + c)$ in S, we form a triangle consisting of three vertices, $i1$, $i2$, and $i3$, and three edges, $(i1, i2)$, $(i2, i3)$, and $(i3, i1)$.

Note that any vertex cover will have to include at least two of the vertices in $\{i1, i2, i3\}$ for each such triangle. Each such triangle is a "satisfaction-enforcing" component. We then connect these two types of components, by adding, for each clause $C_i = (a + b + c)$, the edges $(i1, a)$, $(i2, b)$, and $(i3, c)$. (See Figure 17.9.) Finally, we set the integer parameter $k = n + 2m$, where n is the number of variables in S and m is the number of clauses. Thus, if there is a vertex cover of size at most k, it must have size exactly k. This completes the construction of an instance of the VERTEX-COVER problem. This construction clearly runs in polynomial time, so let us consider its correctness.

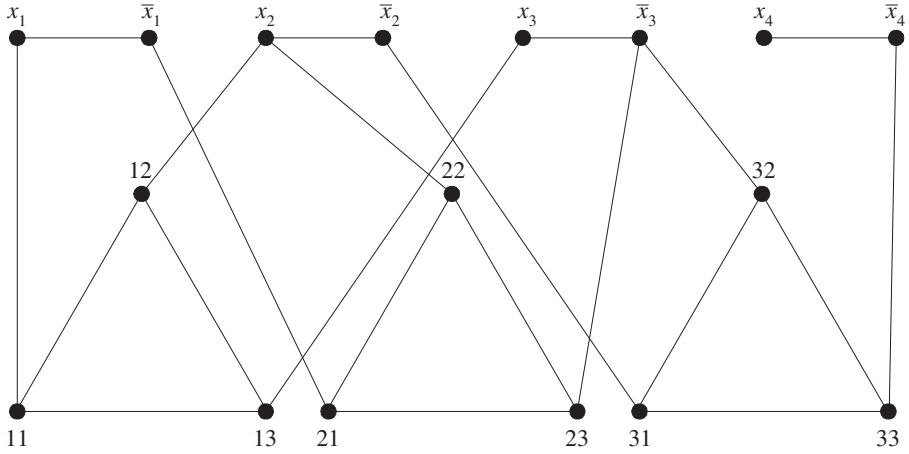


Figure 17.9: Example graph G as an instance of the VERTEX-COVER problem constructed from the formula $S = (x_1 + x_2 + x_3) \cdot (\overline{x_1} + x_2 + \overline{x_3}) \cdot (\overline{x_2} + \overline{x_3} + \overline{x_4})$.

Suppose there is an assignment of Boolean values to variables in S so that S is satisfied. From the graph G constructed from S, we can build a subset of vertices C that contains each literal a (in a truth-setting component) that is assigned 1 by the satisfying assignment. Likewise, for each clause $C_i = (a + b + c)$, the satisfying assignment sets at least one of a, b, or c to 1. Whichever one of a, b, or c is 1 (picking arbitrarily if there are ties), we include the other two in our subset C. This C is of size $n + 2m$. Moreover, notice that each edge in a truth-setting component and clause-satisfying component is covered, and two of every three edges incident on a clause-satisfying component are also covered. In addition, notice that an edge incident to a component associated clause C_i that is not covered by a vertex in the component must be covered by the node in C labeled with a literal, for the corresponding literal in C_i is 1.

Suppose then the converse, namely, that there is a vertex cover C of size at most $n+2m$. By construction, this set must have size exactly $n+2m$, for it must contain one vertex from each truth-setting component and two vertices from each clause-satisfying component. This leaves one edge incident to a clause-satisfying component that is not covered by a vertex in the clause-satisfying component; hence, this edge must be covered by the other endpoint, which is labeled with a literal. Thus, we can assign the literal in S associated with this node 1 and each clause in S is satisfied; hence, all of S is satisfied. Therefore, S is satisfiable if and only if G has a vertex cover of size at most k. This gives us the following.

Theorem 17.9: VERTEX-COVER *is* ***NP****-complete.*

As mentioned before, the above reduction illustrates the component-design technique. We constructed truth-setting and clause-satisfying components in our graph G to enforce important properties in the clause S.

CLIQUE

As with the VERTEX-COVER problem, there are several other problems that involve selecting a subset of objects from a larger set so as to optimize the size the subset can have while still satisfying an important property. The next such problem we consider is the CLIQUE problem.

A ***clique*** in a graph G is a subset C of vertices such that, for each v and w in C, with $v \neq w$, (v, w) is an edge. That is, there is an edge between every pair of distinct vertices in C. Problem CLIQUE takes a graph G and an integer k as input and asks whether there is a clique in G of size at least k.

We leave as a simple exercise (R-17.7) to show that CLIQUE is in ***NP***. To show CLIQUE is ***NP***-hard, we reduce the VERTEX-COVER problem to it. Therefore, let (G, k) be an instance of the VERTEX-COVER problem. For the CLIQUE problem, we construct the complement graph G^c, which has the same vertex set as G, but has the edge (v, w), with $v \neq w$, if and only if (v, w) is not in G. We define the integer parameter for CLIQUE as $n-k$, where k is the integer parameter for VERTEX-COVER. This construction runs in polynomial time and serves as a reduction, for G^c has a clique of size at least $n - k$ if and only if G has a vertex cover of size at most k. (See Figure 17.10.)

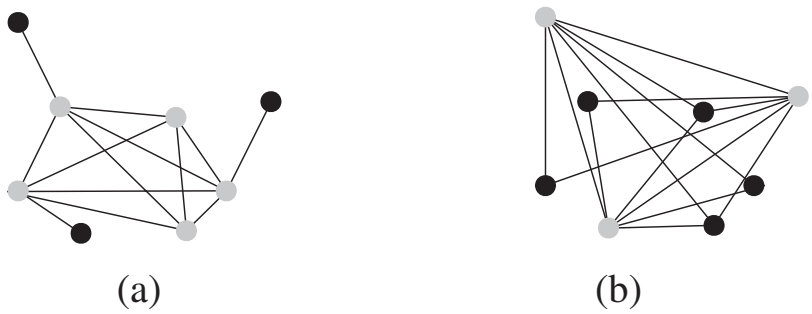


Figure 17.10: A graph G illustrating the proof that CLIQUE is ***NP***-hard. (a) Shows the graph G with the nodes of a clique of size 5 shaded in gray. (b) Shows the graph G^c with the nodes of a vertex cover of size 3 shaded in gray.

Therefore, we have the following.

Theorem 17.10: CLIQUE *is* ***NP****-complete.*

Note how simple the above proof by local replacement is. Interestingly, the next reduction, which is also based on the local-replacement technique, is even simpler.

SET-COVER

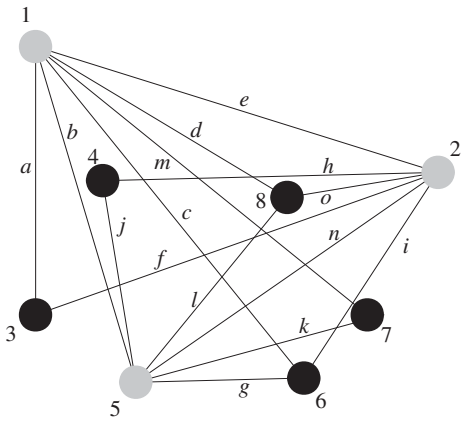
Problem SET-COVER takes a collection of m sets S_1, S_2, ..., S_m and an integer parameter k as input and asks whether there is a subcollection of k sets S_{i_1}, S_{i_2},

$\ldots, S_{i_k}$, such that

$$\bigcup_{i=1}^{m} S_i = \bigcup_{j=1}^{k} S_{i_j}.$$

That is, the union of the subcollection of k sets includes every element in the union of the original m sets.

We leave it to an exercise (R-17.14) to show SET-COVER is in ***NP***. As to the reduction, we note that we can define an instance of SET-COVER from an instance G and k of VERTEX-COVER. Namely, for each vertex v of G, there is set S_v, which contains the edges of G incident on v. Clearly, there is a set cover among these sets S_v's of size k if and only if there is a vertex cover of size k in G. (See Figure 17.11.)



(a)

$$\begin{aligned}
S_1 &= \{a, b, c, m, d, e\} \\
S_2 &= \{e, h, o, n, i\} \\
S_3 &= \{a, f\} \\
S_4 &= \{j, m\} \\
S_5 &= \{b, j, l, n, k, g\} \\
S_6 &= \{g, c, i\} \\
S_7 &= \{k, m\} \\
S_8 &= \{d, o, l\}
\end{aligned}$$

(b)

Figure 17.11: A graph G illustrating the proof that SET-COVER is ***NP***-hard. The vertices are numbered 1 through 8 and the edges are given letter labels a through o. (a) Shows the graph G with the nodes of a vertex cover of size 3 shaded in gray. (b) Shows the sets associated with each vertex in G, with the subscript of each set identifying the associated vertex. Note that $S_1 \cup S_2 \cup S_5$ contains all the edges of G.

Thus we have the following.

Theorem 17.11: SET-COVER *is* ***NP****-complete.*

This reduction illustrates how easily we can covert a graph problem into a set problem. In the next subsection, we show how we can actually reduce graph problems to number problems.

17.5 SUBSET-SUM and KNAPSACK

Some hard problems involve only numbers. In such cases, we must take extra care to use the size of the input in bits, for some numbers can be very large. To clarify the role that the size of numbers can make, researchers say that a problem L is ***strongly NP-hard*** if L remains ***NP***-hard even when we restrict the value of each number in the input to be bounded by a polynomial in the size (in bits) of the input. An input x of size n would satisfy this condition, for example, if each number i in x was represented using $O(\log n)$ bits. Interestingly, the number problems we study in this section are not strongly ***NP***-hard. (See Exercises C-17.14 and C-17.15.)

In the SUBSET-SUM problem, we are given a set S of n integers and an integer k, and we are asked whether there is a subset of integers in S that sum to k. This problem could arise, for example, as in the following.

Example 17.12: *Suppose we have an Internet web server, and we are presented with a collection of download requests. For each download request we can easily determine the size of the requested file. Thus, we can abstract each web request simply as an integer—the size of the requested file. Given this set of integers, we might be interested in determining a subset of them that exactly sums to the bandwidth our server can accommodate in one minute. Unfortunately, this problem is an instance of SUBSET-SUM. Moreover, because it is **NP**-complete, this problem will actually become harder to solve as our web server's bandwidth and request-handling ability improves.*

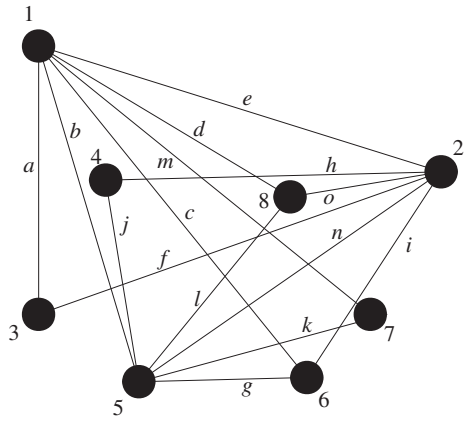
SUBSET-SUM might at first seem easy, and indeed showing that it belongs to ***NP*** is straightforward. (See Exercise R-17.15.) Unfortunately, it is ***NP***-complete, as we now show. Let G and k be given as an instance of the VERTEX-COVER problem. Number the vertices of G from 1 to n and the edges G from 1 to m, and construct the ***incidence matrix*** H for G, defined so that $H[i, j] = 1$ if and only if the edge numbered j is incident on the vertex numbered i; otherwise, $H[i, j] = 0$. (See Figure 17.12.)

We use H to define some admittedly large (but still polynomial-sized) numbers to use as inputs to the SUBSET-SUM problem. Namely, for each row i of H, which encodes all the edges incident on vertex i, we construct the number

$$a_i = 4^{m+1} + \sum_{j=1}^{m} H[i, j]4^j.$$

Note that this number adds in a different power of 4 for each 1-entry in the ith row of $H[i, j]$, plus a larger power of 4 for good measure. The collection of a_i's defines an "incidence component" to our reduction, for each power of 4 in an a_i, except for the largest, corresponds to a possible incidence between vertex i and some edge.

In addition to the above incidence component, we also define an "edge-covering



(a)

H	1	2	3	4	5	6	7	8
a	1	0	1	0	0	0	0	0
b	1	0	0	0	1	0	0	0
c	1	0	0	0	0	1	0	0
d	1	0	0	0	0	0	0	1
e	1	1	0	0	0	0	0	0
f	0	1	1	0	0	0	0	0
g	0	0	0	0	1	1	0	0
h	0	1	0	1	0	0	0	0
i	0	1	0	0	0	1	0	0
j	0	0	0	1	1	0	0	0
k	0	0	0	0	1	0	1	0
l	0	0	0	0	1	0	0	1
m	0	1	0	1	0	0	0	0
n	0	1	0	0	1	0	0	0
o	0	1	0	0	0	0	0	1

(b)

Figure 17.12: A graph G illustrating the proof that SUBSET-SUM is ***NP***-hard. The vertices are numbered 1 through 8 and the edges are given letter labels a through o. (a) Shows the graph G; (b) shows the incidence matrix H for G. Note that there is a 1 for each edge in one or more of the columns for vertices 1, 2, and 5.

component," where, for each edge j, we define a number

$$b_j = 4^j.$$

We then set the sum we wish to attain with a subset of these numbers as

$$k' = k4^{m+1} + \sum_{j=1}^{m} 2 \cdot 4^j,$$

where k is the integer parameter for the VERTEX-COVER instance.

Let us consider how this reduction, which clearly runs in polynomial time, actually works. Suppose graph G has a vertex cover $C = \{i_1, i_2, \ldots, i_k\}$, of size k. Then we can construct a set of values adding to k' by taking every a_i with an index in C, that is, each a_{i_r} for $r = 1, 2, \ldots, k$. In addition, for each edge numbered j in G, if only one of j's endpoints is included in C, then we also include b_j in our subset. This set of numbers sums to k', for it includes k values of 4^{m+1} plus 2 values of each 4^j (either from two a_{i_r}'s such that this edge has both endpoints in C or from one a_{i_r} and one b_j if C contains just one endpoint of edge j).

Suppose there is a subset of numbers suming to k'. Since k' contains k values of 4^{m+1}, it must include exactly k a_i's. Let us include vertex i in our cover for each such a_i. Such a set is a cover, for each edge j, which corresponds to a power 4^j, must contribute two values to this sum. Since only one value can come from a b_j, one must have come from at least one of the chosen a_i's. Thus we have the following:

Theorem 17.13: SUBSET-SUM *is* ***NP****-complete.*

KNAPSACK

In the KNAPSACK problem, illustrated in Figure 17.13, we are given a set S of items, numbered 1 to n. Each item i has an integer size, s_i, and worth, w_i. We are also given two integer parameters, s, and w, and are asked whether there is a subset, T, of S such that

$$\sum_{i \in T} s_i \leq s, \quad \text{and} \quad \sum_{i \in T} w_i \geq w.$$

Problem KNAPSACK defined above is the decision version of the optimization problem "0-1 knapsack" discussed in Section 12.6.

We can motivate the KNAPSACK problem with the following Internet application.

Example 17.14: *Suppose we have s widgets that we are interested in selling at an Internet auction website. A prospective buyer i can bid on multiple lots by saying that he or she is interested in buying s_i widgets at a total price of w_i dollars. If multiple-lot requests, such as this, cannot be broken up (that is, buyer i wants exactly s_i widgets), then determining if we can earn w dollars from this auction gives rise to the KNAPSACK problem. (If lots can be broken up, then our auction optimization problem gives rise to the fractional knapsack problem, which can be solved efficiently using the greedy method of Section 10.1.)*

The KNAPSACK problem is in ***NP***, for we can construct a nondeterministic polynomial-time algorithm that guesses the items to place in our subset T and then verifies that they do not violate the s and w constraints, respectively.

KNAPSACK is also ***NP***-hard, as it actually contains the SUBSET-SUM problem as a special case. In particular, any instance of numbers given for the SUBSET-SUM problem can correspond to the items for an instance of KNAPSACK with each $w_i = s_i$ set to a value in the SUBSET-SUM instance and the targets for the size s and worth w both equal to k, where k is the integer we wish to sum to for the SUBSET-SUM problem. Thus, by the restriction proof technique, we have the following.

Theorem 17.15: KNAPSACK *is **NP**-complete.*

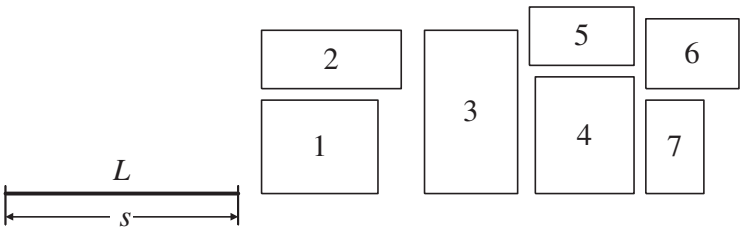


Figure 17.13: A geometric view of the KNAPSACK problem. Given a line L of length s, and a collection of n rectangles, can we translate a subset of the rectangles to have their bottom edge on L so that the total area of the rectangles touching L is at least w? Here, the width of rectangle i is s_i and its area is w_i.

17.6 HAMILTONIAN-CYCLE and TSP

The last two ***NP***-complete problems we consider ask about the existence of certain kinds of cycles in a graph. Such problems are useful for optimizing the travel of robots and circuit-board drills, as discussed at the start of this chapter.

HAMILTONIAN-CYCLE

HAMILTONIAN-CYCLE is the problem that takes a graph G and asks whether there is a cycle in G that visits each vertex in G exactly once, returning to its starting vertex. (See Figure 17.14a.) It is relatively easy to show that HAMILTONIAN-CYCLE is in ***NP***—guess a sequence of vertices and verify that each consecutive pair of vertices in this sequence is connected by an edge and that every vertex (other than the starting and ending vertex) is visited exactly once. To show that this problem is ***NP***-complete, we will reduce VERTEX-COVER to it, using a component-design type of reduction.

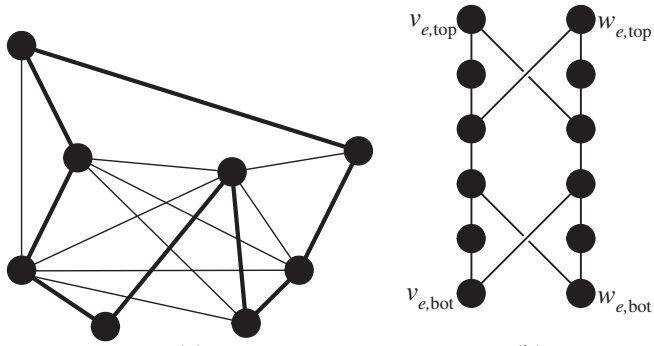


Figure 17.14: Illustrating the HAMILTONIAN-CYCLE problem and its ***NP***-completeness proof. (a) Shows an example graph with a Hamiltonian cycle shown in bold. (b) Illustrates a cover-enforcer subgraph H_e used to show that HAMILTONIAN-CYCLE is ***NP***-hard.

Let G and k be a given instance of the VERTEX-COVER problem. We will construct a graph H that has a Hamiltonian cycle if and only if G has a vertex cover of size k. We begin by including a set of k initially disconnected vertices $X = \{x_1, x_2, \ldots, x_k\}$ to H. This set of vertices will serve as a "cover-choosing" component, for they will serve to identify which nodes of G should be included in a vertex cover. In addition, for each edge $e = (v, w)$ in G we create a "cover-enforcer" subgraph H_e in H. This subgraph H_e has 12 vertices and 14 edges as shown in Figure 17.14b.

Six of the vertices in the cover-enforcer H_e for $e = (v, w)$ correspond to v and the other six correspond to w. Moreover, we label two vertices in cover-enforcer

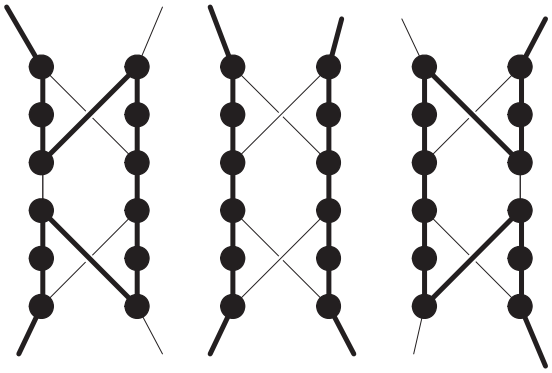


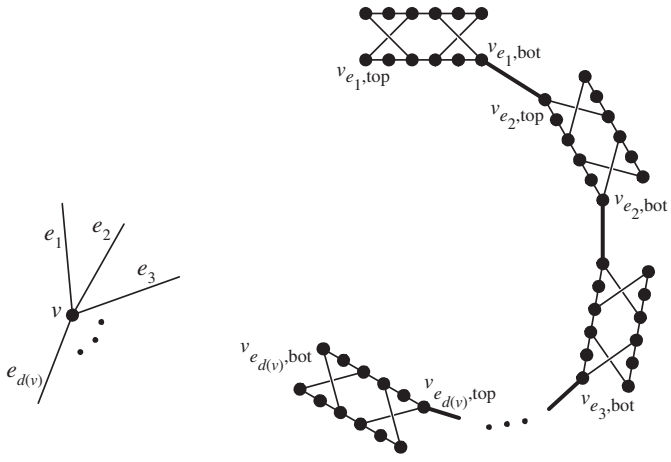
Figure 17.15: The three possible ways that a Hamiltonian cycle can visit the edges in a cover-enforcer H_e.

H_e that correspond to v as $v_{e,\text{top}}$ and $v_{e,\text{bot}}$, and we label two vertices in H_e that correspond to w as $w_{e,\text{top}}$ and $w_{e,\text{bot}}$. These are the only vertices in H_e that will be connected to any other vertices in H outside of H_e. Thus, a Hamiltonian cycle can visit the nodes of H_e in only one of three possible ways, as shown in Figure 17.15.

We join the important vertices in each cover-enforcer H_e to other vertices in H in two ways, one that corresponds to the cover-choosing component and one that corresponds to the cover-enforcing component. For the cover-choosing component, we add an edge from each vertex in X to every vertex $v_{e,\text{top}}$ and every vertex $v_{e,\text{bot}}$. That is, we add $2kn$ edges to H, where n is the number of vertices in G.

For the cover-enforcing component, we consider each vertex v in G in turn. For each such v, let $\{e_1, e_2, \ldots, e_{d(v)}\}$ be a listing of the edges of G that are incident upon v. We use this listing to create edges in H by joining $v_{e_i,\text{bot}}$ in H_{e_i} to $v_{e_{i+1},\text{top}}$ in $H_{e_{i+1}}$, for $i = 1, 2, \ldots, d-1$. (See Figure 17.16.) We refer to the H_{e_i} components joined in this way as belonging to the ***covering thread*** for v. This completes the construction of the graph H. Note that this computation runs in polynomial time in the size of G.

We claim that G has a vertex cover of size k if and only if H has a Hamiltonian cycle. Suppose, first, that G has a vertex cover of size k. Let $C = \{v_{i_1}, v_{i_2}, \ldots, v_{i_k}\}$ be such a cover. We construct a Hamiltonian cycle in H, by connecting a series of paths P_j, where each P_j starts at x_j and ends at x_{j+1}, for $j = 1, 2, \ldots, k-1$, except for the last path P_k, which starts at x_k and ends at x_1. We form such a path P_j as follows. Start with x_j, and then visit the entire covering thread for v_{i_j} in H, returning to x_{j+1} (or x_1 if $j = k$). For each cover-enforcer subgraph H_e in the covering thread for v_{i_j}, which is visited in this P_j, we write, without loss of generality, e as (v_{i_j}, w). If w is not also in C, then we visit this H_e as in Figure 17.15a or Figure 17.15c (with respect to v_{i_j}). Instead, if w is also in C, then we visit this H_e as in Figure 17.15b. In this way we will visit each vertex in H exactly once, since C is a vertex cover for G. Thus, this cycle is in fact a Hamiltonian cycle.



(a) (b)

Figure 17.16: Connecting the cover-enforcers. (a) A vertex v in G and its set of incident edges $\{e_1, e_2, \ldots, e_{d(v)}\}$. (b) The connections made between the H_{e_i}'s in H for the edges incident upon v.

Suppose, conversely, that H has a Hamiltonian cycle. Since this cycle must visit all the vertices in X, we break this cycle up into k paths, P_1, P_2, ..., P_k, each of which starts and ends at a vertex in X. Moreover, by the structure of the cover-enforcer subgraphs H_e and the way that we connected them, each P_j must traverse a portion (possibly all) of a covering thread for a vertex v in G. Let C be the set of all such vertices in G. Since the Hamiltonian cycle must include the vertices from every cover-enforcer H_e and every such subgraph must be traversed in a way that corresponds to one (or both) of e's endpoints, C must be a vertex cover in G.

Therefore, G has a vertex cover of size k if and only if H has a Hamiltonian cycle. This gives us the following.

Theorem 17.16: *H* AMILTONIAN-CYCLE *is **NP**-complete.*

TSP

In the ***traveling salesperson problem***, or TSP, we are given an integer parameter k and a graph G, such that each edge e in G is assigned an integer cost $c(e)$, and we are asked whether there is a cycle in G that visits all the vertices in G (possibly more than once) and has total cost at most k. We have already established that TSP is in ***NP***, in Lemma 17.2. Given this fact, showing that TSP is ***NP***-complete is easy, as the TSP problem contains the HAMILTONIAN-CYCLE problem as a special case. Namely, given an instance G of the HAMILTONIAN-CYCLE problem, we can create an instance of TSP by assigning each edge in G the cost $c(e) = 1$ and setting the integer parameter $k = n$, where n is the number of vertices in G. Therefore, using the restriction form of reduction, we get the following.

Theorem 17.17: TSP *is **NP**-complete.*