# Contents

# 2
# Tools from algorithms and complexity theory

David P. Williamson
*Cornell University*

Given the taxonomy of deterministic scheduling problems in Chapter 1, we could immediately begin the discussion of their solution, but this would be a bit like a naturalist heading out into the forest with just a field guide and no equipment. Instead, we spend a brief moment in this chapter familiarizing ourselves with many tools and concepts from the field of combinatorial optimization that will be useful in designing scheduling algorithms in subsequent chapters. The field of combinatorial optimization is now sufficiently broad that any given topic we cover has already had a book written about it. Thus we will not cover any topic in depth, but rather cover the basic material that will be needed for the rest of this book. We will give references to more comprehensive treatments in the notes at the end of the chapter.

In the first part of the chapter we consider some well-studied algorithmic techniques: namely, linear programming, network flow, and dynamic programming. These techniques are sufficiently general that they can be used to solve several scheduling problems, and can be used as subroutines within algorithms for other scheduling problems. Furthermore, the techniques are efficient in practice. In the second part of the chapter, we turn to the concept of the complexity of algorithms and problems. Just as the naturalist, possessing a taxonomy of plants, would like indications of whether a plant is poisonous or not, the theory of NP-completeness helps determine whether a given scheduling problem is likely to have an efficient algorithm to solve it. We end the chapter by discussing a few techniques that can be brought to bear on the more poisonous varieties of scheduling problems.

## 2.1. Algorithmic techniques

**Linear programming.** One of the most useful tools from combinatorial optimization is *linear programming*. In linear programming, we find a non-negative, rational vector $x$ that minimizes a given linear objective function in $x$ subject to linear constraints on $x$. More formally, given an $n$-vector $c \in \mathbb{Q}^n$, an $m$-vector $b \in \mathbb{Q}^m$, and an $m \times n$ matrix $A = (a_{ij}) \in \mathbb{Q}^{m \times n}$, an optimal solution to the linear programming problem

$$
\begin{array}{llll}
 & \text{Min} & \displaystyle\sum_{j=1}^{n} c_j x_j & \\
 & \text{subject to:} & & \\
(P) & & \displaystyle\sum_{j=1}^{n} a_{ij} x_j \geq b_i \quad \text{for } i = 1, \ldots, m & (2.1) \\
 & & x_j \geq 0 \qquad\quad \text{for } j = 1, \ldots, n & (2.2)
\end{array}
$$

is an $n$-vector $x$ that minimizes the linear *objective function* $\sum_{j=1}^{n} c_j x_j$ subject to the *constraints* (2.1) and (2.2). The vector $x$ is called the *variable*. Any $x$ which satisfies the constraints is said to be *feasible*, and if such an $x$ exists, the linear program is said to be *feasible*. If there does not exist any feasible $x$, the linear program is called *infeasible*. The term “linear program” is frequently abbreviated to *LP*. Sometimes LPs are expressed more compactly in matrix/vector notation as follows:

$$
\begin{array}{c}
\text{Min } c^T x \\
\text{subject to:} \\
Ax \geq b \\
x \geq 0,
\end{array}
$$

where $c^T$ denotes the transpose of $c$. There are very efficient, practical algorithms to solve linear programs; LPs with tens of thousands of variables and constraints are solved routinely.

One could imagine variations and extensions of the linear program above: for example, maximizing the objective function rather than minimizing it, having equations in addition to inequalities, and allowing variables $x_j$ to take on negative values. However, the linear program $(P)$ above is sufficiently general that it can capture all these variations, and so is said to be in *canonical form*. To see this, observe that maximizing $\sum_{j=1}^{n} c_j x_j$ is equivalent to minimizing $-\sum_{j=1}^{n} c_j x_j$, and that an equation $\sum_{j=1}^{n} a_{ij} x_j = b_i$ can be expressed as a pair of inequalities $\sum_{j=1}^{n} a_{ij} x_j \geq b_i$ and $-\sum_{j=1}^{n} a_{ij} x_j \geq -b_i$. Finally, a variable $x_j$ which is allowed to be negative can be expressed in terms of two non-negative variables $x_j^+$ and $x_j^-$ by substituting $x_j^+ - x_j^-$ for $x_j$ in the objective function and the constraints.

Another variation of linear programming, called *integer linear programming* or *integer programming*, allows constraints requiring a variable $x_j$ to be an integer. For instance, we can require that $x_j \in \mathbb{N}$, or that $x_j$ be in a bounded range of integers, such as $x_j \in \{0,1\}$. Unlike linear programming, there is currently no efficient, practical algorithm to solve general integer programs; in fact, many quite small integer programs are very difficult to solve. In Section 2.4, we will see evidence that it is unlikely that such an algorithm can exist. Nevertheless, integer programming remains a useful tool because it is a compact way to model problems in combinatorial optimization, and because there are several important special cases that do have efficient algorithms.

To illustrate the usefulness of linear programming in solving scheduling problems, consider the problem $R|pmtn|C_{\max}$. We will show that the problem of deciding how to allocate portions of each job to each machine can be formulated as a linear program, although we defer until Chapter 9 the problem of constructing a feasible schedule from the allocations. Let the variable $C$ denote the makespan of the schedule, which we wish to minimize, and let the variable $x_{ij}$ denote the fraction of the $j$th job to be allocated to the $i$th machine. Thus for any job $j$, $\sum_{i=1}^{m} x_{ij} = 1$. The total amount of processing to be performed by machine $i$ is then $\sum_{j=1}^{n} p_{ij}x_{ij}$, so that $C \geq \sum_{j=1}^{n} p_{ij}x_{ij}$. Finally, no job can be processed for more than $C$ units of time, so that $C \geq \sum_{i=1}^{m} p_{ij}x_{ij}$. In Chapter 9, we will see that any feasible values of $x$ and $C$ can be converted into a schedule of makespan $C$. Thus we can formulate the problem as the following linear program,

$$
\begin{array}{lll}
\text{Min} & C & \\
\text{subject to:} & & \\
& \displaystyle\sum_{i=1}^{m} x_{ij} = 1 & j = 1,\ldots,n \\
(R) & C - \displaystyle\sum_{j=1}^{n} p_{ij}x_{ij} \geq 0 & i = 1,\ldots,m \\
& C - \displaystyle\sum_{i=1}^{m} p_{ij}x_{ij} \geq 0 & j = 1,\ldots,n \\
& x_{ij} \geq 0 & i = 1,\ldots,m; \quad j = 1,\ldots,n,
\end{array}
$$

and the optimal solution to the linear program, $C$, is the minimum possible makespan.

Observe that if we add the constraints $x_{ij} \in \{0,1\}$ for all $i,j$ to the LP $(R)$, then this integer program *models* the problem $R||C_{\max}$, in the sense that the optimal solution to the integer program has the same value as the optimal solution to the problem $R||C_{\max}$, and the solution $x_{ij}$ indicates which machines should process which jobs. However, adding these constraints makes the problem much harder to solve than the linear program.

Linear programming has a very interesting and useful concept of *duality*. To explain it, we begin with a small example. Consider the following linear program in canonical form:

$$\begin{array}{rll} \text{Min} & 6x_1 + 4x_2 + 2x_3 & \\ \text{subject to:} & & \\ & 4x_1 + 2x_2 + x_3 \geq 5 & \\ & x_1 + x_2 \geq 3 & \\ & x_2 + x_3 \geq 4 & \\ & x_i \geq 0 & \text{for } i = 1,2,3. \end{array}$$

Observe that because all variables $x_j$ are non-negative, it must be the case that the objective function $6x_1 + 4x_2 + 2x_3 \geq 4x_1 + 2x_2 + x_3$. Furthermore, $4x_1 + 2x_2 + x_3 \geq 5$ by the first constraint. Thus we know that the value of the objective function of an optimal solution to this linear program (called the *optimal value* of the linear program) is at least 5. We can get an improved lower bound by considering combinations of the constraints. It is also the case that $6x_1 + 4x_2 + 2x_3 \geq (4x_1 + 2x_2 + x_3) + 2 \cdot (x_1 + x_2) \geq 5 + 2 \cdot 3 = 11$, which is the first constraint summed together with twice the second constraint. Even better, $6x_1 + 4x_2 + 2x_3 \geq (4x_1 + 2x_2 + x_3) + (x_1 + x_2) + (x_2 + x_3) \geq 5 + 3 + 4 = 12$, by summing all three constraints together. Thus the optimal value of the LP is at least 12.

In fact, we can set up a linear program to determine the best lower bound obtainable by various combinations of constraints. Suppose we take $y_1$ times the first constraint, $y_2$ times the second, and $y_3$ times the third, where the $y_i$ are non-negative. Then the lower bound achieved is $5y_1 + 3y_2 + 4y_3$. We need to ensure that

$$6x_1 + 4x_2 + 2x_3 \geq y_1(4x_1 + 2x_2 + x_3) + y_2(x_1 + x_2) + y_3(x_2 + x_3),$$

which we can do by ensuring that no more than 6 copies of $x_1$, 4 copies of $x_2$, and 2 copies of $x_3$ appear in the sum; that is, $4y_1 + y_2 \leq 6$, $2y_1 + y_2 + y_3 \leq 4$, and $y_1 + y_3 \leq 2$. We want to maximize the lower bound achieved subject to these constraints, which gives the linear program

$$\begin{array}{rll} \text{Max} & 5y_1 + 3y_2 + 4y_3 & \\ \text{subject to:} & & \\ & 4y_1 + y_2 \leq 6 & \\ & 2y_1 + y_2 + y_3 \leq 4 & \\ & y_1 + y_3 \leq 2 & \\ & y_i \geq 0 & i = 1,2,3 \end{array}$$

This maximization linear program is called the *dual* of the previous minimization linear program, which is referred to as the *primal*. It is not hard to see that any feasible solution to the dual gives an objective function value that is a lower bound on the optimal value of the primal.

We can create a dual for any linear program; the dual of the canonical form LP $(P)$ above is

$$\begin{array}{lrll}
 & \text{Max} & \displaystyle\sum_{i=1}^{m} b_i y_i & \\
 & \text{subject to:} & & \\
(D) & & \displaystyle\sum_{i=1}^{m} a_{ij} y_i \le c_j & \text{for } j = 1, \ldots, n \\
 & & y_i \ge 0 & \text{for } i = 1, \ldots, m.
\end{array}$$

As in our small example, we introduce a variable $y_i$ for each linear constraint in the primal, and try to maximize the lower bound achieved by summing $y_i$ times the $i$th constraint, subject to the constraint that the variable $x_j$ not appear more than $c_j$ times in the sum. In matrix/vector notation this is

$$\begin{array}{rl}
\text{Max} & y^T b \\
\text{subject to:} & \\
 & y^T A \le c \\
 & y \ge 0.
\end{array}$$

We now formalize our argument above that the value of the dual of the canonical form LP is a lower bound on the value of the primal. This fact is called *weak duality*.

**Theorem 2.1** [ Weak duality ]. *If $x$ is a feasible solution to the LP $(P)$, and $y$ a feasible solution to the LP $(D)$, then $\sum_{j=1}^{n} c_j x_j \ge \sum_{i=1}^{m} b_i y_i$.*

*Proof.*

$$\begin{aligned}
\sum_{j=1}^{n} c_j x_j &\ge \sum_{j=1}^{n} \left( \sum_{i=1}^{m} a_{ij} y_i \right) x_j \qquad (2.3) \\
&= \sum_{i=1}^{m} \left( \sum_{j=1}^{n} a_{ij} x_j \right) y_i \\
&\ge \sum_{i=1}^{m} b_i y_i, \qquad (2.4)
\end{aligned}$$

where the first inequality follows by the feasibility of $y$, the next equality by an interchange of summations, and the last inequality by the feasibility of $x$. □

A very surprising, interesting, and useful fact is that when both primal and dual LPs are feasible, their values are exactly the same! This is sometimes called *strong duality*.

**Theorem 2.2** [ Strong duality ]. *If the LPs $(P)$ and $(D)$ are feasible, then for any optimal solution $x^*$ to $(P)$ and any optimal solution $y^*$ to $(D)$, $\sum_{j=1}^{n} c_j x_j^* = \sum_{i=1}^{m} b_i y_i^*$.*

As an example of this, for the small, three-variable LP and its dual we saw earlier, the optimal value is 14, achieved by setting $x_1^* = 0$, $x_2^* = 3$, and $x_3^* = 1$ in the primal, and $y_1^* = 0$, $y_2^* = 2$, and $y_3^* = 2$ in the dual. A proof of Theorem 2.2 is beyond the scope of this chapter, but one can be found in the textbooks on linear programming referenced in the notes at the end of the chapter.

An easy but useful corollary of strong duality is a set of implications called the *complementary slackness conditions*. Let $\bar{x}$ and $\bar{y}$ be feasible solutions to $(P)$ and $(D)$, respectively. We say that $\bar{x}$ and $\bar{y}$ obey the complementary slackness conditions if $\sum_{i=1}^m a_{ij}\bar{y}_i = c_j$ for each $j$ such that $\bar{x}_j > 0$ and if $\sum_{j=1}^n a_{ij}\bar{x}_j = b_i$ for each $i$ such that $\bar{y}_i > 0$. In other words, whenever $\bar{x}_j > 0$ the dual constraint that corresponds to the variable $x_j$ is met with equality, and whenever $\bar{y}_i > 0$ the primal constraint that corresponds to the variable $y_i$ is met with equality.

**Corollary 2.3** [ Complementary slackness ]. *Let $\bar{x}$ and $\bar{y}$ be feasible solutions to the LPs $(P)$ and $(D)$, respectively. Then $\bar{x}$ and $\bar{y}$ obey the complementary slackness conditions if and only if they are optimal solutions to their respective LPs.*

*Proof.* If $\bar{x}$ and $\bar{y}$ are optimal solutions, then by strong duality the two inequalities (2.3) and (2.4) must hold with equality, which implies that the complementary slackness conditions are obeyed. Similarly, if the complementary slackness conditions are obeyed, then (2.3) and (2.4) must hold with equality, and it must be the case that $\sum_{j=1}^n c_j\bar{x}_j = \sum_{i=1}^m b_i\bar{y}_i$. By weak duality, $\sum_{j=1}^n c_j x_j \geq \sum_{i=1}^m b_i y_i$ for any feasible $x$ and $y$ so therefore $\bar{x}$ and $\bar{y}$ must be optimal. □

So far we have only discussed the case in which the LPs $(P)$ and $(D)$ are feasible, but of course it is possible that one or both of them are infeasible. The following theorem tells us that if the primal is infeasible and the dual is feasible, the dual must be *unbounded*: that is, given a feasible $y$ with objective function value $z$, then for any $z' > z$ there exists a feasible $y'$ of value $z'$. Similarly, if the dual is infeasible and the primal is feasible, then the primal is unbounded: given feasible $x$ with objective function value $z$, then for any $z' < z$ there exists a feasible $x'$ with value $z'$. If an LP is not unbounded, we say it is *bounded*.

**Theorem 2.4.** *For primal and dual LPs $(P)$ and $(D)$, one of the following four statements must hold: $(i)$ both $(P)$ and $(D)$ are feasible; $(ii)$ $(P)$ is infeasible and $(D)$ is unbounded; $(iii)$ $(P)$ is unbounded and $(D)$ is infeasible; or $(iv)$ both $(P)$ and $(D)$ are infeasible.*

Sometimes in the design of scheduling algorithms it is helpful to take advantage of the fact if an LP is feasible, there exist feasible solutions of a particular form, called *basic* solutions. Furthermore, if an optimal solution exists, then there exists an optimal solution that is basic. Most linear programming algorithms will return a basic optimal solution. Suppose for a moment that in the canonical primal LP, there are more variables than constraints, that is, $n \geq m$. A basic solution to the LP is obtained by setting $n - m$ of the variables $x_j$ to zero, treating the inequalities as equalities, and solving the resulting $m \times m$ linear system (assuming the system

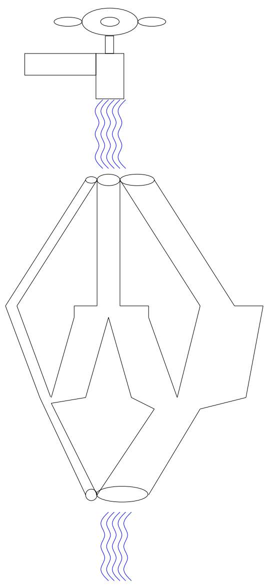

**Figure 2.1.** Example of a maximum flow problem.

is consistent and the given $m$ columns are linearly independent). In fact, the oldest and most frequently used linear programming algorithm, called the *simplex method*, works by moving from basic solution to basic solution, at each step swapping a variable set to zero for a variable in the linear system in a particular manner until an optimal solution is reached. When there are more constraints than variables ($m \geq n$), a basic solution is obtained by selecting $n$ of the constraints, treating them as equalities, and solving the resulting $n \times n$ linear system (assuming the system is consistent and the $n$ constraints are linearly independent). The solution obtained might not be feasible (since we ignored some constraints), but if an optimal solution exists, there will exist one of this form.

**Network flow.** We now turn to another useful tool from combinatorial optimization, called *network flow*. An example of the most fundamental problem in this area is shown in Figure 2.1. We have a source of fluid and a destination for it joined by a network of pipes, each pipe with its own capacity. We would like to know at what rate we can send a flow of fluid from the source to the destination given the capacity of the pipes. The problem is usually abstracted as a directed graph $G = (V, E)$, with two distinguished nodes, a *source* node $s$ and a *sink* node $t$, such that no arc enters the source, and no arc leaves the sink. A *capacity* $u_{ij}$ is associated with each arc $(i, j)$ of the directed graph (see Figure 2.2). This *maximum flow problem* is used to model flow in pipes, traffic on streets, and the movement of goods via various modes of transportation, among other things.

It is not hard to see that the maximum flow problem can be modelled as a lin-

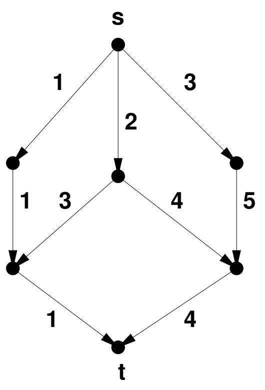

**Figure 2.2.** Abstraction of the maximum flow problem in Figure **??**.

ear program. Create a variable $x_{ij}$ for each arc $(i,j)$ to denote the flow on the arc. Then we wish to maximize the total flow out of the source, $\sum_{j\in V:(s,j)\in E} x_{sj}$, subject to two types of constraints. First, for any arc $(i,j)$ the flow on arc $(i,j)$ must not exceed its capacity; that is, $x_{ij} \le u_{ij}$. Second, for any node $k \neq s,t$, the flow coming into node $k$ must be equal to the flow going out of node $k$; that is, $\sum_{i\in V:(i,k)\in E} x_{ik} - \sum_{j\in V:(k,j)\in E} x_{kj} = 0$. The first type of constraint is called a *capacity constraint*, and the second type is called a *flow conservation constraint*. The maximum flow problem can thus be modelled as the linear program

$$
\begin{array}{llll}
 & \text{Max} & \displaystyle\sum_{j\in V:(s,j)\in E} x_{sj} & \\
 & \text{subject to:} & & \\
(MF) & & \displaystyle\sum_{i\in V:(i,k)\in E} x_{ik} - \sum_{j\in V:(k,j)\in E} x_{kj} = 0 & k \in V : k \neq s,t \\
 & & 0 \le x_{ij} \le u_{ij} & (i,j) \in E,
\end{array}
$$

and solving the linear program gives a maximum flow.

Although the maximum flow problem and the other network flow problems considered here can all be modelled as linear programs, we consider them separately for two reasons: first, they form an extremely useful subclass of linear programs; and second, as we discuss at more length later on, there are special-purpose algorithms for network flow problems that are much more efficient than the general linear programming algorithms.

Unlike most linear programs, the maximum flow LP above has the property that if the capacities $u$ are integer, then the basic solutions $x$ are also integer. In particular, if an optimal solution exists, then there is an optimal solution such that the values of the $x_{ij}$ are integer. In other words, if the capacities $u$ are integer, then adding integrality constraints $x_{ij} \in \mathbb{Z}$ does not change the optimum value. This turns out to be true for all the network flow problems we discuss in this section, and, as we will see, this is a useful fact for designing scheduling algorithms.

Network flow problems also turn out to have interesting combinatorial dual problems, which are sometimes useful in their own right. For example, there is a natural combinatorial structure to the maximum flow problem that gives upper bounds on the amount of flow we can send from $s$ to $t$. Let $S$ be a set of vertices containing $s$ but not $t$. Let $u(S)$ denote the total capacity of all the arcs with their tails in $S$ and their heads in $\bar{S}$; that is, $u(S) = \sum_{i\in S, j\in \bar{S}} u_{ij}$. We call $S$ an *s-t cut* of the graph, and $u(S)$ the *value* of the *s-t* cut. It is not difficult to see that for any *s-t* cut $S$ the total amount of flow going from $s$ to $t$ cannot exceed $u(S)$. The strongest upper bound on the flow is then a *minimum s-t cut* – the *s-t* cut $S$ that minimizes $u(S)$.

Of course, the maximum flow problem also has a linear programming dual, which is as follows:

$$\begin{array}{llll} & \text{Min} & \displaystyle\sum_{(i,j)\in E} u_{ij} z_{ij} & \\ & \text{subject to:} & & \\ & & z_{ij} + y_j - y_i \geq 0 & \text{for } (i,j) \in E; i \neq s,t; j \neq s,t \\ (MFD) & & z_{sj} + y_j \geq 1 & \text{for } (s,j) \in E \\ & & z_{it} - y_i \geq 0 & \text{for } (i,t) \in E \\ & & z_{ij} \geq 0 & \text{for } (i,j) \in E. \end{array}$$

We will shortly prove that in fact this linear programming dual corresponds to the minimum *s-t* cut problem, so that the value of the linear programming dual and the combinatorial dual are precisely the same.

**Lemma 2.5.** *The value of the dual LP $(MFD)$ is exactly the value of a minimum s-t cut.*

By strong duality, this immediately implies the following celebrated *max flow/min cut theorem.*

**Theorem 2.6.** *The value of a maximum flow in a directed graph is the same as the value of its minimum s-t cut.*

This theorem can be extended to undirected graphs. The maximum flow in an undirected graph is the maximum flow in the directed graph obtained by replacing each undirected edge $(i,j)$ of capacity $u_{ij}$ with two directed arcs $(i,j)$ and $(j,i)$ of capacity $u_{ij}$. We remove arcs entering $s$ and leaving $t$. The capacity of an *s-t* cut $S$ in an undirected graph is simply the sum of the capacities of all edges with one endpoint in $S$ and the other not in $S$. Then the theorem holds as before.

We now prove Lemma 2.5.

*Proof of Lemma 2.5.* We prove equality by proving first that the value of LP is no greater than the value of a minimum *s-t* cut, and then the reverse. Given a minimum

$s$-$t$ cut $S$, we create a solution to the LP $(MFD)$ of value $u(S)$ by setting $z_{ij} = 1$ if $i \in S, j \notin S$, and $z_{ij} = 0$ otherwise. We set $y_i = 1$ if $i \in S$ and $y_i = 0$ otherwise. It is easy to verify that $(y,z)$ is a feasible solution of value $u(S)$, so that the value of the LP must be no greater than that of a minimum $s$-$t$ cut.

We now prove that there exists an $s$-$t$ cut of value no greater than the value of an optimal solution $(y^*,z^*)$. For notational convenience, we add variables $y^*_s = 1$ and $y^*_t = 0$. We select a value $U$ uniformly at random from the interval $(0,1]$, and use it to create an $s$-$t$ cut $S$: if $y^*_i \geq U$, then $i \in S$, otherwise $i \notin S$. Obviously $s \in S$ and $t \notin S$. The probability that arc $(i,j)$ ends up in the cut is then $\max(0,\min(y^*_i,1) - \max(y^*_j,0))$, which is no greater than $z^*_{ij}$ by the feasibility of LP solution $y^*$. Thus the expected value of the $s$-$t$ cut produced is at most $\sum_{(i,j)\in E} u_{ij}z^*_{ij}$. Therefore, there exists an $s$-$t$ cut of value at most $\sum_{(i,j)\in E} u_{ij}z^*_{ij}$, and we are done. $\square$

We can use the maximum flow problem to determine whether or not a feasible solution exists for the problem $P|pmtn,r_j,\bar{d}_j|-$. In Chapter 9, we will see a rule by McNaughton that shows that a schedule of makespan $T$ can be constructed for the problem $P|pmtn|C_{\max}$ if and only if $T \geq \max\{\max_j p_j, \frac{1}{m}\sum_{j=1}^n p_j\}$, where $m$ is the number of machines. We now show how the maximum flow problem can be used to reduce the feasibility of $P|pmtn,r_j,\bar{d}_j|-$ to McNaughton's rule. First, sort the release dates $r_j$ and the deadlines $\bar{d}_j$ into increasing order, and obtain a list of times (without repetitions) $0 = t_0 < t_1 < t_2 < \cdots < t_q$, such that for every $r_j$ and $\bar{d}_j$ there exists some $k$ and $l$ such that $t_k = r_j$ and $t_l = \bar{d}_j$. Construct a network with a source node $s$, a sink node $t$, $n$ nodes $j$ (one for each job $j$), and $q$ nodes $T_k$ (one for each time interval $[t_{k-1},t_k]$). For each job $j$, add an arc $(s,j)$ of capacity $p_j$ to the network, and for each node $T_k$, add an arc $(T_k,t)$ of capacity $m(t_k - t_{k-1})$. Finally, for each job $j$, let $k$ and $l$ be such that $t_k = r_j$ and $t_l = \bar{d}_j$, and for each $h$, $k+1 \leq h \leq l$, add an arc $(j,T_h)$ of capacity $t_h - t_{h-1}$. We call this network $N$, and we can show the following theorem.

**Theorem 2.7.** *There is a feasible solution to an instance of $P|pmtn,r_j,\bar{d}_j|-$ if and only if the corresponding network $N$ has maximum flow value exactly $\sum_{j=1}^n p_j$.*

*Proof.* Given a schedule, we can construct a flow of the required value. On each arc $(s,j)$ put a flow of value of value $p_j$, while on each arc $(j,T_k)$ put a flow of value equal to the amount of time job $j$ was processed in time interval $[t_{k-1},t_k]$. Clearly the capacity constraints for these arcs are obeyed, since the capacity of arc $(s,j)$ is $p_j$, and the capacity of arc $(j,T_k)$ is $t_k - t_{k-1}$. For each arc $(T_k,t)$ put a flow of value equal to the amount of processing performed by the $m$ machines in time interval $[t_{k-1},t_k]$. This flow cannot have value more than $m(t_k - t_{k-1})$, and so the capacity constraint on arc $(T_k,t)$ is obeyed. The flow is of value $\sum_{j=1}^n p_j$, since that is the total amount of flow out of the source $s$. The flow conservation constraints are obeyed at each node: for each node $j$, $p_j$ units of flow enter from the source $s$, and $p_j$ units leave since $p_j$ units of time are scheduled for job $j$ overall. Similarly, for node $T_k$, the total flow entering and leaving $T_k$ is equal to the amount of processing performed during the corresponding time interval.

In a similar fashion, we can show that given a flow of value $\sum_{j=1}^{n} p_j$, we can construct a feasible schedule by using McNaughton's rule. Let $p_j^k$ be the amount of flow on arc $(j, T_k)$ (and 0 if the arc does not exist). Since the flow has value $\sum_{j=1}^{n} p_j$, there must be flow of value $p_j$ on each arc $(s, j)$, so that by flow conservation, $\sum_{k=1}^{q} p_j^k = p_j$ for all jobs $j$. By the capacity constraints on arcs $(j, T_k)$ we know that $p_j^k \leq t_k - t_{k-1}$ for each $j$ and $k$, and by capacity constraints on arcs $(T_k, t)$, we know that $\sum_{j=1}^{n} p_j^k \leq m(t_k - t_{k-1})$ for each $k$. Thus by McNaughton's rule, during the time interval $t_k - t_{k-1}$ we can construct a schedule in which $p_j^k$ units of job $j$ are feasibly scheduled, since $t_k - t_{k-1} \geq \max\{\max_j p_j^k, \frac{1}{m}\sum_{j=1}^{n} p_j^k\}$. Putting these schedules together gives an overall feasible schedule for the instance from time 0 to time $t_q$, since for each job $j$, $p_j$ units are scheduled, and by the construction of the network, they must be scheduled between time $r_j$ and $\bar{d}_j$. $\square$

There are other network flow problems which are useful for solving scheduling problems. In Chapter 9 we will look at a variation on the maximum flow problem in which the capacities of the arcs leaving the source are a linearly increasing function of a parameter $\lambda$ (that is, $u_{sj} = a_{sj} + \lambda \cdot b_{sj}$, where $a_{sj}, b_{sj} \geq 0$), and the capacities of the arcs entering the sink are a linearly decreasing function of $\lambda$ (that is, $u_{it} = a_{it} - \lambda \cdot b_{it}$, where $a_{it}, b_{it} \geq 0$). In the *parametric maximum flow problem*, we would like to compute the value of the maximum flow for $k$ different non-negative values of $\lambda$. Of course, this can be done by computing a maximum flow $k$ times, but it turns out that there are more efficient algorithms. We discuss this further later in the chapter.

One of the most general network flow problems assigns costs to the arcs of the directed graph, so that sending a unit of flow through arc $(i, j)$ costs $c_{ij}$ units. The objective function is then to find the flow that minimizes the total cost, and so is called the *minimum-cost flow problem*. For each node $i$ in the directed graph there is a specified *supply* value $b_i$, which must be the difference between the flow leaving node $i$ and the flow entering $i$. Those nodes for which $b_i > 0$ are called *sources* and those for which $b_i < 0$ are called *sinks*. Note that in order for the problem to have a feasible solution, it must be the case that $\sum_{i \in V} b_i = 0$. Arcs $(i, j)$ can have lower bounds $l_{ij}$ in addition to capacities $u_{ij}$: that is, there must be at least $l_{ij}$ units of flow through arc $(i, j)$. The problem can then be formulated as the following linear program:

$$\begin{array}{lll}
\text{Min} & \displaystyle\sum_{(i,j)\in E} c_{ij} x_{ij} & \\
\text{subject to:} & & \\
& \displaystyle\sum_{(k,j)\in E} x_{kj} - \sum_{(i,k)\in E} x_{ik} = b_k & \forall k \in V \\
& l_{ij} \leq x_{ij} \leq u_{ij} & \forall (i,j) \in E.
\end{array}$$

As in the case of the maximum flow problem, whenever the supplies $b_i$, the capacities $u_{ij}$, and the lower bounds $l_{ij}$ are integer, the basic solutions $x$ of this linear program are also integer.

One particularly useful special case of the minimum-cost flow problem is when the network is a complete directed bipartite graph, with arcs from sources to sinks, each source with supply 1 and each sink with supply $-1$, and $l_{ij} = 0$ and $u_{ij} = \infty$ for each arc. The assumption that the supplies sum to zero implies that there must be the same number of sources and sinks. Since $b$, $l$, and $u$ are integer, a basic optimal solution $x$ must be integer, and thus for each source $i$, there is exactly one sink $j$ such that $x_{ij} = 1$ (for all other sinks $k \neq j$, $x_{ik} = 0$). That is, each source is assigned to exactly one sink, and hence this special case is called the *assignment problem*, and solutions to this problem are called *assignments*.

We can use the assignment problem to solve the scheduling problem $1|p_j = 1|\sum_j f_j$. For each of the $n$ jobs, we create a source and a sink node. The $j$th source represents the $j$th job, and the $k$th sink represents the $k$th position in the sequence of jobs. Since the job in the $k$th position will complete at time $k$, the cost of the $j$th job finishing in the $k$th position is $c_{jk} = f_j(k)$. Thus finding the minimum-cost assignment gives a sequence of the jobs that minimizes $\sum_{j=1}^{n} f_j(C_j)$.

Another variant of the minimum-cost flow problem is the *transportation problem*; although we will not prove it, this variant is not a special case, but is entirely equivalent to the minimum-cost flow problem. As in the assignment problem, the directed graph is a complete directed bipartite graph, with arcs from sources to sinks, and $l_{ij} = 0$ and $u_{ij} = \infty$ for each arc. In the transportation problem, however, the sources have arbitrary positive integer supplies, and the sinks have arbitrary negative integer supplies. A variation of this problem allows the amount of flow entering each sink $i$ to be at most $|b_i|$, rather than exactly $|b_i|$. In this case $\sum_{i \in V} b_i \leq 0$ in order for the problem to have a feasible solution.

We can use this variation of the transportation problem to extend our algorithm for the problem $1|p_j = 1|\sum_j f_j$ to an algorithm for $P|p_j = 1|\sum_j f_j$. As before, we create a source node of supply 1 for each of the $n$ jobs. If we have $m$ machines, then since the schedule will have no idle time, every job will complete by time $\lceil n/m \rceil$, and so we create a sink node for each of the $\lceil n/m \rceil$ possible times at which a job can complete. Since at most $m$ jobs can be scheduled to complete at a given time $k$, we set $b_k = -m$ for sink $k$, and enforce that at most $m$ units of flow can enter any sink. Since the cost of the $j$th job finishing at time $k$ is $f_j(k)$, we set $c_{jk} = f_j(k)$. A basic optimal solution $x$ to this transportation problem is integer, so for each $j$, $x_{jk} = 1$ for some sink $k$ (and 0 for all others). Thus we can construct a schedule of the same cost as the flow by scheduling job $j$ at time $k$ on some machine. Because $\sum_{j=1}^{n} x_{jk} \leq m$, the schedule uses at most $m$ machines at any given time $k$.

The area of network flows is very rich, and there are many other flow variants – such as generalized flow and multicommodity flow – whose details are not as pertinent to the topic of this book. The notes at the end of the chapter contain suggested further reading.

**Dynamic programming.** We now turn to *dynamic programming*, which is a technique for solving problems, rather than a class of problems, as in linear programming and network flows. In particular, it is a method for solving *multi-stage decision problems*, in which choices must be made in a sequence of stages. The underlying idea is that for such problems we can sometimes divide the decision in the $j$th stage into several subproblems. These subproblems should have the property that given the optimum solutions to the subproblems in the $(j-1)$st stage, we can easily compute the optimum solution to the subproblems in the $j$th stage. One of the subproblems of the $n$th stage is the overall problem we want to solve, and thus the optimum sequence of choices can be found.

Usually a dynamic programming algorithm for a problem is embodied in a multidimensional array $A(j,\cdots)$, with the additional coordinates indexing the subproblems for the $j$th stage. Boundary conditions specify the values of $A(0,\cdots)$, and given the values of $A(j-1,\cdots)$, the values of $A(j,\cdots)$ are easily computed. The desired value is one of the entries of $A(n,\cdots)$.

To illustrate this method, we will consider how it applies to the *knapsack problem*. In the knapsack problem, we are given $n$ items, each of which has a size $s_j \geq 0$ and an integer value $v_j \geq 0$. We are also given a knapsack of capacity $B$. We assume $s_j \leq B$ for all items $j$. The goal is to find the subset of items of the greatest total value that can be placed in the knapsack; that is, the sum of their sizes must not exceed the capacity of the knapsack. The problem can be viewed as a multi-stage decision problem, since we can consider the items in order, from 1 to $n$, and decide whether or not to include the $j$th item in the knapsack.

Let us see how we can apply the dynamic programming technique to solve this problem. The main difficulty is breaking the decision for the $j$th stage into useful subproblems. In this case we consider the subproblems $(j,s)$ of finding the most valuable set of items in $\{1,\ldots,j\}$ that use total size, or space, no more than $s$. We store the value of the most valuable such set in a two-dimensional array $A(j,s)$, where the first coordinate will range over the items 1 to $n$, and the second coordinate will range over the possible total sizes of the items in the knapsack (from 0 to $B$). Supposing that we can compute the values in the array $A(j,s)$, then the value of the optimum solution is the most valuable subset of all the items that fits in space at most $B$; that is, the optimum value is $A(n,B)$.

Now let us see how to set the boundary conditions and how to compute $A(j,\cdot)$ from $A(j-1,\cdot)$. Certainly $A(0,s)=0$ for any $s$ between 0 and $B$; that is, by using no items, the most valuable set of items that uses space at most $s$ has value 0. Now suppose that we know the values of $A(j-1,s)$ for $0 \leq s \leq B$, and we want to compute $A(j,s)$. To achieve the most valuable subset of items from $\{1,\ldots,j\}$ that has size at most $s$, either the $j$th item is used or it is not used. If it is not used, the most valuable subset of items from $\{1,\ldots,j\}$ having space at most $s$ must be the same as the most valuable subset of items from $\{1,\ldots,j-1\}$ having space at most $s$; that is, $A(j,s)=A(j-1,s)$. If the $j$th item is used, then the value of the subset of items is $v_j$ plus the most valuable subset of items from $\{1,\ldots,j-1\}$ that occupies space at most $s-s_j$ if this amount of space is non-negative; that is, $A(j,s)=v_j+A(j-1,s-s_j)$ if $s-s_j \geq 0$. The optimum choice of whether to use item $j$ or not is whichever maximizes the overall value; that is, $A(j,s)=\min[A(j-1,s),v_j+A(j-1,s-s_j)]$

```
For $s \leftarrow 0$ to $B$
    $A(0,s) \leftarrow 0$
For $j \leftarrow 1$ to $n$
    For $s \leftarrow 0$ to $B$
        if $s_j \leq s$
            $A(j,s) \leftarrow \min(A(j-1,s), v_j + A(j-1,s-s_j))$
        else
            $A(j,s) \leftarrow A(j-1,s)$.
```

**Figure 2.3.** Dynamic programming algorithm for the knapsack problem.

(if $s - s_j \geq 0$, otherwise $A(j,s) = A(j-1,s)$).

Given this discussion, the algorithm for computing the value of an optimal solution is straightforward, and we give it in Figure 2.3. If we would like to know the set of items that gives this value, we can compute it from the array $A(j,s)$ by backtracking through the decisions that were made. We leave this as an exercise for the reader.

We can use this algorithm to solve the scheduling problem $1|\bar{d}_j = d|\sum_j w_j U_j$, since this scheduling problem is just a knapsack problem in which the capacity of the knapsack is $B = d$, and each job corresponds to an item of size $s_j = p_j$ and value $v_j = w_j$. Minimizing the total weight of late jobs is equivalent to maximizing the total weight of jobs that complete before the deadline $d$, so that a maximum-weight set of jobs $S$ that complete by time $d$ corresponds to a maximum-weight set of items whose total size is no more than $d$. In Chapter 5 we will see that the dynamic programming algorithm for the knapsack problem can be extended to solve the more general problem $1||\sum_j w_j U_j$.

We can improve on the algorithm of Figure 2.3 by observing that for a given $j$, it is possible that $A(j,s)$ is the same for many consecutive values of $s$; that is, the most valuable subset of items of $\{1,\ldots,j\}$ that uses size at most $s$ is the same for sizes $s = s'$ up to $s''$. Then rather than storing an entry for every value of $s$, we can create a new array $A'$, where $A'(j)$ contains a list $(t_1,w_1),(t_2,w_2),\ldots,(t_k,w_k)$, with the understanding that $A(j,s) = w_i$ for $t_i \leq s < t_{i+1}$ (where $t_{k+1} = B+1$); in other words, for each pair $(t_i,w_i)$ there is a subset of items in $\{1,\ldots,j\}$ that has value $w_i$ and uses space at most $t_i$. Since it is the case that $t_1 < t_2 < \cdots < t_k$ and $w_1 < w_2 < \cdots < w_k$, the number of elements in the list is no more than $B+1$ and no more than one plus the maximum possible value of the knapsack. Certainly the maximum possible value of the knapsack is no more than $V = \sum_{j=1}^n v_j$. So the length of the list is at most $\min(B,V)+1$.

Given the list for $A'(j-1)$, it is easy to compute the list for $A'(j)$. For each pair $(t_i,w_i)$ in the list $A'(j-1)$, we create a new pair $(t_i + s_j, w_i + v_j)$ if $t_i + s_j \leq B$, since if it is possible to have a knapsack of value $w_i$ using space at most $t_i$ using a subset of items from $\{1,\ldots,j-1\}$, one can have a knapsack of value $w_i + v_j$ using space at most $t_i + s_j$ using items from $\{1,\ldots,j\}$ by including item $j$ to the previous knapsack.

```
$A'(0) \leftarrow \{(0,0)\}$
for $j \leftarrow 1$ to $n$
    for each pair $(t,w)$ in $A'(j-1)$
        if $t + s_j \leq B$
            add new pair $(t+s_j, w+v_j)$ to $A'(j)$
    merge pairs from $A'(j-1)$ into $A'(j)$
    discard dominated pairs from $A'(j)$
```

**Figure 2.4.** Another dynamic programming algorithm for the knapsack problem.

We then merge the list of old pairs and new pairs to obtain a list $(t'_1, w'_1), \ldots, (t'_l, w'_l)$ with $t'_1 \leq t'_2 \leq \cdots t'_l$. We then check to see whether we can discard some pairs, since in the final list it must be the case that $t'_1 < t'_2 < \cdots < t'_l$ and $w'_1 < w'_2 < \cdots < w'_l$. Thus we discard any *dominated* pair $(t'_i, w'_i)$; that is, any pair $(t'_i, w'_i)$ such that there exists another pair $(t'_k, w'_k)$ of value at least as great that uses no more space ($w'_k \geq w'_i$ but $t'_k \leq t'_i$). Clearly the resulting list is correct: if $(t'_i, w'_i)$ is on the list, we can pack items from a subset of $\{1, \ldots, j\}$ of value $w'_i$ in space $t'_i$. If we can pack items from $\{1, \ldots, j\}$ of value $w$ in space $t$, then either the packing uses item $j$ or not; if not, the old list of $A'(j-1)$ implied that value $w$ could be packed in space $t$, and if so, value $w - v_j$ could be packed in space $t - s_j$ which was implied by the old list of $A'(j-1)$, and therefore one of the new pairs implies that value $w$ can be packed in space $t$. We summarize the new algorithm in Figure 2.4.

## 2.2. Analysis of algorithms

So far we have had little to say about the efficiency of the algorithmic techniques described above, other than to give assurances that the techniques are efficient in practice. In this section we would like to make those assurances somewhat more precise, to the extent possible.

The caveat "to the extent possible" is necessary. When we have given algorithms, they have been in English-like descriptions rather than specific computer programs, since the algorithm is independent of a particular implementation of it, just as the text of a play is independent of a particular staging. Yet of course the particular implementation (computer language, choice of data structures, etc.) will affect the efficiency of the algorithm. Even a particular implementation on a specific machine will be affected by the compiler used, and even a given compilation of a particular program targeted to a particular computer architecture will be affected by issues such as cache size, memory latency, disk speed, and so forth.

Thus although we could in principle state how many "instructions" – arithmetic operations, memory fetches and stores, comparisons, branches, etc. – need to be executed by the English-like statements of our algorithms on a particular input, this might not correspond precisely to the amount of time taken by a computer running some implementation of the algorithm on that input. Furthermore, the time taken by each type of instruction might differ. For these reasons, we discuss the efficiency of algorithms in terms of their *order of growth*, a somewhat cruder measure than precise instruction counts. The order of growth indicates how the running time of an algorithm varies as the size of its input varies (e.g. the number of jobs, the number of machines, the size of the jobs). The order of growth of an algorithm is expressed in terms of *big-oh notation*, which we define as follows.

**Definition 2.8** [ Big-oh notation ]. *Given two functions $f,g : \mathbb{N} \to \mathbb{N}$, we say that $f(n) = O(g(n))$ if there exist some positive constants $c, n_0$ such that $f(n) \le c \cdot g(n)$ for all $n \ge n_0$.*

Thus if an algorithm for a scheduling problem with $n$ jobs and $m$ machines executes at most $f(n,m) = 8nm^2 + 5nm + 6n + 3m + 2$ instructions, we simplify this using big-oh notation by saying that it takes $O(nm^2)$ *time*, or that its *running time* is $O(nm^2)$. Sometimes we refer to the instruction counts as the *time complexity* of the algorithm.

When specifying the time complexity of an algorithm, we usually refer to its *worst-case* behavior. For instance, one might have an algorithm that executes $f(n)$ instructions, where

$$f(n) = \begin{cases} c_1 n & \text{if } n \text{ composite} \\ c_2 n^2 & \text{if } n \text{ prime.} \end{cases}$$

Then $f(n) = O(n^2)$, but it is not the case that $f(n) = O(n)$. Hence we say that the algorithm takes $O(n^2)$ time.

Let us now examine the time complexity of the algorithms in Figure 2.3 and 2.4. First, we consider the algorithm of Figure 2.3. Lines 1–2 initialize the array $A$; each time through the loop takes at most some constant $c_1$ number of instructions; hence these lines execute in $O(B)$ time. Lines 3–8 calculate the array. The instructions in lines 5–8 take at most some constant $c_2$ number of instructions, which are executed $nB$ times, so that lines 3–8 take $O(nB)$ time. The overall algorithm executes at most $c_1 B + c_2 nB$ instructions; thus the running time is $O(nB)$. Now consider the algorithm of Figure 2.4. Line 1 takes a constant $d_1$ number of instructions. Lines 2–7 calculate the array. Lines 4–5 take a constant $d_2$ number of instructions, and are executed $nL$ times, where $L$ is the maximum length of any list $A'(j)$. We argued earlier that $L \le \min(B,V) + 1$, so that lines 2–5 take $O(n \min(B,V))$ time. It is possible to implement lines 6 and 7 in $d_3 L$ instructions, so that overall lines 2–7 take $O(n \min(B,V))$ time. Thus the overall running time of the algorithm in Figure 2.4 is $O(n \min(B,V))$ time.

Sometimes it is useful to give a lower bound on the order of growth of the running time of an algorithm. This can be done using *big-omega* notation.

**Definition 2.9** [ Big-omega notation ]. *Given two functions $f,g : \mathbb{N} \to \mathbb{N}$, we say that $f(n) = \Omega(g(n))$ if there exist some positive constants $c, n_0$ such that $f(n) \ge c \cdot g(n)$ for all $n \ge n_0$.*

The reader can verify that the knapsack algorithm in Figure 2.3 takes $\Omega(nB)$ time. Recall our hypothetical algorithm that takes $c_1 n$ instructions when $n$ is composite and $c_2 n^2$ when $n$ is prime. This algorithm takes $\Omega(n)$ time, but not $\Omega(n^2)$ time.

Sometimes it is useful to capture the fact that, as in the case of the knapsack algorithm of Figure 2.3, both $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$; we do this with *big-theta* notation.

**Definition 2.10** [ Big-theta notation ]. *Given two functions $f, g : \mathbb{N} \to \mathbb{N}$, we say that $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.*

Since the knapsack algorithm in Figure 2.3 takes $O(nB)$ time and $\Omega(nB)$ time, it takes $\Theta(nB)$ time.

Big-oh notation is also used for the amount of memory (sometimes called *space*) that an algorithm uses. For instance, the space required by the knapsack algorithm given in Figure 2.3 is dominated by the space needed to store the array $A(j,s)$. Since there are $n(B+1)$ items in this array, we say that the algorithm requires $O(nB)$ space. This amount is referred to the *space complexity* of the algorithm.

**The complexity of the algorithmic techniques.** We now turn to a discussion of the time complexity of the algorithmic techniques discussed in the previous section. The most popular algorithm for solving linear programs is called the *simplex method*, and it has a number of variants. Recall that in our discussion of basic solutions to linear programs, we observed that the simplex method moves from basic solution to basic solution by swapping a variable set to 0 for a variable in the linear system. This step is called *pivoting*, and the means by which simplex decides which two variables to swap is called the *pivot rule*. Many textbook pivot rules can be shown to require $\Omega(2^n)$ pivots in the worst case, where $n$ is the maximum of the number of rows and columns, and thus the simplex method can take a large amount of time in the worst case. However, these examples are pathological; in practice, the simplex method is very fast, and problems with tens of thousands of variables and constraints are solved routinely.

*Interior-point methods* constitute another class of algorithms for solving linear programs. Interior-point algorithms have much better worst-case time complexity than the simplex method: the fastest algorithm currently known needs to solve $O(\sqrt{n}L)$ linear systems, each of which takes $O(n^3)$ time in the worst case, where $L$ is the "size" of the linear program (that is, the number of bits needed to encode in binary the matrix $A$, the right-hand side $b$, and the objective function $c$). More practical variants of interior-point methods have larger time complexity ($O(nL)$ iterations instead of $O(\sqrt{n}L)$), but in practice the number of iterations required for these variants seems to be essentially constant, and indeed these algorithms sometimes outperform the simplex method.

Many algorithms have been devised for solving the maximum flow problem. As of the writing of this chapter, the algorithm for the maximum flow problem with the lowest worst-case time complexity in most cases has a running time of $O(\min(n^{2/3}, m^{1/2}) m \log(\frac{n^2}{m}) \log U)$, where $n$ is the number of vertices in the graph, $m$ is the number of edges, and $U$ is size of the largest capacity. However, a class of algorithms called *push-relabel* algorithms (also called *preflow-push* algorithms) have been shown to work very well in practice, and appear to be faster than algorithms whose theoretical worst-case complexity is lower. The lowest known worst-case complexity of a push-relabel algorithm is $O(nm\log(\frac{n^2}{m}))$, and hence is theoretically faster than the previously mentioned algorithm only when $U$ is very large. The notes at the end of this chapter contain pointers to descriptions of these algorithms and studies of them. Interestingly, it has been shown that the parametric maximum flow problem can be solved in the same worst-case running time as a push-relabel algorithm.

As is the case with maximum flow algorithms, there are many minimum-cost flow algorithms. One type that has had successful implementations uses the simplex method with a special pivot rule to solve the associated linear program; this type is called a *network simplex* algorithm. Another type that works as well or better in practice uses the push-relabel algorithm as a subroutine. Specialized algorithms have been developed for the assignment and transportation problems, and again, many different types of algorithms have been proposed. See the notes at the end of the chapter for references.

## 2.3. Complexity theory and a notion of efficiency

Given the current set of algorithms for solving linear programming and network flow problems as described above, it is natural to ask whether better algorithms might exist. For example, is it inherent in the nature of the knapsack problem that all algorithms to solve it must take $\Omega(n\min(B,V))$ time? This asks whether the *complexity* of the knapsack problem is such that no faster running time is possible. The study of such questions is called *complexity theory*.

There is an immediate difficulty with posing such questions, since the time bound given will depend on our model of a computer. For instance, one can imagine a parallel computer executing the knapsack algorithm in Figure 2.3 with $O(n)$ processors. Such a computer could execute the loop in lines 1–2 in a constant number of instructions. Perhaps a parallel machine with a reasonable number of processors could solve the knapsack problem in $O(n)$ time.

There are two possible ways to make the question well-posed. One is to fix a model of a computer. We have so far implicitly considered a single processor machine in which each arithmetic operation, comparison, and memory access takes a single instruction, regardless of the size of the numbers involved. This model is known as the *random access machine*, or the *RAM* model of computation. We can consider the inherent time complexity of problems with respect to the RAM.

Another way to make the question well-posed is to broaden it to whether a given problem has an efficient algorithm or not, and define efficiency in such a way that it is invariant under any reasonable model of a computer. This line of thinking has led to an emphasis on the class of *polynomial-time* algorithms.

**Definition 2.11** [ Polynomial-time algorithm ]. *An algorithm for a problem is said to run in polynomial time, or said to be a polynomial-time algorithm, with respect* *to a particular model of computer (such as a RAM) if the number of instructions executed by the algorithm can be bounded by a polynomial in the size of the input.*

More formally, let $x$ denote an *instance* of a given problem; for example, an instance of the knapsack problem is the number $n$ of items, the numbers $s_j$ and $v_j$ giving the sizes and values of the items, and the number $B$ giving the size of the knapsack. To present the instance as an input to an algorithm $A$ for the problem, we must encode it in bits in some fashion; let $|x|$ be the number of bits in the encoding of $x$. Then $|x|$ is called the *size* of the instance or the *instance size*. Furthermore, we say that $A$ is a polynomial-time algorithm if there exists a polynomial $p(n)$ such that the running time of $A$ is $O(p(|x|))$.

We consider an algorithm to be efficient if it runs in polynomial time, and we consider a problem to be efficiently solvable if it has a polynomial-time algorithm. This is an imperfect measure: for instance, we have already noted that the simplex method has a worst-case running time that is exponential in the size of the instance, but is efficient in practice. However, the correspondence between theory and practice is good enough that the equivalence is useful.

One benefit of equating efficiency with polynomial time is that given reasonable models of a computer, efficiency is invariant: that is, an algorithm that runs in polynomial time on one will run in polynomial time on another. Of course, the two polynomials might be different for the two different models of computation. For instance, consider any parallel RAM with a number of processors at most a polynomial in the size of the instance. This parallel machine can achieve a speed-up factor of at most the number of processors it has; thus a polynomial-time algorithm on an ordinary RAM will run in polynomial-time on a parallel RAM and vice versa.

It will be useful in later sections to refer to the class of problems P. The class P contains all *decision problems* that have polynomial-time algorithms. A decision problem is one whose output is either "Yes" or "No". It is not difficult to think of decision problems related to optimization problems. For instance, consider a decision variant of the knapsack problem in which, in addition to inputs $B$, and $v_j$ and $s_j$ for every item $j$, there is also an input $C$, and the problem is to output "Yes" if the optimum solution to the knapsack instance has value at least $C$, and "No" otherwise. The instances of a decision problem can be divided into "Yes" instances and "No" instances; that is, instances in which the correct output for the instance is "Yes" (or "No").

Such a decision problem for the knapsack problem is clearly no harder than the optimization version, but also is no easier: if we had a polynomial-time algorithm for the decision problem, we would also be able to get a polynomial-time algorithm for the optimization problem. To see this, first observe that we can use an algorithm for the decision problem to determine the optimum value $Z$ by performing a bisection search over the range $[0,V]$ (recall that $V = \sum_{j=1}^{n} v_j$ so that $V$ is an upper bound on the value of the knapsack). Once we have determined the optimum value $Z$, we can loop through the items, using the algorithm for the decision problem to see if the value of the optimum solution remains $Z$ when the $j$th item is removed from the instance. If so, we omit item $j$ for the remaining iterations of the loop. The items remaining at the end of the loop must belong to an optimal knapsack, and thus we have determined an optimal knapsack. We use $O(n + \log V)$ calls to the decision algorithm, so if that algorithm runs in polynomial time, there is a polynomial-time algorithm for the optimization problem.

Let us now return to the question of the complexity of the knapsack problem: does it have a polynomial-time algorithm? At first glance, it would appear that the answer is yes, since we have an $O(nB)$ algorithm, where $n$ and $B$ are part of the input. However, this gets us into some of the subtlety of the definition. Usually the data put into a computer are encoded in binary, so that the size of a number $B$ is $\lceil \log_2 B \rceil$ bits. Thus the running time $O(nB)$ is exponential in the size of $B$, not polynomial. If we encode the numeric inputs to the knapsack problem in *unary* rather than binary (that is, we use $B$ bits to encode $B$), then the running time $O(nB)$ is polynomial in the instance size. Algorithms which have this property are called *pseudopolynomial*.

**Definition 2.12** [ Pseudopolynomial-time algorithm ]. *An algorithm for a problem is said to run in pseudopolynomial time, or said to be a pseudopolynomial-time algorithm, with respect to a particular model of computer (such as a RAM) if the number of instructions executed by the algorithm can be bounded by a polynomial in the size of the instance when the numeric data is encoded in unary.*

## 2.4. Complexity theory and a notion of hardness

So, does the knapsack problem have a polynomial-time algorithm? As of the writing of this chapter, the answer is unknown, but there are substantial reasons to think not. Similar evidence suggests that many scheduling problems of interest do not have polynomial-time algorithms. In this section, we present this evidence via the class of problems NP, and the theory of NP-completeness.

Roughly speaking, the class NP is the set of all decision problems such that for any “Yes” instance of the problem, there is a short, easily verifiable “proof” that the answer is “Yes”. Additionally, for each “No” instance of the problem, no such “proof” is convincing. What kind of “short proof” do we have in mind? Take the example of the decision variant of the knapsack problem given above. For any “Yes” instance, in which there is a feasible subset of items of value at least $C$, a short proof of this fact is a list of the items in the subset. Given the knapsack instance and the list, an algorithm can quickly verify that the items in the list have total size at most $B$, and total value at least $C$. Note that for any “No” instance, then no possible list of items will be convincing.

We now attempt to formalize this rough idea as follows. A short proof is one whose encoding is bounded by some polynomial in the size of the instance. An easily verifiable proof is one that can be verified in time bounded by a polynomial in the size of the instance and the proof. This gives the following definition.

**Definition 2.13** [ NP ]. *A decision problem is said to be in the problem class* NP *if* *there exists a verification algorithm $A(\cdot,\cdot)$ and two polynomials, $p_1$ and $p_2$, such that:*

1. *for every "Yes" instance $x$ of the problem, there exists a proof $y$ with $|y| \leq p_1(|x|)$ such that $A(x,y)$ outputs "Yes";*
2. *for every "No" instance $x$ of the problem, for all proofs $y$ with $|y| \leq p_1(|x|)$, $A(x,y)$ outputs "No";*
3. *the running time of $A(x,y)$ is $O(p_2(|x|+|y|))$.*

NP stands for *non-deterministic polynomial time*. Most of the scheduling problems in this book have decision variants that are in the class NP.

Observe that nothing precludes a decision problem in NP from having a polynomial-time algorithm. However, the central problem of complexity theory is whether *every* problem in NP has a polynomial-time algorithm. This is usually expressed as the question of whether the class P of decision problems with polynomial-time algorithms is the same as the class NP, or, more succinctly, as whether P = NP. Although this question has been a matter of intense research for almost thirty years, its answer is unknown as of the writing of this chapter.

To tackle this problem, in the early 1970s several researchers showed that there are problems in NP that are representative of the entire class, in the sense that if they have polynomial-time algorithms, then P = NP, and if they do not, then P $\neq$ NP. These are the NP-*complete* problems; we will be more precise about their definition in a moment. Since then, thousands of problems have been shown to be NP-complete, yet none of them is known to have a polynomial-time algorithm. Most complexity theorists take this as strongly suggestive evidence that P $\neq$ NP. However, complexity theory is not yet able to prove this inequality, and thus the strongest statement that can be made is that it seems likely that every NP-complete problem does not have a polynomial-time algorithm.

Let us now turn to the definition of NP-*completeness*. To do this, we will need the notion of a *polynomial-time reduction*.

**Definition 2.14** [ Polynomial-time reduction ]. *Given two decision problems A and B, there is a polynomial-time reduction from A to B (or A reduces to B in polynomial time) if there is a polynomial-time algorithm that takes as input an instance of A and produces as output an instance of B and has the property that a "Yes" instance of B is output if and only if a "Yes" instance of A is input.*

We will use the symbol $\preceq$ to denote a polynomial-time reduction so that we write $A \preceq B$ if $A$ reduces to $B$ in polynomial time. Sometimes the symbol $\leq_m^P$ is used in the literature to denote a polynomial-time reduction. We can now give a formal definition of NP-completeness.

**Definition 2.15** [ NP-complete ]. *A problem B is* NP-*complete if B is in* NP*, and for every problem A in* NP*, there is a polynomial-time reduction from A to B.*

The following theorem is now easy to show.

**Theorem 2.16.** *Let $B$ be an* NP*-complete problem. If $B$ has a polynomial-time algorithm, then* P = NP.

*Proof.* Given any problem $A$ in NP, we can create a polynomial-time algorithm for it as follows: the algorithm takes an instance $x$ of $A$ as input, uses the polynomial-time reduction from $A$ to $B$ to transform it into an instance $y$ of $B$, then uses the polynomial-time algorithm for $B$ on that instance. The algorithm outputs "Yes" if and only if the algorithm for $B$ outputs "Yes" for $y$. First, observe that the algorithm outputs "Yes" on $x$ if and only if $x$ is a "Yes" instance of $A$, by the properties of the reduction. Second, we will show that $A$ runs in polynomial time. Suppose that the running time for the reduction algorithm is bounded by polynomial $p_1(|x|)$ and the running time of the algorithm for $B$ is bounded by polynomial $p_2(|y|)$. Certainly $|y| \leq p_1(|x|)$, since the running time for the reduction must include the time spent writing down the bits of $y$. Thus the overall running time is $p_1(|x|) + p_2(p_1(|x|))$, which is a polynomial in $|x|$. □

A useful property of NP-complete problems is that once we have an NP-complete problem $B$ it is often easy to prove that other problems are also NP-complete. As we will see, all we have to do is show that a problem $A$ is in NP, and that $B \preceq A$. This follows as an easy corollary of the transitivity of polynomial-time reductions.

**Theorem 2.17.** *Polynomial-time reductions are transitive: that is, if $A \preceq B$ and $B \preceq C$, then $A \preceq C$.*

**Corollary 2.18.** *If $A$ is in* NP*, $B$ is* NP*-complete, and $B \preceq A$, then $A$ is also* NP*-complete.*

*Proof.* All we need to show is that for each problem $C$ in NP, there is a polynomial-time reduction from $C$ to $A$. Because $B$ is NP-complete, we know that $C \preceq B$. By hypothesis, $B \preceq A$. By Theorem 2.17, $C \preceq A$. □

*Proof of Theorem 2.17.* Since $A \preceq B$, there is an algorithm such that for some polynomial $p_1$ the algorithm takes an instance $x$ of $A$ as input and produces an instance $y$ of $B$ in time no more than $p_1(|x|)$, and $y$ is a "Yes" instance of $B$ if and only if $x$ is a "Yes" instance of $A$. Furthermore, $|y| \leq p_1(|x|)$. Because $B \preceq C$, there exists an algorithm such that for some polynomial $p_2$ an instance $y$ of $B$ can be transformed into an instance $z$ of $C$ in time no more than $p_2(|y|)$, such that $z$ is a "Yes" instance of $C$ if and only if $y$ is a "Yes" instance of $B$. Additionally, $|z| \leq p_2(|y|)$. Thus there is an algorithm such that any instance $x$ of $A$ can be transformed into an instance $z$ of $C$ in time no more than $p_3(|x|)$ such that $x$ is a "Yes" instance of $A$ if and only if $z$ is a "Yes" instance of $C$, where $p_3(\cdot)$ is the polynomial $p_2(p_1(\cdot))$. Thus $A \preceq C$. □

Theorem 2.18 shows that given one NP-complete problem, it is possible to prove that other problems are NP-complete. How do we prove a problem is NP-complete in the first place? Ingenious proofs providing the first NP-complete problems were developed independently by Cook and Levin in the early 1970s, but it is beyond the scope of this chapter to provide any details of the proofs. Interested readers can consult the chapter notes for references. A paper of Karp then showed that many problems of interest are NP-complete, and since then thousands of problems have been shown to be NP-complete. We list a few of them below.

**Partition**
**Input:** Positive integers $a_1,\ldots,a_n$ such that $\sum_{i=1}^n a_i$ is even
**Question:** Does there exist a partition of $\{1,\ldots,n\}$ into sets $S$ and $T$ such that $\sum_{i\in S} a_i = \sum_{i\in T} a_i$?

**3-Partition**
**Input:** Positive integers $a_1,\ldots,a_{3n},b$, such that $b/4 < a_i < b/2$ for all $i$, and $\sum_{i=1}^{3n} a_i = nb$.
**Question:** Does there exist a partition of $\{1,\ldots,3n\}$ into $n$ sets $T_j$ such that $\sum_{i\in T_j} a_i = b$ for all $j = 1,\ldots,n$? (By the condition on the $a_i$, each $T_j$ must contain exactly 3 elements.)

**Clique**
**Input:** An undirected graph $G = (V,E)$, and a positive integer $K$
**Question:** Does there exist a set of vertices $S \subseteq V$ with $|S| \geq K$ such that for all $i,j \in S$ with $i \neq j$, then $(i,j) \in E$? (Such a set $S$ is called a *clique* or an $|S|$-clique).

The decision version of the knapsack problem given at the beginning of the section is also NP-complete. However, we know that this problem has a pseudopolynomial-time algorithm. This brings up an interesting distinction among the NP-complete problems. Some NP-complete problems, such as the knapsack and partition problems, are NP-complete only when it is assumed that their numeric data is encoded in binary. As we've seen, the knapsack problem has a polynomial-time algorithm if the input is encoded in unary; so does the partition problem. Other problems, however, such as the 3-partition problem above, are NP-complete even when their numeric data is encoded in unary. We call such problems *strongly* NP-*complete*, or, sometimes, *unary* NP-*complete*. In contrast, problems such as the knapsack and partition problems are called *weakly* NP-*complete* or *binary* NP-*complete*.

Given the NP-completeness of the problems above, we give proofs showing the NP-completeness of three different scheduling problems. From here on we sometimes use the notation $\alpha|\beta|\gamma$ to refer to the decision version of the problem as well as the optimization version. The reader should be able to tell which version is intended from the context.

**Theorem 2.19.** *The problem* $1|r_j|L_{\max}$ *is* NP-*complete.*

*Proof.* The decision version of $1|r_j|L_{\max}$ has an input parameter $B$, and a "Yes" instance is one for which the maximum lateness of an optimal schedule is no more than $B$. Certainly this problem is in NP: a short proof for the problem is a list of the starting time of every job. Because each start time is at most $\max_j r_j + \sum_{j=1}^n p_j$, the starting time of each is a number whose size is at most a polynomial in the instance size. Since there are $n$ start times, the size of the proof can be bounded by a polynomial in the size of the instance. We can verify in polynomial time whether such a list is a valid schedule and whether $L_{\max} \leq B$, and output "Yes" or "No" as appropriate.

To prove that the problem is NP-complete, we give a polynomial-time reduction to it from the partition problem. Given an instance of the partition problem $a_1, \ldots, a_n$, we construct a scheduling instance with $n+1$ jobs. Let $A = \sum_{i=1}^{n} a_i$. Job $j$, for $1 \leq j \leq n$, has processing time $p_j = a_j$, release date $r_j = 0$, and deadline $\bar{d}_j = A+1$. For job $n+1$, set $p_{n+1} = 1$, $r_{n+1} = A/2$, and $\bar{d}_{n+1} = A/2+1$. Finally, set $B = 0$. Certainly this reduction can be performed in polynomial time.

We now need to show that this is a "Yes" instance of $1|r_j|L_{\max}$ if and only if $a_1, \ldots, a_n$ is a "Yes" instance of the partition problem. If $a_1, \ldots, a_n$ is a "Yes" instance of the partition problem, then there are two sets $S$ and $T$ that partition $\{1, \ldots, n\}$ such that $\sum_{i \in T} a_i = \sum_{i \in S} a_i = A/2$. Thus the jobs in $S$ can be scheduled from time 0 to time $A/2$, job $n+1$ from time $A/2$ to $A/2+1$, and the jobs in $T$ from time $A/2+1$ to $A+1$; in this case, each job will complete by its deadline. Hence the constructed instance of $1|r_j|L_{\max}$ is a "Yes" instance.

Similarly, if the constructed instance of $1|r_j|L_{\max}$ is a "Yes" instance, then job $n+1$ must be processed from time $A/2$ to $A/2+1$. Some set $S$ of jobs is processed from time 0 to $A/2$, and the remaining set $T$ is processed from time $A/2+1$ to $A+1$. Then $S$ and $T$ partition $\{1, \ldots, n\}$ and $\sum_{i \in S} a_i \leq A/2$ and $\sum_{i \in T} a_i \leq A/2$. Hence $\sum_{i \in S} a_i = \sum_{i \in T} a_i = A/2$, and $a_1, \ldots, a_n$ is a "Yes" instance of the partition problem. □

**Theorem 2.20.** *The problem $P||C_{\max}$ is* NP-*complete.*

*Proof.* The decision version of the problem $P||C_{\max}$ has an input $B$, and the instance is a "Yes" instance if $C_{\max} \leq B$ in an optimal schedule. It is easy to show that this problem is in NP: the short proof contains for each job the starting time of the job and machine on which the job executes. As argued above, this proof has size that can be bounded by a polynomial in the instance size. A polynomial-time algorithm can check whether this is a feasible schedule and whether $C_{\max} \leq B$ and output "Yes" or "No" as appropriate.

To prove that the problem is NP-complete, we give a polynomial-time reduction to it from the 3-partition problem. Given an instance of the 3-partition problem, $a_1, \ldots, a_{3n}$, and $b$, construct a scheduling instance with $3n$ jobs and $n$ machines. Each job $j$ has processing time $p_j = a_j$, and $B$ is set to $b$. Certainly this reduction can be performed in polynomial time.

We now need to show that this a "Yes" instance of $P||C_{\max}$ if and only if $a_1, \ldots, a_{3n}$, $b$ is a "Yes" instance of the 3-partition problem. If $a_1, \ldots, a_{3n}, b$ is a "Yes" instance of 3-partition problem, then there are $n$ sets $T_j$ that partition $\{1, \ldots, 3n\}$ such that $\sum_{i \in T_j} a_i = b$. For each set $T_j$, schedule the jobs $i \in T_j$ in any order on the $j$th machine from time 0 to time $\sum_{i \in T_j} a_i = b$. This implies that $C_{\max} = b$, and thus the constructed instance of $P||C_{\max}$ is a "Yes" instance.

Similarly, suppose the constructed instance of $P||C_{\max}$ is a "Yes" instance. Let $T_j$ denote the set of indices of jobs processed on the $j$th machine. Since the scheduling instance is a "Yes" instance, it must be the case that $\sum_{i \in T_j} a_i \leq b$ for each $j$, $1 \leq j \leq n$. But since $\sum_{j=1}^{n} \sum_{i \in T_j} a_i = \sum_{i=1}^{3n} a_i = nb$, it must be the case that $\sum_{i \in T_j} a_i = b$ for each $j$, $1 \leq j \leq n$. Hence $a_1, \ldots, a_{3n}, b$ is a "Yes" instance of the 3-partition problem. □

One consequence of the proof of Theorem 2.20 is that $P||C_{\max}$ is strongly NP-complete. This follows since 3-partition is strongly NP-complete; that is, it is NP-complete even if the $a_i$ and $b$ are encoded in unary. We can use the reduction of the Theorem 2.20 to reduce instances of the 3-partition problem with data encoded in unary to instances of $P||C_{\max}$ with data encoded in unary in polynomial time. Thus the problem $P||C_{\max}$ with data encoded in unary is also NP-complete.

**Theorem 2.21.** *The problem $P|prec, p_j = 1|C_{\max}$ is* NP-*complete.*

*Proof.* The decision version of $P|prec, p_j = 1|C_{\max}$ has an input $B$, and an instance is a "Yes" instance if $C_{\max} \leq B$ in an optimal schedule. It is easy to verify that this problem is in NP.

We now give a polynomial-time reduction from the clique problem to $P|prec, p_j = 1|C_{\max}$. Given the input graph $G = (V, E)$ and the bound $K$ of the clique problem, create one job $v$ for each vertex $v \in V$ and one job $e$ for each edge $e \in E$. Set $v \to e$ if $v$ is one of the two endpoints of $e$. Set $N = |V|$, $M = |E|$, and $L = K(K-1)/2$, so that $L$ is the number of edges in a clique of size $K$. For the moment we leave the number of machines $m$ unspecified, and create extra "dummy" jobs: $m - K$ jobs $X_a$, $m - (L + N - K)$ jobs $Y_b$, and $m - (M - L)$ jobs $Z_c$. We set $X_a \to Y_b \to Z_c$ for every $a, b, c$. Set $m$ to guarantee that there will be at least one $X_a$ job, one $Y_b$ job, and one $Z_c$ job; that is, $m = \max\{K, L + N - K, M - L\} + 1$. Finally, set $B = 3$.

For this instance of $P|prec, p_j = 1|C_{\max}$, first observe the following: in any schedule of length 3, it must be the case that the dummy jobs $X_a$ are scheduled at time 0, the jobs $Y_b$ at time 1, and the jobs $Z_c$ at time 2, leaving only $K$ machines free at time 0, $L + N - K$ free at time 1, and $M - L$ free at time 2.

We now show that this is a "Yes" instance of $P|prec, p_j = 1|C_{\max}$ if there is a clique of size at least $K$ in $G$. If this is the case, then the non-dummy jobs can be completed by time 3 by scheduling $K$ of the jobs $v$ corresponding to $K$ vertices of the clique at time 0, scheduling $L$ of the jobs $e$ corresponding to the $L$ edges between the $K$ vertices of the clique at time 1, scheduling the remaining $N - K$ jobs corresponding to vertices at time 1, and scheduling the remaining $M - L$ jobs corresponding to edges at time 2.

Finally, this instance is a "No" instance of $P|prec, p_j = 1|C_{\max}$ if there is no clique of size $K$ in $G$. In this case, then even if $K$ jobs $v$ are scheduled at time 0, at most $L - 1$ jobs $e$ can be scheduled at time 1. Even if the remaining $N - K$ jobs $v$ are scheduled at time 1, this means there are still $M - (L - 1)$ jobs $e$ left to schedule at time 2, but there is only enough room for $M - L$ jobs. Hence not all jobs can complete by time 3, and therefore the constructed instance of $P|prec, p_j = 1|C_{\max}$ is a "No" instance. □

We conclude this section by defining the term NP-*hard*, which can be applied to either optimization or decision problems. Roughly speaking, it means "as hard as the hardest problem in NP". To be more precise, we need to define an *oracle*. Given a decision or optimization problem $A$, we say that an algorithm has $A$ as an oracle (or has *oracle access* to $A$) if we suppose that the algorithm can solve instance of $A$ with a single instruction.

**Definition 2.22** [ NP-hard ]. *A problem $A$ is* NP-*hard if there is a polynomial-time algorithm for an* NP-*complete problem $B$ when the algorithm has oracle access to $A$.*

The term "NP-hard" is most frequently applied to optimization problems whose corresponding decision problems are NP-complete; it is easy to see that such optimization problems are indeed NP-hard. It is also easy to see that if $A$ is NP-hard and there is a polynomial-time algorithm for $A$, then $\mathrm{P} = \mathrm{NP}$.

## 2.5. Coping with NP-completeness

We now turn from complexity theory back to algorithmic techniques. We have seen that the theoretical measure of whether an optimization problem is efficiently solvable is whether it has a polynomial-time algorithm, and have seen that many scheduling problems are NP-complete and thus unlikely to have such algorithms. Yet such problems continue to arise in practice and need solution, whether they have efficient algorithms or not.

What then can be done? In this section, we explore some of the possible answers to this question. One possibility is to give an efficient *heuristic* that does not always find an optimal solution to the problem, but does find one that is "good enough". An *approximation algorithm* is a type of heuristic that comes with a proven performance guarantee. Another possibility is to give an algorithm that finds the optimal solution, and although it is not guaranteed to run in polynomial time, it runs quickly enough for instances arising in practice. Some such algorithms use *branch-and-bound* techniques, which are discussed below.

One last possibility is to make probabilistic statements about the kinds of instances that arise in practice, and then design algorithms that with high probability run quickly and find a good or optimal solution. We will not discuss probabilistic techniques in this section, although this approach is considered in Chapter 9.

**Approximation algorithms.** Although there are many heuristics and metaheuristic techniques for scheduling problems (such as simulated annealing), in this chapter we will only consider *approximation algorithms*. An approximation algorithm for an optimization problem is one that runs in polynomial time, and is guaranteed to provide a solution whose value is close to the optimum value. If the algorithm produces a solution whose value is always within a factor of $\alpha$ of the value of an optimal solution, then the algorithm is called an $\alpha$-approximation algorithm. The value $\alpha$ is called the *performance guarantee* or the *approximation factor* of the algorithm. In this chapter, we will use the convention that $\alpha > 1$ for minimization problems, and

$\alpha < 1$ for maximization problems. For example, a 2-approximation algorithm for a minimization problem produces solutions of value no more than twice the optimum value, while a $\frac{1}{2}$-approximation algorithm for a maximization problem produces a solution of value at least $\frac{1}{2}$ the optimum value. However, no standard convention exists for maximization problems, and in the literature it is common to see $1/\alpha$ used as the performance guarantee so that the $\frac{1}{2}$-approximation algorithm above would also be referred to as a 2-approximation algorithm.

Sometimes it is possible to give a family of approximation algorithms that can produce solutions whose value comes arbitrarily close to the optimum value. That is, for each $\varepsilon > 0$, we can give a $(1+\varepsilon)$-approximation algorithm $A_\varepsilon$ for a minimization problem or a $(1-\varepsilon)$-approximation algorithm for a maximization problem. The running time of the family of algorithms $\{A_\varepsilon\}$ may depend on $\varepsilon$ in some nasty way (e.g., it may be exponential in $1/\varepsilon$), but since $\varepsilon$ is a fixed constant for each algorithm $A_\varepsilon$ this does not matter. We call such a family a *polynomial-time approximation scheme*, sometimes abbreviated as *PTAS*.

**Definition 2.23** [ Polynomial-time approximation scheme ]. *A polynomial-time approximation scheme (PTAS) is a family of algorithms $\{A_\varepsilon\}$ for an optimization problem such that for each $\varepsilon > 0$, $A_\varepsilon$ is a $(1+\varepsilon)$-approximation algorithm (for a minimization problem) or a $(1-\varepsilon)$-approximation algorithm (for a maximization problem).*

If a family of algorithms is a polynomial-time approximation scheme, and it is also the case that the running time of the family is also polynomial in $1/\varepsilon$, we call the family a *fully polynomial-time approximation scheme*, sometimes abbreviated as *FPTAS* or *FPAS*.

**Definition 2.24** [ Fully polynomial-time approximation scheme ]. *If $\{A_\varepsilon\}$ is a polynomial-time approximation scheme such that the running time of the family of algorithms is polynomial in $1/\varepsilon$, then $\{A_\varepsilon\}$ is called a fully polynomial-time approximation scheme.*

To illustrate these ideas, we show how we can obtain a fully polynomial-time approximation scheme for the knapsack problem (and thus for the problem $1|\bar{d}_j = d|\sum w_j U_j$). In Figure 2.4 of Section 2.1, we gave a dynamic programming algorithm for the knapsack problem that requires $O(n\min(B,V))$ time, where $n$ is the number of items, $B$ is the size of the knapsack, and $V = \sum_{i=1}^{n} v_j$ is the maximum possible value of the knapsack. If we could transform any instance of the knapsack problem into one for which the optimum solution was not too different in value, and for which $V$ was polynomial in $n$ and $1/\varepsilon$, then in time polynomial in the instance size we could find the optimum of the transformed instance, and it would be a good approximation of the original optimum solution. This is precisely what we do in our fully polynomial-time approximation scheme for the knapsack problem. The algorithm shown in Figure 2.5 rounds the values $v_j$ down to the nearest multiple of $\varepsilon W/n$, where $W$ is the maximum value of any item; call these new values $v'_j$. Then $V' = \sum_{j=1}^{n} v'_j = \sum_{j=1}^{n} \lfloor \frac{v_j}{\varepsilon W/n} \rfloor \leq n^2/\varepsilon$, so that invoking the dynamic programming algorithm on the transformed instance takes $O(nV') = O(n^3/\varepsilon)$ time. We now only

```
1    $W \leftarrow 0$
2    For $j \leftarrow 1$ to $n$
3        If $v_j > W$
4            $W \leftarrow v_j$
5    $K \leftarrow \varepsilon W/n$
6    For $j \leftarrow 1$ to $n$
7        $v'_j \leftarrow \lfloor v_j/K \rfloor$
8    Run dynamic programming algorithm in Figure 2.4 on instance with
         sizes $s_j$ and values $v'_j$.
```

**Figure 2.5. Fully polynomial-time approximation scheme for the knapsack problem.**

need to show that the value of the solution found on the transformed instance is close to the optimum value of the original instance.

**Theorem 2.25.** *The algorithm in Figure 2.5 produces a solution of value at least* $(1-\varepsilon)$ *times the optimum value.*

*Proof.* Let $S$ be the set of items found by running the algorithm of Figure 2.4 with the modified values $v'_j = \lfloor v_j/K \rfloor$, where $K = \varepsilon W/n$. Let $O$ be an optimal set of items for the original instance, and let OPT denote its value. We know that $W \leq$ OPT, since one possible knapsack is to take the most valuable item (recall that $s_j \leq B$ for all items $j$). We also know, by the definition of the $v'_j$, that

$$Kv'_j \leq v_j < K(v'_j + 1), \tag{2.5}$$

which implies that $Kv'_j > v_j - K$. Then

$$\begin{aligned}
\sum_{j \in S} v_j &\geq K \sum_{j \in S} v'_j \qquad \text{(by (2.5))} \\
&\geq K \sum_{j \in O} v'_j \qquad \text{(since $S$ is an optimal solution for the values $v'_j$)} \\
&\geq \sum_{j \in O} v_j - |O|K \qquad \text{(by (2.5))} \\
&\geq \sum_{j \in O} v_j - nK \\
&= OPT - \varepsilon W \\
&\geq (1-\varepsilon) OPT.
\end{aligned}$$

□

Sometimes it is possible to show that a particular performance guarantee $\alpha$ cannot be achieved unless P = NP; that is, we can't even get an approximate solution whose value is within a factor $\alpha$ of the optimum in polynomial time unless we could solve the problem optimally in polynomial time. For instance, it is not difficult to show that under quite reasonable conditions, no optimization problem whose decision variant is strongly NP-complete can have a fully polynomial-time approximation scheme.

**Theorem 2.26.** *Given an optimization problem whose decision version is strongly* NP*-complete, let $|x|_u$ denote the length of the encoding of an instance $x$ with the numeric data encoded in unary. Let* $\mathrm{OPT}(x)$ *denote the optimum value of the instance $x$. If* $\mathrm{OPT}(x)$ *is an integer-valued function, and there exists a polynomial $p$ such that* $\mathrm{OPT}(x) < p(|x|_u)$ *for all instances $x$, then there is no fully polynomial-time approximation scheme for the problem unless* $\mathrm{P} = \mathrm{NP}$.

*Proof.* Assume the problem is a minimization problem; a trivially modified proof will work for a maximization problem. The decision version of the problem asks whether or not $\mathrm{OPT}(x) \le B$ for an instance $(x, B)$. Suppose there is a fully polynomial-time approximation scheme for the problem. For a given instance $x$, set $\varepsilon = 1/p(|x|_u)$. Then the fully polynomial-time approximation scheme returns a solution of value between $\mathrm{OPT}(x)$ and

$$(1+\varepsilon)\,\mathrm{OPT}(x) = \left(1 + \frac{1}{p(|x|_u)}\right)\mathrm{OPT}(x) < \mathrm{OPT}(x) + 1.$$

Since $\mathrm{OPT}(x)$ is integer valued, the algorithm returns a solution of value $OPT(x)$. Thus we can correctly decide if $\mathrm{OPT}(x) \le B$. Furthermore, the algorithm takes time polynomial in $|x|$ and $\frac{1}{\varepsilon} = p(|x|_u)$, which is polynomial in $|x|_u$. However, the decision variant is NP-complete even if the instance is encoded in unary, so we have a polynomial-time algorithm to decide an NP-complete problem, implying $\mathrm{P} = \mathrm{NP}$. □

As an example of the applications of this theorem, consider the problem $P||C_{\max}$. The makespan of an optimal schedule is at most $\sum_{j=1}^{n} p_j$. If the data for $P||C_{\max}$ is encoded in unary, then certainly the encoding will require at least $\sum_{j=1}^{n} p_j$ bits to represent all the processing times $p_j$. Thus $\mathrm{OPT}(x) < |x|_u + 1$, and one condition of the theorem is met. The other condition of the theorem is met since the decision version of $P||C_{\max}$ is strongly NP-complete. Thus there is no FPTAS for $P||C_{\max}$ unless $\mathrm{P} = \mathrm{NP}$.

Sometimes a proof of NP-completeness will imply that no $\alpha$-approximation algorithm can exist for a given $\alpha$ unless $\mathrm{P} = \mathrm{NP}$. As an example, we can use the proof of Theorem 2.21 to show the following theorem.

**Theorem 2.27.** *There can be no $\alpha$-approximation algorithm for $P|prec, p_j = 1|C_{\max}$ with* $\alpha < 4/3$ *unless* $\mathrm{P} = \mathrm{NP}$.

*Proof.* In the proof of Theorem 2.21, we showed that a "Yes" instance of the clique problem could be mapped to an instance of $P|prec, p_j = 1|C_{\max}$ with makespan 3, and a "No" instance was mapped to an instance with makespan greater than 3. Since $p_j = 1$ for all jobs, an instance with makespan greater than 3 must have makespan at least 4. Now suppose we have an $\alpha$-approximation algorithm for $P|prec, p_j = 1|C_{\max}$ with $\alpha < 4/3$. Then a polynomial-time algorithm for the clique problem is as follows: given an instance of the clique problem, transform the instance into an instance of $P|prec, p_j = 1|C_{\max}$ as in Theorem 2.21 and run the $\alpha$-approximation algorithm. If the approximation algorithm gives a schedule of length 3, output "Yes", otherwise "No". The algorithm runs in polynomial time, and gives the correct answer since whenever it is given a "Yes" instance of the clique problem, the approximation algorithm creates a schedule of length less than $\frac{4}{3} \cdot 3 = 4$, which must be a schedule of length 3. Thus the algorithm correctly outputs "Yes". If the algorithm is given a "No" instance of the clique problem, the makespan of the schedule must be at least 4, and so the algorithm correctly returns "No". Thus we cannot have an $\alpha$-approximation algorithm for $P|prec, p_j = 1|C_{\max}$ for $\alpha < 4/3$ unless P = NP.□

Similar theorems can be shown for other scheduling problems; see Chapters 11–13 for examples.

**Branch and bound.** Another approach to dealing with NP-complete problems is to guarantee optimality but not efficiency, rather than guaranteeing efficiency but not optimality. One technique of this type is called *branch-and-bound.* This technique is a combination of an algorithm which searches the space of all possible solutions to a problem (the "branch" part) and an algorithm that bounds the optimum value, to rule out parts of the search space (the "bound" part). More sophisticated variations of this technique have been developed (e.g. branch-and-cut, branch-and-price), but we will only discuss the most basic version.

We begin by showing how this technique can be applied to the knapsack problem. First, we formulate the problem as an integer program:

$$\begin{aligned} \text{Max} \quad & \sum_{j=1}^{n} v_j x_j \\ \text{subject to:} \quad & \\ & \sum_{j=1}^{n} s_j x_j \le B \\ & x_j \in \{0,1\} \qquad\qquad 1 \le j \le n. \end{aligned}$$

Replacing the constraints $x_j \in \{0,1\}$ with constraints $0 \le x_j \le 1$ gives a linear programming relaxation. Solving this LP is our "bounding" algorithm, since the optimal value of this LP solution is an upper bound on the cost of an optimal solution. In fact, the optimal solution to this LP can be obtained easily: suppose the items are indexed such that $v_1/s_1 \ge v_2/s_2 \ge \cdots \ge v_n/s_n$, and $\sum_{j=1}^{k-1} s_j \le B$, but $\sum_{j=1}^{k} s_j > B$. Then an optimal LP solution will set $x_1 = x_2 = \cdots = x_{k-1} = 1$, $x_{k+1} = x_{k+2} = \cdots = x_n = 0$, and $x_k = \frac{1}{s_k}\left(B - \sum_{j=1}^{k-1} s_k\right)$. If $x_k = 0$, then the solution is integral, and the solution is an optimal solution to the knapsack problem as well.

If $x_k \neq 0$, then we enter our "branching" algorithm. We create the two subproblems of our knapsack problem, one in which the $k$th item cannot be included in the knapsack (equivalent to setting $x_k = 0$) and the other in which the $k$th item must be included in the knapsack (equivalent to $x_k = 1$); this latter problem is equivalent to a knapsack problem with item $k$ omitted and a knapsack of capacity $B - s_k$. Observe that if we can find the solution to these two subproblems, then whichever solution has the maximum value is the optimal solution to the original knapsack problem, since the $k$th item must either be included or excluded. We say that the two subproblems *partition the solution space* of the original problem. Since both subproblems are knapsack problems, we can apply the same approach as above to find their solution; eventually either the LP relaxation returns an integer solution, or is infeasible, in which case we no longer need to consider the subproblem.

This process is usually described in terms of a tree. The tree's root corresponds to the original knapsack problem, the root's children to the subproblems of the original problem, and in general the children of a node correspond to the subproblems of that node.

This overall algorithm might be quite slow, but the following ideas can be used to speed it up. We can keep track of the value of the *incumbent* solution, the best integral solution found thus far. If we reach a subproblem such that the value of the LP relaxation for that subproblem is less than the value of the incumbent solution, then we do not need to find an integral solution to that subproblem; whatever it is, it will not be the solution to the original problem because it will have value less than that of the incumbent solution. So we can "cut off" this part of the solution space; we say that the node in the tree corresponding to this subproblem has been *fathomed*. Obviously the more nodes we can fathom, the less work we will have to do. Since we do not initially have any integral solution to the problem (and we might have to solve many subproblems before we find one), it is common to run a heuristic either before or immediately after solving initial LP relaxation of the original problem, so that we have some incumbent solution with which to cut off parts of the solution space. For example, we can run the polynomial-time approximation scheme for knapsack of the previous section to get a good initial incumbent solution.

The discussion above illustrates one way of applying branch-and-bound to a particular problem, the knapsack problem. The ideas there can be generalized in a number of different ways. For instance, we needn't use a linear programming relaxation of an integer programming formulation of the problem of interest; we can also use some other easily computed bound on the value of an optimal solution, as long as we know when the bound is equal to the value of an optimal solution. In the example above, we "branched" on a fractional variable $x$ from the linear programming solution, creating two subproblems in which $x$ was set to 0 and $x$ was set to 1. In general, any way of creating two or more subproblems that partition the solution space will do.

Other methods exist to speed up branch-and-bound. The running time of branch-and-bound depends on the amount of time it takes to compute a bound and the number of subproblems created. Getting good bounds and having a good incumbent solution cut down on the number of subproblems, but it is also helpful to speed up the bound algorithm. One way to do this is to use a bound which can be computed more quickly than an LP relaxation. In the case of the knapsack problem, we had an LP bound which could be computed by inspection, but this is not always the case.

One way to get a bound which can be computed more quickly is to further relax the linear programming relaxation. We now consider one technique for doing so, called *Lagrangean relaxation*.

**Lagrangean relaxation.** *Lagrangean relaxation* is a technique that can be used to speed up the "bound" computation in branch-and-bound. Usually it is applied to integer or linear programs in which there are some "nice" inequalities and some "nasty" inequalities, in the sense that if the "nasty" inequalities were not present we would be able to solve the integer or linear program more easily (perhaps using a combinatorial algorithm, such as a network flow algorithm). The basic idea of Lagrangean relaxation is that we drop the nasty constraints from the integer program or linear program, but add penalties for their violation to the objective function.

For example, suppose we wish to solve the following integer program

$$\begin{array}{llll} & Z_A^* = \text{Min} & \sum_{j=1}^{n} c_j x_j & \\ & \text{subject to:} & & \\ (A) & & \sum_{j=1}^{n} a_{ij} x_j \geq d_i & i = 1, \ldots, p \\ & & \sum_{j=1}^{n} b_{ij} x_j \geq e_i & i = 1, \ldots, q \\ & & x_j \in \{0,1\} & j = 1, \ldots, n, \end{array}$$

where the integer program is easier to solve without the $q$ constraints $\sum_{j=1}^{n} b_{ij} x_j \geq e_i$. Then in Lagrangean relaxation, we introduce constants $\lambda_1, \ldots, \lambda_q \geq 0$ and create the following integer program

$$\begin{array}{lll} L^*(\lambda) = \text{Min} & \sum_{j=1}^{n} c_j x_j + \sum_{i=1}^{q} \lambda_i \left( e_i - \sum_{j=1}^{n} b_{ij} x_j \right) & \\ \text{subject to:} & & \\ & \sum_{j=1}^{n} a_{ij} x_j \geq d_i & i = 1, \ldots, p \\ & x_j \in \{0,1\} & j = 1, \ldots, n. \end{array}$$

Observe that we can rearrange terms in the objective function in the following way:

$$\begin{aligned} L(x, \lambda) &= \sum_{j=1}^{n} c_j x_j + \sum_{i=1}^{q} \lambda_i \left( e_i - \sum_{j=1}^{n} b_{ij} x_j \right) \\ &= \sum_{j=1}^{n} \left( c_j - \sum_{i=1}^{q} \lambda_i b_{ij} \right) x_j + \sum_{i=1}^{q} \lambda_i e_i. \end{aligned}$$

Thus if we define $\tilde{c}_j = c_j - \sum_{i=1}^{q} \lambda_i b_{ij}$, the Lagrangean relaxation is of the same form as the easily solvable integer program, except that it has a constant term in the objective function:

$$L^*(\lambda) = \quad \text{Min} \quad \sum_{j=1}^{n} \tilde{c}_j x_j + \sum_{i=1}^{q} \lambda_i e_i$$

subject to:

$$\begin{aligned} \sum_{j=1}^{n} a_{ij} x_j &\geq d_i & i &= 1, \ldots, p \\ x_j &\in \{0,1\} & j &= 1, \ldots, n. \end{aligned}$$

By hypothesis, we can solve this integer program easily. Furthermore, the value of its solution, $L^*(\lambda)$, is a lower bound on $(A)$, $Z_A^*$. To see this, note that for any feasible solution $x$ to $(A)$,

$$\sum_{i=1}^{q} \lambda_i \left( e_i - \sum_{j=1}^{n} b_{ij} x_j \right) \leq 0,$$

and so $L(x, \lambda) \leq \sum_{j=1}^{n} c_j x_j$. Since any $x$ feasible for $(A)$ is feasible for the Lagrangean relaxation and the Lagrangrean relaxation is a minimization problem, $L^*(\lambda) \leq Z_A^*$.

The technique can be applied to linear programs as well as integer programs; the discussion above carries through if the constraints $x_j \in \{0,1\}$ are replaced with $x_j \geq 0$. As a concrete example of how Lagrangean relaxation applies to linear programs, consider the problem $1|prec|\sum_j w_j C_j$. For the problem without precedence constraints, there is a linear program

$$\text{Min} \quad \sum_{j=1}^{n} w_j C_j$$

subject to:

$$\begin{aligned} \sum_{j=1}^{n} a_{ij} C_j &\geq d_i & &\forall i \\ C_j &\geq 0 & j &= 1, \ldots, n, \end{aligned}$$

which gives the optimal solution, where the variable $C_j$ gives the completion time of job $j$. There is a simple $O(n \log n)$ time algorithm to compute the solution (see Chapter 4 for details of the linear program and algorithm). We can get a better lower bound on $1|prec|\sum_j w_j C_j$ by adding constraints $C_l - C_k \geq p_k$ whenever $k \to l$. However, adding these constraints to the linear program makes it much harder to solve. Applying Lagrangean relaxation gives the linear program

$$L^*(\lambda) = \quad \text{Min} \quad \sum_{j=1}^{n} w_j C_j + \sum_{k,l:k\to l} \lambda_{kl}\,(p_k - C_l + C_k)$$

subject to:

$$\sum_{j=1}^{n} a_{ij} C_j \ge d_i \qquad \forall i$$
$$C_j \ge 0 \qquad j = 1,\dots,n.$$

Rearranging the objective function yields

$$\begin{aligned} L^*(\lambda) &= \sum_{j=1}^{n}\left(w_j - \sum_{k:k\to j}\lambda_{kj} + \sum_{l:j\to l}\lambda_{jl}\right)C_j + \sum_{k,l:k\to l}\lambda_{kl}p_k \\ &= \sum_{j=1}^{n}\tilde{w}_j C_j + \sum_{k,l:k\to l}\lambda_{kl}p_k, \end{aligned}$$

where $\tilde{w}_j = w_j - \sum_{k:k\to j}\lambda_{kj} + \sum_{l:j\to l}\lambda_{jl}$. We can then quickly obtain a lower bound on the cost of optimal solution for the instance of $1|prec|\sum_j w_j C_j$ by running the $O(n\log n)$ time algorithm on the $1||\sum_j w_j C_j$ instance with weights $\tilde{w}_j$, and adding $\sum_{k,l:k\to l}\lambda_{kl}p_k$ to the resulting value.

Of course, in this case and in general, we would like to compute the best bound possible; that is, we would like to compute the maximum of $L^*(\lambda)$ over all $\lambda \ge 0$. In some cases, the given problem has enough structure that we are able to find quickly the $\lambda$ that maximizes $L^*(\lambda)$. In general, however, we can use *subgradient optimization* to find a good value of $\lambda$. First, we claim that $L^*(\lambda)$ is a concave function of $\lambda$, so that any local maximum of $L^*(\lambda)$ is also a global maximum. We could thus use standard gradient ascent methods to find the global maximum, except that $L^*(\lambda)$ is not everywhere differentiable. Instead, we use a generalization of a gradient called a *subgradient*. A vector $\alpha \in \mathbb{R}^q$ is a subgradient of $L^*(\cdot)$ at $\lambda$ if

$$L^*(\lambda') \le L^*(\lambda) + (\lambda' - \lambda)^T \alpha$$

for all $\lambda' \ge 0$. For a subgradient $\alpha$, the set $\{\lambda' : (\lambda' - \lambda)^T\alpha \ge 0\}$ contains any $\lambda'$ for which $L^*(\lambda') > L^*(\lambda)$. Thus the subgradient gives a direction of ascent. Fortunately, a subgradient is easy to find. Consider the LP $(A)$. Let $x^*$ be an optimal solution for $\lambda$ so that $L^*(\lambda) = L(x^*, \lambda)$. Then the vector $\alpha$ with $\alpha_i = e_i - \sum_{j=1}^{n} b_{ij} x_j^*$ is a subgradient. To see this, choose some $\lambda' \ge 0$ and let $x'$ be an optimal solution for $\lambda'$

so that $L^*(\lambda') = L(x', \lambda')$. Then it is the case that

$$\begin{aligned}
L^*(\lambda') = L(x', \lambda') &\leq L(x^*, \lambda') \\
&= \sum_{j=1}^{n} c_j x_j^* + \sum_{i=1}^{q} \alpha_i \lambda_i' \\
&= L^*(x^*, \lambda) - \sum_{i=1}^{q} \alpha_i \lambda_i + \sum_{i=1}^{q} \alpha_i \lambda_i' \\
&= L^*(\lambda) + (\lambda' - \lambda)^T \alpha.
\end{aligned}$$

Subgradient optimization generally works by producing a sequence of $\lambda^i \geq 0$ in the following way: we start from some $\lambda^0$, compute $L^*(\lambda^i)$, find a subgradient $\alpha^i$, and then set $\lambda^{i+1} = \lambda^i + \delta^i \alpha^i \geq 0$, where $\delta^i > 0$ is a scalar step length. Note that in this case,

$$L^*(\lambda^{i+1}) \leq L^*(\lambda^i) + \delta^i \|\alpha^i\|^2,$$

so potentially the value of $L^*(\lambda^{i+1})$ has increased from $L^*(\lambda^i)$. The following theorem states that this sequence $\lambda^i$ converges to the optimum under certain conditions.

**Theorem 2.28.** *If $\delta^i \to 0$ as $i \to \infty$ and $\sum_{i=1}^{\infty} \delta^i = \infty$, then*

$$L^*(\lambda^i) \to \max_{\lambda \geq 0} L^*(\lambda).$$

The notes at the end of the chapter give references about Lagrangean relaxation and subgradient optimization.

## Acknowledgments

I am very grateful to Eugene Lawler, whose existing outline and drafts of this chapter proved a solid foundation on which I was able to build in my own fashion. I hope he would be pleased by the results. I am grateful to the editors, Jan Karel Lenstra and David Shmoys, for allowing me to participate in this volume. In addition, David Shmoys gave me many helpful suggestions on how to structure (and restructure) the chapter. I am also grateful to Greg Sorkin for many useful comments.

## Notes

2.1. *Algorithmic techniques*

*Combinatorial optimization.* For overviews of the field of combinatorial optimization, consult the books of Bertsimas and Tsitsiklis (1997), Cook, Cunningham, Pulleyblank, and Schrijver (1998), Lawler (1976), Nemhauser and Wolsey (1988), and Papadimitriou and Steiglitz (1982).

*Linear programming.* There are many books on linear programming. A nice introduction for beginners is the book by Chvátal (1983). Schrijver (1986) gives a technical, dense, and comprehensive overview of the area, although many of the recent developments in interior point methods for solving linear programs are not included. Wright (1997) gives an accessible treatment of this topic.

*Network flows.* For further reading on network flows, we recommend the book of Ahuja, Magnanti, and Orlin (1993). The books above on combinatorial optimization all include sections on network flows, as do many introductory texts on algorithms (see Cormen, Leiserson, and Rivest (1990), for example). The first proof of the maximum flow/minimum cut theorem was given by Ford and Fulkerson (1956). The proof of Theorem 2.5 given here is due to Bertsimas, Teo, and Vohra (1995). The algorithm for deciding the feasibility of $P|pmtn, r_j, \bar{d}_j|-$ given McNaughton's rule is due to Horn (1974).

*Dynamic programming.* The term "dynamic programming" was coined by Bellman (1957). He considers problems in continuous optimization. Other examples of dynamic programming applied to discrete optimization problems can be found in standard textbooks in algorithms, such as Cormen, Leiserson, and Rivest (1990). The knapsack algorithm given in Figure 2.3 is due to Bellman and Dreyfus (1962). The algorithm in Figure 2.4 is due to Lawler (1979).

2.2. *Analysis of algorithms.* The first example showing that the simplex method requires an exponential number of pivots for a certain pivot rule is due to Klee and Minty (1972); other researchers followed with examples for other pivot rules.

For further information about interior-point methods, consult the books of Wright (1997) and Ye (1997).

Ahuja, Magnanti, and Orlin (1993) discuss various types of network flow algorithms and their running times. The maximum flow algorithm with running time $O(\min(n^{2/3}, m^{1/2}) m \log(n^2/m) \log U)$ is due to Goldberg and Rao (1998). The more practical push-relabel algorithm was discovered by Goldberg and Tarjan (1988). Implementations of various maximum flow algorithms are studied in the volume edited by Johnson and McGeoch (1993). See also Cherkassky and Goldberg (1997) for an implementation of the push-relabel algorithm. The result showing that the parametric maximum flow problem can be solved in the same running time as the push-relabel algorithm is due to Gallo, Grigoriadis, and Tarjan (1989). The successive approximation push-relabel algorithm for minimum-cost flows is described in Goldberg and Tarjan (1990) and evaluated in Goldberg (1997). The implementation there is compared to network simplex codes of Grigoriadis (1986) and Kennington and Helgason (1980).

2.3. *Complexity theory and a notion of efficiency.* For overviews of the field of complexity theory, consult the books of Papadimitriou (1994) and Sipser (1997).

2.4. *Complexity theory and a notion of hardness.* A slightly dated, but still very worthwhile introduction to the theory of NP-completeness can be found in the book of Garey and Johnson (1979). Perhaps the most useful feature of the book is an appendix listing some 300 NP-complete problems. The notion of NP-completeness and the first NP-complete problems were given independently by Cook (1971) and

Levin (1973). However, Karp (1972) was the first to show that many problems from combinatorial optimization are NP-complete.

Lenstra, Rinnooy Kan, and Brucker (1977) give the proof of the NP-completeness of $1|r_j|L_{\max}$ in Theorem 2.19. The proof of the NP-completeness of $P||C_{\max}$ in Theorem 2.20 is due to Garey and Johnson (1978). The first proof of the NP-completeness of $P|prec, p_j = 1|C_{\max}$ is due to Ullman (1975). The proof we give in Theorem 2.21 is due to Lenstra and Rinnooy Kan (1978).

2.5. *Coping with* NP-*completeness*

*Approximation algorithms.* Shmoys (1995) gives an excellent survey of this area. For more in-depth treatments of approximation algorithms, we refer to the collection of surveys edited by Hochbaum (1997) and the textbooks by Vazirani (2001) and Williamson and Shmoys (2011).

The polynomial-time approximation scheme we give for the knapsack problem is due to Ibarra and Kim (1975).

*Branch-and-bound.* Surveys of Lagrangean relaxation are given by Geoffrion (1974) and Fisher (1981). The example of applying Lagrangean relaxation to $1|prec|\sum_j w_jC_j$ is due to Van de Velde (1990). Theorem 2.28 follows from work of Polyak (1967).