Design Techniques for Approximation Algorithms

IN THE preceding chapter we observed that many relevant optimization problems are NP-hard, and that it is unlikely that we will ever be able to find efficient (i.e., polynomial-time) algorithms for their solution. In such cases it is worth looking for algorithms that always return a feasible solution whose measure is not too far from the optimum.

According to the general formal framework set up in Chap. 1, given an input instance x of an optimization problem, we say that a feasible solution $y \in \text{SOL}(x)$ is an *approximate solution* of the given problem and that any algorithm that always returns a feasible solution is an *approximation algorithm*.

Observe that, in most cases, it is not difficult to design a polynomial-time approximation algorithm that returns a (possibly) trivial feasible solution (e.g., in the case of MINIMUM GRAPH COLORING it is sufficient to assign each node a different color). However, we are mainly interested in approximate solutions that satisfy a specific quality criterion. The quality of an approximate solution can be defined in terms of the "distance" of its measure from the optimum, which we would like to be as small as possible. A precise characterization of how to define such distance is deferred to Chap. 3; in this chapter we will use as quality measure the *performance ratio*, that is, the ratio between the value of the approximate solution returned by the algorithm and the optimum.

In the following we present fundamental techniques for the design of approximation algorithms. In the case of problems in PO, these techniques are commonly used to design polynomial-time algorithms that always re-

turn optimal solutions: in this chapter, instead, we will use them to design efficient algorithms for the approximate solution of NP-hard optimization problems.

Each section of the chapter considers a specific algorithmic strategy that has been fruitfully used in the design of approximation algorithms. As we will see, each strategy can be applied to a variety of problems. For some problems, it is possible to show that the measure of the approximate solution is, in all cases, close enough to the optimum. For other problems, we will see that, for some instances, the measure of the obtained solution is arbitrarily far from the optimum (in such cases, a positive result will be proved for some special classes of the input instances).

We will also see that each technique allows us in many cases to define an algorithmic scheme that can be applied to obtain several algorithms for the same problem with possibly different approximation properties.

Finally, at the end of the chapter, we will briefly describe several possible approaches to the analysis of algorithms yielding approximate solutions that will be thoroughly presented in the remainder of the book.

2.1 The greedy method

THE FIRST algorithmic strategy that we consider is the *greedy method* that can be applied to optimization problems such that an instance of the problem specifies a set of items and the goal is to determine a subset of these items that satisfies the problem constraints and maximizes or minimizes the measure function. The greedy method can be applied to a variety of problems: we first consider examples of the technique applied to maximization problems whose set of feasible solutions forms an independent set. Namely, for each instance x, any feasible solution y of x is a subset of the items defined in x and any subset of y is a feasible solution as well. Note that this implies that if x has at least one feasible solution, then the empty set is a feasible solution.

The greedy method initially sorts the items according to some criterion and then incrementally builds the solution starting from the empty set. Namely, it considers items one at a time, always maintaining a set of "selected" items: when a new item is considered, it is either added to the set or it is eliminated from any further consideration. The decision whether adding an item to the set is only based on the items that have been already selected (and on the item under consideration). Clearly, different solutions might be obtained depending on the order in which items are initially sorted.

Assuming that the comparison of two items can be done in constant time,

Problem 2.1: Maximum Knapsack

INSTANCE: Finite set X of items, for each $x_i \in X$, value $p_i \in Z^+$ and size $a_i \in Z^+$, positive integer b.

SOLUTION: A set of items $Y \subseteq X$ such that $\sum_{x_i \in Y} a_i \leq b$.

MEASURE: Total value of the chosen items, i.e., $\sum_{x_i \in Y} p_i$.

the running time required by the greedy algorithm is $O(n \log n)$ for sorting the n items plus the cost for n feasibility tests. Clearly this latter cost depends on the considered problem. Moreover, the quality of the approximate solution obtained depends on the initial ordering in which objects are considered; clearly, for each instance of the problem there is always an optimal ordering (i.e., an ordering that allows the greedy algorithm to find an optimal solution) but we do not expect to be able to find such an ordering in polynomial time for all instances of a computationally hard problem. However, as we will see in this section, in some cases we can find simple orderings that allow the greedy method to find good approximate solutions.

The greedy technique also applies to minimization problems. For instance, the well-known Kruskal's algorithm for the minimum spanning tree problem is based on sorting edges according to their weights and then greedily selecting edges until a spanning tree is obtained. In Sect. 2.1.3 we will see how the greedy approach can be used to design an approximation algorithm for MINIMUM TRAVELING SALESPERSON.

2.1.1 Greedy algorithm for the knapsack problem

MAXIMUM KNAPSACK (see Problem 2.1) models the problem of finding the optimal set of items to be put in a knapsack of limited capacity: to each item we associate a *profit* p_i (that represents the advantage of taking it) and an *occupancy* a_i. In general, we cannot take all items because the total occupancy of the chosen items cannot exceed the knapsack capacity b (in the sequel, without loss of generality, we assume that $a_i \leq b$, for $i = 1, 2, \ldots, n$).

Program 2.1 is a greedy algorithm for the MAXIMUM KNAPSACK problem that considers items in non-increasing order with respect to the profit/occupancy ratio (i.e., p_i/a_i). Since, after the items have been sorted, the complexity of the algorithm is linear in their number, the total running time is $O(n \log n)$. As far as the performance ratio is concerned, let us denote by $m_{Gr}(x)$ the value of the solution found by the algorithm when

Program 2.1: Greedy Knapsack

input Set X of n items, for each $x_i \in X$, values p_i, a_i, positive integer b;
output Subset $Y \subseteq X$ such that $\sum_{x_i \in Y} a_i \leq b$;
begin
    Sort X in non-increasing order with respect to the ratio p_i/a_i;
    (* Let $(x_1, x_2, \ldots, x_n)$ be the sorted sequence *)
    $Y := \emptyset$;
    for $i := 1$ **to** n **do**
      if $b \geq a_i$ **then**
      begin
        $Y := Y \cup \{x_i\}$;
        $b := b - a_i$
      end
    return Y
end.

applied to instance x. The following example shows that $m_{Gr}(x)$ can be arbitrarily far from the optimal value.

Example 2.1 ▶ Let us consider the following instance x of MAXIMUM KNAPSACK defined over n items: $p_i = a_i = 1$ for $i = 1, \ldots, n-1$, $p_n = b - 1$, and $a_n = b = kn$ where k is an arbitrarily large number. In this case, $m^*(x) = b - 1$ while the greedy algorithm finds a solution whose value is $m_{Gr}(x) = n - 1$: hence, $m^*(x)/m_{Gr}(x) > k$.

An analogous result can be easily shown if the items are sorted in non-decreasing order with respect to their profit or in non-increasing order with respect to their occupancy.

A closer look at the previous example shows that the poor behavior of Program 2.1 is due to the fact that the algorithm does not include the element with highest profit in the solution while the optimal solution contains only this element. This suggests a simple modification of the greedy procedure that has a better performance, as shown in the following theorem.

Theorem 2.1 ▶ *Given an instance x of the* MAXIMUM KNAPSACK *problem, let* $m_H(x) = \max(p_{max}, m_{Gr}(x))$, *where* p_{max} *is the maximum profit of an item in x. Then* $m_H(x)$ *satisfies the following inequality:*

$$m^*(x)/m_H(x) < 2.$$

PROOF   Let j be the index of the first item not inserted in the knapsack by the greedy algorithm with input x. The profit achieved by the algorithm at that point is

$$\bar{p}_j = \sum_{i=1}^{j-1} p_i \leq m_{Gr}(x)$$

and the overall occupancy is

$$\bar{a}_j = \sum_{i=1}^{j-1} a_i \leq b.$$

We first show that any optimal solution of the given instance must satisfy the following inequality: $m^*(x) < \bar{p}_j + p_j$. Since items are ordered in non-increasing order with respect to the ratio p_i/a_i, it follows that the exchange of any subset of the chosen items $x_1, x_2, \ldots, x_{j-1}$ with any subset of the unchosen items $x_j, \ldots, x_n$, that does not increase the sum of the occupancies $\bar{a}_j$, does not increase the overall profit. Therefore, the optimal value is bounded by $\bar{p}_j$ plus the maximum profit obtainable by filling the remaining available space of the knapsack (that is, $b - \bar{a}_j$) with items whose profit/weight ratio is at most p_j/a_j. Since $\bar{a}_j + a_j > b$, we obtain $m^*(x) \leq \bar{p}_j + (b - \bar{a}_j)p_j/a_j < \bar{p}_j + p_j$. To complete the proof we consider two possible cases. If $p_j \leq \bar{p}_j$ then $m^*(x) < 2\bar{p}_j \leq 2m_{Gr}(x) \leq 2m_H(x)$. On the other hand, if $p_j > \bar{p}_j$ then $p_{max} > \bar{p}_j$; in this case we have that

$$m^*(x) \leq \bar{p}_j + p_j \leq \bar{p}_j + p_{max} < 2p_{max} \leq 2m_H(x).$$

Thus, in both cases the theorem follows. QED

As a consequence of the above theorem a simple modification of Program 2.1 allows us to obtain a provably better algorithm. The modification consists of adding one more step, in which the solution is chosen to be either the greedy solution or the item with largest profit. In Sect. 2.5 we will see that much better approximation algorithms can be obtained by applying a different approach.

2.1.2 Greedy algorithm for the independent set problem

In this section we consider the behavior of the greedy approach applied to MAXIMUM INDEPENDENT SET (see Problem 2.2). If we use a greedy algorithm for this problem, then it is reasonable to assume that vertices with smaller degree should be preferred to vertices of higher degree. In fact, whenever we add a vertex to the current solution, we also have to eliminate all its neighbors from any further consideration. Hence, first choosing vertices with smaller degrees might allow us to obtain a largest independent set: Program 2.2 is based on this criterion. Note that this procedure differs from the previously discussed greedy algorithm: indeed, at each iteration of Program 2.2 the chosen vertex is the one with minimum degree in the subgraph induced by vertices that might be added to the current independent set.

Problem 2.2: Maximum Independent Set

INSTANCE: Graph $G = (V, E)$.

SOLUTION: An independent set on G, i.e., a subset $V' \subseteq V$ such that, for any $(u, v) \in E$ either $u \notin V'$ or $v \notin V'$.

MEASURE: Cardinality of the independent set, i.e., $|V'|$.

Program 2.2: Greedy Independent Set

input Graph $G = (V, E)$;
output Independent set V' in G;
begin
   $V' := \emptyset$;
   $U := V$;
   while U is not empty **do**
   begin
      $x :=$ vertex of minimum degree in the graph induced by U;
      $V' := V' \cup \{x\}$;
      Eliminate x and all its neighbors from U
   end;
   return V'
end.

It is possible to see that there exists a sequence of graphs with increasing number of vertices for which this algorithm achieves solutions whose measures are arbitrarily far from the optimal values.

Example 2.2 ▶ Let us consider the graph of Fig. 2.1 where K_4 is a clique of four nodes and I_4 is an independent set of four nodes. In this case, the rightmost node is the first to be chosen by Program 2.2 and the resulting solution contains this node and exactly one node of K_4. The optimal solution, instead, contains the four nodes of I_4. This example can be easily generalized by substituting I_4 and K_4 with I_k and K_k, for any $k \geq 2$.

Note that, until now, no variant of the greedy approach has allowed us to obtain a result similar to Theorem 2.1 in the case of MAXIMUM INDEPENDENT SET. This limitation is not due to the used method: in fact, it can be proved that, unless $P = NP$, no polynomial-time algorithm exists that finds a good approximate solution for all graphs. As we will see in Chap. 6, this is due to the intrinsic difficulty of the problem.

We now show that the performance of the greedy approach can be bounded by a function of the "density" of the graph.
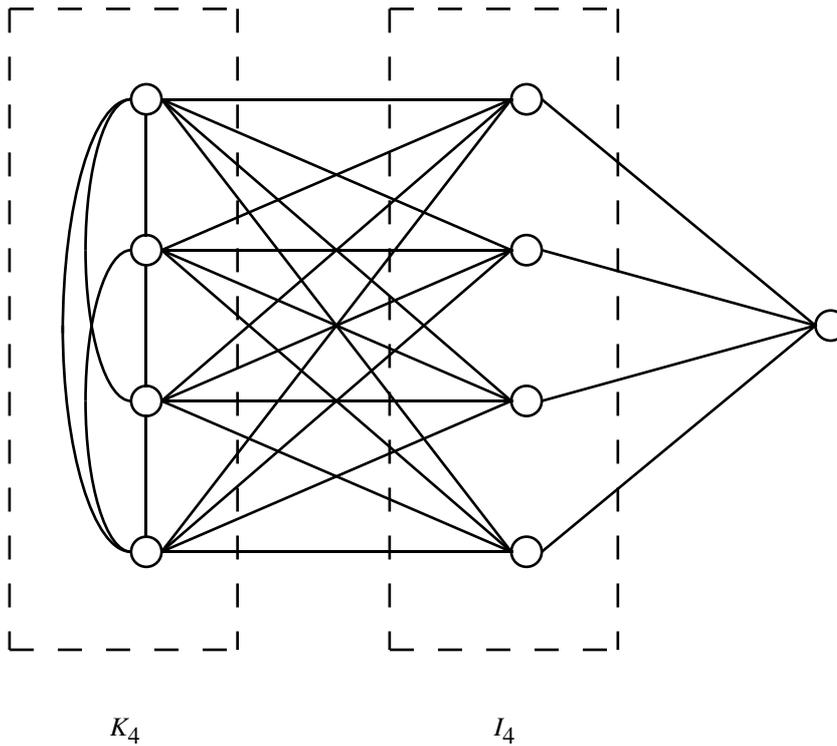
Figure 2.1
A bad example for
Program 2.2

K_4 I_4

Given a graph G with n vertices and m edges, let $\delta = m/n$ be the density   ◄ Theorem 2.2
of G. The value $m_{Gr}(G)$ of the solution found by Program 2.2 is at least
$n/(2\delta + 1)$.

Let x_i be the vertex chosen at the i-th iteration of the **while** loop of Pro-   PROOF
gram 2.2 and let d_i be the degree of x_i. The algorithm then deletes x_i and all
its d_i neighbors from G. Hence, the number of eliminated edges is at least
$d_i(d_i + 1)/2$ (since x_i is the minimum degree vertex in the graph currently
induced by U).

Summing up over all iterations, we have that

$$\sum_{i=1}^{m_{Gr}(G)} \frac{d_i(d_i + 1)}{2} \leq m = \delta n. \tag{2.1}$$

Since the algorithm stops when all vertices are eliminated, the following
equality holds:

$$\sum_{i=1}^{m_{Gr}(G)} (d_i + 1) = n. \tag{2.2}$$

By adding Eq. (2.2) and twice Eq. (2.1) we obtain that

$$\sum_{i=1}^{m_{Gr}(G)} (d_i + 1)^2 \leq n(2\delta + 1).$$

The left-hand side of the above inequality is minimized when $d_i + 1 = n/m_{Gr}(G)$, for all i (this is an application of the Cauchy-Schwarz inequality; see Appendix A). It follows that

$$n(2\delta + 1) \geq \sum_{i=1}^{m_{Gr}(G)} (d_i + 1)^2 \geq \frac{n^2}{m_{Gr}(G)}.$$

QED   Hence, $m_{Gr}(G) \geq n/(2\delta + 1)$ and the theorem is proved.

The following theorem provides a relationship between the measure of the solution found by the greedy algorithm and the optimal measure.

Theorem 2.3 ▶ *Given a graph G with n vertices and m edges, let $\delta = m/n$. Program 2.2 finds an independent set of value $m_{Gr}(G)$ such that*

$$m^*(G)/m_{Gr}(G) \leq (\delta + 1).$$

PROOF   The proof is similar to that of the preceding theorem: in this case, we additionally count in Eq. (2.1) the number of edges that are incident to vertices of some optimal solution.

Namely, fix a maximum independent set V^* and let k_i be the number of vertices in V^* that are among the $d_i + 1$ vertices deleted in the i-th iteration of the **while** loop of Program 2.2.

Clearly, we have that

$$\sum_{i=1}^{m_{Gr}(G)} k_i = |V^*| = m^*(G). \tag{2.3}$$

Since the greedy algorithm selects the vertex with minimum degree, the sum of the degree of the deleted vertices is at least $d_i(d_i + 1)$. Since an edge cannot have both its endpoints in V^*, it follows that the number of deleted edges is at least $(d_i(d_i + 1) + k_i(k_i - 1))/2$.

Hence we can improve Eq. (2.1) to obtain

$$\sum_{i=1}^{m_{Gr}(G)} \frac{d_i(d_i + 1) + k_i(k_i - 1)}{2} \leq \delta n. \tag{2.4}$$

Adding Eqs. (2.2), (2.3) and twice (2.4), we obtain the following bound:

$$\sum_{i=1}^{m_{Gr}(G)} \left((d_i+1)^2 + k_i^2\right) \leq n(2\delta+1) + m^*(G).$$

By applying the Cauchy-Schwarz inequality, it is possible to show that the left-hand side of the above inequality is minimized when $d_i + 1 = n/m_{Gr}(G)$ and $k_i = m^*(G)/m_{Gr}(G)$, for $i = 1, 2, \ldots, m_{Gr}(G)$. Hence,

$$n(2\delta+1) + m^*(G) \geq \sum_{i=1}^{m_{Gr}(G)} \left((d_i+1)^2 + k_i^2\right) \geq \frac{n^2 + m^*(G)^2}{m_{Gr}(G)},$$

that is,

$$m_{Gr}(G) \geq m^*(G)\frac{(n/m^*(G)) + (m^*(G)/n)}{2\delta + 1 + (m^*(G)/n)}.$$

To complete the proof it is sufficient to observe that the fractional term on the right-hand side of the above inequality is minimized when $m^*(G) = n$. By substituting this term, the theorem follows.   QED

2.1.3   Greedy algorithm for the salesperson problem

We now show how the idea of a "greedy" selection can be applied to MIN-IMUM TRAVELING SALESPERSON (see Problem 1.8). Recall that an instance of this problem can be represented by a complete graph $G = (V, E)$ with positive weights on the edges. Feasible solutions of the problem are subsets I of edges such that the graph (V, I) is a cycle.

The idea of the greedy algorithm is first to find a Hamiltonian path and then to form a tour by adding the edge that connects the first and the last vertex of the path.

The Hamiltonian path is found incrementally: at the beginning, the algorithm arbitrarily chooses the first city of the path, say c_{i_1}, and then executes a loop $(n-1)$ times. In the first iteration the algorithm selects the vertex that follows c_{i_1} by choosing vertex c_{i_2} such that edge (c_{i_1}, c_{i_2}) has minimum weight among all edges with an endpoint in c_{i_1}; edge (c_{i_1}, c_{i_2}) is then added to the path. In the r-th iteration of the loop, after a path through vertices $c_{i_1}, c_{i_2}, \ldots, c_{i_r}$ has been obtained, the algorithm chooses vertex $c_{i_{r+1}}$ that follows c_{i_r} in the path as the vertex such that $(c_{i_r}, c_{i_{r+1}})$ is the minimum weight edge from c_{i_r} to a vertex that is not in the path; edge $(c_{i_r}, c_{i_{r+1}})$ is added to the path. After $(n-1)$ iterations a Hamiltonian path $c_{i_1}, c_{i_2}, \ldots, c_{i_n}$

is obtained and the last step of the algorithm completes the tour by adding edge (c_{i_n}, c_{i_1}). Since the underlying graph is complete, the algorithm is guaranteed to find a feasible solution.

The "greediness" of this algorithm lies in the fact that during the construction of the Hamiltonian path the algorithm selects the edge that minimizes the cost of extending the current path with a new vertex. For this reason, we call the algorithm *Nearest Neighbor*.

No precise estimation is known on the quality of the solutions returned by *Nearest Neighbor* in the general case (see Exercise 2.3 for a constant lower bound). However, if we restrict the set of problem instances, then it is possible to obtain bounds on the quality of the solution found by the algorithm. We do so by considering instances of MINIMUM TRAVELING SALESPERSON in which the distance matrix is symmetric (i.e., $D(i, j) = D(j, i)$, for any pair of cities c_i and c_j) and the triangular inequality is satisfied (i.e., for all triples of cities c_i, c_j, and c_k, $D(i, j) \leq D(i, k) + D(k, j)$). In the following, we will denote this problem as MINIMUM METRIC TRAVELING SALESPERSON.

In order to provide a bound on the performance of *Nearest Neighbor* we need the following technical lemma.

Lemma 2.4 ▶ *For any instance x of* MINIMUM METRIC TRAVELING SALESPERSON *with n cities, assume that there exists a mapping $l : \{c_1, \ldots, c_n\} \mapsto \mathbb{Q}$ such that:*

1. *for any two distinct cities c_i and c_j, $D(i, j) \geq \min(l(c_i), l(c_j))$;*

2. *for any city c_i, $l(c_i) \leq \frac{1}{2} m^*(x)$.*

Then such a function satisfies the following inequality:

$$\sum_{i=1}^{n} l(c_i) \leq \frac{1}{2} (\lceil \log n \rceil + 1) m^*(x).$$

PROOF Without loss of generality, we assume that cities are sorted in nonincreasing order with respect to their l-values. Let us first prove that, for all k with $1 \leq k \leq n$,

$$m^*(x) \geq 2 \sum_{i=k+1}^{\min(2k,n)} l(c_i). \tag{2.5}$$

Let I^* be an optimal tour of length $m^*(x)$ and consider the subset of cities $C_k = \{c_i \mid 1 \leq i \leq \min(2k, n)\}$. Let I_k be a tour (of length m_k^*) that traverses the cities in C_k in the same order as I^*. For each pair c_r and c_s of adjacent

cities in I_k, either c_r and c_s were adjacent also in I^* or, by the triangle inequality, there exists a path in I^* starting from c_r and ending at c_s of length at least $D(r,s)$. As a consequence, $m^*(x) \geq m_k^*$, for each k.

Since $D(i,j) \geq \min(l(c_i), l(c_j))$ for all pairs of cities c_i and c_j, by summing over all edges (c_i, c_j) that belong to I_k we obtain

$$m_k^* \geq \sum_{(c_i,c_j)\in I_k} \min(l(c_i), l(c_j)) = \sum_{c_i \in C_k} \alpha_i l(c_i),$$

where α_i is the number of cities $c_j \in C_k$ adjacent to c_i in I_k and such that $i > j$ (hence, $l(c_i) \leq l(c_j)$). Clearly, $\alpha_i \leq 2$ and $\sum_{c_i \in C_k} \alpha_i$ equals the number of cities in I_k. Since the number of cities in I_k is at most $2k$, we may derive a lower bound on the quantity $\sum_{c_i \in C_k} \alpha_i l(c_i)$ by assuming that $\alpha_i = 0$ for the k cities $c_1, \ldots, c_k$ with largest values $l(c_i)$ and that $\alpha_i = 2$ for all the other $|C_k| - k$ cities. Hence, we obtain Eq. (2.5) since

$$m^*(x) \geq m_k^* \geq \sum_{c_i \in C_k} \alpha_i l(c_i) \geq 2 \sum_{i=k+1}^{\min(2k,n)} l(c_i).$$

Summing all Eqs. (2.5) with $k = 2^j$, for $j = 0, 1, \ldots, \lceil \log n \rceil - 1$, we obtain that

$$\sum_{j=0}^{\lceil \log n \rceil - 1} m^*(x) \geq \sum_{j=0}^{\lceil \log n \rceil - 1} 2 \sum_{i=2^j+1}^{\min(2^{j+1},n)} l(c_i),$$

which results in

$$\lceil \log n \rceil m^*(x) \geq 2 \sum_{i=2}^{n} l(c_i).$$

Since, by hypothesis, $m^*(x) \geq 2l(c_1)$, the lemma is then proved.   QED

For any instance x of MINIMUM METRIC TRAVELING SALESPERSON ◄ Theorem 2.5
with n cities, let $m_{NN}(x)$ be the length of the tour returned by Nearest
Neighbor *with input x. Then $m_{NN}(x)$ satisfies the following inequality:*

$$m_{NN}(x)/m^*(x) \leq \frac{1}{2}(\lceil \log n \rceil + 1).$$

Let $c_{k_1}, c_{k_2}, \ldots, c_{k_n}$ be a tour resulting from the application of *Nearest*   PROOF
Neighbor to x. The proof consists in showing that, if we associate to each city c_{k_r} ($r = 1, \ldots, n$) the value $l(c_{k_r})$ corresponding to the length of edge $(c_{k_r}, c_{k_{r+1}})$ (if $r < n$) or of edge (c_{k_n}, c_{k_1}) (if $r = n$), then we obtain a mapping l that satisfies the hypothesis of Lemma 2.4. Since $\sum_{i=1}^{n} l(c_{k_i}) = m_{NN}(x)$, the theorem follows immediately by applying the lemma.

Program 2.3: Partitioning Sequential Scheme

input Set I of items;
output Partition P of I;
begin
   Sort I (according to some criterion);
   (* Let $(x_1, x_2 \ldots, x_n)$ be the obtained sequence *)
   $P := \{\{x_1\}\}$;
   for $i := 2$ **to** n **do**
      if x_i can be added to a set p in P **then**
         add x_i to p
      else
         $P := P \cup \{\{x_i\}\}$;
   return P
end.

The first hypothesis of Lemma 2.4 can be proved as follows. Let c_{k_r} and c_{k_s} be any pair of cities: if $r < s$ (that is, c_{k_r} has been inserted in the tour before c_{k_s}), then $l(c_{k_r}) \leq D(k_r, k_s)$, since c_{k_s} was a possible choice as city $c_{k_{r+1}}$. Analogously, if $s < r$, then $l(c_{k_s}) \leq D(k_r, k_s)$; hence $D(k_r, k_s) \geq \min(l(c_{k_r}), l(c_{k_s}))$ and the first hypothesis of Lemma 2.4 holds.

To show the second hypothesis, assume that $l(c_{k_r}) = D(k_r, k_s)$. Observe that any optimal tour I^* is composed of two disjoint paths connecting c_{k_r} and c_{k_s}: by the triangle inequality, each such path must have length at least $D(k_r, k_s)$. The inequality $m^*(x) \geq 2D(k_r, k_s) = 2l(c_{k_r})$ hence follows and the proof of the theorem is completed.

QED

2.2 Sequential algorithms for partitioning problems

IN THIS section we consider *partitioning problems*, that is, problems in which feasible solutions are suitably defined partitions (not necessarily bipartitions) of a set of items $I = \{x_1, x_2 \ldots, x_n\}$ defined in the input instance and that satisfy the constraints specified by the particular problem.

A sequential algorithm for a partitioning problem initially sorts the items according to a given criterion and then builds the output partition P sequentially. The general scheme of such sequential procedure is given in Program 2.3.

Notice that, in the above scheme, when the algorithm considers item x_i it is not allowed to modify the partition of items x_j, for $j < i$. Therefore, if two objects are assigned to the same set of the partition then they will be in the same set of the final partition as well.

Problem 2.3: Minimum Scheduling on Identical Machines

INSTANCE: Set of jobs T, number p of machines, length l_j for executing job $t_j \in T$.

SOLUTION: A p-machine schedule for T, i.e., a function $f : T \mapsto [1..p]$.

MEASURE: The schedule's makespan, i.e.,

$$\max_{i \in [1..p]} \sum_{t_j \in T : f(t_j) = i} l_j.$$

To design a sequential algorithm for a particular problem, we need to specify the criteria used for (a) finding the order in which items are processed, and (b) including items in the partition. For each particular problem there are several possible criteria that can be used to sort the items. As in the case of the greedy approach, there exists at least one ordering that will allow us to obtain an optimal solution; however, we do not expect to find such an ordering in polynomial time for NP-hard combinatorial problems and we look for orderings of the items that allow us to find good approximate solutions.

2.2.1 Scheduling jobs on identical machines

In this section we apply the sequential algorithm scheme to the problem of scheduling a set of jobs on p identical machines with the goal of minimizing the time necessary to complete the execution of the jobs (see Problem 2.3). This problem is NP-hard even in the case of $p = 2$ (see Bibliographical notes).

Assume that we are given the order in which jobs are processed. In order to obtain a scheduling algorithm from the sequential scheme it is necessary to determine a rule for assigning jobs to machines. The obvious rule is to assign each job to the machine with smallest load. Namely, suppose that the first $j-1$ jobs have already been assigned and let $A_i(j-1)$ be the finish time necessary for the execution of the subset of jobs assigned to the i-th machine (i.e., $A_i(j-1) = \sum_{1 \leq k \leq j-1 : f(t_k) = i} l_k$). The j-th job is then assigned to the machine with minimum finish time (ties are arbitrarily broken). In this way the algorithm, called *List Scheduling*, minimizes the increase in the finish time required for the execution of a new job.

Given an instance x of MINIMUM SCHEDULING ON IDENTICAL MACHINES *with p machines, for any order of the jobs, the List Scheduling* ◀ Theorem 2.6

algorithm finds an approximate solution of measure $m_{LS}(x)$ such that

$$m_{LS}(x)/m^*(x) \le (2 - \frac{1}{p}).$$

PROOF Let W be the sum of the processing time of all jobs, i.e., $W = \sum_{k=1}^{|T|} l_k$; the value of the optimal solution clearly satisfies the bound $m^*(x) \ge W/p$.

To bound the value of the approximate solution found by the algorithm assume that the h-th machine is the one with the highest finish time (i.e., $A_h(|T|) = m_{LS}(x)$) and let j be the index of the last job assigned to this machine. Since job t_j was assigned to the least loaded machine, then the finish time of any other machine is at least $A_h(|T|) - l_j$. This implies that $W \ge p(A_h(|T|) - l_j) + l_j$ and that the following bound on $m_{LS}(x)$ holds:

$$m_{LS}(x) = A_h(|T|) \le \frac{W}{p} + \frac{(p-1)l_j}{p}.$$

Since $m^*(x) \ge W/p$ and $m^*(x) \ge l_j$ we have that

$$m_{LS}(x) \le \frac{W}{p} + \frac{(p-1)l_j}{p} \le m^*(x) + \frac{p-1}{p}m^*(x) = \left(2 - \frac{1}{p}\right)m^*(x)$$

QED and the theorem follows.

The following example shows that no better bound can be proved to hold for any ordering of the jobs: indeed, it shows that, for any number p of machines, there exists an instance of MINIMUM SCHEDULING ON IDENTICAL MACHINES on p machines with $p(p-1)+1$ jobs and an ordering of the jobs for which the bound of Theorem 2.6 is tight.

Example 2.3 ► Given p, let us consider the instance of MINIMUM SCHEDULING ON IDENTICAL MACHINES with p machines, $p(p-1)$ jobs of length 1 and one job of length p. Clearly, the optimal measure is p. On the other side, if the job with largest processing time is the last in the order, then *List Scheduling* will find a solution with value $2p-1$ (see Fig. 2.2).

The bad behavior of the previous example is due to the fact that the job with largest processing time is scheduled last. A simple ordering that allows us to find a better solution is based on the *LPT* (Largest Processing Time) rule that considers jobs in non-increasing order with respect to their processing time (i.e., $l_1 \ge l_2 \ge \cdots \ge l_{|T|}$).
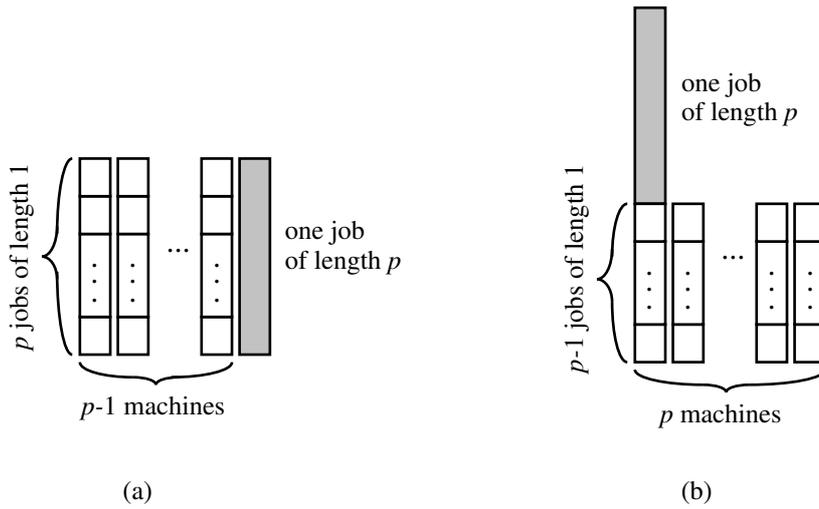
(a)

(b)

Figure 2.2
Example of (a) an optimal
scheduling and (b) a
scheduling computed by the
sequential algorithm

Given an instance x of MINIMUM SCHEDULING ON IDENTICAL MA-   ◄ Theorem 2.7
CHINES *with p machines, the LPT rule provides a schedule of measure*
$m_{LPT}(x)$ *such that* $m_{LPT}(x)/m^*(x) \le (4/3 - 1/3p)$.

Assume that the theorem does not hold and consider the instance x that   PROOF
violates the claim having the minimum number of jobs. Let j be the job of
x that is last considered by the *LPT* rule and let l_{min} be its length (which is
among the shortest ones). Two cases may arise. Either $l_{min} > m^*(x)/3$
or $l_{min} \le m^*(x)/3$. In the first case, it is simple to show that at most
two jobs may have been assigned to any machine and that the *LPT* rule
provides the optimal solution (see Exercise 2.5). In the second case (i.e.,
$l_{min} \le m^*(x)/3$), since x is a minimum counter-example it follows that the
instance x' obtained from x by removing job j satisfies the claim. This im-
plies that $m_{LPT}(x) > m_{LPT}(x')$ and, therefore, that the *LPT* rule assigns job
j to a machine that will have the largest processing time at the end of the
algorithm. By reasoning as in the proof of the previous theorem, we obtain
that

$$m_{LPT}(x) \le \frac{W}{p} + \frac{p-1}{p}l_{min}$$

where W is the sum of the processing times of all jobs. Since $m^*(x) \ge W/p$
and $l_{min} \le m^*(x)/3$, we obtain

$$m_{LPT}(x) \le m^*(x) + \frac{p-1}{3p}m^*(x) = \left(\frac{4}{3} - \frac{1}{3p}\right)m^*(x).$$

The theorem thus follows.   QED

Problem 2.4: Minimum Bin Packing

INSTANCE: Finite multiset I of rational numbers $\{a_1, a_2, \ldots, a_n\}$ with $a_i \in (0, 1]$ for $i = 1, \ldots, n$ (in a multiset the same number may appear more than once).

SOLUTION: A partition $\{B_1, B_2, \ldots, B_k\}$ of I such that $\sum_{a_i \in B_j} a_i \leq 1$ for $j = 1, \ldots, k$.

MEASURE: The cardinality of the partition, i.e., k.

2.2.2 Sequential algorithms for bin packing

MINIMUM BIN PACKING (see Problem 2.4) looks for a packing of a set of weighted items using the minimum number of bins of unit capacity. The total weight of the items assigned to a bin cannot exceed its capacity.

A simple sequential algorithm for MINIMUM BIN PACKING, called *Next Fit*, processes the items one at a time in the same order as they are given in input. The first item a_1 is placed into bin B_1. Let B_j be the last used bin, when the algorithm considers item a_i: *Next Fit* assigns a_i to B_j if it has enough room, otherwise a_i is assigned to a new bin B_{j+1}.

Theorem 2.8 ▶ *Given an instance x of* MINIMUM BIN PACKING, Next Fit *returns a solution with value $m_{NF}(x)$ such that $m_{NF}(x)/m^*(x) < 2$.*

PROOF   The proof estimates the value of the optimal and approximate solutions as functions of the sum of the item sizes, denoted by A (i.e., $A = \sum_{i=1}^{n} a_i$).

Observe that the number of bins used by *Next Fit* is less than $2\lceil A \rceil$: this is due to the fact that, for each pair of consecutive bins, the sum of the sizes of the items included in these two bins is greater than 1. On the other hand, since the number of bins used in each feasible solution is at least the total size of the items, we have that $m^*(x) \geq \lceil A \rceil$. It follows that

QED $m_{NF}(x) < 2m^*(x)$.

Example 2.4 ▶ The bound stated in Theorem 2.8 is asymptotically tight. In fact, for each integer n, there exists an instance of $4n$ items and an ordering of these items such that $m^*(x) = n + 1$ and $m_{NF}(x) = 2n$. The instance and the order of the items are as follows: $I = \{1/2, 1/(2n), 1/2, 1/(2n), \ldots, 1/2, 1/(2n)\}$ (each pair is repeated $2n$ times). Figure 2.3 shows both the optimal and the approximate solution found by *Next Fit*.

An obvious weakness of *Next Fit* is that it tries to assign an item only to the last used bin. This suggests a new algorithm, called *First Fit*, that processes items in the input order according to the following rule: item a_i

is assigned to the first used bin that has enough available space to include it; if no bin can contain a_i, a new bin is opened.
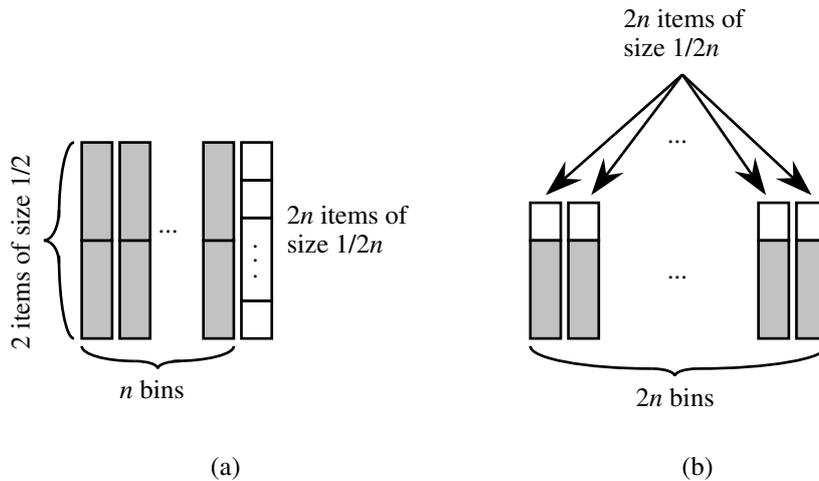

Figure 2.3
Example of (a) an optimal packing and (b) a packing found with *Next Fit*

The *First Fit* algorithm has a better performance than *Next Fit*: indeed, it finds a solution that is at most 70% far away from the optimal solution. Namely, it can be shown that *First Fit* finds a solution with value $m_{FF}(x)$ such that $m_{FF}(x) \leq 1.7m^*(x) + 2$ (see Bibliographical notes).

An even better algorithm for MINIMUM BIN PACKING is *First Fit Decreasing*: this algorithm first sorts items in non-increasing order with respect to their size and then processes items as *First Fit* (see Program 2.4).

Given an instance x of MINIMUM BIN PACKING, *the* First Fit Decreasing *algorithm finds a solution with measure* $m_{FFD}(x)$ *such that*   ◀ Theorem 2.9

$$m_{FFD}(x) \leq 1.5m^*(x) + 1.$$

Let us partition the ordered list of items $\{a_1, a_2, \ldots, a_n\}$, according to their value, into the following sets: PROOF

$$
\begin{aligned}
A &= \{a_i \mid a_i > 2/3\}, \\
B &= \{a_i \mid 2/3 \geq a_i > 1/2\}, \\
C &= \{a_i \mid 1/2 \geq a_i > 1/3\}, \\
D &= \{a_i \mid 1/3 \geq a_i\}.
\end{aligned}
$$

Consider the solution obtained by *First Fit Decreasing*. If there is at least one bin that contains only items belonging to D, then there is at most one

Program 2.4: First Fit Decreasing

input Set I of n positive rationals less than or equal to 1;
output Partition of I in subsets of unitary weight;
begin
   Sort elements of I in non-increasing order;
   (* Let $(a_1, a_2, \ldots, a_n)$ be the obtained sequence *)
   for $i := 1$ **to** n **do**
      if there is a bin that can contain a_i **then**
         Insert a_i into the first such bin
      else
         Insert a_i into a new bin ;
   return the partition
end.

bin (the last opened one) with total occupancy less than $2/3$, and the bound follows.

If there is no bin containing only items belonging to D, then *First Fit Decreasing* finds the optimal solution. In fact, let x' be the instance obtained by eliminating all items belonging to D. Since the value of the solution found by *First Fit Decreasing* for x and x' is the same, it is sufficient to prove the optimality of *First Fit Decreasing* for x'. To this aim, we first observe that, in each feasible solution of x', items from A cannot share a bin with any other item and that every bin contains at most two items (and only one of these two items can belong to B). The thesis follows by observing that *First Fit Decreasing* processes items in non-increasing order with respect to their weight. Therefore, it packs each item belonging to C with the largest possible item in B that might fit with it and that does not already share a bin with another item. This implies that the number of bins in the optimal solution and in the solution found by *First Fit Decreasing* are the same.

QED

By means of a detailed case analysis, the bound given in the previous theorem can be substituted by $11 m^*(x)/9 + 7/9$ (see Bibliographical notes). The following example shows that no better multiplicative factor can be obtained, for sufficiently large instances.

Example 2.5 ▶ For any $n > 0$, let us consider the following instance x_n. The instance includes $5n$ items: n items of size $1/2 + \varepsilon$, n items of size $1/4 + 2\varepsilon$, n items of size $1/4 + \varepsilon$ and $2n$ items of size $1/4 - 2\varepsilon$. As can be seen in Fig. 2.4, $m_{FFD}(x_n) = \frac{11}{6}n$, while $m^*(x_n) = \frac{3}{2}n$.

Note that if there is more than one bin that can contain an item without violating the feasibility constraint, the *First Fit Decreasing* algorithm

chooses the first opened bin and does not try to optimize the choice. This might give rise to bins filled only partly and, therefore, to an increased number of bins (the division of the available space into many small fragments is called *fragmentation*).
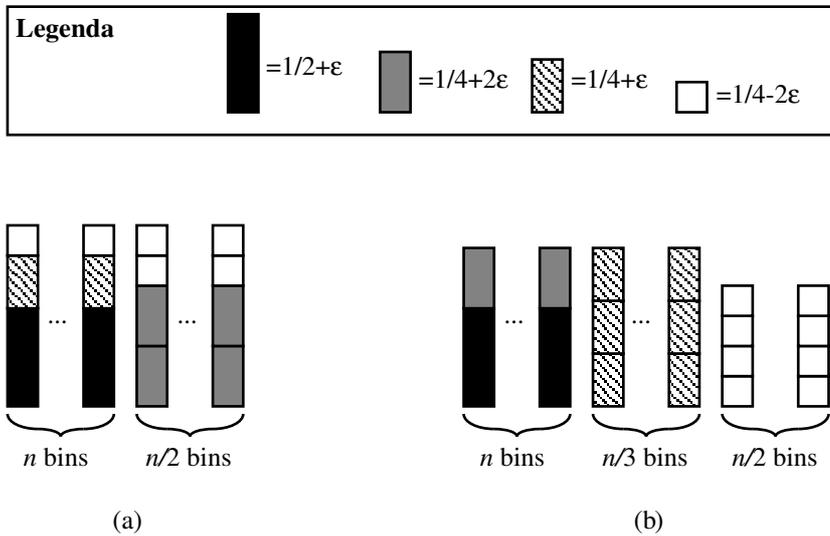

Figure 2.4
An example of (a) an optimal packing that uses $3n/2$ bins and (b) a packing produced by FFD that uses $11n/6$ bins

An apparently better algorithm for MINIMUM BIN PACKING is the *Best Fit Decreasing* algorithm. Like *First Fit Decreasing*, *Best Fit Decreasing* initially sorts the items in non-increasing order with respect to their weight and then considers them sequentially. The difference between the two algorithms is the rule used for choosing the bin in which the new item a_i is inserted: while trying to pack a_i, *Best Fit Decreasing* chooses one bin whose empty space is minimum. In this way, the algorithm tries to reduce the fragmentation by maximizing the number of bins with large available capacity.

It is possible to see that the quality of the solution found by *Best Fit Decreasing* may be worse than the quality of the solution found by *First Fit Decreasing* (see Exercise 2.9). On the other hand, as shown in the following example, it is possible to define instances for which *Best Fit Decreasing* finds an optimal solution while *First Fit Decreasing* returns a non-optimal solution.

For any $n > 0$, let us us consider the following instance x_n. The instance includes ◄ Example 2.6
$6n$ items: n items of size $7/10$, $2n$ items of size $2/5$, n items of size $1/5$, and $2n$ items of size $3/20$. As can be easily seen, $m_{BFD}(x_n) = 2n$, while $m_{FFD}(x_n) = 2n + \lceil n/6 \rceil$.

However, the 11/9-bound holds for *Best Fit Decreasing* as well and the sequence of instances given in Example 2.5 shows that the bound is tight (i.e., 11/9 is the smallest multiplicative factor).

As a final remark we observe that both *First Fit Decreasing* and *Best Fit Decreasing* are *off-line* algorithms while *Next Fit* and *First Fit* are *on-line* algorithms. Indeed, these latter two algorithms assign item a_i to a bin without knowing the size of items a_j, for any $j > i$. On the other side, in an off-line algorithm the packing of the items starts when all sizes are known (as in the case of *First Fit Decreasing* and *Best Fit Decreasing* in which the first step to be performed is sorting the items).

2.2.3 Sequential algorithms for the graph coloring problem

In this section we consider the behavior of sequential algorithms for MIN-IMUM GRAPH COLORING. In order to apply the sequential scheme to design a coloring algorithm it is necessary to specify an order of the vertices. Assuming $(v_1, v_2, \ldots, v_n)$ is the sequence obtained, it is straightforward to obtain a sequential algorithm. Vertex v_1 is colored with color 1 and the remaining vertices are colored as follows: when vertex v_i is considered and colors $1, 2, \ldots, k$ have been used the algorithm attempts to color v_i using one of the colors $1, 2, \ldots, k$ (if there are several possibilities it chooses the minimum color); otherwise (that is, v_i has at least k adjacent vertices which have been assigned different colors) a new color $k + 1$ is used to color v_i. The algorithm proceeds in this way until all vertices have been colored.

In the sequel we will see that the quality of the solution obtained using this algorithm is not as good as in the case of MINIMUM SCHEDULING ON IDENTICAL MACHINES and MINIMUM BIN PACKING. In fact, unless $P = NP$, there is no ordering of the vertices that can be found in polynomial time and that allows us to obtain a constant bound on the performance ratio that holds for all graphs.

Nevertheless, we will consider two different criteria to order the vertices of the graph: the *decreasing degree* order and the *smallest last* order. To better motivate these criteria we first analyze the number of colors used by a sequential algorithm as a function of the degree of the vertices. Namely, given an ordering $(v_1, v_2, \ldots, v_n)$ of the vertex set of G, let G_i be the graph induced by vertices $\{v_1, \ldots, v_i\}$ (clearly, $G_n = G$). Let k_i be the number of colors used by the sequential algorithm to color G_i (hence, k_n denotes the number of colors used by the algorithm with input G), and let $d_i(v)$ be the degree of v in G_i (hence, $d_n(v)$ denotes the degree of v in G).

Theorem 2.10 ▶ *Let k_n be the number of colors used by the sequential coloring algorithm*

when applied to an input graph G whose vertices are considered in the order $(v_1, v_2, \ldots, v_n)$. Then, the following inequality holds:

$$k_n \leq 1 + \max_{1 \leq i \leq n} \min(d_n(v_i), i - 1).$$

PROOF

If the algorithm does not introduce a new color to color vertex v_i, then $k_i = k_{i-1}$. Otherwise, $k_i = k_{i-1} + 1$ and the degree of v_i in G_i must satisfy the inequality

$$d_i(v_i) \geq k_{i-1}.$$

By induction, it thus follows that

$$k_n \leq 1 + \max_{1 \leq i \leq n} (d_i(v_i)). \tag{2.6}$$

Since $d_i(v_i)$ is clearly bounded both by $d_n(v_i)$ and by $i - 1$ (that is, the number of other nodes in G_i), the theorem follows.

QED

An immediate consequence of the above theorem is the following result.

For any ordering of the vertices, the sequential coloring algorithm uses at most $\Delta + 1$ colors to color a graph G, where Δ denotes the highest degree of the vertices of G.

◄ Corollary 2.11

Observe that a non-increasing ordering of the vertices with respect to their degree minimizes the upper bound in Theorem 2.10. Hence, we use this ordering to obtain the sequential algorithm called *Decreasing Degree*. Unfortunately, the number of colors used by this algorithm can be much larger than the number of colors used by the optimal solution. In fact, the following example shows that there exist 2-colorable graphs with $2n$ vertices and maximum degree $n - 1$ for which *Decreasing Degree* could require n colors (as a side effect, the example also shows that the bound of Corollary 2.11 is tight).

Let n be an integer and $G(V, E)$ be a graph with $V = \{x_1, \ldots, x_n, y_1, \ldots, y_n\}$ and $E = \{(x_i, y_j) \mid i \neq j\}$ (see Fig. 2.5 where $n = 4$). Note that all vertices have the same degree $n - 1$ and that G can be easily colored with 2 colors. However, if the initial ordering of the vertices is
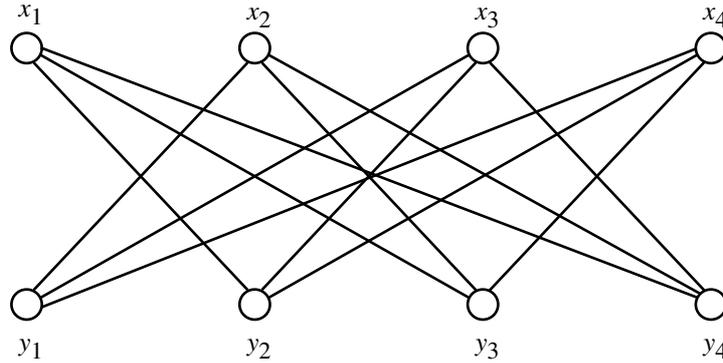
◄ Example 2.7

$$(x_1, y_1, x_2, y_2, \ldots, x_n, y_n),$$

then *Decreasing Degree* uses n colors.

Figure 2.5
A graph for which
Decreasing Degree behaves
poorly

Many other heuristic criteria for finding an initial ordering of the vertices have been proposed for improving the bad behavior of *Decreasing Degree*. In the following we consider the *smallest last* order defined as follows: v_n, the last vertex to be considered, is the one with minimum degree in G (breaking ties arbitrarily). The order of the remaining vertices is defined backward: after vertices $v_{i+1}, \ldots, v_n$ have been inserted in the ordered sequence, v_i is the vertex with minimum degree in the subgraph induced by $V - \{v_{i+1}, \ldots, v_n\}$ (breaking ties arbitrarily). Using this ordering we obtain the *Smallest Last* algorithm).

Recall that, for any ordering $(v_1, \ldots, v_n)$ of the vertices, Eq. (2.6) implies the following bound on the number of colors used by the sequential algorithm:

$$k_n \leq 1 + \max_{1 \leq i \leq n} (d_i(v_i)).$$

The smallest last ordering of the vertices minimizes the above bound since we have that, for each i with $1 \leq i \leq n$, $d_i(v_i) = \min_{v_j \in G_i} d_i(v_j)$. Unfortunately, it is possible to show that *Smallest Last* fails to find good colorings for all graphs (see Exercise 2.14). However, its performance is satisfactory when applied to planar graphs.

Theorem 2.12 ▶ *The* Smallest Last *algorithm colors a planar graph with at most six colors.*

PROOF   It is sufficient to prove that $\max_{1 \leq i \leq n}(d_i(v_i)) \leq 5$. Indeed, this is due to the fact that, for any planar graph G, Euler's theorem implies that the vertex of smallest degree has at most 5 neighbors and that deleting a vertex from a planar graph yields a planar graph.

QED

From the above theorem, the next result follows.

Corollary 2.13 ▶ *There exists a polynomial-time coloring algorithm $\mathcal{A}$ that, when applied to a planar graph G, finds a solution with measure $m_{\mathcal{A}}(G)$ such that $m_{\mathcal{A}}(G)/m^*(G) \leq 2$.*

Program 2.5: Local Search Scheme

input Instance *x*;
output Locally optimal solution *y*;
begin
   y := initial feasible solution;
   while there exists a neighbor solution *z* of *y* better than *y* **do**
     y := *z*;
   return *y*
end.

Since there exists a polynomial-time algorithm for optimally coloring a 2-colorable graph (see Exercise 1.15), it follows that the joint use of this algorithm and of *Smallest Last* allows us to obtain an algorithm $\mathcal{A}$ such that:

PROOF

- if $m^*(G) \leq 2$, then $m_{\mathcal{A}}(G) = m^*(G)$;

- if $m^*(G) \geq 3$, then $m_{\mathcal{A}}(G) \leq 6$.

The corollary thus follows.

QED

2.3 Local search

L OCAL SEARCH algorithms start from an initial solution (found using some other algorithm) and iteratively improve the current solution by moving to a better "neighbor" solution (see Program 2.5).

Roughly speaking, if *y* is a feasible solution, then *z* is a neighbor solution if it does not differ substantially from *y*. Given a feasible solution *y*, the local search algorithm looks for a neighbor solution with an improved value of the measure function. When no improvement is possible (i.e., when the algorithm reaches a solution *y* such that all neighbor solutions are no better than *y*) then the algorithm stops in a *local optimum* (with respect to the chosen neighborhood).

To obtain a local search algorithm for a specific problem we thus need an algorithm for finding the initial feasible solution (in many cases this can be a trivial task) and a neighborhood structure of any feasible solution *y* (notice that it is not necessary to find all neighbors of *y* but it is sufficient to determine whether there exists one neighbor solution that has a better measure).

Observe that if the neighborhood structure allows us to move from one solution to a better neighbor solution in polynomial time and to find an

Problem 2.5: Maximum Cut

INSTANCE: Graph $G = (V, E)$.

SOLUTION: Partition of V into disjoint sets V_1 and V_2.

MEASURE: The cardinality of the cut, i.e., the number of edges with one endpoint in V_1 and one endpoint in V_2.

optimal solution (i.e., a global optimum) after a polynomial number of solutions have been considered, then the local search algorithm solves the problem optimally in polynomial time. Clearly, in the case of NP-hard problems, we do not expect to find a neighborhood structure that allows us to find an optimal solution in polynomial time (unless $P = NP$). In these cases, we look for a local search algorithm that finds an approximate solution corresponding to a local optimum with respect to the neighborhood structure.

2.3.1 Local search algorithms for the cut problem

In this section, we will describe a local search algorithm for MAXIMUM CUT (see Problem 2.5) that achieves a solution that is guaranteed to be at most a constant factor away from the optimal solution. To this aim, we need to define both the procedure for obtaining an initial feasible solution and the neighborhood structure. In the case of the MAXIMUM CUT the first task is trivial, since the partition $V_1 = \emptyset$ and $V_2 = V$ is a feasible solution.

Furthermore, we define the following neighborhood structure $\mathcal{N}$: the neighborhood of a solution (V_1, V_2) consists of all those partitions (V_{1k}, V_{2k}), for $k = 1, \ldots, |V|$, such that:

1. if vertex $v_k \in V_1$, then

$$V_{1k} = V_1 - \{v_k\} \qquad \text{and} \qquad V_{2k} = V_2 \cup \{v_k\};$$

2. if vertex $v_k \notin V_1$, then

$$V_{1k} = V_1 \cup \{v_k\} \qquad \text{and} \qquad V_{2k} = V_2 - \{v_k\}.$$

The important property of the above neighborhood structure is that each local optimum has a measure that is at least half of the optimum.

Theorem 2.14 ▶ *Given an instance x of* MAXIMUM CUT, *let* (V_1, V_2) *be a local optimum*

with respect to the neighborhood structure $\mathcal{N}$ and let $m_{\mathcal{N}}(x)$ be its measure. Then

$$m^*(x)/m_{\mathcal{N}}(x) \leq 2.$$

PROOF

Let m be the number of edges of the graph. Since $m^*(x) \leq m$, it is sufficient to prove that $m_{\mathcal{N}}(x) \geq m/2$.

We denote by m_1 and m_2 the number of edges connecting vertices inside V_1 and V_2 respectively. We have that

$$m = m_1 + m_2 + m_{\mathcal{N}}(x). \qquad (2.7)$$

Given any vertex v_i, we define

$$m_{1i} = \{v \mid v \in V_1 \text{ and } (v, v_i) \in E\}$$

and

$$m_{2i} = \{v \mid v \in V_2 \text{ and } (v, v_i) \in E\}.$$

If (V_1, V_2) is a local optimum then, for any vertex v_k, the solution provided by (V_{1k}, V_{2k}) has a value at most $m_{\mathcal{N}}(x)$. This implies that, for every node $v_i \in V_1$,

$$|m_{1i}| - |m_{2i}| \leq 0$$

and, for every node $v_j \in V_2$,

$$|m_{2j}| - |m_{1j}| \leq 0.$$

By summing over all vertices in V_1 and V_2, we obtain

$$\sum_{v_i \in V_1} (|m_{1i}| - |m_{2i}|) = 2m_1 - m_{\mathcal{N}}(x) \leq 0$$

and

$$\sum_{v_j \in V_2} (|m_{2j}| - |m_{1j}|) = 2m_2 - m_{\mathcal{N}}(x) \leq 0.$$

Hence $m_1 + m_2 - m_{\mathcal{N}}(x) \leq 0$. From this inequality and from Eq. 2.7 it follows that $m_{\mathcal{N}}(x) \geq m/2$ and the theorem is proved.

QED

For a given optimization problem, there are many possible definitions of neighborhood structure that may determine both the quality of the approximate solution and the running time of the algorithm. The main issues involved in the definition of the neighborhood structure are:

- the quality of the solution obtained (that is, how close is the value of the local optimum to the global optimal value);

- the order in which the neighborhood is searched;

- the complexity of verifying that the neighborhood does not contain any better solution;

- the number of solutions generated before a local optimum is found.

The above four issues are strongly related. In fact, if the neighborhood structure is "large" then it is likely that the value of the solution obtained is close to the optimal value. However, in this case we expect the task of checking that a better neighbor does not exist to become more complex.

As an extreme case, we can assume that any solution is neighbor of any other solution: in this case, the algorithm will find an optimal solution, but if the problem is computationally hard, then either the complexity of looking for a better neighbor is computationally hard or there is an exponential number of solutions to be considered before the global optimum is found.

2.3.2 Local search algorithms for the salesperson problem

The interesting property of the neighborhood structure defined for MAXIMUM CUT is that all local optima have a measure that is at least half of the optimum. Unfortunately, this nice property is not shared by many other combinatorial optimization problems. However, local search approximation algorithms are applied successfully due to their good practical behavior and their simple structure. For example, experimental work has shown that, in the case of MINIMUM TRAVELING SALESPERSON, there exist local search algorithms that find approximate good solutions in a reasonable amount of time for very large instances (even though there are instances in which the performance of the algorithm is quite bad).

Obtaining an initial feasible solution for an instance of MINIMUM TRAVELING SALESPERSON is easy since any permutation of the n cities is a solution: therefore, the identity permutation of the n cities $(c_1, c_2, \ldots, c_n)$ is a feasible tour. Alternatively, we can consider the solution found by any other algorithm (e.g, *Nearest Neighbor*).

A simple neighborhood structure for the symmetric case of MINIMUM TRAVELING SALESPERSON is the *2-opt* structure which is based on the following observation: given a tour I, replacing two edges (x,y) and (v,z) in I by the two edges (x,v) and (y,z) yields another tour I'. This swap operation is called a *2-move*. The 2-opt neighborhood structure of a solution I consists of those solutions that can be obtained from I using a 2-move. Observe that, according to this neighborhood structure, any solution has

$O(n^2)$ neighbors: hence, searching in a neighborhood for a better solution requires at most $O(n^2)$ operations.
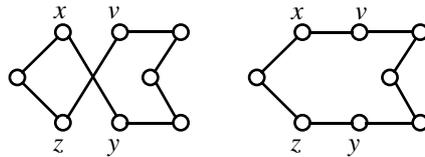

Figure 2.6
Improving a tour by
performing a 2-move

Let us consider the Euclidean version of MINIMUM TRAVELING SALESPERSON in which it is assumed that cities correspond to points in the plane and that the distance between cities x and y is given by the Euclidean distance between the corresponding points in the plane. Figure 2.6 represents a tour: since edges (x,y) and (v,z) cross each other, the triangle inequality implies that replacing (x,y) and (v,z) by (x,v) and (y,z) yields a better solution.

◄ Example 2.8

Note that given a tour I the absence of crossing edges does not guarantee that I is a local optimum with respect to the 2-opt neighborhood structure (see, for instance, Fig. 2.7).

Experimental work has shown that the quality of the solution obtained by using the 2-opt neighborhood structure depends on the solution initially chosen and that if the local search algorithm is executed several times using different initial solutions, then the overall algorithm can be very effective. For example, in the case of randomly distributed points in the plane, such an algorithm finds a solution with expected value within 5.5% of the optimal value.
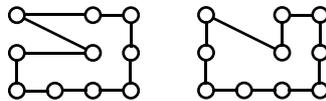

Figure 2.7
A tour with no crossing
edges and its improvement

A more detailed presentation of local search algorithms for MINIMUM TRAVELING SALESPERSON is given in Chap. 10.

2.4 Linear programming based algorithms

L INEAR PROGRAMMING is one of the most successful areas of operations research and has been applied in many application contexts (we refer to Appendix A for related definitions).

Observe that, since a linear program can be solved in polynomial time, given a hard combinatorial optimization problem $\mathcal{P}$, we do not expect to

find a linear programming formulation of $\mathcal{P}$ such that, for any instance x of $\mathcal{P}$, the number of constraints of the corresponding linear program is polynomial in the size of x: in fact, this would imply that P= NP.[1]

Nevertheless, linear programming can be used as a computational step in the design of approximation algorithms. Several approaches are possible that are based on the fact that most combinatorial optimization problems can be formulated as *integer* linear programming problems. This equivalent formulation does not simplify the problem but allows us to design algorithms that, in some cases, give good approximate solutions.

2.4.1 Rounding the solution of a linear program

The simplest approach to obtain approximation algorithms based on linear programming can be described as follows. Given an integer linear program, by relaxing the integrality constraints we obtain a new linear program, whose optimal solution can be found in polynomial time. This solution, in some cases, can be used to obtain a feasible solution for the original integer linear program, by "rounding" the values of the variables that do not satisfy the integrality constraints.

As an example, let us consider the weighted version of MINIMUM VERTEX COVER (which we will denote as MINIMUM WEIGHTED VERTEX COVER) in which a non-negative weight c_i is associated with each vertex v_i and we look for a vertex cover having minimum total weight. Given a weighted graph $G = (V, E)$, MINIMUM WEIGHTED VERTEX COVER can be formulated as the following integer linear program $ILP_{VC}(G)$:

$$
\begin{aligned}
\text{minimize} \quad & \sum_{v_i \in V} c_i x_i \\
\text{subject to} \quad & x_i + x_j \geq 1 \quad \forall (v_i, v_j) \in E \\
& x_i \in \{0, 1\} \quad \forall v_i \in V.
\end{aligned}
$$

Let LP_{VC} be the linear program obtained by relaxing the integrality constraints to simple non-negativeness constraints (i.e., $x_i \geq 0$ for each $v_i \in V$). Let $x^*(G)$ be an optimal solution of LP_{VC}. Program 2.6 obtains a feasible solution V' for the MINIMUM WEIGHTED VERTEX COVER problem by rounding up all components of $x^*(G)$ with a sufficiently large value. Namely, it includes in the vertex cover all vertices corresponding to components whose value is at least 0.5.

Theorem 2.15 ▶ *Given a graph G with non-negative vertex weights, Program 2.6 finds a*

---

[1]Note that we might be able to find a linear programming formulation such that the number of constraints is exponential in the number of variables.

Program 2.6: Rounding Weighted Vertex Cover

input Graph $G = (V, E)$ with non-negative vertex weights;
output Vertex cover V' of G;
begin
   Let ILP_{VC} be the linear integer programming formulation of the problem;
   Let LP_{VC} be the problem obtained from ILP_{VC} by relaxing
     the integrality constraints;
   Let $x^*(G)$ be the optimal solution for LP_{VC};
   $V' := \{v_i \mid x_i^*(G) \geq 0.5\}$;
   return V'
end.

feasible solution of MINIMUM WEIGHTED VERTEX COVER *with value* $m_{LP}(G)$ *such that* $m_{LP}(G)/m^*(G) \leq 2$.

Let V' be the solution returned by the algorithm. The feasibility of V' can   PROOF
be easily proved by contradiction. In fact, assume that V' does not cover
edge (v_i, v_j). This implies that both $x_i^*(G)$ and $x_j^*(G)$ are less than 0.5,
thus contradicting the fact that $x^*(G)$ is a feasible solution of the relaxed
linear program.

   In order to prove that V' is a solution whose value is at most twice the
optimal value, we first observe that the value $m^*(G)$ of an optimal solution
satisfies the inequality:

$$m^*(G) \geq m_{LP}^*(G)$$

where $m_{LP}^*(G)$ denotes the optimal measure of the relaxed linear program.
Since

$$\sum_{v_i \in V'} c_i \leq 2 \sum_{v_i \in V} c_i x_i^*(G) = 2m_{LP}^*(G) \leq 2m^*(G),$$

the theorem then follows.   QED

2.4.2   Primal-dual algorithms

The implementation of Program 2.6 requires the solution of a linear pro-
gram with a possibly large number of constraints (in fact, the number of
constraints is equal to the number of edges of the graph) and, therefore, it is
computationally expensive. A different approach (still based on linear pro-
gramming) allows us to obtain an approximate solution more efficiently.

   The method is known as *primal-dual* and uses the dual of the linear pro-
gram obtained by relaxing the integrality constraints. Recall that, given
a minimization linear program *LP*, its dual *DLP* is a maximization linear

program whose optimal value coincides with the optimal value of *LP* (see Appendix A). Therefore, if we consider a minimization *integer* linear program *ILP* whose relaxation provides *LP*, any feasible solution of *DLP* has a measure no greater than the optimal measure of *ILP* (which, in turn, is no greater than the value of any feasible solution of *ILP*) and can, thus, be used as a lower bound when estimating the quality of an approximate solution of *ILP* (see Fig. 2.8).

A primal-dual algorithm exploits this property to find approximate solutions of an integer linear program *ILP*: in particular, it simultaneously maintains a (possibly non-feasible) integer solution *x* of *ILP* and a (not necessarily optimal) feasible solution of *DLP*. At each step, *x* and *y* are examined and modified to derive a new pair of solutions x' and y' where x' is "more feasible" than *x* and y' has a better measure than *y*. The algorithm ends when the integer solution becomes feasible: the quality of this solution is evaluated by comparing it with the final dual solution. This approach allows us to obtain faster algorithms because it is not necessary to optimally solve either *ILP* or *DLP*. Moreover, as we will see in the rest of this section, there are cases in which the method allows us to obtain solutions with a good performance ratio.

Figure 2.8
The space of values of feasible solutions of *ILP* and *DLP*
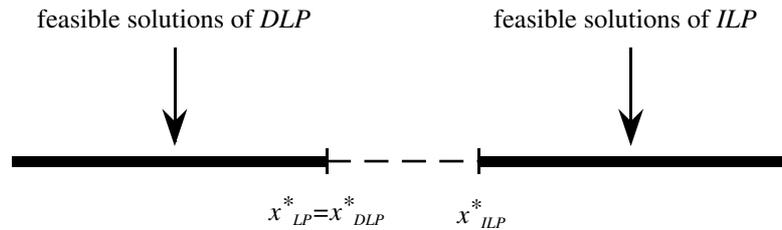
In particular, let us formulate a primal-dual algorithm for MINIMUM WEIGHTED VERTEX COVER. First observe that, given a weighted graph $G = (V, E)$, the dual of the previously defined relaxation LP_{VC} is the following linear program DLP_{VC}:

$$
\begin{aligned}
\texttt{maximize} \quad & \sum_{(v_i,v_j)\in E} y_{ij} \\
\texttt{subject to} \quad & \sum_{j:(v_i,v_j)\in E} y_{ij} \leq c_i \quad \forall v_i \in V \\
& y_{ij} \geq 0 \qquad\qquad \forall (v_i, v_j) \in E.
\end{aligned}
$$

Note that the empty set is an unfeasible integer solution of MINIMUM WEIGHTED VERTEX COVER (that is, of the initial integer linear program) while the solution in which all y_{ij} are zero is a feasible solution with value 0 of DLP_{VC}. The primal-dual algorithm starts from this pair of solutions

Program 2.7: Primal-Dual Weighted Vertex Cover

input Graph $G = (V, E)$ with non-negative vertex weights;
output Vertex cover V' of G;
begin
   Let ILP_{VC} be the integer linear programming formulation of the problem;
   Let DLP_{VC} be the dual of the linear programming relaxation of ILP_{VC};
   for each dual variable y_{ij} of DLP_{VC} **do** $y_{ij} := 0$;
   $V' := \emptyset$;
   while V' is not a vertex cover **do**
   begin
     Let (v_i, v_j) be an edge not covered by V';
     Increase y_{ij} until a constraint of DLP_{VC} becomes tight for either i or j;
     if $\sum_{k:(v_i,v_k) \in E} y_{ik} = c_i$ **then**
       $V' := V' \cup \{v_i\}$ (* the i-th dual constraint is tight *)
     else
       $V' := V' \cup \{v_j\}$ (* the j-th dual constraint is tight *)
   end;
   return V'
end.

and constructs a vertex cover (i.e., a feasible primal solution) by looking for a better dual solution (see Program 2.7).

Given a graph G with non-negative weights, Program 2.7 finds a feasible solution of MINIMUM WEIGHTED VERTEX COVER *with value $m_{PD}(G)$ such that $m_{PD}(G)/m^*(G) \leq 2$.* ◄ Theorem 2.16

Let V' be the solution obtained by the algorithm. By construction V' is a   PROOF feasible solution. For the analysis of its quality, first observe that for every $v_i \in V'$ we have $\sum_{j:(v_i,v_j) \in E} y_{ij} = c_i$. Therefore,

$$m_{PD}(G) = \sum_{v_i \in V'} c_i = \sum_{v_i \in V'} \sum_{j:(v_i,v_j) \in E} y_{ij} \leq \sum_{v_i \in V} \sum_{j:(v_i,v_j) \in E} y_{ij} = 2 \sum_{(v_i,v_j) \in E} y_{ij}.$$

Since $\sum_{(v_i,v_j) \in E} y_{ij} \leq m^*(G)$ (that is, the value of the feasible dual solution obtained by the algorithm is always at most the value of the primal optimal solution), the theorem hence follows.   QED

2.5 Dynamic programming

DYNAMIC PROGRAMMING is an algorithmic technique that, in some cases, allows us to reduce the size of the search space while looking

for an optimal solution and that, for this reason, has been applied to many combinatorial problems.

Roughly speaking, dynamic programming can be applied to any problem for which an optimal solution of the problem can be derived by composing optimal solutions of a limited set of "subproblems", regardless of how these solutions have been obtained (this is generally called the *principle of optimality*). Due to efficiency reasons, this top-down description of the technique is usually translated into a bottom-up programming implementation in which subproblems are defined with just a few indices and "subsolutions" are optimally extended by means of iterations over these indices.

In this section we will present the dynamic programming technique by first giving an (exponential-time) exact algorithm for MAXIMUM KNAPSACK and by subsequently using this algorithm to design a family of polynomial-time approximation algorithms.

In order to apply the dynamic programming technique to MAXIMUM KNAPSACK, we need to specify the notion of subproblem so that the principle of optimality is satisfied. Recall that an instance of the problem consists of a positive integer knapsack capacity b and of a finite set X of n items such that, for each $x_i \in X$, a profit $p_i \in Z^+$ and a size $a_i \in Z^+$ are specified.

For any k with $1 \le k \le n$ and for any p with $0 \le p \le \sum_{i=1}^{n} p_i$, we then consider the problem of finding a subset of $\{x_1, \ldots, x_k\}$ which minimizes the total size among all those subsets having total profit equal to p and total size at most b: we denote with $M^*(k, p)$ an optimal solution of this problem and with $S^*(k, p)$ the corresponding optimal size (we assume that, whenever $M^*(k, p)$ is not defined, $S^*(k, p) = 1 + \sum_{i=1}^{n} a_i$).

Clearly, $M^*(1, 0) = \emptyset$, $M^*(1, p_1) = \{x_1\}$, and $M^*(1, p)$ is not defined for any positive integer $p \ne p_1$. Moreover, for any k with $2 \le k \le n$ and for any p with $0 \le p \le \sum_{i=1}^{n} p_i$, the following relationship holds (this is the formal statement of the principle of optimality in the case of MAXIMUM KNAPSACK):

$$
M^*(k, p) = \begin{cases} M^*(k-1, p-p_k) \cup \{x_k\} & \text{if } p_k \le p,\ M^*(k-1, p-p_k) \\ & \text{is defined, } S^*(k-1, p) \text{ is at} \\ & \text{least } S^*(k-1, p-p_k) + a_k, \\ & \text{and } S^*(k-1, p-p_k) + a_k \le b, \\ \\ M^*(k-1, p) & \text{otherwise.} \end{cases}
$$

That is, the best subset of $\{x_1, \ldots, x_k\}$ that has total profit p is either the best subset of $\{x_1, \ldots, x_{k-1}\}$ that has total profit $p - p_k$ plus item x_k or the best subset of $\{x_1, \ldots, x_{k-1}\}$ that has total profit p. Since the best subset

Program 2.8: Dynamic Programming Maximum Knapsack

input Set X of n items, for each $x_i \in X$, values p_i, a_i, positive integer b;
output Subset $Y \subseteq X$ such that $\sum_{x_i \in Y} a_i \leq b$;
begin
  for $p := 0$ **to** $\sum_{i=1}^{n} p_i$ **do**
  begin
    $M^*(1,p) := $ undefined;
    $S^*(1,p) := 1 + \sum_{i=1}^{n} a_i$;
  end;
  $M^*(1,0) := \emptyset; S^*(1,0) := 0$;
  $M^*(1,p_1) := \{x_1\}; S^*(1,p_1) := a_1$;
  for $k := 2$ **to** n **do**
    for $p := 0$ **to** $\sum_{i=1}^{n} p_i$ **do**
    begin
      if $(p_k \leq p)$ **and** $(M^*(k-1, p - p_k) \neq$ undefined$)$
        and $(S^*(k-1, p - p_k) + a_k \leq S^*(k-1,p))$
        and $(S^*(k-1, p - p_k) + a_k \leq b)$ **then**
      begin
        $M^*(k,p) := M^*(k-1, p - p_k) \cup \{x_k\}$;
        $S^*(k,p) := S^*(k-1, p - p_k) + a_k$
      end
      else
      begin
        $M^*(k,p) := M^*(k-1,p)$;
        $S^*(k,p) := S^*(k-1,p)$
      end
    end;
  $p^* := $ maximum p such that $M^*(n,p) \neq$ undefined;
  return $M^*(n, p^*)$
end.

of $\{x_1, \ldots, x_k\}$ that has total profit p must either contain x_k or not, one of these two choices must be the right one.

From the above relationship, it is now possible to derive an algorithm that, for any instance of MAXIMUM KNAPSACK, computes an optimal solution: this algorithm is shown in Program 2.8.

Given an instance x of MAXIMUM KNAPSACK *with n items, Program 2.8* ◄ Theorem 2.17
finds an optimal solution of x in time $O(n \sum_{i=1}^{n} p_i)$ *where p_i denotes the profit of the i-th item.*

The correctness of the algorithm is implied by the principle of optimality   PROOF
in the case of MAXIMUM KNAPSACK. In order to bound the running time

Program 2.9: Knapsack Approximation Scheme

input Set X of n items, for each $x_i \in X$, values p_i, a_i, positive integer b,
   rational number $r > 1$;
output Subset $Y \subseteq X$ such that $\sum_{x_i \in Y} a_i \le b$;
begin
   $p_{max} := $ maximum among values p_i;
   $t := \lfloor \log(\frac{r-1}{r} \frac{p_{max}}{n}) \rfloor$;
   $x' := $ instance with profits $p'_i = \lfloor p_i/2^t \rfloor$;
   remove from x' items with zero profit;
   $Y := $ solution returned by Program 2.8 with input x';
   return Y
end.

QED

it is sufficient to observe that the execution of the body of both the first and
the third **for** loop of Program 2.8 requires a constant number of steps.

The running time of Program 2.8 is polynomial in the values of the prof-
its associated with the items of the instance of the problem. These values
are exponential in the length of the input if we use any reasonable encoding
scheme (in fact $\log p_i$ bits are sufficient to encode the value p_i) and for this
reason the algorithm is not a polynomial-time one. However, in order to
stress the fact that the running time is polynomial in the value of the profits,
we will say that the running time of the algorithm is *pseudo-polynomial*.

Besides being interesting by itself, Program 2.8 can be used to obtain a
polynomial-time algorithm that, given an instance x of MAXIMUM KNAP-
SACK and a bound on the desired performance ratio, returns a feasible
solution of x whose quality is within the specified bound. The algorithm
works in the following way: instead of directly solving the given instance
x, it solves an instance x' which is obtained by scaling down all profits
by a power of 2 (depending on the desired degree of approximation). In-
stance x' is then solved by Program 2.8 and from its optimal solution the
approximate solution of the original instance x is finally derived. The al-
gorithm is shown in Program 2.9: since the algorithm's behavior depends
on the required performance ratio, it is called an *approximation scheme* for
MAXIMUM KNAPSACK.

Theorem 2.18 ► *Given an instance x of* MAXIMUM KNAPSACK *with n items and a rational
number $r > 1$, Program 2.9 returns a solution in time $O(rn^3/(r-1))$ whose
measure $m_{AS}(x,r)$ satisfies the inequality $m^*(x)/m_{AS}(x,r) \le r$.*

PROOF

Let $Y(x,r)$ be the approximate solution computed by Program 2.9 with
input x and r and let $Y^*(x)$ be an optimal solution of x, with measure $m^*(x)$.

It is easy to see that, for any item inserted in $Y(x,r)$, the largest additive error introduced by pruning the profit is at most 2^t; therefore, we have that $m^*(x) - m_{AS}(x,r) \leq n2^t$. Moreover, if p_{max} is the largest profit of an item, we then have that $np_{max} \geq m^*(x) \geq p_{max}$. Hence,

$$\frac{m^*(x) - m_{AS}(x,r)}{m^*(x)} \leq \frac{n2^t}{p_{max}},$$

that is,

$$m^*(x) \leq \frac{p_{max}}{p_{max} - n2^t} m_{AS}(x,r).$$

If we take into account that $t = \lfloor \log(\frac{r-1}{r} \frac{p_{max}}{n}) \rfloor$, then we obtain $m^*(x) \leq r \cdot m_{AS}(x,r)$.

As far as the running time is concerned, we have that it corresponds to the running time of Program 2.8 with input the scaled instance, that is, $O(n \sum_{i=1}^{n} p_i') = O(n \sum_{i=1}^{n} p_i/2^t)$. Since p_{max} is the maximum profit of an item and because of the definition of t, we have that the running time of Program 2.9 is $O(rn^3/(r-1))$ and the theorem thus follows. QED

The preceding result shows that Program 2.9 is an approximation scheme whose running time is polynomial both in the length of the instance and in the required quality of the solution: in the next chapter, we will see that this kind of algorithms are called *fully polynomial-time approximation schemes*.

Moreover, observe that in the preceding proof the behavior of the algorithm heavily depends (both from the point of view of the performance ratio and from that of the running time) on the upper and lower bounds to the estimated value of the optimal value $m^*(x)$, defined by the relationship $np_{max} \geq m^*(x) \geq p_{max}$. Actually, by using a tighter bound on $m^*(x)$, a better algorithm can be derived with a different definition of t whose running time is $O(rn^2/(r-1))$ (see Exercise 2.18).

The scaling technique that we have just seen is strongly related to another technique referred to as *fixed partitioning*, which we now sketch briefly. Suppose we are given an instance x of MAXIMUM KNAPSACK and a rational number $r > 1$. We divide the range of possible values of the measure function, which is bounded by np_{max}, into $\lceil np_{max}/\delta \rceil$ equal intervals of size δ, where δ is chosen according to the desired bound r on the performance ratio. Then the dynamic programming algorithm is applied by considering the problem of finding, for any k with $1 \leq k \leq n$ and for any i with $1 \leq i \leq \lceil np_{max}/\delta \rceil$, a subset of $\{x_1, \ldots, x_k\}$ which minimizes the total size among all those subsets whose total profit belongs to interval $(\delta(i-1), \delta i]$ and whose total size is at most b.

It is then possible to modify Program 2.8 so that a solution of the problem relative to k and i can be computed by means of the solutions of a finite set of problems relative to $k-1$ and $j < i$ (see Exercise 2.21). At the end, the solution achieved by this procedure has a performance ratio at most r if $\delta \leq \frac{r-1}{r} p_{max}/n$. Indeed, each time we consider a new item (that is, k is increased), the absolute error of the current best solution may increase by at most δ. Thus, the absolute error of the returned solution is at most $n\delta$. Hence,

$$\frac{m^*(x) - m(x,Y)}{m^*(x)} \leq \frac{n\delta}{p_{max}} \leq \frac{r-1}{r},$$

that is,

$$1 - \frac{m(x,Y)}{m^*(x)} \leq \frac{r-1}{r}$$

which implies that the performance ratio is at most r. Notice that, since the number of intervals is $O(n^2 r/(r-1))$, the above procedure is a fully polynomial-time approximation scheme.

Clearly, the result is the same as we would have achieved by adopting a scaling factor δ: we are indeed dealing with two ways to look at the same technique rather than with two different techniques.

The scaling (or fixed partitioning) technique can be applied to construct fully polynomial-time approximation schemes for many other optimization problems (see Exercise 2.22).

2.6 Randomized algorithms

IN RECENT years there has been increased interest in the area of randomized algorithms, which have found widespread use in many areas of computer science. The main reason for this interest is the fact that, for many applications, a randomized algorithm is either simpler or faster (or both) than a deterministic algorithm. In this section we will consider randomized techniques for combinatorial optimization problems, and we will see an example of how they can be used to design simple approximation algorithms.

Roughly speaking, a randomized algorithm is an algorithm that during some of its steps performs random choices. Note that the random steps performed by the algorithm imply that by executing the algorithm several times with the same input we are not guaranteed to find the same solution. Actually, in the case of optimization problems we can find solutions whose measure might differ significantly. More precisely, given an instance of a combinatorial optimization problem, the value of the solution found by a

Program 2.10: Random Satisfiability

input Set C of disjunctive clauses on a set of variables V;
output Truth assignment $f : V \mapsto \{\text{TRUE}, \text{FALSE}\}$;
begin
  For each $v \in V$, independently set $f(v) = \text{TRUE}$ with probability $1/2$;
  return f
end.

randomized approximation algorithm is a random variable: when estimating the expected value of this random variable, we thus have to consider the behavior of the algorithm averaging over all possible executions.

In this section, we present a very simple randomized algorithm for MAXIMUM SATISFIABILITY (see Problem 1.16) which has a good average performance. This algorithm is shown in Program 2.10.

Let $m_{RS}(x)$ be the random variable denoting the value of the solution found by the algorithm with input x. The following result provides a bound on its expected value, assuming that all clauses in C have at least k literals.

Given an instance x of MAXIMUM SATISFIABILITY *in which all c clauses* ◄ Theorem 2.19
have at least k literals, the expected measure $m_{RS}(x)$ of the solution found by Program 2.10 satisfies the following inequality:

$$E[m_{RS}(x)] \geq \left(1 - \frac{1}{2^k}\right) c.$$

The probability that any clause with k literals is not satisfied by the truth   PROOF
assignment found by the algorithm is 2^{-k} (which is the probability that all literals in the clause have been assigned the value FALSE). Therefore the probability that a clause with at least k literals is satisfied is at least $1 - 2^{-k}$. It follows that the expected contribution of a clause to $m_{RS}(x)$ is at least $1 - 2^{-k}$. By summing over all clauses we obtain

$$E[m_{RS}(x)] \geq \left(1 - \frac{1}{2^k}\right) c$$

and the theorem follows.   QED

Given an instance x of MAXIMUM SATISFIABILITY, *the expected mea-* ◄ Corollary 2.20
sure $m_{RS}(x)$ of the solution found by Program 2.10 satisfies the following inequality:

$$m^*(x)/E[m_{RS}(x)] \leq 2.$$

The corollary derives immediately from the previous theorem and from the fact that the optimal solution has always measure at most c.

Note that the above corollary holds for all instances of the problem and that the average is taken over different executions of the algorithm. However, it does not guarantee that the algorithm *always* finds a good approximate solution.

2.7 Approaches to the approximate solution of problems

IN THE previous sections, given a computationally hard optimization problem, we have evaluated the performance of an approximation algorithm, that is, an algorithm that provides a non-optimal solution. In particular, given an instance of the problem, we have shown how far the solution found by the algorithm was from the optimal solution. Clearly, we have only been able to derive upper bounds on this distance since an exact evaluation of the performance ratio would require running both the approximation algorithm and an exact algorithm.

Several possible approaches to the analysis of approximation algorithms will be thoroughly presented in the remainder of the book. In this section we describe these approaches briefly.

2.7.1 Performance guarantee: chapters 3 and 4

In the performance guarantee approach, we require that, for all instances of the problem, the performance ratio of the solution found by the algorithm is bounded by a suitable function. In particular, we are mainly interested in the cases in which this function is a constant. As an example, Theorem 2.15 shows that, for any graph with positive weights on the vertices, Program 2.6 finds a feasible solution of MINIMUM WEIGHTED VERTEX COVER whose measure is bounded by twice the optimum (i.e., $m_{LP}(G) \leq 2m^*(G)$).

According to the performance guarantee approach we are interested in determining the algorithm with the minimum performance ratio. Note that if algorithm $\mathcal{A}_1$ has a better performance ratio than algorithm $\mathcal{A}_2$ this does not imply that, for all instances of the problem, the solution found by $\mathcal{A}_1$ is always closer to the optimal solution than the solution found by $\mathcal{A}_2$. In fact the performance guarantee approach uses the worst possible case to measure the performance ratio of the algorithm.

We have also seen examples in which we cannot prove similar results: in

the case of MINIMUM GRAPH COLORING, all the approximate algorithms we have presented have "bad" instances in which the number of colors used cannot be bounded by a linear function of the minimum number of colors. In fact, for this problem it is possible to prove a strong negative result: if there exists an approximation algorithm $\mathcal{A}$ that, for each instance x of the problem with n vertices, finds a solution with measure $m_{\mathcal{A}}(x)$ such that $m_{\mathcal{A}}(x) \leq n^c m^*(x)$, where c is any constant with $0 < c < 1/7$, then $P = NP$.

Therefore we do not expect to design a polynomial-time algorithm for MINIMUM GRAPH COLORING that always finds a good approximate solution with respect to the performance guarantee approach. Again, this does not imply that approximation algorithms for the problem with good practical behavior cannot be designed. As we will see, similar considerations apply to MAXIMUM INDEPENDENT SET as well.

For such hard problems, it is sometimes possible to analyze the behavior of approximation algorithms by properly restricting attention to specific classes of input instances. As an example, we have seen that we can bound the performance ratio of approximation algorithms for MINIMUM GRAPH COLORING when we restrict ourselves to planar graphs.

2.7.2 Randomized algorithms: chapter 5

A randomized algorithm is an algorithm that might perform random choices among different possibilities (note that no probabilistic assumption is made on the set of instances). Clearly, repeated executions of the algorithm with the same input might give different solutions with different values depending on the random choices. More precisely, for each given instance of the problem, the value of the solution found by a randomized algorithm is a random variable defined over the set of possible choices of the algorithm. As in Theorem 2.19, we are interested in studying the expected value of this random variable by obtaining bounds on the expected quality of the solution found by the algorithm. A randomized algorithm with good expected behavior is not guaranteed to find a good solution for all executions and for all problem instances: however, in most cases, randomized algorithms are simple to implement and very efficient.

2.7.3 Probabilistic analysis: chapter 9

The performance guarantee approach analyzes the quality of an algorithm on the basis of its worst possible behavior. This might not be realistic if

"bad" instances do not occur often and, for all remaining cases, the algorithm is able to find a solution that is very close to the optimum. Often we are interested in knowing the behavior of the algorithm for the set of instances that might actually arise in practice. Since, in most cases, it is impossible to define such a class of instances, we might be interested in analyzing the behavior of the algorithm with respect to the "average input" of the problem.

Before performing the analysis, we have to choose a probability distribution on the set of the instances of the problem. In some cases, an equiprobability assumption can be acceptable, and therefore the average instance is simply determined by evaluating the statistical mean over all instances of a fixed size. However, there are problems for which it is not reasonable to assume equiprobability, and it might not be clear which input distribution is realistic for practical applications.

A final remark concerning the probabilistic analysis of approximation algorithms is that it requires the use of sophisticated mathematical tools. In fact, at the present state of the art, only simple algorithms for simple input distributions have been analyzed: in most cases, an analysis of sophisticated algorithms with respect to practical applications has not been made.

2.7.4 Heuristics: chapter 10

The previously described approaches have considerably enlarged our ability to cope with NP-hard optimization problems. However, the theoretical analysis performed using these approaches is not completely satisfactory for three main reasons.

The first reason is that there exist problems that are not efficiently solvable with any of the above approaches. The second reason is that, in the solution of large instances of a problem, it is not sufficient to devise approximation algorithms that are polynomial-time bounded because we need algorithms with a low level of complexity: in these cases, even a quadratic-time algorithm might be too expensive. Finally, there exist approximation algorithms for which we are not able to prove accurate bounds using the previous approaches.

In the last chapter of the book, we will then consider *heuristics*, that is, approximation algorithms that are characterized by a good practical behavior even though the value of the solution obtained is not provably good (either from a worst case or from a probabilistic point of view). Heuristics are examples of an approach that is different from all the others already considered and that, at the same time, has shown its usefulness for impor-

tant problems.

The evaluation of a heuristic algorithm is based on executing the algorithm on a large set of instances and averaging the behavior on this set: the set of instances can be either randomly generated or based on instances arising in real problems. The 2-opt algorithm for MINIMUM TRAVELING SALESPERSON is an example. We have seen examples in which this algorithm finds a solution that is far from the optimum and we are currently unable to perform a precise probabilistic analysis of its behavior. Moreover, no current implementation of the algorithm guarantees that the running time of the algorithm is polynomial in the number of cities. However, practical experiments with this algorithm are very satisfactory: the execution of the algorithm on an extremely large set of instances shows that the observed running time is almost linear in the number of cities and that the solution found is, on average, a few percent away from the optimum.

2.7.5 Final remarks

Before concluding this chapter, we observe that there exist approaches to the approximate solution of combinatorial problems that are not considered in this book.

For example, we are not interested in "on-line" problems in which the information concerning the input is not complete before the execution of the algorithm. Indeed, in an on-line problem the input instance is disclosed to the algorithm one piece at a time and the algorithm must take decisions concerning some input variables before knowing the next piece of information.

Furthermore, since we assume that the input instance can be represented in an exact way, we are not interested in a notion of approximation such as that of real numbers with rational numbers. It is well known that round-off errors may cause serious mistakes, and a theory of errors was created with the aim of deriving reliable computations in this setting.

Finally, we will not consider stochastic optimization that deals with the solution of problems for which the information concerning an instance is given by random variables or by a stochastic process.

2.8 Exercises

Exercise 2.1 Prove that the bound of Theorem 2.1 is tight, namely, that, for any $\varepsilon > 0$, there exists an instance x of MAXIMUM KNAPSACK such that $m^*(x)/m_H(x) > (2 - \varepsilon)$.

Exercise 2.2 (*) Prove that the bounds provided by Theorems 2.2 and 2.3 are tight.

Exercise 2.3 Prove that for any constant $c > 1$ and for any $n > 3$, there exists an instance $x_{c,n}$ of MINIMUM TRAVELING SALESPERSON with n cities such that *Nearest Neighbor* achieves a solution of measure $m_{NN}(x_{c,n})$ such that $m_{NN}(x_{c,n})/m^*(x_{c,n}) > c$.

Exercise 2.4 (*) Prove that for any constant $c > 1$, there are infinitely many instances $x_{c,n}$ of MINIMUM METRIC TRAVELING SALESPERSON with n cities such that *Nearest Neighbor* achieves a solution of measure $m_{NN}(x_{c,n})$ such that $m_{NN}(x_{c,n})/m^*(x_{c,n}) > c$. (Hint: for any $i > 0$, derive an instance for which *Nearest Neighbor* finds a tour whose length is at least $(i+2)/6$ times the optimal length.)

Exercise 2.5 Consider the MINIMUM SCHEDULING ON IDENTICAL MACHINES problem. Show that the LPT rule provides an optimal solution in the case in which $l_{min} > m^*(x)/3$.

Exercise 2.6 Prove that the bound of Theorem 2.7 is tight if $p = 2$.

Exercise 2.7 Consider the variation of the LPT rule that assigns the two longest jobs to the first machine and, subsequently, applies the LPT rule to the remaining jobs. Prove that if $p = 2$ then the performance ratio of the solution provided by choosing the best solution between the one given by this variation and the one returned by the original LPT rule is strictly better than the performance ratio of the solution provided by the LPT rule.

Exercise 2.8 Prove that the bound of Theorem 2.9 is tight in the sense that no better multiplicative factor can be obtained if the additive factor is 1.

Exercise 2.9 Show that there exists an instance of MINIMUM BIN PACKING for which the number of bins used by the solution computed by *Best Fit Decreasing* is greater than the number of bins in the solution computed by *First Fit Decreasing*.

Exercise 2.10 Consider the following variant of MINIMUM BIN PACKING: the input instance is defined by a set of n items $\{x_1, x_2, \ldots, x_n\}$ whose sum is at most m. The goal is to maximize the number of items that are packed in m bins of unitary capacity. A sequential algorithm for this problems that is similar to *First Fit* considers items in the given order and tries to pack each item in the first available bin that can include it. If none of the m bins can accomodate item x_i then x_i is not packed. Prove that the above algorithm achieves a solution that packs at least $n/2$ items.

Problem 2.6: Minimum Multicover

INSTANCE: Set $S = \{s_1, \ldots, s_n\}$, collection C of multisubsets $S_1, \ldots, S_m$ of S, where each item s_i appears with multiplicity a_{ij} in S_j, weight function $w : C \mapsto \mathbb{N}$, multiplicity function $b : S \mapsto \mathbb{N}$.

SOLUTION: A subcollection $C' \subseteq C$ such that each item s_i appears at least $b(s_i)$ times in C'.

MEASURE: The overall weight of C'.

Exercise 2.11 (*) Let us consider the generalization of MINIMUM BIN PACKING to higher dimensions, known as vector packing. In this problem the size of each item x is not a single number but a d-dimensional vector $(x_1, x_2, \ldots, x_d)$; the bin capacity is also a d-dimensional vector $(1, 1, \ldots, 1)$ and the goal is to pack all items in the minimum number of bins given that the content of any given bin must have a vector sum less than or equal to the bin capacity. Show that the approximation algorithm that generalizes *First Fit* to higher dimensions achieves a solution whose measure is not greater than $(d + 1)$ times the optimal value.

Exercise 2.12 Prove that, for any graph G,

$$2\sqrt{n} \leq \chi(G) + \chi(G^c) \leq n + 1$$

where $\chi(\cdot)$ denotes the chromatic number of a graph, that is, the minimum number of colors needed to color the graph, and G^c denotes the complement graph of G. (Hint: for the second inequality, use Theorem 2.10.)

Exercise 2.13 (**) Show that Δ colors are sufficient to color a graph G with maximum degree Δ if G is neither complete nor a cycle with an odd number of vertices.

Exercise 2.14 Show that, for any sufficiently large n, there exists a 3-colorable graph with n vertices such that *Smallest Last* makes use of $\Omega(n)$ colors.

Exercise 2.15 Let us consider the special case of MAXIMUM CUT in which the required partition of the node set must have the same cardinality. Define a polynomial-time local search algorithm for this problem and evaluate its performance ratio.

Exercise 2.16 Consider Problem 2.6. Define an algorithm, based on rounding the linear programming relaxation, that finds a solution whose measure is at most p times the optimal measure, where $p = \max_i(\sum_j a_{ij})$.

Problem 2.7: Minimum Hitting Set

INSTANCE: Collection C of subsets of a finite set S.

SOLUTION: A hitting set for C, i.e., a subset $S' \subseteq S$ such that S' contains at least one element from each subset in C.

MEASURE: Cardinality of the hitting set, i.e., $|S'|$.

Problem 2.8: Maximum Integer d-dimensional Knapsack

INSTANCE: Non-negative integer $d \times n$ matrix A, vector $b \in \mathbb{N}^d$, vector $c \in \mathbb{N}^n$.

SOLUTION: A vector $x \in \mathbb{N}^n$ such that $Ax \leq b$.

MEASURE: The scalar product of c and x, i.e., $\sum_{i=1}^{n} c_i x_i$.

Exercise 2.17 (*) Consider Problem 2.7. Design a primal-dual algorithm that achieves a solution whose measure is at most k times the optimal measure, where k is the maximum cardinality of an element of C. (Hint: observe that MINIMUM HITTING SET is a generalization of MINIMUM VERTEX COVER.)

Exercise 2.18 Define a modified version of Program 2.9 in order to provide an approximation scheme for MAXIMUM KNAPSACK whose running time is $O(rn^2/(r-1))$. (Hint: make use of the greedy solution found by Program 2.1 in order to define the value of t.)

Exercise 2.19 Design a dynamic programming algorithm for finding the optimal solution of Problem 2.8.

Exercise 2.20 Design a dynamic programming algorithm for finding the optimal solution of Problem 2.9.

Exercise 2.21 Define a modified version of Program 2.8 which makes use of the fixed partitioning technique.

Exercise 2.22 Construct a polynomial-time approximation scheme for MINIMUM SCHEDULING ON IDENTICAL MACHINES in the case in which we have a fixed number of machines (that is, p is not part of the instance), by making use of the fixed partitioning technique.

Problem 2.9: Maximum Integer k-choice Knapsack

INSTANCE: Non-negative integer $n \times k$ matrices A and C, non-negative integer b.

SOLUTION: A vector $x \in \mathbb{N}^n$, function $f : \{1,\dots,n\} \mapsto \{1,\dots,k\}$ such that $\sum_{i=1}^{n} a_{i,f(i)} x_i \le b$.

MEASURE: $\sum_{i=1}^{n} c_{i,f(i)} x_i$.

2.9 Bibliographical notes

MANY BOOKS deal with the design and the analysis of algorithms for combinatorial optimization problems: we refer to [Papadimitriou and Steiglitz, 1982] for general reference on the first five sections. Moreover, we refer to the list of decision problems in [Garey and Johnson, 1979] for all NP-hardness results we have cited in this chapter and in the following ones.

It is well known that the greedy method finds an optimal solution when the set of feasible solutions defines a *matroid*, that is an independent system that satisfies the additional property that all maximal independent sets have the same cardinality (see for example [Edmonds, 1971]). [Graham and Hell, 1985] is a useful source of information on the history of the greedy algorithm.

A generalization of matroids, called greedoids, is studied in [Korte, Lovasz, and Schrader, 1991] where the authors prove that in a greedoid the greedy algorithm finds an optimal solution when the objective function is a bottleneck function.

The analysis of Program 2.2 can be found in [Halldórsson and Radhakrishnan, 1994b] where also tight bounds are shown. The analysis of *Nearest Neighbor* in the symmetric case of MINIMUM METRIC TRAVELING SALESPERSON is due to [Rosenkrantz, Stearns, and Lewis, 1977], that study the behavior of several other heuristics for the problem.

The early work of Graham on multiprocessor scheduling inaugurated the study of performance guarantee approximation algorithms: [Graham, 1966, Graham, 1969] give bounds on the performance of the *List Scheduling* algorithm and of other heuristics.

In [Matula, Marble, and Isaacson, 1972, Johnson, 1974c, Matula and Beck, 1983] the behavior of sequential algorithms for MINIMUM GRAPH COLORING is analysed. In particular the authors analyzed the algorithms

presented in Sect. 2.2 together with some other ones. Also, experimental results for some heuristics are given. The statement of Exercise 2.13 is due to [Brooks, 1941], and a constructive proof can be found in [Lovász, 1975b].

The worst case performance of *First Fit* for MINIMUM BIN PACKING is analyzed in [Johnson, 1972], while [Johnson, 1972, Baker, 1983] study *First Fit Decreasing*: the currently tightest analysis of *First Fit Decreasing* is presented in [Li and Yue, 2000]. Many more algorithms have been proposed and analyzed from the worst case point of view: we refer to [Johnson, 1974b, Johnson et al., 1974, Johnson and Garey, 1985, Karmarkar and Karp, 1982]. In particular, in [Johnson et al., 1974] the authors show that *Best Fit Decreasing* is at least as good as *First Fit Decreasing* if item sizes are at least 1/6, and that the two algorithms produce identical results if item sizes are ate least 1/5: moreover, they give examples that show that sometimes *Best Fit Decreasing* and sometimes *First Fit Decreasing* produces better results. [Garey and Johnson, 1981] gave a useful survey presenting many results concerning approximation algorithms for MINIMUM BIN PACKING and scheduling problems: the survey has since then been updated (see [Coffman, Garey, and Johnson, 1997]). The survey also presents complexity results for MINIMUM VECTOR PACKING (see Exercise 2.11 for the definition of the problem). [Garey et al., 1976] analyzed appropriate variations of *First Fit* and *First Fit Decreasing* and showed that an algorithm that generalizes *First Fit* achieves a solution that is at most $(d+7/10)m^*(x)+c$, where d is the number of dimensions and c is an absolute constant (in the one-dimensional case this reduces to the known $17/10$ bound); in the case of *First Fit Decreasing* the measure of the solution is at most $(d+1/3)m^*+c$.

An early description of local search can be found in [Dunham et al., 1961]. However, the first proposal of a local search heuristic originates from the idea of using edge exchange procedures for improving the length of a TSP tour [Croes, 1958, Lin, 1965]. For a general introduction to MINIMUM TRAVELING SALESPERSON we refer to [Lawler et al., 1985, Junger, Reinelt, and Rinaldi, 1994]. Computational results on these and other heuristics are reported in [Golden and Stewart, 1985, Johnson, 1990, Reinelt, 1994b] (see also Chap. 10).

Linear programming has had a strong influence on the design of algorithms for combinatorial optimization problems (in Appendix A an outline of the main concepts and methods of linear programming are provided). We refer to [Papadimitriou and Steiglitz, 1982] for general references and for several examples of how to use linear programming to obtain exact polynomial-time algorithms for combinatorial optimization problems. The use of linear programming for the analysis of approximation algorithms

dates back to the 1970s (see [Lovász, 1975a]). [Wolsey, 1980] shows that previously known approximation algorithms can be analyzed using linear programming. The use of linear programming in the design of approximation algorithms originates in [Bar-Yehuda and Even, 1981, Hochbaum, 1982a]: Program 2.6 was proposed in [Hochbaum, 1982a], while [Bar-Yehuda and Even, 1981] proposed Program 2.7, which is the first example of a primal-dual algorithm for obtaining an approximate solution. The above two papers analysed also approximation algorithms for the MINIMUM HITTING SET problem (see Exercise 2.17), while MINIMUM MULTICOVER was considered in [Hall and Hochbaum, 1986] (see Exercise 2.16). Recently, primal-dual algorithms have been fruitfully applied to the design of approximation algorithms for several other problems (see the survey [Goemans and Williamson, 1997]).

The algorithmic method known as dynamic programming was originally proposed by Bellman: we refer to [Dreyfus and Law, 1977] for a survey of the early applications of dynamic programming.

A reference book for the analysis and the design of randomized algorithms is [Motwani and Raghavan, 1995]. Further examples of randomized algorithms for the design of approximation algorithms will be given in Chap. 5.