Chapter 17

Approximation Algorithms

In Chapter 16, we examined decision problems that appear to be intractable. As we might expect, there are other types of problems that are also intractable. For example, consider the following version of the vertex cover problem (cf. Section 16.5). Instead of being given a target size as input, we are given simply an undirected graph from which we must find a vertex cover of minimum size. Let us call this optimization problem $\text{VCOpt}$. We can easily reduce $\text{VC}$ to $\text{VCOpt}$, though because an optimization problem is not a decision problem, the reduction is not a many-one reduction. However, it is clear that if $\text{VCOpt}$ has a polynomial-time solution, then so does $\text{VC}$. We can therefore conclude that unless $\mathcal{P} = \mathcal{NP}$, $\text{VCOpt}$ cannot be solved in polynomial time.

With hard optimization problems, however, it may not be necessary to obtain an exact solution. In this chapter, we will explore techniques for obtaining approximate solutions to hard optimization problems. We will see that for some problems, we can obtain reasonable approximation algorithms. On the other hand, we will use the theory of $\mathcal{NP}$-completeness to show limitations to these techniques. Before looking at specific problems, however, we must first extend some of the definitions from Chapter 16 to include problems other than decision problems.

17.1 Polynomial Turing Reducibility

In this section, we will extend the definition of $\mathcal{NP}$-hardness to include problems that are not decision problems. As we have already observed, we can reduce $\text{VC}$ to $\text{VCOpt}$ in a way that proves that $\text{VCOpt}$ cannot be solved in polynomial time unless $\mathcal{P} = \mathcal{NP}$; however, this reduction is not
CHAPTER 17. APPROXIMATION ALGORITHMS

a many-one reduction. We will therefore define a new kind of reducibility that will include this kind of reduction.

Suppose we can reduce a problem \( X \) to another problem \( Y \) in such a way that for some polynomial \( p(n) \) and any instance \( x \) of \( X \):

- the time required to obtain a solution for \( x \), excluding any time needed to solve instances of \( Y \), is bounded above by \( p(|x|) \); and

- the values of all variables are bounded above by \( p(|x|) \).

We then say that \( X \) is polynomially Turing reducible to \( Y \), or \( X \leq^{pT} Y \). Note that if \( X \leq^{p} Y \), then clearly \( X \leq^{pT} Y \).

It is easily seen that \( \text{VC} \leq^{pT} \text{VCOPT} \). More generally, consider any minimization problem \( Y \) with objective function \( f \). We can construct a decision problem \( X \) from \( Y \) by adding an additional natural number input, \( k \). \( X \) is simply the set of all pairs \((y, k)\) such that \( y \) is an instance of \( Y \) with a candidate solution \( s \) for which \( f(s) \leq k \). It is easily seen that \( X \leq^{pT} Y \), for if we can find the minimum value of \( f \) for a given instance \( y \), then we can quickly decide whether there is a candidate solution \( s \) for which \( f(s) \leq k \).

We can now extend the notion of \( \mathcal{NP} \)-hardness by saying that \( Y \) is \( \mathcal{NP} \)-hard with respect to Turing reducibility if for every \( X \in \mathcal{NP} \), \( X \leq^{pT} Y \). Note that we have not modified the definition of \( \mathcal{NP} \) — it contains only decision problems. As a result, it makes no sense to extend the definition of \( \mathcal{NP} \)-completeness beyond decision problems. Because \( \mathcal{NP} \)-hardness with respect to Turing reducibility is the natural version of \( \mathcal{NP} \)-hardness to use when discussing optimization problems, we will simply refer to this version as “\( \mathcal{NP} \)-hardness” in this chapter. The following theorem can now be shown in a manner similar to the proof of Theorem 16.2.

**Theorem 17.1** If \( X \leq^{pT} Y \) and there is a deterministic polynomial-time algorithm for solving \( Y \), then there is a deterministic polynomial-time algorithm for solving \( X \).

The above theorem shows that if there is a deterministic polynomial-time algorithm for solving an \( \mathcal{NP} \)-hard problem \( Y \), then there is a deterministic polynomial-time algorithm for deciding every problem in \( \mathcal{NP} \). We therefore have the following corollary, which highlights the importance of the notion of \( \mathcal{NP} \) hardness with respect to Turing reducibility.

**Corollary 17.2** If \( Y \) is \( \mathcal{NP} \)-hard and there is a deterministic polynomial-time algorithm for solving \( Y \), then \( \mathcal{P} = \mathcal{NP} \).
17.2 Knapsack

The first problem we will examine is the 0-1 knapsack problem, as defined in Section 12.4. As is suggested by Exercise 16.18, the associated decision problem is $\mathcal{NP}$-complete; hence, the optimization problem is $\mathcal{NP}$-hard.

Consider the following greedy strategy for filling the knapsack. Suppose we take an item whose ratio of value to weight is maximum. If this item won’t fit, we discard it and solve the remaining problem. Otherwise, we include it in the knapsack and solve the problem that results from removing this item and decreasing the capacity by its weight. We have thus reduced the problem to a smaller instance of itself. Clearly, this strategy results in a set of items whose total weight does not exceed the weight bound. Furthermore, it is not hard to implement this strategy in $O(n \lg n)$ time, where $n$ is the number of items.

Because the problem is $\mathcal{NP}$-hard, we would not expect this greedy strategy to yield an optimal solution in all cases. What we need is a way to measure how good an approximation to an optimal solution it provides. In order to motivate an analysis, let us consider a simple example. Consider the following instance consisting of two items:

- The first item has weight 1 and value 2.
- The second item has weight 10 and value 10.
- The weight bound is 10.

The value-to-weight ratios of the two items are 2 and 1, respectively. The greedy algorithm therefore takes the first item first. Because the second item will no longer fit, the solution provided by the greedy algorithm consists of the first item by itself. The value of this solution is 2. However, it is easily seen that the optimal solution is the second item by itself. This solution has a value of 10.

A common way of measuring the quality of an approximation is to form a ratio with the actual value. Specifically, for a maximization problem, we define the approximation ratio of a given approximation to be the ratio of the optimal value to the approximation. Thus, the approximation ratio for the above example is 5. For a minimization problem, we use the reciprocal of this ratio, so that the approximation ratio is always at least 1. As the approximation ratio approaches 1, the approximation approaches the optimal value.

Note that for a minimization problem, the approximation ratio cannot take a finite value if the optimal value is 0. For this reason, we will restrict
our attention to optimization problems whose optimal solutions always make their objective functions positive. In addition, we will restrict our attention to problems whose objective functions have integer values for all candidate solutions.

We would like to show some fixed upper bound on the approximation ratio of our greedy algorithm. However, we can modify the above example by replacing 10 with an arbitrarily large \( x \) in order to achieve an arbitrarily large approximation ratio of \( x/2 \). Thus, this approximation algorithm can perform arbitrarily poorly.

With a bit more work, however, we can modify this algorithm so that it has a bounded approximation ratio. Specifically, we find \( n \) different packings and take the one with the highest value. For the \( i \)th packing, we take the \( i \)th item first, then apply the greedy strategy to finish the packing. Thus, we expend additional work in making sure that we get started correctly. The algorithm is shown in Figure 17.1. For simplicity, we assume that the items are given in nondecreasing order of value-to-weight ratios, and that no item’s weight exceeds the weight bound. It is easily seen that this algorithm produces a solution in \( \Theta(n^2) \) time. The following theorem shows how well it approximates an optimal solution in the worst case.

**Theorem 17.3** \textsc{KnapsackApprox} yields an approximation ratio of at most 2 on all inputs that satisfy the precondition. Furthermore, for every \( \epsilon \in \mathbb{R}^>0 \), there is some input for which the approximation ratio is at least \( 2 - \epsilon \).

**Proof:** We begin by showing the lower bound. Let \( \epsilon \in \mathbb{R}^>0 \), and without loss of generality, assume \( \epsilon < 1 \). We first define the weight bound as

\[
W = 2 \left\lceil \frac{4}{\epsilon} \right\rceil.
\]

We then construct the following set of three items:

- The first item has weight 1 and value 2.
- The second and third items each have a weight and value of \( W/2 \).

The optimal solution clearly consists of the second and third items. This solution has value \( W \). Each iteration of the outer loop of \textsc{KnapsackApprox} yields a solution containing the first item and one of the other two. The solution returned by this algorithm therefore has a value of \( W/2 + 2 \).
Figure 17.1 An approximation algorithm for the 0-1 knapsack problem

**Precondition:** \( W \) is a positive \( \text{Nat} \), \( n \geq 1 \), and \( w[1..n] \) and \( v[1..n] \) are arrays of positive \( \text{Nats} \) such that for \( 1 \leq i \leq j \leq n \), \( v[i]/w[i] \geq v[j]/w[j] \) and \( w[i] \leq W \).

**Postcondition:** Returns an array \( A[1..n] \) of \( \text{Bools} \) such that if

\[
S = \{i \mid 1 \leq i \leq n, A[i] = \text{true}\},
\]

then

\[
\sum_{i \in S} w[i] \leq W.
\]

**KnapsackApprox** \((W, w[1..n], v[1..n])\)

\[
\text{maxValue} \leftarrow 0
\]

\[
\text{for } i \leftarrow 1 \text{ to } n
\]

\[
A \leftarrow \text{new ARRAY}[1..n]
\]

\[
\text{for } j \leftarrow 1 \text{ to } n
\]

\[
A[j] \leftarrow \text{false}
\]

\[
A[i] \leftarrow \text{true}; \text{value} \leftarrow v[i]; \text{weight} \leftarrow w[i]
\]

\[
\text{for } j \leftarrow 1 \text{ to } n
\]

\[
\text{if } j \neq i \text{ and weight} + w[j] \leq W
\]

\[
\text{weight} \leftarrow \text{weight} + w[j]; \text{value} \leftarrow \text{value} + v[j]; A[j] \leftarrow \text{true}
\]

\[
\text{if } \text{value} > \text{maxValue}
\]

\[
M \leftarrow A
\]

\[
\text{return } M
\]
The approximation ratio is therefore
\[
\frac{W}{\frac{W}{2} + 2} = \frac{2W}{W + 4} = 2 - \frac{8}{W + 4} = 2 - \frac{8}{2(\frac{4}{\epsilon}) + 4} \geq 2 - \frac{8}{8/\epsilon} = 2 - \epsilon.
\]

Now consider an arbitrary input to KnapsackApprox. For a given solution \(X\), let \(V(X)\) denote the value of \(X\). Suppose KnapsackApprox returns a solution \(A\), and let \(S\) be an optimal solution. Let \(i\) be the index of some element with maximum value in \(S\), and consider iteration \(i\) of the outer loop. Let \(A_i\) be the solution chosen by this iteration. We will show \(A_i\) has an approximation ratio of at most 2. Because \(V(A) \geq V(A_i)\), the theorem will follow.

In computing an upper bound \(V(S) - V(A_i)\), we can ignore all items that belong to \(S \cap A_i\). Suppose these common items have a total weight of \(C\). Suppose further that \(V(A_i) < V(S)\). Then the greedy loop must reject at least one element belonging to \(S\). Let item \(k\) be the first element from \(S\) to be rejected by the greedy loop in the construction of \(A_i\). Then the total weight of all items in \(A_i \setminus S\) chosen prior to item \(k\) is greater than \(W - C - w[k]\). Because their value-to-weight ratios are all at least \(v[k]/w[k]\), their total value is greater than
\[
\frac{v[k](W - C - w[k])}{w[k]}.
\]

The items in \(S \setminus A_i\) must have total weight at most \(W - C\). Furthermore, all of their value-to-weight ratios are at most \(v[k]/w[k]\); hence their total value is at most
\[
\frac{v[k](W - C)}{w[k]}.
\]

We therefore have
\[
V(S) - V(A_i) < \frac{v[k](W - C)}{w[k]} - \frac{v[k](W - C - w[k])}{w[k]} = v[k].
\]
Because items $i$ and $k$ both belong to $S$ and $v[i] \geq v[k]$, $v[k] \leq V(S)/2$. We therefore have

\[
V(S) - V(A_i) < V(S)/2 \\
V(S) < 2V(A_i) \\
V(S)/V(A_i) < 2 \\
V(S)/V(A) < 2.
\]

$\square$

Though we have a bounded approximation ratio, an approximation ratio of 2 may seem unsatisfactory, as in the worst case we may only achieve half the actual maximum value. It turns out that we can improve the approximation ratio by examining all pairs of items, then using the greedy algorithm to complete each of these packings. More generally, we can achieve an upper bound of $1 + \frac{1}{k}$ by examining all sets of $k$ items and completing each packing using the greedy algorithm. (If there are fewer than $k$ items, we simply do an exhaustive search and return the optimal solution.) The proof is a straightforward generalization of the proof of Theorem 17.3 — the details are left as an exercise.

It is not hard to see that the algorithm outlined above can be implemented to return a solution in $\Theta(n^{k+1})$ time. If $k$ is a fixed constant, the running time is polynomial. We therefore have an infinite sequence of algorithms, each of which is polynomial, such that if an approximation ratio of $1 + \epsilon$ is needed (for some positive $\epsilon$), then one of these algorithms will provide such an approximation. Such a sequence of algorithms is called a polynomial approximation scheme.

Although each of the algorithms in the above sequence is polynomial in the length of the input, it is somewhat unsatisfying that to achieve an approximation ratio of $1 + \frac{1}{k}$, a running time in $\Theta(n^{k+1})$ is required. We would be more satisfied with a running time that is polynomial in both $n$ and $k$. More generally, suppose we have an approximation algorithm that takes as an extra input a natural number $k$ such that for any fixed $k$, the algorithm yields an approximation ratio of no more than $1 + \frac{1}{k}$. Suppose further that this algorithm runs in a time polynomial in $k$ and the length of its input. We call such an algorithm a fully polynomial approximation scheme.

We can obtain a fully polynomial approximation scheme for the 0-1 knapsack problem using one of the dynamic programming algorithms suggested in Section 12.4. The algorithm based on recurrence (12.5) on page 403 runs
in $\Theta(nV)$ time, where $n$ is the number of items and $V$ is the sum of their values. We can make $V$ as small as we wish by replacing each value $v$ by $\lfloor v/d \rfloor$ for some positive integer $d$. If some of the values become 0, we remove these items. Observe that because we don’t change any weights or the weight bound, any packing for the new instance is a packing for the original. However, because we take the floor of each $v/d$, the optimal packing for the new instance might not be optimal for the original. The smaller we make $d$, the better our approximation, but the less efficient our dynamic programming algorithm.

In order to determine an appropriate value for $d$, we need to analyze the approximation ratio of this approximation algorithm. Let $S$ be some optimal set of items. The optimal value is then

$$V^* = \sum_{i \in S} v_i.$$  

With the modified values, this packing has a value of

$$\sum_{i \in S} \left\lfloor \frac{v_i}{d} \right\rfloor \geq \sum_{i \in S} \frac{v_i - d}{d}$$

$$= \sum_{i \in S} \frac{v_i}{d} - \sum_{i \in S} 1$$

$$\geq \frac{V^*}{d} - n.$$  

If we remove from $S$ the items whose new values are 0, we obtain a packing for the revised instance with same value as above. Because the dynamic programming algorithm selects an optimal packing for the revised instance, it will yield a packing with a value at least this large. If we substitute the original values into the packing chosen by the dynamic programming algorithm, we obtain a value of at least $V^* - nd$. The approximation ratio is therefore at most

$$\frac{V^*}{V^* - nd}.$$  

We need to ensure that the approximation ratio is at most $1 + \frac{1}{k}$ for some
positive integer \( k \). We therefore need

\[
\frac{V^*}{V^* - nd} \leq 1 + \frac{1}{k}
\]

\[
V^* \leq V^* - nd + \frac{V^* - nd}{k}
\]

\[
0 \leq \frac{V^* - (k+1)nd}{k}
\]

\[
0 \leq V^* - (k+1)nd
\]

\[
d \leq \frac{V^*}{(k+1)n}.
\]

Let \( v \) be the largest value of any item in the original instance. Assuming that no item’s weight exceeds the weight bound, we can conclude that \( V^* \geq v \). Thus, if \( v \geq (k+1)n \), we can satisfy the above inequality by setting \( d \) to \( \lfloor v/(k+1)n \rfloor \). However, if \( v < (k+1)n \), this value becomes 0. In this case, we can certainly set \( d \) to 1, as the dynamic programming algorithm would then give the optimal solution. We therefore set

\[
d = \max \left( \left\lfloor \frac{v}{(k+1)n} \right\rfloor, 1 \right).
\]

We can clearly compute the scaled values in \( O(n) \) time. If \( v \geq 2(k+1)n \), the sum of the scaled values is no more than

\[
\frac{nv}{d} = \frac{nv}{\left\lfloor \frac{v}{(k+1)n} \right\rfloor}
\]

\[
\leq \frac{nv}{v-(k+1)n}
\]

\[
= \frac{(k+1)n^2v}{v-(k+1)n}
\]

\[
\leq \frac{(k+1)n^2v}{v/2}
\]

\[
= 2(k+1)n^2.
\]

In this case, the dynamic programming algorithm runs in \( O(kn^3) \) time.

If \( v < 2(k+1)n \), then \( d = 1 \), so that we use the original values. In this case, the sum of the values is no more than

\[
nv < 2(k+1)n^2,
\]
so that again, the dynamic programming algorithm runs in $O(kn^3)$ time. Thus, the total running time of the approximation algorithm is in $O(kn^3)$. Because this running time is polynomial in $k$ and $n$, and because the approximation ratio is no more than $1 + \frac{1}{k}$, this algorithm is a fully polynomial approximation scheme.

17.3 Bin Packing

Exercise 16.27 introduced the bin packing problem as a decision problem. Its input consists of a set of items, each having a positive integer weight $w_i$, a positive integer weight bound $W$, and a positive integer $k$. The question we ask is whether the items can be partitioned into $k$ disjoint subsets, each having a total weight of no more than $W$. The corresponding optimization problem does not include the input $k$, but instead asks for the minimum number of subsets into which the items can be partitioned such that the weight bound is satisfied. As is suggested by Exercise 16.27, the decision problem BP is strongly $NP$-complete. As a result, it is easily seen that the optimization problem is $NP$-hard in the strong sense.

Ideally, we would like to have a fully polynomial approximation scheme for bin packing. However, the following theorem tells us that unless $P = NP$, a fully polynomial approximation scheme does not exist.

**Theorem 17.4** Let $p(x, y)$ be an integer-valued polynomial, and let $X$ be an optimization problem whose optimal value on any input $x$ is a natural number bounded above by $p(|x|, \mu(x))$. If there is a fully polynomial approximation scheme for $X$, then there is a pseudopolynomial algorithm for obtaining an optimal solution for $X$.

**Proof:** The pseudopolynomial algorithm operates as follows. Given an input $x$, it first computes $k = p(|x|, \mu(x))$. It then uses the fully polynomial approximation scheme to approximate a solution with an approximation ratio bounded by $1 + \frac{1}{k}$. Let $V$ be the value of the approximation, and let $V^*$ be the value of an optimal solution. If the problem is a minimization

\[ |x| \] denotes the number of bits in the encoding of $x$ and $\mu(x)$ denotes the maximum value of any integer encoded within $x$.\n
Recall that $|x|$
problem, we have
\[
\frac{V}{V^*} \leq 1 + \frac{1}{k}
\]
\[
V \leq V^* + \frac{V^*}{k}
\]
\[
V - V^* \leq \frac{V^*}{k}
\]
< 1.
Because both \( V \) and \( V^* \) are natural numbers and \( V \geq V^* \), we conclude that \( V = V^* \). Furthermore, because the fully polynomial approximation scheme runs in time polynomial in \( |x| \) and \( p(|x|, \mu(x)) \), it is a pseudopolynomial algorithm.

An analogous argument applies to maximization problems. \( \square \)

Because the minimum number of bins needed is clearly no more than the length of the input to the bin packing problem, Theorem 17.4 applies to this problem. Indeed, the condition that the optimal solution is bounded by a polynomial in the length of the input and the largest integer in the input holds for most optimization problems. In these cases, if the given problem is strongly \( \mathcal{NP} \)-hard (as is bin packing), there can be no fully polynomial approximation scheme unless \( \mathcal{P} = \mathcal{NP} \).

If we cannot obtain a fully polynomial approximation scheme for bin packing, we might still hope to find a polynomial approximation scheme. However, the theory of \( \mathcal{NP} \)-hardness tells us that this is also unlikely. In particular, for a fixed positive integer \( k \), let \( k \)-BP denote the problem of deciding whether, for a given instance of bin packing, there is a solution using at most \( k \) bins. It is easily seen that \( \text{PART} \leq^p_m 2\text{-BP} \), so that \( 2\text{-BP} \) is \( \mathcal{NP} \)-hard. Now for a fixed positive real number \( \epsilon \), let \( \epsilon \text{-APPROXBP} \) be the problem of approximating a solution to a given instance of bin packing with an approximation ratio of no more than \( 1 + \epsilon \). We will now show that for any \( \epsilon < 1/2 \), \( 2\text{-BP} \leq^p_T \epsilon \text{-APPROXBP} \), so that \( \epsilon \text{-APPROXBP} \) is \( \mathcal{NP} \)-hard. As a result, there can be no polynomial approximation scheme for bin packing unless \( \mathcal{P} = \mathcal{NP} \).

**Theorem 17.5** For \( 0 < \epsilon < 1/2 \), \( \epsilon \text{-APPROXBP} \) is \( \mathcal{NP} \)-hard.

**Proof:** As we noted above, we will show that \( 2\text{-BP} \leq^p_T \epsilon \text{-APPROXBP} \). Given an instance of \( 2\text{-BP} \), we first find an approximate solution with approximation ratio at most \( \epsilon \). If the approximate solution uses no more than
2 bins, then we can answer “yes”. If the approximate solution uses 3 or more bins, then the optimal solution uses at least

\[ \frac{3}{1 + \epsilon} > \frac{3}{3/2} \]

\[ = 2 \]

bins. We can therefore answer “no”.

Ignoring the time needed to compute the approximation, this algorithm runs in \( \Theta(1) \) time. Therefore, \( 2\text{-BP} \leq^P \epsilon\text{-APPROXBP} \), and \( \epsilon\text{-APPROXBP} \) is \( \mathcal{NP} \)-hard. \( \square \)

From Theorem 17.5, we can conclude that there is no approximation algorithm for bin packing with approximation ratio less than \( 3/2 \) unless \( \mathcal{P} = \mathcal{NP} \). As a result, there can be no polynomial approximation scheme for bin packing unless \( \mathcal{P} = \mathcal{NP} \).

On the other hand, there do exist approximation algorithms which yield approximation ratios that come close to the lower bound of \( 3/2 \) for bin packing. The algorithm we will present here is a simple greedy strategy known as first fit. For each item, we try each bin in turn to see if the item will fit. If we find a bin in which the item fits, we place it in that bin; otherwise, we place it in a new bin. The algorithm is shown in Figure 17.2. This algorithm is easily seen to run in \( \Theta(n^5) \) time in the worst case. We will now show that it yields an approximation ratio of at most 2.

**Theorem 17.6** BinPackingFF yields an approximation ratio of no more than 2 on all inputs that satisfy the precondition.

**Proof:** We will first show as an invariant of the for loop that at most one bin is no more than half full. This clearly holds initially. Suppose it holds at the beginning of some iteration. If \( w[i] > W/2 \), then no matter where \( w[i] \) is placed, it cannot increase the number of bins that are no more than half full. Suppose \( w[i] \leq W/2 \). Then if there is a bin that is no more than half full, \( w[i] \) will fit into this bin. Thus, the only case in which the number of bins that are no more than half full increases is if there are no bins that are no more than half full. In this case, the number cannot be increased to more than one.

We conclude that the packing returned by this algorithm has at most one bin that is no more than half full. Suppose this packing consists of \( k \) bins. The total weight must therefore be strictly larger than \( (k - 1)W/2 \).
**Precondition:** \( W \) is a positive \( \text{Nat} \), and \( w[1..n] \) is an array of positive \( \text{Nats} \) such that for \( 1 \leq i \leq n \), \( w[i] \leq W \).

**Postcondition:** Returns an array \( B[1..k] \) of \( \text{ConsLists} \) of \( \text{Nats} \) \( i \) such that \( 1 \leq i \leq n \). For \( 1 \leq i \leq n \), \( i \) occurs in exactly one \( \text{ConsList} \) in \( B[1..k] \). For \( 1 \leq i \leq k \), if \( S \) is the set of integers in \( B[i] \), then

\[
\sum_{j \in S} w_j \leq W.
\]

**BinPackingFF** (\( W, w[1..n] \))

\[
B \leftarrow \text{new Array}[1..n]; \ slack \leftarrow \text{new Array}[1..n]; \ numBins \leftarrow 0
\]

for \( i \leftarrow 1 \) to \( n \)

\[
j \leftarrow 1
\]

while \( j \leq \text{numBins} \) and \( w[i] > \text{slack}[j] \)

\[
j \leftarrow j + 1
\]

if \( j > \text{numBins} \)

\[
\text{numBins} \leftarrow \text{numBins} + 1; \ B[j] \leftarrow \text{new ConsList}(); \ slack[j] \leftarrow W
\]

\[
B[j] \leftarrow \text{new ConsList}(i, \ B[j]); \ slack[j] \leftarrow \text{slack}[j] - w[i]
\]

return \( B[1..\text{numBins}] \)

The optimal packing must therefore contain more than \( (k - 1)/2 \) bins. Thus, the number of bins in the optimal packing is at least

\[
\left\lceil \frac{k - 1}{2} \right\rceil + 1 = \left\lceil \frac{k + 1}{2} \right\rceil \geq k/2.
\]

The approximation ratio is therefore at most 2. \( \square \)

It can be shown via a much more complicated argument that if the optimal packing uses \( B^* \) bins, then **BinPackingFF** gives a packing using no more than \( \left\lceil \frac{17}{10} B^* \right\rceil \) bins. Thus, as \( B^* \) increases, the upper bound on the approximation ratio approaches 17/10. If we first sort the items by nonincreasing weight, it can be shown that this strategy (known as **first-fit decreasing**) gives a packing using no more than \( \frac{11}{9} B^* + 4 \) bins. Note that although this upper bound is less than \( 3/2 \) as \( B^* \) increases, this does
not give a polynomial-time algorithm for $\epsilon$-ApproxBP for any $\epsilon < 3/2$, as the proof of Theorem 17.5 essentially shows the hardness of deciding whether $B^* = 2$. Furthermore, Theorem 17.5 does not preclude the existence of a pseudopolynomial algorithm with an approximation ratio bounded by some value less than 3/2. We leave it as an exercise to show that dynamic programming can be combined with the first-fit decreasing strategy to yield, for any positive $\epsilon$, an approximation algorithm with an approximation ratio bounded by $\frac{11}{9} + \epsilon$.

17.4 The Traveling Salesperson Problem

Exercise 16.28 introduced the traveling salesperson problem as a decision problem, TSP. Its input consists of a complete undirected graph $G$ with positive integer edge weights and a positive integer $k$. The question we ask is whether there is a Hamiltonian cycle in $G$ with total weight no more than $k$. As is suggested by Exercise 16.28, TSP is strongly $\mathcal{NP}$-complete. The corresponding optimization problem does not include the input $k$, but instead asks for the Hamiltonian cycle in $G$ with minimum weight. It is easily seen that this problem is $\mathcal{NP}$-hard in the strong sense. Clearly, a minimum weight Hamiltonian cycle has weight no more than $nW$, where $n$ is the number of vertices in $G$ and $W$ is the maximum weight of any edge in $G$; hence, by Theorem 17.4, there can be no fully polynomial approximation scheme for the optimization problem unless $P = \mathcal{NP}$.

For $\epsilon > 0$, let $\epsilon$-ApproxTSP be the problem of finding, for a given undirected graph $G$ with positive integer edge weights, a Hamiltonian cycle with approximation ratio no more than $1 + \epsilon$. In what follows, we will show that $\epsilon$-ApproxTSP is $\mathcal{NP}$-hard in the strong sense for every positive $\epsilon$. As a result, there can be no polynomial or pseudopolynomial algorithm for finding an approximation with any bounded approximation ratio unless $P = \mathcal{NP}$.

**Theorem 17.7** For every positive $\epsilon$, $\epsilon$-ApproxTSP is $\mathcal{NP}$-hard in the strong sense.

**Proof:** Let $\epsilon > 0$, and let HC be the problem of deciding whether a given undirected graph $G$ contains a Hamiltonian cycle. By Exercise 16.9, HC is $\mathcal{NP}$-complete. Since there are no integers in the problem instance, it is strongly $\mathcal{NP}$-complete. We will show that HC $\leq_{PP}^{P}$ $\epsilon$-ApproxTSP, where $\leq_{PP}^{P}$ denotes a pseudopolynomial Turing reduction. It will then follow that $\epsilon$-ApproxTSP is $\mathcal{NP}$-hard in the strong sense.
Let $G = (V, E)$ be an undirected graph. We first construct a complete undirected graph $G' = (V, E')$. Let $k = \lfloor \epsilon \rfloor + 2$. We define the weight of an edge $e \in E'$ as follows:

- If $e \in E$, then the weight of $e$ is 1.
- If $e \notin E$, then the weight of $e$ is $nk$, where $n$ is the size of $V$.

Note that because $k$ is a fixed constant, the weights are bounded by a polynomial in the size of $G$.

We now show how we can use an approximation of a minimum-weight Hamiltonian cycle in $G'$ to decide whether $G$ has a Hamilton cycle. Suppose we can obtain an approximation with an approximation ratio of no more than $1 + \epsilon$. If the weight of this approximation is $n$, then the corresponding Hamiltonian cycle must contain only edges with weight 1; hence, it is a Hamiltonian cycle in $G$, so we can answer “yes”. Otherwise, the approximation contains at least one edge with weight $nk$, and $n > 0$. The weight of the approximation is therefore at least $nk + n - 1$. Because the approximation ratio is no more than $1 + \epsilon$, the minimum-weight Hamiltonian path has a weight of at least

$$\frac{nk + n - 1}{1 + \epsilon} = \frac{n(|\epsilon| + 2) + n - 1}{1 + \epsilon} > \frac{n(1 + \epsilon)}{1 + \epsilon} = n.$$

Hence, there is no Hamiltonian cycle whose edge weights are all 1. Because this implies that $G$ contains no Hamiltonian cycle, we can answer “no”.

The running time for this algorithm, excluding any time needed to compute the approximation, is linear in the size of $G$. Furthermore, all integers constructed have values polynomial in the size of $G$. We therefore conclude that $\epsilon$-APPROX-TSP is NP-hard in the strong sense.

As a result of Theorem 17.7, we have little hope of finding a polynomial-time approximation algorithm yielding a bounded approximation ratio for the traveling salesman problem. However, if we make a certain restriction to the problem, we can find such an algorithm. The metric traveling salesperson problem is the restriction of the traveling salesperson problem to inputs in which the edges of the graph satisfy the triangle inequality; i.e., if $u$, $v$, and $w$ are vertices, then

$$\text{weight} \{u, w\} \leq \text{weight} \{u, v\} + \text{weight} \{v, w\}.$$
The triangle inequality is satisfied, for example, if the vertices represent points in the plane, and the edge weights represent distances. In what follows, we will present a polynomial-time approximation algorithm yielding an approximation ratio bounded by 2 for this problem.

We first observe that if we remove any edge from a Hamiltonian cycle, we obtain a spanning tree of the graph. Furthermore, the weight of this spanning tree must be less than the weight of the Hamiltonian cycle. Hence, an MST will have a weight strictly less than the weight of a minimum-weight Hamiltonian cycle. Now consider a tour of an MST that follows a depth-first search — that is, we go from vertex $u$ to vertex $v$ when the call on $u$ makes a call on $v$, and we go from $v$ to $u$ when the call on $v$ returns. In this way, we traverse each edge exactly twice and reach each vertex at least once, returning to the vertex from which we started. Clearly, the weight of the edges in this tour (counting each edge exactly twice) is less than twice the weight of a minimum-weight Hamiltonian cycle.

We now wish to convert this tour to a Hamiltonian cycle by taking shortcuts. Specifically, when the tour would return to a vertex that it has already reached, we skip ahead to the next vertex in the tour that has not yet been reached (see Figure 17.3). When we have reached all vertices, we return to the starting point.

It is easily seen by induction that if the triangle inequality is satisfied, then the weight of edge \{u, v\} is no more than the sum of the weights of the edges on any simple path from $u$ to $v$. It is easily seen that when a path from $u$ to $v$ is replaced by edge \{u, v\} in the above conversion, that path is a simple path, because all edges in the tour that reach vertices that have already been reached must go from children to parents; hence, the path in the tour from $u$ to $v$ takes edges from children to parents, followed by a single edge from a parent to $v$, which is reached for the first time in the tour. Clearly, no vertex can be repeated in such a path. As a result, the weight of this Hamiltonian cycle is less than twice the weight of an optimal Hamiltonian cycle.

Notice that because this Hamiltonian cycle reaches the vertices in the same order that they are first reached in the depth-first search, the vertices are ordered by their preorder traversal numbers. Therefore, it is easy to construct this Hamiltonian cycle while doing the depth-first search on the MST. A Searcher for the depth-first search needs only a VisitCounter pre for recording the preorder traversal numbers and a readable array order[0..n−1] such that the Hamiltonian cycle will be \{order[0], order[1],..., order[n−1], order[0]\}. Such a Searcher is defined in Figure 17.4.

For constructing an MST, we can use either Kruskal’s algorithm (Figure
**Figure 17.3** Conversion of an MST to a Hamiltonian cycle

![Conversion of an MST to a Hamiltonian cycle](image)

**Figure 17.4** An implementation of Searcher for use in the metric traveling salesman approximation algorithm

\[
\text{MetricTspSearcher}(n) \quad \text{pre} \leftarrow \text{new VisitCounter}(n); \text{order} \leftarrow \text{new Array}[0..n-1]
\]

\[
\text{MetricTspSearcher.PrepProc}(i) \quad \text{pre.Visit}(i); \text{order[pre.Num(i)]} \leftarrow i
\]
Precondition:  \( G \) is a Graph representing a complete undirected graph with at least one vertex, whose edges contain positive \( \text{Nat} \) weights satisfying the triangle inequality.

Postcondition:  Returns an array \( \text{order}[0..n - 1] \) in which each \( \text{Nat} \) less than \( n \) occurs exactly once. The sum of the weights of the edges \( \{\text{order}[0], \text{order}[1]\}, \{\text{order}[1], \text{order}[2]\}, \ldots, \{\text{order}[n - 1], \text{order}[0]\} \) is less than twice the weight of an optimal Hamiltonian cycle.

\[
\text{MetricTsp}(G)
\]
\[
\begin{align*}
n &\leftarrow G.\text{Size}; L \leftarrow \text{Prim}(G) \\
G' &\leftarrow \text{new ListMultigraph}(n) \\
\text{while not } L.\text{isEmpty}() \\
&\begin{align*}
e &\leftarrow L.\text{head}(); L \leftarrow L.\text{tail}() \\
i &\leftarrow e.\text{source}(); j \leftarrow e.\text{destination}(); x &\leftarrow e.\text{data}() \\
G'.\text{put}(i, j, x); G'.\text{put}(j, i, x)
\end{align*}
\end{align*}
\]
\[
G'' &\leftarrow \text{new ListGraph}(G'); \text{sel} \leftarrow \text{new Selector}(n) \\
s &\leftarrow \text{new MetricTspSearcher}(n); \text{Dfs}(G'', 0, \text{sel}, s) \\
\text{return } s.\text{order}()
\]

Assuming \( G \) is a \text{MatrixGraph}, the call to \text{Prim} runs in \( \Theta(n^2) \) time. The \text{ListMultigraph} constructor then runs in \( \Theta(n) \) time. Because the \text{ConsList} returned by \text{Prim} contains exactly \( n - 1 \) edges, and the \text{ListMultigraph.put} operation runs in \( \Theta(1) \) time, the loop runs in \( \Theta(n) \) time. As was shown in Section 9.5, the \text{ListGraph} constructor runs in \( \Theta(n) \) time. The \text{Selector} constructor runs in \( \Theta(n) \) time, and the \text{MetricTspSearcher} constructor clearly runs in \( \Theta(1) \) time. Because the \text{MetricTsp}-
Searcher.PreProc operation runs in \( \Theta(1) \) time and \( G'' \) is a ListGraph, with \( n - 1 \) edges, the call to Dfs runs in \( \Theta(n) \) time. The total running time is therefore in \( \Theta(n^2) \).

### 17.5 The Maximum Cut and Minimum Cluster Problems

We conclude this chapter by examining two optimization problems that are essentially the same, but which yield entirely different results with respect to approximation algorithms. Let \( G = (V, E) \) be a complete undirected graph with positive integer edge weights. For a natural number \( k \geq 2 \), a \( k \)-cut for \( G \) is a partition of \( V \) into \( k \) disjoint sets, \( S_1, S_2, \ldots, S_k \). The weight of this cut is the sum of the weights of all edges \( \{u, v\} \in E \) such that \( u \) and \( v \) are in different partitions. The **maximum cut problem** is to find, for a given natural number \( k \geq 2 \) and complete undirected graph \( G = (V, E) \) with edge weights and more than \( k \) vertices, a \( k \)-cut with maximum weight. The **minimum cluster problem** is to find a \( k \)-cut that minimizes the sum of the weights of all edges \( \{u, v\} \in E \) such that \( u \) and \( v \) are in the same partition. Clearly, a \( k \)-cut has maximum weight iff it minimizes this latter sum.

Let \( \text{Cut} \) be the problem of deciding, for given natural numbers \( k \geq 2 \) and \( B \), and complete undirected graph \( G = (V, E) \) with positive integer edge weights and more than \( k \) vertices, whether there is a \( k \)-cut with weight at least \( B \). For each \( k \geq 2 \), we also define the \( k \)-\text{Cut} problem to be the \text{Cut} problem restricted to cuts of exactly \( k \) sets (i.e., \( k \) is not given as input, but is fixed). We further define the \text{Cluster} and \( k \)-\text{Cluster} problems analogously. We leave as exercises to show that \( k \)-\text{Cluster} is strongly \( \mathcal{NP} \)-complete for every \( k \geq 2 \), and that \text{Cluster} is strongly \( \mathcal{NP} \)-complete. It then follows that \text{Cut} and \( k \)-\text{Cut} for \( k \geq 2 \) are all strongly \( \mathcal{NP} \)-complete.

Given the above results, the problems of finding either a maximum cut or a minimum cluster are \( \mathcal{NP} \)-hard in the strong sense. Thus, from Theorem 17.4, there is no fully polynomial approximation scheme for either of these problems unless \( \mathcal{P} = \mathcal{NP} \). However, there is a simple greedy strategy that yields good approximation ratios for maximum cut. We begin with \( k \) empty sets, and add vertices one by one to the set that gives us the largest cut. The algorithm is shown in Figure 17.6. Note that this algorithm actually uses the amount by which the total weight of the clusters would increase when choosing the set in which to place a given vertex — these values are easier to compute than the weights of the resulting cuts. It is easily seen that this algorithm runs in \( \Theta(n^2) \) time, if \( G \) is represented by a MatrixGraph. The
Precondition: \( G \) is a \texttt{Graph} representing a complete undirected graph with positive \texttt{Nat} edge weights, and \( k \) is a \texttt{Nat} such that \( 2 \leq k < n \), where \( n \) is the number of vertices in \( G \).

Postcondition: Returns an array \( \text{cut}[0..n-1] \) such that for \( 0 \leq i < n \), \( 1 \leq \text{cut}[i] \leq k \). If \( W \) is the sum of the weights of the edges in \( G \), then the weight of the cut described by \( \text{cut}[0..n-1] \) is at least \( W(k-1)/k \).

\[
\text{MaxCut}(G,k)
\]
\[
n \leftarrow G.\text{Size}(); \text{cut} \leftarrow \text{new Array}[0..n-1]
\]
\[
\text{clusterInc} \leftarrow \text{new Array}[1..k]
\]
\[
\text{for } i \leftarrow 0 \text{ to } n-1
\]
\[
\text{for } j \leftarrow 1 \text{ to } k
\]
\[
\text{clusterInc}[j] \leftarrow 0
\]
\[
\text{for } j \leftarrow 0 \text{ to } i-1
\]
\[
\text{clusterInc}[	ext{cut}[j]] \leftarrow \text{clusterInc}[	ext{cut}[j]] + G.\text{Get}(i,j)
\]
\[
m \leftarrow 1
\]
\[
\text{for } j \leftarrow 2 \text{ to } k
\]
\[
\text{if } \text{clusterInc}[j] < \text{clusterInc}[m]
\]
\[
m \leftarrow j
\]
\[
\text{cut}[i] \leftarrow m
\]
\[
\text{return } \text{cut}[0..n-1]
\]

following theorem gives bounds for its approximation ratio.

**Theorem 17.8** For each \( k \geq 2 \), \text{MAXCut} yields an approximation ratio of no more than

\[
1 + \frac{1}{k-1},
\]

thus, the approximation ratio is never more than 2.

**Proof:** For a given vertex \( i \), let \( W_i \) denote the sum of the weights of all edges \( \{i,j\} \) such that \( 0 \leq j < i \). At the end of iteration \( i \), the value of the cut increases by \( W_i - \text{clusterInc}[m] \), where \( \text{clusterInc}[m] \) is the sum of the weights of the edges from \( i \) to other vertices in partition \( m \). \( m \) is chosen so that \( \text{clusterInc}[m] \) is minimized; hence, for each partition other than \( m \), the sum of the weights of the edges from \( i \) to vertices in that partition is at
least \( \text{clusterInc}[m] \). We therefore have

\[
\text{clusterInc}[m] \leq W_i/k.
\]

The value of the cut therefore increases by at least \( W_i(k - 1)/k \) on iteration \( i \). Because the value of the cut is initially 0, the final value of the cut is at least

\[
\sum_{i=0}^{n-1} \frac{W_i(k - 1)}{k} = \frac{k - 1}{k} \sum_{i=0}^{n-1} W_i
\]

\[
= \frac{k - 1}{k} W_i,
\]

where \( W \) is the sum of all edge weights in \( G \). Clearly, the maximum cut can be no more than \( W \). The approximation ratio is therefore bounded above by

\[
\frac{W}{(k-1)W/k} = \frac{k}{k - 1}
\]

\[
= 1 + \frac{1}{k-1}.
\]

\[\square\]

Though the algorithm MaxCut yields a fixed bound on the approximation ratio for approximating a maximum cut, it is perhaps surprising, that the same algorithm yields unbounded approximation ratios for approximating a minimum cluster, even though the two optimization problems are essentially the same. We can see why this is the case by examining instances that cause the approximation ratio for MaxCut to approach the upper bound shown in Theorem 17.8.

For \( k \geq 2 \), consider a complete undirected graph \( G = (V, E) \), where \( V = \{i \in \mathbb{N} \mid i < k^2\} \). We partition \( V \) into \( k \) groups such that for \( 0 \leq j < k \), group \( j \) is the set \( \{i \in \mathbb{N} \mid jk \leq i < (j + 1)k\} \); thus, each group contains \( k \) vertices. We now assign weights to the edges in \( G \) such that if vertices \( u \) and \( v \) are in the same group, then \( \{u, v\} \) has weight 1; otherwise, \( \{u, v\} \) has weight \( x \), where \( x \) is some sufficiently large natural number. (See Figure 17.7 for the case in which \( k = 2 \).)

We claim that if \( x \geq k^3 \), the maximum cut for \( G \) partitions the vertices so that each group forms a cluster. To see this, first note that when we partition \( G \) in this way, the resulting cut includes exactly those edges with
weight $x$. Because the sum of all other edges is less than $x$, the maximum cut must contain all of the edges with weight $x$. Note that connecting any two of the vertices $0, k, \ldots, k^2 - k$ is an edge with weight $x$; hence, a maximum cut must place all of these $k$ vertices into different clusters. Then for $0 \leq i < k$ and $1 \leq j < k$, there is an edge of weight $x$ between vertex $ik + j$ and $i'k$ for each $i' \neq i$. As a result, vertex $ik + j$ must be placed in the same cluster as vertex $ik$. Hence, the maximum cut partitions the vertices so that each group forms a cluster.

Now consider the behavior of MaxCut on $G$. It will first place vertices $0, 1, \ldots, k - 1$ into different clusters. Then vertex $k$ is adjacent to exactly one vertex in each of the clusters via an edge with weight $x$. Because placing $k$ in any of the clusters would increase the weight of that cluster by $x$, $k$ is placed in the first cluster with vertex 0. Placing $k + 1$ in this cluster would increase its weight by $x + 1$; however, placing $k + 1$ in any other cluster would increase that cluster’s weight by only $x$. As a result, $k + 1$ is placed in the second cluster with vertex 1. It is easily seen that the algorithm continues by placing vertices into clusters in round-robin fashion, so that each cluster ultimately contains exactly one vertex from each group.

We can use symmetry to help us to evaluate the approximation ratio of MaxCut on $G$. In the maximum cut, each vertex is adjacent to $(k - 1)k$
vertices in other clusters via edges whose weights are all $x$. In the cut produced by \textsc{MaxCut}, each vertex is also adjacent to $(k - 1)k$ vertices in other clusters; however, only $(k - 1)^2$ of these edges have weight $x$, while the remaining edges each have weight 1. The approximation ratio is therefore

$$\frac{(k - 1)kx}{(k - 1)^2x + k - 1} = \frac{kx}{(k - 1)x + 1},$$

which approaches

$$\frac{k}{k - 1} = 1 + \frac{1}{k - 1},$$

as $x$ approaches $\infty$. The bound of Theorem 17.8 is therefore tight.

Let us now analyze the approximation ratio for \textsc{MaxCut} as an approximation algorithm for the minimum cluster problem. Again we can use symmetry to simplify the analysis. In the optimal solution, each vertex is adjacent to $k - 1$ vertices in the same cluster via edges whose weights are all 1. In the solution given by \textsc{MaxCut}, each vertex is adjacent to $k - 1$ vertices in the same cluster via edges whose weights are all $x$. Thus, the approximation ratio is $x$, which can be chosen to be arbitrarily large. We can therefore see that even though the maximum cut and minimum cluster optimization problems are essentially the same, the \textsc{MaxCut} algorithm yields vastly different approximation ratios relative to the two problems.

To carry this idea a step further, we will now show that the minimum cluster problem has no approximation algorithm with a bounded approximation ratio unless $\mathcal{P} = \mathcal{NP}$. For a given $\epsilon \in \mathbb{R}^{>0}$ and integer $k \geq 2$, let the $\epsilon$-\textsc{Approx-$k$-Cluster} problem be the problem of finding, for a given complete undirected graph $G$ with positive integer edge weights, a $k$-cut whose sum of cluster weights is at most $W^*(1 + \epsilon)$, where $W^*$ is the minimum sum of cluster weights. Likewise, let $\epsilon$-\textsc{Approx-Cluster} be the corresponding problem with $k$ provided as an input. We will show that for every positive $\epsilon$ and every integer $k \geq 3$, the $\epsilon$-\textsc{Approx-$k$-Cluster} problem is $\mathcal{NP}$-hard in the strong sense. Because $\epsilon$-\textsc{Approx-3-Cluster} $\leq_T^{op} \epsilon$-\textsc{Approx-Cluster}, it will then follow that this latter problem is also $\mathcal{NP}$-hard in the strong sense. Whether the result extends to $\epsilon$-\textsc{Approx-2-Cluster} is unknown at the time of this writing.

**Theorem 17.9** For every $\epsilon \in \mathbb{R}^{>0}$ and every $k \geq 3$, $\epsilon$-\textsc{Approx-$k$-Cluster} is $\mathcal{NP}$-hard in the strong sense.

**Proof:** As is suggested by Exercises 16.23 and 16.24, the problem of deciding whether a given undirected graph is $k$-colorable is $\mathcal{NP}$-complete for
each $k \geq 3$. Let us refer to this problem as $k$-Col. Because $k$-Col contains no large integers, it is $\mathcal{NP}$-complete in the strong sense. We will now show that $k$-Col $\leq_{\mathcal{PP}}^{p}$ $\varepsilon$-Approx-$k$-Cluster, so that $\varepsilon$-Approx-$k$-Cluster is $\mathcal{NP}$-hard in the strong sense for $k \geq 3$.

Let $G = (V, E)$ be a given undirected graph. Let $n$ be the number of vertices in $G$. We can assume without loss of generality that $n > k$, for otherwise $G$ is clearly $k$-colorable. We construct $G' = (V, E')$ to be the complete graph on $V$. We assign an edge weight of $n^2 [1 + \varepsilon]$ to edge $\{u, v\} \in E'$ if $\{u, v\} \in E$; otherwise, we assign it a weight of 1. Clearly, this construction can be completed in time polynomial in $n$. Furthermore, because $\varepsilon$ is a fixed constant, all integers have values polynomial in $n$. Suppose we have a $k$-cut of $G'$ such that the ratio of its cluster weight to the minimum cluster weight is at most $1 + \varepsilon$. If the given cluster weight is less than $n^2 [1 + \varepsilon]$, then all edges connecting vertices in the same cluster must have weight less than $n^2 [1 + \varepsilon]$; hence none of them belong to $E$. This $k$-cut is therefore a $k$-coloring of $G$. In this case, we can answer “yes”. Suppose the cluster weight is at least $n^2 [1 + \varepsilon]$. Because the approximation ratio is no more than $1 + \varepsilon$, the minimum sum of cluster weights is at least $n^2$. It is therefore impossible to $k$-color $G$, for a $k$-coloring of $G$ would be a $k$-cut of $G'$ in which each cluster contains only edges with weight 1, and which would therefore have total weight less than $n^2$. In this case, we can answer “no”. We conclude that $k$-Col $\leq_{\mathcal{PP}}^{p}$ $\varepsilon$-Approx-$k$-Cluster, so that $\varepsilon$-Approx-$k$-Cluster is $\mathcal{NP}$-hard in the strong sense. □

17.6 Summary

Using Turing reducibility, we can extend the definition of $\mathcal{NP}$-hardness from Chapter 16 to apply to problems other than decision problems in a natural way. We can then identify certain optimization problems as being $\mathcal{NP}$-hard, either in the strong sense or the ordinary sense. One way of coping with $\mathcal{NP}$-hard optimization problems is by using approximation algorithms.

For some $\mathcal{NP}$-hard optimization problems we can find polynomial approximation schemes, which take as input an instance $x$ of the problem and a positive real number $\varepsilon$ and return, in time polynomial in $|x|$, an approximate solution with approximation ratio no more than $1 + \varepsilon$. If this algorithm runs in time polynomial in $|x|$ and $1/\varepsilon$, it is called a fully polynomial approximation scheme.

However, Theorem 17.4 tells us that for most optimization problems, if
the problem admits a fully polynomial approximation scheme, then there is a pseudopolynomial algorithm to solve the problem exactly. As a result, we can use strong $\mathcal{NP}$-hardness to show for a number of problems that unless $\mathcal{P} = \mathcal{NP}$, that problem cannot have a fully polynomial approximation scheme. Furthermore, by showing $\mathcal{NP}$-hardness of certain approximation problems, we can show that unless $\mathcal{P} = \mathcal{NP}$, the corresponding optimization problem has no approximation algorithm with approximation ratio bounded by some — or in some cases any — given value.

Finally, there are some pairs of optimization problems, such as the maximum cut problem and the minimum cluster problem, that are essentially the same problem, but which yield vastly different results concerning approximation algorithm. For example, the maximum $k$-cut, can be approximated in $\Theta(n^2)$ time with an approximation ratio of no more than $1 + \frac{1}{k-1}$; however, unless $\mathcal{P} = \mathcal{NP}$, there is no polynomial-time algorithm with any bounded approximation ratio for finding the minimum weight of clusters formed by a $k$-cut if $k \geq 3$.

### 17.7 Exercises

**Exercise 17.1** Give an approximation algorithm that takes an instance of the knapsack problem and a positive integer $k$ and returns a packing with an approximation ratio of no more than $1 + \frac{1}{k}$. Your algorithm must run in $O(n^{k+1})$ time. Prove that both of these bounds (approximation ratio and running time) are met by your algorithm.

**Exercise 17.2** Suppose we were to modify the knapsack problem to allow as many copies as we wish of any of the items. Show that the greedy algorithm yields an approximation ratio of no more than 2 for this variation.

**Exercise 17.3** The best-fit algorithm for bin packing considers the items in the given order, always choosing the largest-weight bin in which the item will fit. Show that the approximation ratio for this algorithm is no more than 2.

**Exercise 17.4** Let $\epsilon$ be any fixed positive real number. Demonstrate how dynamic programming can be combined with the first-fit-decreasing algorithm to obtain a pseudopolynomial-time approximation algorithm for bin packing with approximation ratio no more than $\frac{11}{9} + \epsilon$. You may use the fact that the first-fit-decreasing algorithm always produces a packing using at most $\frac{11}{9}B^* + 4$ bins, where $B^*$ is the minimum number of bins possible.
* Exercise 17.5

a. Prove that \( \text{NOT-ALL-EQUAL-3-SAT} \leq_{\text{pp}} \text{\( k \)-CLUSTER} \) for each fixed \( k \geq 2 \), so that, from the result of Exercise 16.6, \( k \)-CLUSTER is \( \mathcal{NP} \)-hard in the strong sense for \( k \geq 2 \).

b. Prove that \( \text{CLUSTER} \) is strongly \( \mathcal{NP} \)-complete.

* Exercise 17.6

Give an approximation algorithm with approximation ratio bounded by 2 for the problem of finding a minimum-sized vertex cover. Your algorithm should run in \( O(n + a) \) time, assuming the graph is implemented as a \text{ListGraph}. Prove both the approximation ratio and the running time.

* Exercise 17.7

a. Given an undirected graph \( G = (V, E) \), we define \( G^2 = (V^2, E') \) such that for \( u = \langle u_1, u_2 \rangle \) and \( v = \langle v_1, v_2 \rangle \) in \( V^2 \), \( \{u, v\} \in E' \) iff for every \( i, 1 \leq i \leq 2 \), either \( u_i = v_i \) or \( \{u_i, v_i\} \in E \). Prove that the size of the largest clique in \( G^2 \) is \( k^2 \), where \( k \) is the size of the largest clique in \( G \).

b. Use part a to prove that if there is a polynomial-time algorithm with a bounded approximation ratio for approximating the size of a largest clique in a given graph, then there is a polynomial approximation scheme for this problem.

17.8 Chapter Notes

The concept of a polynomial-time approximation algorithm was first formalized by Garey, Graham, and Ullman [48] and Johnson [70]. In fact, much of the foundational work in this area is due to Garey and Johnson — see their text [52] for a summary of the early work. For example, they proved Theorem 17.4 [51]. A detailed analysis of bin packing, including the \( \frac{11}{9}B^* + 4 \) upper bound on the approximation ratio for first-fit decreasing, is given by Johnson [69]. The \( \lceil \frac{17}{16}B^* \rceil \) upper bound for the first-fit-decreasing algorithm is due to Garey, Graham, Johnson, and Yao [47], and a close relationship between best-fit and first-fit was established by Johnson, Demers, Ullman, Garey, and Graham [68].

The polynomial approximation scheme suggested by Exercise 17.1 for the knapsack problem is due to Sahni [95]. The fully polynomial approximation
scheme of Section 17.2 is due to Ibarra and Kim [66]. Theorem 17.7 was shown by Sahni and Gonzalez [94].