2 Basics

In this chapter we provide the reader with basic notions used throughout the book. After a short introduction into sets and relations, decision problems, optimization problems and the encoding of problem instances are discussed. The way algorithms will be represented and problem membership of complexity classes are other essential issues which will be discussed. Afterwards graphs, especially certain types such as precedence graphs and networks that are important for scheduling problems, are presented. The last two sections deal with algorithmic methods used in scheduling such as enumerative algorithms (e. g. dynamic programming and branch and bound) and heuristic approaches (e. g. tabu search, simulated annealing, ejection chains, and genetic algorithms).

2.1 Sets and Relations

Sets are understood to be any collection of distinguishable objects, such as the set $\{1, 2, \cdots\}$ of natural numbers, denoted by $\mathbb{N}$, the set $\mathbb{N}_0$ of non-negative integers, the set of real numbers, $\mathbb{R}$, or the set of non-negative reals $\mathbb{R}_{\geq 0}$. Given real numbers a and b, $a \leq b$, then $[a, b]$ denotes the *closed interval* from a to b, i.e. the set of reals $\{x \mid a \leq x \leq b\}$. *Open intervals* ($(a, b) := \{x \mid a < x < b\}$) and *half open intervals* are defined similarly.

In scheduling theory we are normally concerned with finite sets; so, unless infinity is stated explicitly, the sets are assumed to be finite.

For set $\mathcal{S}$, $|\mathcal{S}|$ denotes its *cardinality*. The *power set* of $\mathcal{S}$ (i.e. the set of all subsets of $\mathcal{S}$) is denoted by $\mathcal{P}(\mathcal{S})$. For an integer k, $0 \leq k \leq |\mathcal{S}|$, the set of all subsets of cardinality k is denoted by $\mathcal{P}_k(\mathcal{S})$.

The *Cartesian product* $\mathcal{S}_1 \times \cdots \times \mathcal{S}_k$ of sets $\mathcal{S}_1, \cdots, \mathcal{S}_k$ is the set of all tuples of the form $(s_1, s_2, \cdots, s_k)$ where $s_i \in \mathcal{S}_i$, $i = 1, \cdots, k$, i.e. $\mathcal{S}_1 \times \cdots \times \mathcal{S}_k = \{(s_1, \cdots, s_k) \mid s_i \in \mathcal{S}_i, i = 1, \cdots, k\}$. The k-fold Cartesian product $\mathcal{S} \times \cdots \times \mathcal{S}$ is denoted by $\mathcal{S}^k$.

Given sets $\mathcal{S}_1, \cdots, \mathcal{S}_k$, a subset $\mathcal{Q}$ of $\mathcal{S}_1 \times \cdots \times \mathcal{S}_k$ is called a *relation over* $\mathcal{S}_1, \cdots, \mathcal{S}_k$. In the case $k = 2$, $\mathcal{Q}$ is called a *binary relation*. For a binary relation $\mathcal{Q}$ over $\mathcal{S}_1$ and $\mathcal{S}_2$, the sets $\mathcal{S}_1$ and $\mathcal{S}_2$ are called *domain* and *range*, respectively. If $\mathcal{Q}$ is a relation over $\mathcal{S}_1, \cdots, \mathcal{S}_k$, with $\mathcal{S}_1 = \cdots = \mathcal{S}_k = \mathcal{S}$, then we simply say: $\mathcal{Q}$ is a (k-*ary*) *relation over* $\mathcal{S}$. For example, the set of edges of a directed graph (see Section 2.3) is a binary relation over the vertices of the graph.

J. Blazewicz et al., *Handbook on Scheduling*, International Handbooks on Information Systems, https://doi.org/10.1007/978-3-319-99849-7_2

Let $\mathcal{S}$ be a set, and $\mathcal{Q}$ be a binary relation over $\mathcal{S}$. Then, $\mathcal{Q}^{-1} = \{(a, b) \mid (b, a) \in \mathcal{Q}\}$ is the *inverse* to $\mathcal{Q}$. Relation $\mathcal{Q}$ is *symmetric* if $(a, b) \in \mathcal{Q}$ implies $(b, a) \in \mathcal{Q}$. $\mathcal{Q}$ is *antisymmetric* if for $a \neq b$, $(a, b) \in \mathcal{Q}$ implies $(b, a) \notin \mathcal{Q}$. $\mathcal{Q}$ is *reflexive* if $(a, a) \in \mathcal{Q}$ for all $a \in \mathcal{S}$. $\mathcal{Q}$ is *irreflexive* if $(a, a) \notin \mathcal{Q}$ for all $a \in \mathcal{S}$. $\mathcal{Q}$ is *transitive* if for all $a, b, c \in \mathcal{S}$, $(a, b) \in \mathcal{Q}$ and $(b, c) \in \mathcal{Q}$ implies $(a, c) \in \mathcal{Q}$.

A binary relation over $\mathcal{S}$ represents a *partial order*. A set with a binary relation is called a *partially ordered set* or *poset* if and only if it is reflexive, antisymmetric and transitive. A binary relation over $\mathcal{S}$ is called an *equivalence* relation (over $\mathcal{S}$) if it is reflexive, symmetric, and transitive.

Given set $\mathcal{J}$ of n closed intervals of reals, $\mathcal{J} = \{I_i \mid I_i = [a_i, b_i],\ a_i \leq b_i,\ i = 1, \cdots, n\}$, a partial order $\leq_I$ on $\mathcal{J}$ can be defined by

$$I_i \leq_I I_j \quad \Leftrightarrow \quad (I_i = I_j) \text{ or } (b_i \leq a_j),\ i, j \in \{1, \cdots, n\} .$$

A poset $\mathcal{Q}$ over $\mathcal{S}$ is called an *interval order* if and only if there exists a bijection from $\mathcal{S}$ to a set of intervals, $s_i \rightarrow I_i = [a_i, b_i]$, such that for any $s_i, s_j \in \mathcal{S}$ we have $(s_i, s_j) \in \mathcal{Q}$ exactly when $b_i < a_j$.

Let $\boldsymbol{l} = (n_1, \cdots, n_k)$ and $\boldsymbol{l}' = (n'_1, \cdots, n'_{k'})$ be sequences of integers, and $k, k' \geq 0$. If $k = 0$ then $\boldsymbol{l}$ is the empty sequence. We say that $\boldsymbol{l}$ is *lexicographically smaller* than $\boldsymbol{l}'$, written $\boldsymbol{l} \lessdot \boldsymbol{l}'$, if

(*i*) the two sequences agree up to some index j, but $n_{j+1} < n'_{j+1}$ (i.e. there exists j, $0 \leq j \leq k$, such that for all i, $1 \leq i \leq j$, $n_i = n'_i$ and $n_{j+1} < n'_{j+1}$), or if

(*ii*) sequence $\boldsymbol{l}$ is shorter, and the two sequences agree up to the length of $\boldsymbol{l}$ (i.e. $k < k'$ and $n_i = n'_i$ for all i, $1 \leq i \leq k$).

Let $\mathcal{Q}$ and $\mathcal{P}$ be binary relations over set $\mathcal{S}$. Then the *relational product* $\mathcal{Q} \circ \mathcal{P}$, defined as $\{(a, b) \mid \exists\, x \in \mathcal{S},\ (a, x) \in \mathcal{Q},\ (x, b) \in \mathcal{P}\}$, is a relation over $\mathcal{S}$. Generally, we write $\mathcal{Q}^0$ for $\{(a, a) \mid a \in \mathcal{S}\}$, $\mathcal{Q}^1 = \mathcal{Q}$, and $\mathcal{Q}^{i+1} = \mathcal{Q}^i \circ \mathcal{Q}$ for $i > 1$. The union $\mathcal{Q}^* = \cup\{\mathcal{Q}^i \mid i \geq 0\}$ is called the *transitive closure* of $\mathcal{Q}$.

A *function* from $\mathcal{A}$ to $\mathcal{B}$ ($\mathcal{A} \rightarrow \mathcal{B}$; $\mathcal{A}$ and $\mathcal{B}$ are not necessarily finite) is a relation F over $\mathcal{A}$ and $\mathcal{B}$ such that for each $a \in \mathcal{A}$ there exists just one $b \in \mathcal{B}$ for which $(a, b) \in F$; instead of $(a, b) \in F$ we usually write $F(a) = b$. Set $\mathcal{A}$ is called the *domain* of F and set $\{b \mid b \in \mathcal{B}, \exists\, a \in \mathcal{A}, (a, b) \in F\}$ is called the *range* of F. F is called *surjective*, or *onto* $\mathcal{B}$ if for each element $b \in \mathcal{B}$ there is at least one element $a \in \mathcal{A}$ such that $F(a) = b$. Function F is said to be *injective*, or *one-one* if for each pair of elements, $a_1, a_2 \in \mathcal{A}$, $F(a_1) = F(a_2)$ implies $a_1 = a_2$. A function that is both surjective and injective is called *bijective*. A bijective function F: $\mathcal{A} \rightarrow \mathcal{A}$ is called a *permutation* of $\mathcal{A}$. Though we are able to represent functions in special cases by means of tables we usually specify functions in a more or less abbreviated way that specifies how the function values are to be determined. For example, for $n \in \mathbb{N}$, the factorial function $n!$ denotes the set of pairs $\{(n, m) \mid n \in \mathbb{N},\ m = n \cdot (n-1) \cdots 3 \cdot 2\}$. Other examples of functions are polynomials, exponential functions and logarithms.

We will say that function $f\colon \mathbb{N} \to \mathbb{R}^+$ is *of order* g, written $O(g(k))$, if there exist constants c and $k_0 \in \mathbb{N}$ such that $f(k) \le cg(k)$ for all $k \ge k_0$.

2.2 Problems, Algorithms, Complexity

2.2.1 Problems and Their Encoding

In general, the scheduling problems we consider belong to a broader class of combinatorial search problems. A *combinatorial search problem* Π is a set of pairs (I, A), where I is called an *instance* of a problem, i.e. a finite set of *parameters* (understood generally, e.g. numbers, sets, functions, graphs) with specified values, and A is an *answer* (*solution*) to the instance. As an example of a search problem let us consider *merging* two sorted sequences of real numbers. Any instance of this problem consists of two finite sequences of reals $\boldsymbol{e}$ and $\boldsymbol{f}$ sorted in non-decreasing order. The answer is the sequence $\boldsymbol{g}$ consisting of all the elements of $\boldsymbol{e}$ and $\boldsymbol{f}$ arranged in non-decreasing order.

Let us note that among search problems one may also distinguish two subclasses: optimization and decision problems. An *optimization problem* is defined in such a way that an answer to its instance specifies a solution for which a value of a certain objective function is at its optimum (an *optimal solution*). On the other hand, an answer to an instance of a *decision problem* may take only two values, either "yes" or "no". It is not hard to see, that for any optimization problem, there always exists a decision counterpart, in which we ask (in the case of minimization) if there exists a solution with the value of the objective function less than or equal to some additionally given threshold value y. (If in the basic problem the objective function has to be maximized, we ask if there exists a solution with the value of the objective function $\ge y$.) The following example clarifies these notions.

Example 2.2.1 Let us consider an optimization *knapsack problem*.

Knapsack

Instance: A finite set of elements $\mathcal{A} = \{a_1, a_2, \cdots, a_n\}$, each of which has an integer weight $w(a_i)$ and value $v(a_i)$, and an integer capacity b of a knapsack.

Answer: Subset $\mathcal{A}' \subseteq \mathcal{A}$ for which $\sum_{a_i \in \mathcal{A}'} v(a_i)$ is at its maximum, subject to the constraint $\sum_{a_i \in \mathcal{A}'} w(a_i) \leq b$ (i.e. the total value of chosen elements is at its maximum and the sum of weights of these elements does not exceed knapsack capacity b).

The corresponding decision problem is denoted as follows. (To distinguish optimization problems from decision problems the latter will be denoted using capital letters.)

KNAPSACK

Instance: A finite set of elements $\mathcal{A} = \{a_1, a_2, \cdots, a_n\}$, each of which has an integer weight $w(a_i)$ and value $v(a_i)$, an integer knapsack capacity b and threshold value y.
Answer: "Yes" if there exists subset $\mathcal{A}' \subseteq \mathcal{A}$ such that

$$\sum_{a_i \in \mathcal{A}'} v(a_i) \geq y \text{ and } \sum_{a_i \in \mathcal{A}'} w(a_i) \leq b.$$

Otherwise "No". □

When considering search problems, especially in the context of their solution by computer algorithms, one of the most important issues that arises is a question of data structures used to encode problems. Usually to encode instance I of problem Π (that is particular values of parameters of problem Π) one uses a finite *string* of symbols $x(I)$. These symbols belong to a predefined finite set Σ (usually called an *alphabet*) and the way of coding instances is given as a set of encoding rules (called *encoding scheme e*). By *input length* (*input size*) $|I|$ of instance I we mean here the length of string $x(I)$. Let us note that the requirement that an instance of a problem is encoded by a finite string of symbols is the only constraint imposed on the class of search problems which we consider here. However, it is rather a theoretical constraint, since we will try to characterize algorithms and problems from the viewpoint of the application of real computers.

Now the encoding scheme and its underlying data structure is defined in a more precise way. For representation of mathematical objects we use set Σ that contains the usual characters, i.e. capital and small Arabic letters, capital and small Greek letters, digits $(0, \cdots, 9)$, symbols for mathematical operations such as $+$, $-$, $\times$, $/$, and various types of parentheses and separators. The class of mathematical objects, $\mathcal{A}$, is then mapped to the set Σ^* of words over the alphabet Σ by means of a function $\rho: \mathcal{A} \to \Sigma^*$, where Σ^* denotes the set of all finite strings (words) made up of symbols belonging to Σ. Each mathematical object $A \in \mathcal{A}$ is represented as a *structured string* in the following sense: Integers are represented by their decimal representation. A square matrix of dimension n with integer elements will be represented as a finite list whose first component represents matrix dimension n, and the following n^2 components represent the integer matrix elements in some specific order. For example, the list is a structured string of the form $(n, a(1, 1),\cdots, a(1, n), a(2, 1),\cdots, a(2, n),\cdots, a(n, n))$ where n and all the $a(i,j)$ are structured strings representing integers. The length of encoding (i.e. the complexity of storing) an integer k would then be of order $\log k$, and that of a matrix would be of order $n^2\log k$ where k is an upper bound for the absolute value of each matrix element. Real numbers will be represented either in decimal notation (e.g. 3.14159) or in half-logarithmic representation using mantissa and exponent (e.g. $0.314159\cdot 10^1$). Functions may be represented by tables which specify the function (range) value for each domain value. Representations of more complicated objects (e.g. graphs) will be introduced later, together with the definition of these types of objects.

As an example let us consider encoding of a particular instance of the knapsack problem defined in Example 2.2.1. Let the number n of elements be equal to 6 and let an encoding scheme define values of parameters in the following order: n, weights of elements, values of elements, knapsack's capacity b. A string coding an exemplary instance is: 6, 4, 2, 12, 15, 3, 7, 1, 4, 8, 12, 5, 7, 28.

The above remarks do not exclude the usage of any other *reasonable encoding scheme* which does not cause an exponential growth of the input length as compared with other encoding schemes. For this reason one has to exclude unary encoding in which each integer k is represented as a string of k 1's. We see that the length of encoding this integer would be k which is exponentially larger, as compared to the above decimal encoding.

In practice, it is worthwhile to express the input length of an instance as a function depending on the number of elements of some set whose cardinality is dominating for that instance. For the knapsack problem defined in Example 2.2.1 this would be the number of elements n, for the merging problem - the total number of elements in the two sequences, for the scheduling problem - the number of tasks. This assumption, usually made, in most cases reduces practically to the assumption that a computer word is large enough to contain any of the binary encoded numbers comprising an instance. However, in some problems, for example those in which graphs are involved, taking as input size the number of nodes may appear too great a simplification since the number of edges in a graph may be equal to $n(n-1)/2$. Nevertheless, in practice one often makes this simplification to unify computational results. Let us note that this simplification causes no exponential growth of input length.

2.2.2 Algorithms

Let us now pass to the notion of an algorithm and its complexity function. An algorithm is any finite procedure for solving a problem (i.e. for giving an answer). We will say that an algorithm *solves search problem* Π, if it finds a solution for any instance I of Π. In order to keep the representation of algorithms easily understandable we follow a structural approach that uses language concepts known from structural programming, such as case statements, or loops of various kinds. Like functions or procedures, algorithms may also be called in an algorithm. Parameters may be used to import data to or export data from the algorithm. Besides these, we also use mathematical notations such as set-theoretic notations.

In general, an algorithm consists of two parts: a *head* and a *method*. The head starts with the keyword **Algorithm**, followed by an identifying number and, optionally, a descriptor (a name or a description of the purpose of the algorithm) and a reference to the author(s) of the algorithm. Input and output parameters are omitted in cases where they are clear from the context. In other cases, they are specified as a parameter list. In even more complex cases, two fields, *Input* (*Instance*)*:* and *Output* (*Answer*)*:* are used to describe parameters, and a field *Method:* is used to describe the main idea of the algorithm. As in PASCAL, a block is embraced by **begin** and **end**. Each block is considered as a sequence of instructions. An instruction itself may again be a block, an assignment-, an else-, or a case- operation, or a loop (**for**, **while**, **repeat** $\cdots$ **until**, or a general **loop**), a **call** of another algorithm, or an exit instruction to terminate a loop instruction (**exit loop**, etc.) or the algorithm or procedure (just **exit**). The right hand side of an assignment operation may be any mathematical expression, or a function call. Case statements partition actions of the algorithm into several branches, depending on the value of a control variable. Loop statements may contain formulations such as: "**for all** $a \in \mathcal{M}$ **do** $\cdots$" or " **while** $\mathcal{M} \neq \emptyset$ **do** $\cdots$". If a loop is preempted by an exit statement the algorithm jumps to the first statement after the loop. Comments are started with two minus signs and are finished at the end of the line. If a comment needs more than one line, each comment line starts with '--'.

Algorithms should reflect the main idea of the method. Details like output layouts are omitted. Names for procedures, functions, variables etc. are chosen so that they reflect the semantics behind them. As an example let us consider an algorithm solving the problem of merging two sequences as defined at the beginning of this section.

Algorithm 2.2.2 *merge*.
Input: Two sequences of reals, $\boldsymbol{e} = (e[1],\cdots,e[n])$ and $\boldsymbol{f} = (f[1],\cdots,f[m])$, both sorted in non-decreasing order.
Output: Sequence $\boldsymbol{g} = (g[1],\cdots,g[n+m])$ in which all elements are arranged in non-decreasing order.

```
begin
$i := 1;\ j := 1;\ k := 1;$      -- initialization of counters
while $(i \le n)$ and $(j \le m)$ do
        -- the while loop merges elements of sequences $\boldsymbol{e}$ and $\boldsymbol{f}$ into $\boldsymbol{g}$;
        -- the loop is executed until all elements of one of the sequences are merged
```

```
  begin
  if $e[i] < f[j]$
  then begin $g[k] := e[i]$; $i := i+1$; end
  else begin $g[k] := f[j]$; $j := j+1$; end;
  $k := k+1$;
  end;
if $i \le n$          -- not all elements of sequence e have been merged
then for $l := i$ to $n$ do $g[k+l-i] := e[l]$
else
  if $j \le m$        -- not all elements of sequence f have been merged
  then for $l := j$ to $m$ do $g[k+l-j] := f[l]$;
end;
```

The above algorithm returns as an answer sequence $\boldsymbol{g}$ of all the elements of $\boldsymbol{e}$ and $\boldsymbol{f}$, sorted in non-decreasing order of the values of all the elements.

As another example, consider the search problem of sorting in non-decreasing order a sequence $\boldsymbol{e} = (e[1], \cdots, e[n])$ of $n = 2^k$ reals (i.e. n is a power of 2). The algorithm *sort* (Algorithm 2.2.4) uses two other algorithms that operate on sequences: *msort*(i,j) and *merge*$1(i,j,k)$. If the two parameters of *msort*, i and j, obey $1 \le i < j \le n$, then *msort*(i,j) sorts the elements of the subsequence $(e[i], \cdots, e[j])$ of $\boldsymbol{e}$ non-decreasingly. Algorithm *merge*1 is similar to *merge* (Algorithm 2.2.2): *merge*$1(i,j,k)$ $(1 \le i \le j < k \le n)$ takes the elements from the two adjacent and already sorted subsequences $(e[i], \cdots, e[j])$ and $(e[j+1], \cdots, e[k])$ of $\boldsymbol{e}$, and merges their elements into $(e[i], \cdots, e[k])$.

Algorithm 2.2.3 *msort*(i,j).

```
begin
case $(i,j)$ of    -- depending on relative values of i and j,
                   -- three subcases are considered
  $i=j$: exit;      -- terminate msort
  $i=j-1$: if $e[i] > e[j]$ then Exchange $e[i]$ and $e[j]$;
  $i<j-1$:
    begin
    call $msort(i, \lfloor (j+i)/2 \rfloor)$;[1]
      -- sorts elements of subsequence $(e[i], \cdots, e[\lfloor (j+i)/2 \rfloor])$
    call $msort(\lfloor (j+i)/2 \rfloor + 1, j)$;
      -- sorts elements of subsequence $(e[\lfloor (j+i)/2 \rfloor + 1], \cdots, e[j])$
    call $merge1(i, \lfloor (j+i)/2 \rfloor, j)$;
```

[1] $\lfloor x \rfloor$ denotes the largest number less than or equal to x.

```
        -- merges sorted subsequences into sequence $(e[i],\cdots,e[j])$
      end;
    end;
end;
```

Algorithm 2.2.4 *sort*.

```
begin
$read(n)$;
$read((e[1],\cdots,e[n]))$;
call $msort(1,n)$;
end;
```

Notice that in the case of an optimization problem one may also consider an *approximate* (*sub-optimal*) *solution* that is *feasible* (i.e. fulfills all the conditions specified in the description of the problem) but does not extremize the objective function. It follows that one can also consider *heuristic* (*sub-optimal*) algorithms which tend toward but do not guarantee the finding of optimal solutions for any instance of an optimization problem. An algorithm which always finds an optimal solution will be called an *optimization* or *exact* algorithm.

2.2.3 Complexity

Let us turn now to the analysis of the computational complexity of algorithms. By the *time complexity function* of algorithm A solving problem Π we understand the function that maps each input length of an instance I of Π into a maximal number of elementary steps (or time units) of a computer, which are needed to solve an instance of that size by algorithm A.

It is obvious that this function will not be well defined unless the encoding scheme and the model of computation (computer model) are precisely defined. It appears, however, that the choice of a particular reasonable encoding scheme and a particular realistic computer model has no influence on the distinction between polynomial- and exponential time algorithms which are the two main types of algorithms from the computational complexity point of view [AHU74]. This is because all realistic models of computers [2] are equivalent in the sense that if a problem is solved by some computer model in time bounded from above by a polynomial in the input length (i.e. in polynomial time), then any other computer model will solve that problem in time bounded from above by a polynomial (perhaps of different degree) in the input length [AHU74]. Thus, to simplify the

[2] By "realistic" we mean here such computer models which in unit time may perform a number of elementary steps bounded from above by a polynomial in the input length. This condition is fulfilled for example by the one-tape Turing machine, the k-tape Turing machine, or the random access machine (RAM) under logarithmic cost of performing a single operation.

computation of the complexity of polynomial algorithms, we assume that, if not stated otherwise, the operation of writing a number as well as addition, subtraction and comparison of two numbers are elementary operations of a computer that need the same amount of time, if the length of a binary encoded number is bounded from above by a polynomial in the computation time of the whole algorithm. Otherwise, a logarithmic cost criterion is assumed. Now, we define the two types of algorithms.

A *polynomial time* (*polynomial*) *algorithm* is one whose time complexity function is $O(p(k))$, where p is some polynomial and k is the input length of an instance. Each algorithm whose time complexity function cannot be bounded in that way will be called an *exponential time algorithm*.

Let us consider two algorithms with time complexity functions k and 3^k, respectively. Let us assume moreover that an elementary step lasts 1 μs and that the input length of the instance solved by the algorithms is $k = 60$. Then one may calculate that the first algorithm solves the problem in 60 μs while the second needs $1.3 \cdot 10^{13}$ centuries. This example illustrates the fact that indeed the difference between polynomial- and exponential time algorithms is large and justifies definition of the first algorithm as a "good" one and the second as a "bad" one [Edm65].

If we analyze time complexity of Algorithm 2.2.2, we see that the number of instructions being performed during execution of the algorithm is bounded by $c_1(n+m)+c_2$, where c_1 and c_2 are suitably chosen constants, i.e. the number of steps depends linearly on the total number of elements to be merged.

Now we estimate the time complexity of Algorithm 2.2.4. The first two *read* instructions together take $O(n)$ steps, where reading one element is assumed to take constant ($O(1)$) time. During execution of *msort*$(1, n)$, the sequence of elements is divided into two subsequences, each of length $n/2$; *msort* is applied recursively on the subsequences which will thus be sorted. Then, procedure *merge*1 is applied, which combines the two sorted subsequences into one sorted sequence. Now let $T(m)$ be the number of steps *msort* performs to sort m elements. Then, each call of *msort* within *msort* involves sorting of $m/2$ elements, so it takes $T(m/2)$ time. The call of *merge*1 can be performed in a number of steps proportional to $m/2+m/2 = m$, as can easily be seen. Hence, we get the recursion

$$T(m) = 2T(m/2)+cm\,,$$

where c is some constant. One can easily verify that there is a constant c' such that $T(m) = c'm\log m$ solves the recursion[3]. Taking all steps of Algorithm 2.2.4 together we get the time complexity $O(\log n)+O(n)+O(n\log n) = O(n\log n)$.

Unfortunately, it is not always true that we can solve problems by algorithms of linear or polynomial time complexity. In many cases only exponential algorithms are available. We will take now a closer look to inherent complexity of

[3] We may take any fixed base for the logarithm, e.g. 2 or 10.

some classes of search problems to explain the reasons why polynomial algorithms are unlikely to exist for these problems.

As we said before, there exist two broad subclasses of search problems: decision and optimization problems. From the computational point of view both classes may be analyzed much in the same way (strictly speaking when their computational hardness is analyzed). This is because a decision problem is computationally not harder than the corresponding optimization problem. That means that if one is able to solve an optimization problem in an "efficient" way (i.e. in polynomial time), then it will also be possible to solve a corresponding decision problem efficiently (just by comparing an optimal value of the objective function [4] to a given constant y). On the other hand, if the decision problem is computationally "hard", then the corresponding optimization problem will also be "hard" [5].

Now, we can turn to the definition of the most important complexity classes of search problems. Basic definitions will be given for the case of decision problems since their formulation permits an easier treatment of the subject. One should, however, remember the above dependencies between decision and optimization problems. We will also point out the most important implications. In order to be independent of a particular type of a computer we have to use an abstract model of computation. From among several possibilities, we choose the *deterministic Turing machine* (*DTM*) for this purpose. Despite the fact that this choice was somehow arbitrary, our considerations are still *general* because all the realistic models of computations are polynomially related.

Class ***P*** consists of all decision problems that may be solved by the deterministic Turing machine in time bounded from above by a polynomial in the input length. Let us note that the corresponding (broader) class of all search problems solvable in polynomial time, is denoted by ***FP*** [Joh90a]. We see that both, the problem of merging two sequences and that of sorting a sequence belong to that class. In fact, class ***FP*** contains all the search problems which can be solved efficiently by the existing computers.

It is worth noting that there exists a large class of decision problems for which no polynomial time algorithms are known, for which, however, one can verify a positive answer in polynomial time, provided there is some additional information. If we consider for example an instance of the KNAPSACK problem defined in Example 2.2.1 and a subset $\mathcal{A}_1 \subseteq \mathcal{A}$ defining additional information, we may easily check in polynomial time whether or not the answer is "yes" in the case of this subset. This feature of polynomial time verifiability rather than solvability is captured by a *non-deterministic Turing machine* (*NDTM*) [GJ79].

---

[4] Strictly speaking, it is assumed that the objective function may be calculated in polynomial time.

[5] Many decision problems and corresponding optimization problems are linked even more strictly, since it is possible to prove that a decision problem is not easier than the corresponding optimization problem [GJ79].

We may now define *class* ***NP*** of decision problems as consisting of all decision problems which may be solved in polynomial time by an NDTM.

It follows that $\boldsymbol{P} \subseteq \boldsymbol{NP}$. In order to define the most interesting class of decision problems, i.e. the class of NP-complete problems, one has to introduce the definition of a polynomial transformation. A *polynomial transformation* from problem Π_2 to problem Π_1 (denoted by $\Pi_2 \propto \Pi_1$) is a function f mapping the set of all instances of Π_2 into the set of instances of Π_1, that satisfies the following two conditions:

1. for each instance I_2 of Π_2 the answer is "yes" if and only if the answer for $f(I_2)$ of Π_1 is also "yes",
2. f is computable in polynomial time (depending on problem size $|I_2|$) by a DTM.

We say that decision problem Π_1 is *NP-complete* if $\Pi_1 \in \boldsymbol{NP}$ and for any other problem $\Pi_2 \in \boldsymbol{NP}$, $\Pi_2 \propto \Pi_1$ [Coo71].

It follows from the above that if there existed a polynomial time algorithm for some NP-complete problem, then any problem from that class (and also from the ***NP*** class of decision problems) would be solvable by a polynomial time algorithm. Since NP-complete problems include classical hard problems (as for example HAMILTONIAN CIRCUIT, TRAVELING SALESMAN, SATISFIABILITY, INTEGER PROGRAMMING) for which, despite many attempts, no one has yet been able to find polynomial time algorithms, probably all these problems may only be solved by the use of exponential time algorithms. This would mean that ***P*** is a proper subclass of ***NP*** and the classes ***P*** and NP-complete problems are disjoint.

Another consequence of the above definitions is that, to prove the NP-completeness of a given problem Π, it is sufficient to transform polynomially a known NP-complete problem to Π. SATISFIABILITY was the first decision problem proved to be NP-complete [Coo71]. The current list of NP-complete problems contains several thousands, from different areas. Although the choice of an NP-complete problem which we use to transform into a given problem in order to prove the NP-completeness of the latter, is theoretically arbitrary, it has an important influence on the way a polynomial transformation is constructed [Kar72]. Thus, these proofs require a good knowledge of NP-complete problems, especially characteristic ones in particular areas.

As was mentioned, decision problems are not computationally harder than the corresponding optimization ones. Thus, to prove that some optimization problem is computationally hard, one has to prove that the corresponding decision problem is NP-complete. In this case, the optimization problem belongs to the class of *NP-hard problems*, which includes computationally hard search problems. On the other hand, to prove that some optimization problem is easy, it is sufficient to construct an optimization polynomial time algorithm. The order of performing these two steps follows mainly from the intuition of the researcher, which however, is guided by several hints. In this book, by "open problems" from the computational complexity point of view we understand those problems which neither have been proved to be NP-complete nor solvable in polynomial time.

Despite the fact that all NP-complete problems are computationally hard, some of them may be solved quite efficiently in practice (as for example the KNAPSACK problem). This is because the time complexity functions of algorithms that solve these problems are bounded from above by polynomials in two variables: the input length $|I|$ and the maximal number $\max(I)$ appearing in an instance I. Since in practice $\max(I)$ is usually not very large, these algorithms have good computational properties. However, such algorithms, called *pseudopolynomial*, are not really of polynomial time complexity since in reasonable encoding schemes all numbers are encoded binary (or in another integer base greater than 2). Thus, the length of a string used to encode $\max(I)$ is $\log \max(I)$ and the time complexity function of a polynomial time algorithm would be $O(p(|I|, \log \max(I)))$ and not $O(p(|I|, \max(I)))$, for some polynomial p. It is also obvious that pseudopolynomial algorithms may perhaps be constructed for *number problems*, i.e. those problems Π for which there does not exist a polynomial p such that $\max(I) \leq p(|I|)$ for each instance I of Π. The KNAPSACK problem as well as TRAVELING SALESMAN and INTEGER PROGRAMMING belong to number problems; HAMILTONIAN CIRCUIT and SATISFIABILITY do not. However, there might be number problems for which pseudopolynomial algorithms cannot be constructed [GJ78].

The above reasoning leads us to a deeper characterization of a class of NP-complete problems by distinguishing problems which are NP-complete in the strong sense [GJ78, GJ79].

For a given decision problem Π and an arbitrary polynomial p, let Π_p denote the subproblem of Π which is created by restricting Π to those instances for which $\max(I) \leq p(|I|)$. Thus Π_p is not a number problem.

Decision problem Π is *NP-complete in the strong sense* (*strongly NP-complete*) if $\Pi \in \boldsymbol{NP}$ and there exists a polynomial p defined for integers for which Π_p is NP-complete.

It follows that if Π is NP-complete and it is not a number problem, then it is NP-complete in the strong sense. Moreover, if Π is NP-complete in the strong sense, then the existence of a pseudopolynomial algorithm for Π would be equivalent to the existence of polynomial algorithms for all NP-complete problems, and thus would be equivalent to the equality $\boldsymbol{P} = \boldsymbol{NP}$. It has been shown that TRAVELING SALESMAN and 3-PARTITION are examples of number problems that are NP-complete in the strong sense [GJ79, Pap94].

From the above definition it follows that to prove NP-completeness in the strong sense for some decision problem Π, one has to find a polynomial p for which Π_p is NP-complete, which is usually not an easy way. To make this proof easier one may use the concept of pseudopolynomial transformation [GJ78].

To end this section, let us stress once more that the membership of a given search problem in class ***FP*** or in the class of NP-hard problems does not depend on the chosen encoding scheme if this scheme is reasonable as defined earlier. The differences in input lengths for a given instance that follow from particular encoding schemes have only influence on the complexity of the polynomial (if the problem belongs to class ***FP***) or on the complexity of the exponential algorithm (if the problem is NP-hard). On the other hand, if numbers are written unary, then pseudopolynomial algorithms would become polynomial because of the artificial increase in input lengths. However, problems NP-hard in the strong sense would remain NP-hard even in the case of such an encoding scheme. Thus, they are also called *unary NP-hard* [LRKB77].

2.3 Graphs and Networks

2.3.1 Basic Notions

A *graph* is a pair $G = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V}$ is the set of *vertices* or *nodes*, and $\mathcal{E}$ is the set of *edges*. If $\mathcal{E}$ is a binary relation over $\mathcal{V}$, then G is called a *directed* graph (or *digraph*). If $\mathcal{E}$ is a set of two-element subsets of $\mathcal{V}$, i.e. $\mathcal{E} \subseteq \mathcal{P}_2(\mathcal{V})$, then G is an *undirected* graph.

A graph $G' = (\mathcal{V}', \mathcal{E}')$ is a *subgraph* of $G = (\mathcal{V}, \mathcal{E})$ (denoted by $G' \subseteq G$), if $\mathcal{V}' \subseteq \mathcal{V}$, and $\mathcal{E}'$ is the set of all edges of $\mathcal{E}$ that connect vertices of $\mathcal{V}'$.

Let $G_1 = (\mathcal{V}_1, \mathcal{E}_1)$ and $G_2 = (\mathcal{V}_2, \mathcal{E}_2)$ be graphs whose vertex sets $\mathcal{V}_1$ and $\mathcal{V}_2$ are not necessarily disjoint. Then $G_1 \cup G_2 = (\mathcal{V}_1 \cup \mathcal{V}_2, \mathcal{E}_1 \cup \mathcal{E}_2)$ is the *union* graph of G_1 and G_2, and $G_1 \cap G_2 = (\mathcal{V}_1 \cap \mathcal{V}_2, \mathcal{E}_1 \cap \mathcal{E}_2)$ is the *intersection* graph of G_1 and G_2.

Digraphs G_1 and G_2 are *isomorphic* if there is a bijective mapping χ: $\mathcal{V}_1 \to \mathcal{V}_2$ such that $(v_1, v_2) \in \mathcal{E}_1$ if and only if $(\chi(v_1), \chi(v_2)) \in \mathcal{E}_2$.

A (undirected) *path* in a graph or in a digraph $G = (\mathcal{V}, \mathcal{E})$ is a sequence $i_1, \cdots, i_r$ of distinct nodes of $\mathcal{V}$ satisfying the property that either $(i_k, i_{k+1}) \in \mathcal{E}$ or $(i_{k+1}, i_k) \in \mathcal{E}$ for each $k = 1, \cdots, r-1$. A *directed path* is defined similarly, except that $(i_k, i_{k+1}) \in \mathcal{E}$ for each $k = 1, \cdots, r-1$. A (undirected) *cycle* is a path together with an edge (i_r, i_1) or (i_1, i_r). A *directed cycle* is a directed path together with the edge (i_r, i_1). We will call a graph (digraph) G *acyclic* if it contains no (directed) cycle.

Two vertices i and j of G are said to be *connected* if there is at least one undirected path between i and j. G is *connected* if all pairs of vertices are connected; otherwise it is *disconnected.*

Let v and w be vertices of the digraph $G = (\mathcal{V}, \mathcal{E})$. If there is a directed path from v to w, then w is called *successor* of v, and v is called *predecessor* of w. If $(v, w) \in \mathcal{E}$, then vertex w is called *immediate successor* of v, and v is called *immediate predecessor* of w. The set of immediate successors of vertex v is denoted by isucc(v); the sets succ(v), ipred(v), and pred(v) are defined similarly. The cardinality of ipred(v) is called *in-degree* of vertex v, whereas *out-degree* is the cardinality of isucc(v). A vertex v that has no immediate predecessor is called *initial vertex* (i.e. ipred(v) = $\emptyset$); a vertex v having no immediate successors is called *final* (i.e. isucc(v) = $\emptyset$).

Directed or undirected graphs can be represented by means of their adjacency matrix. If $\mathcal{V} = \{v_1, \cdots, v_n\}$, the *adjacency matrix* is a binary $n \times n$-matrix A. In case of a directed graph, $A(i,j) = 1$ if there is an edge from v_i to v_j, and $A(i,j) = 0$ otherwise. In case of an undirected graph, $A(i,j) = 1$ if there is an edge between v_i and v_j, and $A(i,j) = 0$ otherwise. The complexity of storage (space complexity) is $O(n^2)$. If the adjacency matrix is sparse, as e.g. in case of trees, there are better ways of representation, usually based on *linked lists*. For details we refer to [AHU74].

In many situations, it is appropriate to use a generalization of graphs called hypergraphs. Following [Ber73] a finite *hypergraph* is a pair $H = (\mathcal{V}, \mathcal{H})$ where $\mathcal{V}$ is a finite set of vertices, and $\mathcal{H} \subseteq \mathcal{P}(\mathcal{V})$ is a set of subsets of $\mathcal{V}$. The elements of $\mathcal{H}$ are referred to as *hyperedges*. Hypergraphs can be represented as bipartite graphs (see below): Let G_H be the graph whose vertex set is $\mathcal{V} \cup \mathcal{H}$, and the set of edges is defined as $\{\{v, h\} \mid h \in \mathcal{H}, \text{ and } v \in h\}$.

2.3.2 Special Classes of Digraphs

A digraph $G = (\mathcal{V}, \mathcal{E})$ is called *bipartite* if its vertex set $\mathcal{V}$ can be partitioned into two subsets $\mathcal{V}_1$ and $\mathcal{V}_2$ such that for each edge $(i,j) \in \mathcal{E}$, $i \in \mathcal{V}_1$ and $j \in \mathcal{V}_2$.

If a digraph $G = (\mathcal{V}, \mathcal{E})$ contains no directed cycle and no transitive edges (i.e. pairs (u, w) of vertices for which there exists a different directed path from u to w), it will be called a *precedence graph*. A corresponding binary relation will be called a *precedence relation* $\prec$ over set $\mathcal{V}$. A precedence graph $G = (\mathcal{V}, \mathcal{E})$ (we also write $(\mathcal{V}, \prec)$, where $\prec$ is the corresponding precedence relation) can always be enlarged to a partially ordered set (poset, see Section 2.1) $\prec^*$ by adding transitive edges and all reflexive pairs (v, v) $(v \in \mathcal{V})$ to $\mathcal{E}$. On the other hand, given a poset $(\mathcal{V}, Q)$, where Q is a partial order over set $\mathcal{V}$, we can always construct a precedence graph $(\mathcal{V}, \mathcal{E})$ in the following way: $\mathcal{E}$ is obtained by taking those pairs of elements (u, w), $u \neq w$, for which no sequence $v_1, \cdots, v_k$ of elements with $(u, v_1) \in Q$, $(v_i, v_{i+1}) \in Q$ for $i = 1,\cdots,k-1$, and $(v_k, w) \in Q$ can be found. It can be constructed from a given poset in $O(|\mathcal{V}|^{2.8})$ time [AHU74].

A digraph $G = (\mathcal{V}, \mathcal{E})$ is called a *chain* if in the corresponding poset $(\mathcal{V}, Q)$ for any two vertices v and $v' \in \mathcal{V}$, $v \neq v'$, either $(v, v') \in Q$ or $(v', v) \in Q$ (such a poset is usually called a *linear order*). An *anti-chain* is a (directed) graph $(\mathcal{V}, \mathcal{E})$ where $\mathcal{E} = \varnothing$.

An *out-tree* is a precedence graph where exactly one vertex has in-degree 0, and all the other vertices have in-degree 1. If $G = (\mathcal{V}, \mathcal{E})$ is an out-tree, then graph $G' = (\mathcal{V}, \mathcal{E}^{-1})$ is called an *in-tree*. An *out-forest* (*in-forest*) is a disjoint union of out-trees (in-trees), respectively. An *opposing forest* is a disjoint union of in-trees and out-trees.

A precedence graph $(\{a, b, c, d\}, \prec)$ has *N-structure* if $a \prec c$, $b \prec c$, $b \prec d$, $a \nprec d$, $d \nprec a$, $a \nprec b$, $b \nprec a$, $c \nprec d$, and $d \nprec c$ (see also Figure 2.3.1). A precedence graph P is *N-free* if it contains no subset isomorphic to an N-structure.

To define another interesting class of graphs let us consider a finite set $\mathcal{V}$ and a collection $(I_v)_{v\in\mathcal{V}}$ of intervals I_v on the reals. This collection defines a partial order $\prec$ on $\mathcal{V}$ as follows:

$$v \prec w \Leftrightarrow I_v \text{ is entirely before } I_w .$$

Such a partial order is called an *interval order*. Without loss of generality, we may assume that the intervals have the form $[n_1, n_2)$ with n_1 and n_2 integral. It can be shown that $\prec$ is an interval order if and only if the transitive closure of this order does not contain $2K_2$ (see Figure 2.3.2) as an induced subgraph [Fis70].

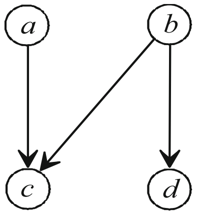

Figure 2.3.1 *N-structured precedence graph.*

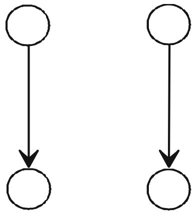

Figure 2.3.2 *Graph* $2K_2$.

Finally we introduce a class of precedence graphs that has been considered frequently in literature. Let $S = (\mathcal{V}, \prec)$ be a precedence graph, and let for each $v \in \mathcal{V}$, $P_v = (\mathcal{V}_v, \prec_v)$ be a precedence graph, where all the sets $\mathcal{V}_v$ $(v \in \mathcal{V})$ and $\mathcal{V}$ are pair-wise disjoint. Let $\mathcal{U} = \bigcup_{v\in\mathcal{V}} \mathcal{V}_v$. Define $(\mathcal{U}, \prec_{\mathcal{U}})$ as the following precedence graph: for $p, q \in \mathcal{U}$, $p \prec_{\mathcal{U}} q$ if either there are $v, v' \in \mathcal{V}$ with $v \prec v'$ such that p is a final vertex in $(\mathcal{V}_v, \prec_v)$ and q is an initial vertex in $(\mathcal{V}_{v'}, \prec_{v'})$, or there is $v \in \mathcal{V}$ with $p, q \in \mathcal{V}_v$, and $p \prec_v q$. Then $(\mathcal{U}, \prec_{\mathcal{U}})$ is called the *lexicographic sum* of $(P_v)_{v \in \mathcal{V}}$ over S. Notice that each vertex v of the digraph $S = (\mathcal{V}, \prec)$ is replaced by the digraph $(\mathcal{V}_v, \prec_v)$, and if vertex v is connected to v' in S (i.e. $v \prec v'$), then each final vertex of $(\mathcal{V}_v, \prec_v)$ is connected to each initial vertex of $(\mathcal{V}_{v'}, \prec_{v'})$.

We need two special cases of lexicographic sums: If $S = (\mathcal{V}, \prec)$ is a chain, the lexicographic sum of $(P_v)_{v \in \mathcal{V}}$ over S is called a *linear sum*. If S is an antichain (i.e. $v_1 \prec v_2 \Rightarrow v_1 = v_2$), then the lexicographic sum of $(P_v)_{v \in \mathcal{V}}$ over S is called *disjoint sum*. A *series-parallel* precedence graph is a precedence graph that can be constructed from one-vertex precedence graphs by repeated application of the operations linear sum and disjoint sum. Opposing forests are examples of series-parallel digraphs. Another example is shown in Figure 2.3.3.

Without proof we mention some properties of series-parallel graphs. A precedence graph $G = (\mathcal{V}, \mathcal{E})$ is series-parallel if and only if it is N-free. The question if a digraph is series-parallel can be decided in $O(|\mathcal{V}| + |\mathcal{E}|)$ time [VTL82].

The structure of a series-parallel graph as it is obtained by successive applications of linear sum and disjoint sum operations can be displayed by a *decomposition tree*. Figure 2.3.4 shows a decomposition tree for the series-parallel graph of Figure 2.3.3. Each leaf of the decomposition tree is identified with a vertex of the series-parallel graph. An *S-node* represents an application of linear sum (series composition) to the sub-graphs identified with its children; the ordering of these children is important: we adopt the convention that left precedes right. A *P-node* represents an application of the operation of disjoint sum (parallel composition) to the subgraphs identified with its children; the ordering of these children is of no relevance for the disjoint sum. The series or parallel relationship of any pair of vertices can be determined by finding their least common ancestor in the decomposition tree.

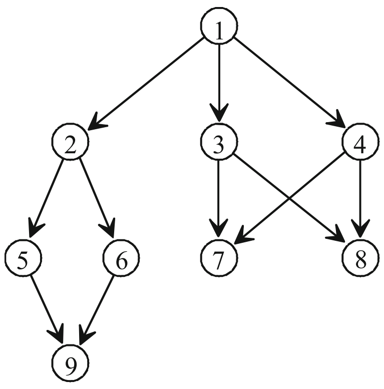

Figure 2.3.3 *Example of a series-parallel digraph.*

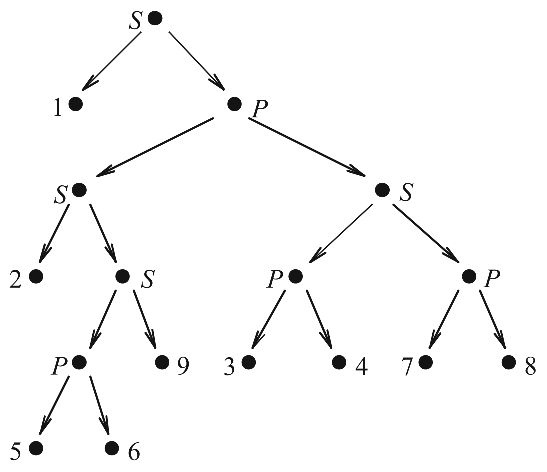

Figure 2.3.4 *Decomposition tree of the digraph of Figure* 2.3.3.

2.3.3 Networks

In this section the problem of finding a maximum flow in a network is considered. We will analyze the subject rather thoroughly because of its importance for many scheduling problems.

By a *network* we will mean a directed graph $G = (\mathcal{V}, \mathcal{E})$ without loops and parallel edges, where each edge $e \in \mathcal{E}$ is assigned a *capacity* $c(e) \in \mathbb{R}_{\geq 0}$, and sometimes a cost of a unit flow. Usually in the network two vertices s and t, called a *source* and a *sink,* respectively, are specified.

A real-valued *flow* function ρ is to be assigned to each edge such that the following conditions hold for some $F \in \mathbb{R}_{\geq 0}$:

$$0 \leq \rho(e) \leq c(e) \text{ for each } e \in \mathcal{E}, \tag{2.3.1}$$

$$\sum_{e \in IN(v)} \rho(e) - \sum_{e \in OUT(v)} \rho(e) = \begin{cases} -F & \text{for } v = s \\ 0 & \text{for } v \in \mathcal{V} - \{s, t\} \\ F & \text{for } v = t, \end{cases} \tag{2.3.2}$$

where $IN(v)$ and $OUT(v)$ are the sets of edges *incoming* to vertex v and *outgoing* from vertex v, respectively. The *total flow* (the *value of flow*) F of ρ is defined by

$$F := \sum_{e \in IN(t)} \rho(e) - \sum_{e \in OUT(t)} \rho(e). \tag{2.3.3}$$

Given a network, in the *maximum flow problem* we want to find a flow function ρ which obeys the above conditions and for which total flow F is at its maximum.

Now, some important notions will be defined and their properties will be discussed. Let $\mathcal{S}$ be a subset of the set of vertices $\mathcal{V}$ such that $s \in \mathcal{S}$ and $t \notin \mathcal{S}$, and let $\overline{\mathcal{S}}$ be the complement of $\mathcal{S}$, i.e. $\overline{\mathcal{S}} = \mathcal{V} - \mathcal{S}$. Let $(\mathcal{S}, \overline{\mathcal{S}})$ denote a set of edges of network G, each of which has its starting vertex in $\mathcal{S}$ and its target vertex in $\overline{\mathcal{S}}$. Set $(\overline{\mathcal{S}}, \mathcal{S})$ is defined in a similar way. Given some subset $\mathcal{S} \subseteq \mathcal{V}$, either set, $(\mathcal{S}, \overline{\mathcal{S}})$ and $(\overline{\mathcal{S}}, \mathcal{S})$, will be called *cut* defined by $\mathcal{S}$.

Following definition (2.3.3) we see that the value of flow is measured at the sink of the network. It is however, possible to measure this value at any cut [Eve79, FF62].

Lemma 2.3.1 *For each subset of vertices $\mathcal{S} \subseteq \mathcal{V}$, we have*

$$F = \sum_{e \in (\mathcal{S}, \overline{\mathcal{S}})} \rho(e) - \sum_{e \in (\overline{\mathcal{S}}, \mathcal{S})} \rho(e) . \tag{2.3.4}$$

□

Let us denote by $c(\mathcal{S})$ the *capacity of a cut* defined by $\mathcal{S}$,

$$c(\mathcal{S}) = \sum_{e \in (\mathcal{S}, \overline{\mathcal{S}})} c(e) . \tag{2.3.5}$$

It is possible to prove the following lemma, which specifies a relation between the value of a flow and the capacity of any cut [FF62].

Lemma 2.3.2 *For any flow function ρ having the value F and for any cut defined by $\mathcal{S}$ we have*

$$F \le c(\mathcal{S}) . \tag{2.3.6}$$

□

From the above lemma we get immediately the following corollary that specifies a relation between maximum flow and a cut of minimum capacity.

Corollary 2.3.3 *If $F = c(\mathcal{S})$, then F is at its maximum, and $\mathcal{S}$ defines a cut of minimum capacity.* □

Let us now define, for a given flow ρ, an *augmenting path* as a path from s to t, (not necessarily directed), which can be used to increase the value of the flow. If an edge e belonging to that path is directed from s to t, then $\rho(e) < c(e)$, otherwise no increase in the flow value on that path would be possible. On the other hand, if such an edge e is directed from t to s, then $\rho(e) > 0$ must be satisfied in order to be able to increase the flow value F by decreasing $\rho(e)$.

Example 2.3.4 As an example let us consider the network given in Figure 2.3.5(a). Each edge of this network is assigned two numbers, $c(e)$ and $\rho(e)$. It is easy to check that flow ρ in this network obeys conditions (2.3.1) and (2.3.2) and its value is equal to 3. An augmenting path is shown in Figure 2.3.5(b). The flow on edge $(5, 4)$ can be decreased by one unit. All the other edge flows on that path can be increased by one unit. The resulting network with a new flow is shown in Figure 2.3.5(c). □

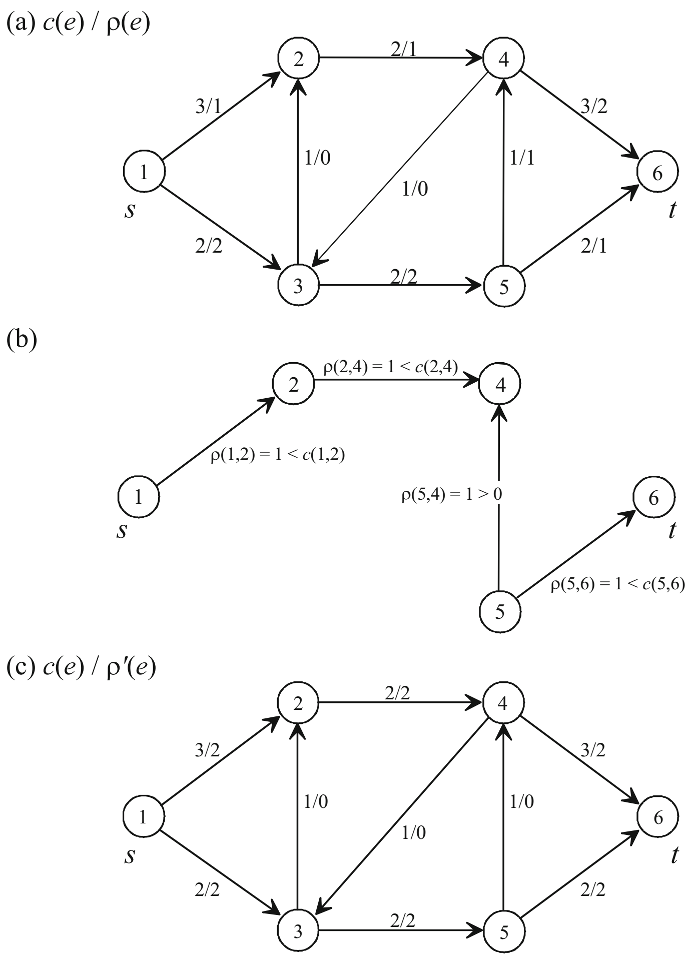

Figure 2.3.5 *A network for Example* 2.3.4:
(a) *a flow* $\rho(e)$ *is assigned to each edge,*
(b) *an augmenting path,*
(c) *a new flow* $\rho'(e)$.

The first method proposed for the construction of a flow of a maximum value was given by Ford and Fulkerson [FF62]. This method consists in finding an augmenting path in a network and increasing the flow value along this path until no such path remains in the network. Convergence of such a general method could be proved for integer capacities only. A corresponding algorithm is of pseudopolynomial complexity [FF62, Eve79].

An important improvement of the above algorithm was made by Edmonds and Karp [EK72]. They showed that if the shortest augmenting path is chosen at every step, then the complexity of the algorithm reduces to $O(|\mathcal{V}|^3|\mathcal{E}|)$, no matter what are the edge capacities. Further improvements in algorithmic efficiency of network flow algorithm were made by Dinic [Din70] and Karzanov [Kar74], whose algorithms' running times are $O(|\mathcal{V}|^2|\mathcal{E}|)$ and $O(|\mathcal{V}|^3)$, respectively. An algorithm proposed by Cherkassky [Che77] allows for solving the max-flow problem in time $O(|\mathcal{V}|^2|\mathcal{E}|^{1/2})$.

Below, Dinic's algorithm will be described, since despite its relatively high worst case complexity function, its average running time is low [Che80], and the idea behind it is quite simple. It uses the notion of a *layered network* which contains all the shortest paths in a network. This allows for a parallel increase of flows in all such paths, which is the main reason of the efficiency of the algorithm.

In order to present this algorithm, the notion of *usefulness* of an edge for a given flow is introduced. We say that edge e having flow $\rho(e)$ is *useful* from u to v, if one of the following conditions is fulfilled:

1) if the edge is directed from u to v then $\rho(e) < c(e)$;
2) otherwise, $\rho(e) > 0$.

For a given network $G = (\mathcal{V}, \mathcal{E})$ and flow ρ, the following algorithm determines a corresponding layered network.

Algorithm 2.3.5 *Construction of a layered network for a given network* $G = (\mathcal{V}, \mathcal{E})$ *and flow function* ρ [Din70].

```
begin
Set $\mathcal{V}_0 := \{s\}$; $\mathcal{T} := \{\emptyset\}$; $i := 0$;
while $t \notin \mathcal{T}$ do
  begin
  Construct subset $\mathcal{T} := \{v \mid v \notin \mathcal{V}_j$ for $j \le i$ and there exists a useful edge
     from any of the vertices of $\mathcal{V}_i$ to $v\}$;
     -- subset $\mathcal{T}$ contains vertices comprising a new layer of the layered network
  $\mathcal{V}_{i+1} := \mathcal{T}$;    -- a new layer of the network has been constructed
  $i := i+1$;
  if $\mathcal{T} = \emptyset$ then exit;
     -- no layered network exists, the flow value $F$ is at its maximum
  end;
$l := i$; $\mathcal{V}_l := \{t\}$;
```

```
for $j := 1$ to $l$ do
  begin
  $\mathcal{E}_j := \{e \mid e$ is a useful edge from a vertex belonging to layer $\mathcal{V}_{j-1}$ to a vertex
     belonging to layer $\mathcal{V}_j\}$;
  for all $e \in \mathcal{E}_j$ do
    if $e = (u,v)$ and $u \in \mathcal{V}_{j-1}$ and $v \in \mathcal{V}_j$
    then $\tilde{c}(e) := c(e) - \rho(e)$
    else
      if $e = (v,u)$ and $u \in \mathcal{V}_{j-1}$ and $v \in \mathcal{V}_j$
      then
        begin
        $\tilde{c}(e) := \rho(e)$;
        Change the orientation of the edge, so that $e = (u,v)$;
        end;
  end;    -- a layered network with new edges and capacities has been constructed
end;
```

In such a layered network a new flow function $\tilde{\rho}$ with $\tilde{\rho} = 0$ for each edge e is assumed. Then a maximal flow is searched for, i.e. one such that for each path $v_0\ (= s),\ v_1,\ v_2, \cdots, v_{l-1},\ v_l\ (= t)$, where $e_j = (v_{j-1}, v_j) \in \mathcal{E}_j$ and $v_j \in \mathcal{V}_j$, $j = 1, 2, \cdots, l$, there exists at least one edge e such that $\tilde{\rho}(e_j) = \tilde{c}(e_j)$.

Let us note, that such a maximal flow may not be of maximum value. This fact is illustrated in Figure 2.3.6 where all capacities $\tilde{c}(e) = 1$. The flow depicted in this figure is maximal and its value $F = 1$. It is not hard, however, to construct a flow of value $F = 2$.

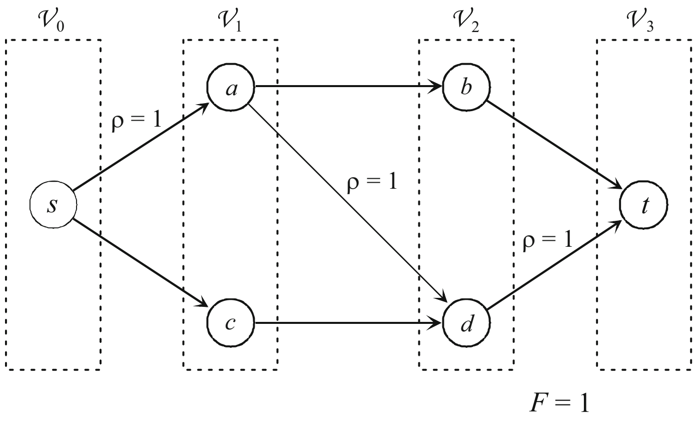

Figure 2.3.6 *An example of a maximal flow which is not of maximum value.*

The construction of a maximal flow for a given layered network is shown below. It consists in finding augmenting paths by means of a *labeling procedure*. For this purpose a depth first search label algorithm is used, that labels all the nodes of the layered network, i.e. assigns to node u, if any, a label $lab(e)$ that corresponds to edge $e = (v, u)$ in a layered network. The algorithm uses for each node v a list isucc(v) of all immediate successors of v (i.e. all nodes u for which an arc (v, u) exists in the layered network). Let us note that, if v belongs to layer $\mathcal{V}_j$, then $u \in$ isucc(v) belongs to layer $\mathcal{V}_{i+1}$, and edge $(v, u) \in \mathcal{E}_j$. The algorithm uses recursively an algorithm *label*(v) that labels nodes being successors of v. Boolean variable *new*(v) is used to check whether or not a given node has been visited and consequently labeled. The algorithms are as follows.

Algorithm 2.3.6 *label*(v).

```
begin
$new(v)$ := false;          -- node $v$ has been visited and labeled
for all $u \in$ isucc($v$) do
if $new(u)$ then
  begin
  if $e = (v,u) \in \bigcup_{j=1}^{l} \mathcal{E}_j$ then $lab(u) := e$;
  call $label(u)$;
  end;          -- all successors of node $v$ have been labeled
end;
```

Algorithm 2.3.7 *label*.

```
begin
$lab(s) := 0$;     -- a source of layered network has been labeled
for all $v \in \mathcal{V}$ do $new(v)$ := true;       -- initialization
call $label(s)$;
end;       -- all successors of $s$ in the layered network are now visited and labeled
```

Using the above algorithms as subroutines the following algorithm constructs a maximal flow in the layered network. The algorithm will stop whenever no augmenting path exists; in this case the flow is maximal [Din70] (see also [Eve79]).

Algorithm 2.3.8 *Construction of a maximal flow in a layered network* [Din70].

```
begin
for all $e \in \bigcup_{j=1}^{l} \mathcal{E}_j$ do
  begin
  $\rho_1(e) := \tilde{\rho}(e) := 0$;
  $c_1(e) := \tilde{c}(e)$;
  end;        -- initialization phase
loop
  call label;     -- all nodes, if any, have been labeled
  if node $t$ is not labeled then exit;
    -- no augmenting path exists
    -- a maximal flow in a layered network has been constructed
  Find an augmenting path $ap$ starting from node $t$ backward and using labels;
  $\Delta := \min\{c_1(e) \mid e \in ap\}$;
  for all $e \in ap$ do
    begin
    $\rho_1(e) := \Delta$;
    $\tilde{\rho}(e) := \tilde{\rho}(e) + \rho_1(e)$;
    $c_1(e) := c_1(e) - \Delta$;
    end;     -- the value of a flow is increased along an augmenting path
  for all $e$ with $c_1(e) = 0$ do Delete $e$ from the layered network;
  repeat
    Delete all nodes which have either no incoming or no outgoing edges;
    Delete all edges incident with such nodes;
  until all such edges and nodes are deleted;
  for all $e \in \bigcup_{j=1}^{l} \mathcal{E}_j$ do $\rho_1(e) := 0$;
end loop;
end;
```

The flow constructed by the above algorithm is used to obtain a new flow in the original network. Next, a new layered network is created and the above procedure is repeated until no new layered network can be constructed. The obtained flow has a maximum value. This is summarized in the next algorithm.

Algorithm 2.3.9 *Construction of a flow of maximum value* [Din70].

```
begin
$\rho(e) := 0$ for all $e \in \mathcal{E}$;
loop
  call Algorithm 2.3.5;
    -- a new layered network is constructed for a flow function $\rho$
    -- if no layered network exists, then the flow has maximum value
  call Algorithm 2.3.8;       -- a new maximal flow $\tilde{\rho}$ is constructed
  for all $e \in \mathcal{E}$ do
    begin
    if $u \in \mathcal{V}_{j-1}$ and $v \in \mathcal{V}_j$ and $e = (u, v) \in \mathcal{E}$
    then $\rho(e) := \rho(e) + \tilde{\rho}(e)$;
      -- the value of the flow increases if edge $e$ has the same direction
```

```
            -- in the original and in the layered network
        if $u \in \mathcal{V}_{j-1}$ and $v \in \mathcal{V}_j$ and $e = (v, u) \in \mathcal{E}$
        then $\rho(e) := \rho(e) - \tilde{\rho}(e)$;
            -- the value of the flow decreases if edge $e$ has opposite directions
            -- in the original and in the layered network
        end;
    -- the flow in the original network is augmented using the
    -- constructed maximal flow values
end loop;
end;
```

To analyze the complexity of the above approach let us call one loop of Algorithm 2.3.9 a *phase*. We see that one phase consists of finding a layered network, constructing a maximal flow $\tilde{\rho}$ in the latter and improving the flow in the original network. It can be proved [Din70, Eve79] that the number of phases is bounded from above by $O(|\mathcal{V}|)$. The most complex part of each phase is to find a maximal flow in a layered network. Since in Algorithm 2.3.8 a depth first search procedure has been used for visiting a network, the complexity of one phase is $O(|\mathcal{V}||\mathcal{E}|)$. The overall complexity of Dinic's approach is thus $O(|\mathcal{V}|^2|\mathcal{E}|)$.

Further generalizations of the subject include networks with lower bounds on edge flows, networks with linear total cost function of the flow where a flow of maximum value and of minimum total cost is looked for, and a transportation problem being a special case of the latter. All these problems can be solved in time bounded from above by a polynomial in the number of nodes and edges of the network. We refer the reader to [AMO93] or [Law76] where a detailed analysis of the subject is presented.

2.4 Enumerative Methods

In this section we describe very briefly two general methods of solving many combinatorial problems [6], namely the method of dynamic programming and the method of branch and bound. Few remarks should be made at the beginning, concerning the scope of this presentation. First, we will not go into details, since both methods are broadly treated in literature, including basic scheduling books [Bak74, Len77, Rin76a], and our presentation should only fulfill the needs of this book. In particular, we will not perform a comparative study of the methods - the interested reader is referred to [Cof76]. We will also not present examples, since they will be given in the later chapters.

[6] Dynamic programming can also be used in a wider context (see e.g. [Den82, How69, DL79]).

Before passing to the description of the methods let us mention that they are of *implicit enumeration* variety, because they consider certain solutions only indirectly, without actually evaluating them explicitly.

2.4.1 Dynamic Programming

Fundamentals of *dynamic programming* were elaborated by Bellman in the 1950's and presented in [Bel57, BD62]. The name "Dynamic Programming" is slightly misleading, but generally accepted. A better description would be "recursive" or "multistage" optimization, since it interprets optimization problems as *multistage decision processes.* It means that the problem is divided into a number of stages, and at each stage a decision is required which impacts on the decisions to be made in later stages. Now, Bellman's principle of optimality is applied to draw up a recursive equation which describes the optimal criterion value at a given stage in terms of the previously obtained one. This principle can be formulated as follows: Starting from any current stage, an optimal policy for the rest of the process, i.e. for subsequent stages, is independent of the policy adopted in the previous stages. Of course, not all optimization problems can be presented as multistage decision processes for which the above principle is true. However, the class of problems for which it works is quite large. For example, it contains problems with an additive optimality criterion, but also other problems as we will show in Sections 5.1.1 and 10.4.3.

If dynamic programming is applied to a combinatorial problem, then in order to calculate the optimal criterion value for any subset of size k, we first have to know the optimal value for each subset of size $k-1$. Thus, if our problem is characterized by a set of n elements, the number of subsets considered is 2^n. It means that dynamic programming algorithms are of exponential computational complexity. However, for problems which are NP-hard (but not in the strong sense) it is often possible to construct pseudopolynomial dynamic programming algorithms which are of practical value for reasonable instance sizes.

2.4.2 Branch and Bound

Suppose that given a finite[7] set $\mathcal{S}$ of feasible solutions and a criterion $\gamma : \mathcal{S} \rightarrow I\!R$, we want to find $S^* \in \mathcal{S}$ such that $\gamma(S^*) = \min_{S \in \mathcal{S}}\{\gamma(S)\}$.

Branch and bound finds S^* by implicit enumeration of all $S \in \mathcal{S}$ through examination of increasingly smaller subsets of $\mathcal{S}$. These subsets can be treated as sets of solutions of corresponding sub-problems of the original problem. This

[7] In general, $|\mathcal{S}|$ can be infinite (see, e.g. [Mit70, Rin76b]).

way of thinking is especially motivated if the considered problems have a clear practical interpretation, and we will adopt this interpretation in the book.

As its name implies, the branch and bound method consists of two fundamental procedures: branching and bounding. *Branching* is the procedure of partitioning a large problem into two or more sub-problems usually mutually exclusive[8]. Furthermore, the sub-problems can be partitioned in a similar way, etc. *Bounding* calculates a *lower bound* on the optimal solution value for each sub-problem generated in the branching process. Note that the branching procedure can be conveniently represented as a *search* (or *branching*) *tree*. At level 0, this tree consists of a single node representing the original problem, and at further levels it consists of nodes representing particular sub-problems of the problem at the previous level. Edges are introduced from each problem node to each of its sub-problems nodes. A list of unprocessed nodes (also called active nodes) corresponding to sub-problems that have not been eliminated and whose own sub-problems have not yet been generated, is maintained.

Suppose that at some stage of the branch and bound process a (complete) solution S of criterion value $\gamma(S)$ has been obtained. Suppose also that a node encountered in the process has an associated lower bound $LB > \gamma(S)$. Then the node needs not be considered any further in the search for S^*, since the resulting solution can never have a value less than $\gamma(S)$. When such a node is found, it is eliminated, and its branch is said to be *fathomed*, since we do not continue the bounding process from it. The solution used for checking if a branch is fathomed is sometimes called a *trial* solution. At the beginning it may be found using a special heuristic procedure, or it can be obtained in the course of the tree search, e.g. by pursuing the tree directly to the bottom as rapidly as possible. At any later stage the best solution found so far can be chosen as a trial one. The value $\gamma(S)$ for a trial solution S is often called an *upper bound.* Let us mention that a node can be eliminated not only on the basis of lower bounds but also by means of so-called elimination criteria provided by dominance properties or feasibility conditions developed for a given problem.

The choice of a node from the set of generated nodes which have so far neither been eliminated nor led to branching is due to the chosen *search strategy.* Two search strategies are used most frequently: jumptracking and backtracking. *Jumptracking* implements a *frontier search* where a node with a minimal lower bound is selected for examination, while *backtracking* implements a *depth first search* where the descendant nodes of a parent node are examined either in an arbitrary order or in order of non-decreasing lower bounds. Thus, in the jumptracking strategy the branching process jumps from one branch of the tree to another, whereas in the backtracking strategy it first proceeds directly to the bottom along some path to find a trial solution and then retraces that path upward up to the first level with active nodes, and so on. It is easy to notice that jumptracking tends to construct a fairly large list of active nodes, while backtracking maintains

[8] If this is not the case, we speak rather about a division of S instead of its partition.

relatively few nodes on the list at any time. However, an advantage of jumptracking is the quality of its trial solutions which are usually much closer to optimum than the trial solutions generated by backtracking, especially at early stages. Deeper comparative discussion of characteristics of the search strategies can be found in [Agi66, LW66].

Summing up the above considerations we can say that in order to implement the scheme of the branch and bound method, i.e. in order to construct a branch and bound algorithm for a given problem, one must decide about

(*i*) the branching procedure and the search strategy,

(*ii*) the bounding procedure or elimination criteria.

Making the above decisions one should explore the problem specificity and observe the compromise between the length of the branching process and time overhead concerned with computing lower bounds or trial solutions. However, the actual computational behavior of branch and bound algorithms remains unpredictable and large computational experiments are necessary to recognize their quality. It is obvious that the computational complexity function of a branch and bound algorithm is exponential in problem size when we search for an optimal solution. However, the approach is often used for finding suboptimal solutions, and then we can obtain polynomial time complexity by stopping the branching process at a certain stage or after a certain time period elapsed.

2.5 Heuristic and Approximation Algorithms

As already mentioned, scheduling problems belong to a broad class of combinatorial optimization problems (cf. Section 2.2.1). To solve these problems one tends to use optimization algorithms which for sure always find optimal solutions. However, not for all optimization problems, polynomial time optimization algorithms can be constructed. This is because some of the problems are NP-hard. In such cases one often uses *heuristic* (*suboptimal*) *algorithms* which tend toward but do not guarantee the finding of optimal solutions for any instance of an optimization problem. Of course, the necessary condition for these algorithms to be applicable in practice is that their worst-case complexity function is bounded from above by a low-order polynomial in the input length. A sufficient condition follows from an evaluation of the distance between the solution value they produce and the value of an optimal solution. This evaluation may concern the worst case or a mean behavior.

2.5.1 Approximation Algorithms

We will call heuristic algorithms with analytically evaluated accuracy *approximation algorithms*. To be more precise, we give here some definitions, starting with the worst case analysis [GJ79].

If Π is a minimization (maximization) problem, and I is any instance of it, we may define the ratio $R_A(I)$ for an approximation algorithm A as

$$R_A(I) = \frac{A(I)}{OPT(I)} \qquad \left(R_A(I) = \frac{OPT(I)}{A(I)}\right),$$

where $A(I)$ is the value of the solution constructed by algorithm A for instance I, and $OPT(I)$ is the value of an optimal solution for I. The *absolute performance ratio* R_A for an approximation algorithm A for problem Π is then given as

$$R_A = \inf\{r \geq 1 \mid R_A(I) \leq r \text{ for all instances of } \Pi\}.$$

The *asymptotic performance ratio* R_A^∞ for A is given as

$$R_A^\infty = \inf\{r \geq 1 \mid \text{for some positive integer } K,\ R_A(I) \leq r \text{ for all instances of } \Pi \text{ satisfying } OPT(I) \geq K\}.$$

The above formulas define a measure of the "goodness" of approximation algorithms. The closer R_A^∞ is to 1, the better algorithm A performs.

More formally, an algorithm A is called *ρ-approximation algorithm* for problem Π, if for all instances I it constructs a feasible solution such that

$$|A(I) - OPT(I)| \leq \varepsilon \cdot OPT(I),$$

where $\varepsilon > 0$, $\rho = 1 + \varepsilon$ for a minimization problem and $\rho = 1 - \varepsilon$ for a maximization problem. For a minimization problem, we have $A(I) \leq (1 + \varepsilon)\ OPT(I)$, while for a maximization problem there is $A(I) \geq (1 - \varepsilon)\ OPT(I)$. The *worst case ratio ρ* (or in other words, the *absolute performance ratio R_A*) is the quality measure for an approximation algorithm. However, for some combinatorial problems it can be proved that there is no hope of finding an approximation algorithm of a specified accuracy, i.e. this question is as hard as finding a polynomial time algorithm for any NP-complete problem. For other combinatorial problems an approximation algorithm can be proposed, and even an approximation scheme can be designed. An *approximation scheme* is a family of $(1 + \varepsilon)$-approximation algorithms over all $0 < \varepsilon < 1$ for a minimization problem, or a family of $(1 - \varepsilon)$-approximation algorithms for a maximization problem. A *polynomial time approximation scheme* (PTAS) is an approximation scheme of the polynomial time complexity in the instance size, while a *fully polynomial time approximation scheme* (FPTAS) is an approximation scheme with the complexity bounded by the polynomial in the instance size and in $1/\varepsilon$. Obviously, such types of approximation methods are especially interesting, since they allow finding a trade-off between the quality of a solution and the time complexity necessary to construct it. Fully polynomial time approximation schemes are the best methods which could be proposed for an NP-hard problem.

The relations between some classes of problems with respect to the existence of methods solving them, discussed e.g. by Shuurman and Woeginger [SW07], are shown in Figure 2.5.1. It completes the presentation of the basic complexity classes of combinatorial problems, given in Section 2.2.3, with some algorithmic issues. Particular classes depicted in this figure correspond to problems from ***NP*** possessing polynomial time algorithms (***P***), pseudopolynomial time algorithms, polynomial time approximation algorithms with finite/positive worst case ratio ρ for minimization/maximization case (*APX*), or approximation schemes (*PTAS* and *FPTAS*).

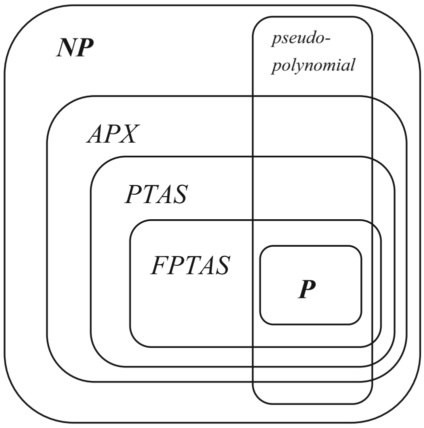

Figure 2.5.1 *Relations between classes of problems possessing various types of solution methods* [SW07].

Analysis of the worst-case behavior of an approximation algorithm may be complemented by an analysis of its mean behavior. This can be done in two ways. The first consists in assuming that the parameters of instances of the considered problem Π are drawn from a certain distribution D and then one analyzes the *mean performance* of algorithm A.

In such an analysis it is usually assumed that all parameter values are realizations of independent probabilistic variables of the same distribution function. Then, for an instance I_n of the considered optimization problem (n being a number of generated parameters) a probabilistic value analysis is performed. The result is an asymptotic value $OPT(I_n)$ expressed in terms of problem parameters. Then, algorithm A is probabilistically evaluated by comparing solution values $A(I_n)$ it produces ($A(I_n)$ being independent probabilistic variables) with $OPT(I_n)$ [Rin87]. The two evaluation criteria used are absolute error and relative error. The *absolute error* is defined as a difference between the approximate and optimal solution values

$$a_n = A(I_n) - OPT(I_n) .$$

On the other hand, the *relative error* is defined as the ratio of the absolute error and the optimal solution value

$$b_n = \frac{A(I_n) - OPT(I_n)}{OPT(I_n)} .$$

Usually, one evaluates the convergence of both errors to zero. Three types of convergence are distinguished. The strongest, i.e. *almost sure convergence* for a sequence of probabilistic variables y_n which converge to constant c is defined as

$$Pr\{\lim_{n\to\infty} y_n = c\} = 1 .$$

The latter implies a weaker *convergence in probability*, which means that for every $\varepsilon > 0$,

$$\lim_{n\to\infty} Pr\{|y_n - c| > \varepsilon\} = 0 .$$

The above convergence implies the first one if the following additional condition holds for every $\varepsilon > 0$:

$$\sum_{j=1}^{\infty} Pr\{|y_n - c| > \varepsilon\} < \infty .$$

Finally, the third type of convergence, *convergence in expectation* holds if

$$\lim_{n\to\infty} |E(y_n) - c| = 0 ,$$

where $E(y_n)$ is the mean value of y_n.

It follows from the above definitions, that an approximation algorithm A is the best from the probabilistic analysis point of view if its absolute error almost surely converges to 0. Algorithm A is then called *asymptotically optimal.*

At this point one should also mention an analysis of the *rate of convergence* of the errors of approximation algorithms which may be different for algorithms whose absolute or relative errors are the same. Of course, the higher the rate, the better the performance of the algorithm.

It is rather obvious that the mean performance can be much better than the worst case behavior, thus justifying the use of a given approximation algorithm. A main obstacle is the difficulty of proofs of the mean performance for realistic distribution functions. Thus, the second way of evaluating the mean behavior of heuristic algorithms are computational experiments, which is still used very often. In the latter approach the values of the given criterion, constructed by the given heuristic algorithm and by an optimization algorithm are compared. This comparison should be made for a representative sample of instances. There are some practical problems which follow from the above statement and they are discussed in [SVW80].

2.5.2 Local Search Heuristics

In recent years more generally applicable heuristic algorithms for combinatorial optimization problems became known under the name *local search*. Primarily, they are designed as universal global optimization methods operating on a high-level solution space in order to guide heuristically lower-level local decision rules' performance to their best outcome. Hence, local search heuristics are often called *meta-heuristics* or strategies with knowledge-engineering and learning capabilities reducing uncertainty while knowledge of the problem setting is exploited and acquired in order to improve and accelerate the optimization process. The desire to achieve a certain outcome may be considered as the basic guide to appropriate knowledge modification and inference as a process of transforming some input information into the desired goal dependent knowledge.

Hence, in order to be able to transform knowledge, one needs to perform inference and to have memory which supplies the background knowledge needed to perform the inference and records the results of the inference for future use. Obviously, an important issue is the extent to which problem-specific knowledge must be used in the construction of *learning* algorithms (in other words the power and quality of inferencing rules) capable to provide significant performance improvements. Very general methods having a wide range of applicability in general are weak with respect to their performance. Problem specific methods achieve a highly efficient learning but with little use in other problem domains. Local search strategies are falling somewhat in between these two extremes, where genetic algorithms or neural networks tend to belong to the former category while tabu search or simulated annealing etc. are counted as examples of the second category. Anyway, these methods can be viewed as tools for searching a space of legal alternatives in order to find a best solution within reasonable time limitations. What is required are techniques for rapid location of high-quality solutions in large-size and complex search spaces and without any guarantee of optimality. When sufficient knowledge about such search spaces is available a priori, one can often exploit that knowledge (inference) in order to introduce problem-specific search strategies capable of supporting to find rapidly solutions of higher quality. Without such an a priori knowledge, or in cases where close to optimum solutions are indispensable, information about the problem has to be accumulated dynamically during the search process. Likewise obtained long-term as well as short-term memorized knowledge constitutes one of the basic parts in order to control the search process and in order to avoid getting stuck in a locally optimal solution. Previous approaches dealing with combinatorially explosive search spaces about which little knowledge is known a priori are unable to learn how to escape a local optimum. For instance, consider a random search. This can be effective if the search space is reasonably dense with acceptable solutions, such that the probability to find one is high. However, in most cases finding an acceptable solution within a reasonable amount of time is impossible because random search is not using any knowledge generated during the search process in order to improve its performance. Consider hill-climbing in which better solutions are found by exploring solutions "close" to a current and best one found so far. Hill-climbing techniques work well within a search space with relatively "few" hills. Iterated hill-climbing from randomly selected solutions can frequently improve the performance, however, any global information assessed during the search will not be exploited. Statistical sampling techniques are typical alternative approaches which emphasize the accumulation and exploitation of more global information. Generally speaking they operate by iteratively dividing the search space into regions to be sampled. Regions unlikely to produce acceptable solutions are discarded while the remaining ones will be subdivided for further sampling. If the number of useful sub-regions is small this search process can be effective. However, in case that the amount of a priori search space knowledge is pretty small, as is the case for many applications in business and engineering, this strategy frequently is not satisfactory.

Combining hill-climbing as well as random sampling in a creative way and introducing concepts of learning and memory can overcome the above mentioned deficiencies. The obtained strategies dubbed "local search based learning" are known, for instance, under the names tabu search and genetic algorithms. They provide general problem solving strategies incorporating and exploiting problem-specific knowledge capable even to explore search spaces containing an exponentially growing number of local optima with respect to the problem defining parameters.

A brief outline of what follows is to introduce the reader into extensions of the hill-climbing concept which are simulated annealing, tabu search, ejection chains, and genetic algorithms. Let us mention that they are particular specifications of the above mentioned knowledge engineering and learning concept reviewed in [Hol75, Mic97, Jon90]. Tabu search develops to become the most popular and successful general problem solving strategy. Hence, attention is drawn to a couple of tabu search issues more recently developed. e.g. ejection chains. Parts of this section can also be found embedded within a problem related setting in [CKP95, PG97].

To be more specific consider the minimization problem min $\{\gamma(x) \mid x \in \mathcal{S}\}$ where γ is the objective function, i.e. the desired goal, and $\mathcal{S}$ is the search space, i.e. the set of feasible solutions of the problem. One of the most intuitive solution approaches to this optimization problem is to start with a known feasible solution and slightly perturb it while decreasing the value of the objective function. In order to realize the concept of slight perturbation let us associate with every x a subset $\mathcal{N}(x)$ of $\mathcal{S}$, called *neighborhood* of x. The solutions in $\mathcal{N}(x)$, or neighbors of x, are viewed as perturbations of x. They are considered to be "close" to x. Now the idea of a simple local search algorithm is to start with some initial solution and move from one neighbor to another neighbor as long as possible while decreasing the objective value. This local search approach can be seen as the basic principle underlying many classical optimization methods, like the gradient method for continuous non-linear optimization or the simplex method for linear programming. Some of the important issues that have to be dealt with when implementing a local search procedure are how to pick the initial solution, how to define neighborhoods and how to select a neighbor of a given solution. In many cases of interest, finding an initial solution creates no difficulty. But obviously, the choice of this starting solution may greatly influence the quality of the final outcome. Therefore, local search algorithms may be run several times on the same problem instance, using different (e.g. randomly generated) initial solutions. Whether or not the procedure will be able to significantly ameliorate a poor solution often depends on the size of the neighborhoods. The choice of neighborhoods for a given problem is conditioned by a trade-off between quality of the solution and complexity of the algorithm, and is generally to be resolved by experiments. Another crucial issue in the design of a local search algorithm is the selection of a neighbor which improves the value of the objective function. Should the first neighbor found improving upon the current solution be picked, the best one, or still some other candidate? This question is rarely to be answered through theoretical considerations. In particular, the effect of the selection criterion on the quality of the final solution, or on the number of iterations of the procedure is often hard to predict (although, in some cases, the number of neighbors can rule out an exhaustive search of the neighborhood, and hence, the selection of the best neighbor). Here again experiments with various strategies are required in order to make a decision. The attractiveness of local search procedures stems from their wide applicability and (usually) low empirical complexity (see [JPY88] and [Yan90] for more information on the theoretical complexity of local search). Indeed, local search can be used for highly intricate problems, for which analytical models would involve astronomical numbers of variables and constraints, or about which little problem-specific knowledge is available. All that is needed here is a reasonable definition of neighborhoods, and an efficient way of searching them. When these conditions are satisfied, local search can be implemented to quickly produce good solutions for large instances of the problem. These features of local search explain that the approach has been applied to a wide diversity of situations, see [PV95, GLTW93, Ree93, AL97]. In the scheduling area we would like to emphasize on two excellent surveys, [AGP95] as well as [VAL96].

Nevertheless, local search in its most simple form, the *hill-climbing*, stops as soon as it encounters a local optimum, i.e., a solution x such that $\gamma(x) \leq \gamma(y)$ for all y in $\mathcal{N}(x)$. In general, such a local optimum is not a global optimum. Even worse, there is usually no guarantee that the value of the objective function at an arbitrary local optimum comes close to the optimal value. This inherent shortcoming of local search can be palliated in some cases by the use of multiple restarts. But, because NP-hard problems often possess many local optima, even this remedy may not be potent enough to yield satisfactory solutions. In view of this difficulty, several extensions of local search have been proposed, which offer the possibility to escape local optima by accepting occasional deteriorations of the objective function. In what follows we discuss successful approaches based on related ideas, namely *simulated annealing* and *tabu search*. Another interesting extension of local search works with a population of feasible solutions (instead of a single one) and tries to detect properties which distinguish good from bad solutions. These properties are then used to construct a new population which hopefully contains a better solution than the previous one. This technique is known under the name *genetic algorithm*.

Simulated Annealing

Simulated annealing was proposed as a framework for the solution of combinatorial optimization problems by Kirkpatrick, Gelatt and Vecchi and, independently, by Cerny, cf. [KGV83, Cer85]. It is based on a procedure originally devised by Metropolis et al. in [MRR+53] to simulate the annealing (or slow cooling) of solids, after they have been heated to their melting point. In simulated annealing procedures, the sequence of solutions does not roll monotonically down towards a local optimum, as was the case with local search. Rather, the solutions trace an up-and-down random walk through the feasible set $\mathcal{S}$, and this walk is loosely guided in a "favorable" direction. To be more specific, we describe the kth iteration of a typical simulated annealing procedure, starting from a current solution x. First, a neighbor of x, say $y \in \mathcal{N}(x)$, is selected (usually, but not necessarily, at random). Then, based on the amplitude of $\Delta := \gamma(x) - \gamma(y)$, a transition from x to y (i.e., an update of x by y) is either accepted or rejected. This decision is made non-deterministically: the transition is accepted with probability $ap_k(\Delta)$, where ap_k is a probability distribution depending on the iteration count k. The intuitive justification for this rule is as follows. In order to avoid getting trapped early in a local optimum, transitions implying a deterioration of the objective function (i.e., with $\Delta < 0$) should be occasionally accepted, but the probability of acceptance should nevertheless increase with Δ. Moreover, the probability distributions are chosen so that $ap_{k+1}(\Delta) \leq ap_k(\Delta)$. In this way, escaping local optima is relatively easy during the first iterations, and the procedure explores the set $\mathcal{S}$ freely. But, as the iteration count increases, only improving transitions tend to be accepted, and the solution path is likely to terminate in a local optimum. The procedure stops if the value of the objective function remains constant in L (a termination parameter) consecutive iterations, or if the number of iterations becomes too large. In most implementations, and by analogy with the original procedure of Metropolis et al. [MRR+53], the probability distributions ap_k take the form:

$$ap_k(\Delta) = \begin{cases} 1 & \text{if } \Delta \geq 0 \\ e^{c_k \Delta} & \text{if } \Delta < 0 \, , \end{cases}$$

where $c_{k+1} \geq c_k \geq 0$ for all k, and $c_k \to \infty$ when $k \to \infty$. A popular choice for the parameter c_k is to hold it constant for a number $L(k)$ of consecutive iterations, and then to increase it by a constant factor: $c_{k+1} = \alpha^{k+1} c_0$. Here, c_0 is a small positive number, and α is slightly larger than 1. The number $L(k)$ of solutions visited for each value of c_k is based on the requirement to achieve a quasi equilibrium state. Intuitively this is reached if a fixed number of transitions is accepted. Thus, as the acceptance probability approaches 0 we would expect $L(k) \to \infty$. Therefore $L(k)$ is supposed to be bounded by some constant B to avoid long chains of trials for large values of c_k. It is clear that the choice of the termination parameter and of the distributions ap_k $(k = 1, 2, \cdots)$ (the so-called *cooling schedule*) strongly influences the performance of the procedure. If the cooling is too rapid (e.g. if B is small and α is large), then simulated annealing tends to behave like local search, and gets trapped in local optima of poor quality. If the cooling is too slow, then the running time becomes prohibitive. Starting from an initial solution x_{start} and parameters c_0 and α a generic simulated annealing algorithm can be presented as follows.

Algorithm 2.5.1 *Simulated annealing* [LA87, AK89].

```
begin
Initialize ($x_{start}$, $c_0$, $\alpha$);
$k := 0$;
$x := x_{start}$;
repeat
  Define $L(k)$ or $B$;
  for $t := 1$ to $L(k)$ do
  begin
    Generate a neighbor $y \in \mathcal{N}(x)$;
    $\Delta := \gamma(x) - \gamma(y)$;
    $ap_k(\Delta) := e^{c_k \Delta}$;
    if random[0,1] $\leq ap_k(\Delta)$ then $x := y$
  end;
  $c_{k+1} := \alpha c_k$;
  $k := k+1$;
until some stopping criterion is met
end;
```

Under some reasonable assumptions on the cooling schedule, theoretical results can be established concerning convergence to a global optimum or the complexity of the procedure (see [LA87, AK89]). In practice, determining appropriate values for the parameters is a part of the fine tuning of the implementation, and still relies on experiments. We refer to the extensive computational studies in

[JAMS89, JAMS91] for the wealth of details on this topic. If the number of iterations during the search process is large, the repeated computation of the acceptance probabilities becomes a time consuming factor. Hence, *threshold accepting* as a deterministic variant of the simulated annealing has been introduced in [DS90]. The idea is not to accept transitions with a certain probability that changes over time but to accept a new solution if the amplitude $-\Delta$ falls below a certain threshold which is lowered over time. Simulated annealing has been applied to several types of combinatorial optimization problems, with various degrees of success (see [LA87, AK89, and JAMS89, JAMS91] as well as the bibliography [CEG88]).

As a general rule, one may say that simulated annealing is a reliable procedure to use in situations where theoretical knowledge is scarce or appears difficult to apply algorithmically. Even for the solution of complex problems, simulated annealing is relatively easy to implement, and usually outperforms a hill-climbing procedure with multiple starts.

Tabu Search

Tabu search is a general framework, which was originally proposed by Glover, and subsequently expanded in a series of papers [GL97, Glo77, Glo86, Glo89, Glo90a, Glo90b, GM86, WH89]. One of the central ideas in this proposal is to guide deterministically the local search process out of local optima (in contrast with the non-deterministic approach of simulated annealing). This can be done using different criteria, which ensure that the loss incurred in the value of the objective function in such an "escaping" step (a move) is not too important, or is somehow compensated for.

A straightforward criterion for leaving local optima is to replace the improvement step in the local search procedure by a "least deteriorating" step. One version of this principle was proposed by Hansen under the name steepest descent mildest ascent (see [HJ90], as well as [Glo89]). In its simplest form, the resulting procedure replaces the current solution x by a solution $y \in \mathcal{N}(x)$ which maximizes $\Delta := \gamma(x)-\gamma(y)$. If during L (a termination parameter) iterations no improvements are found, the procedure stops. Notice that Δ may be negative, thus resulting in a deterioration of the objective function. Now, the major defect of this simple procedure is readily apparent. If Δ is negative in some transition from x to y, then there will be a tendency in the next iteration of the procedure to reverse the transition, and go back to the local optimum x (since x improves on y). Such a reversal would cause the procedure to oscillate endlessly between x and y. Therefore, throughout the search a (dynamic) list of forbidden transitions, called *tabu list* (hence the name of the procedure) is maintained. The purpose of this list is not to rule out cycling completely (this would in general result in heavy bookkeeping and loss of flexibility), but at least to make it improbable. In the framework of the steepest descent mildest ascent procedure, we may for instance implement this idea by placing solution x in a tabu list *TL* after every transition away from x. In effect, this amounts to deleting x from S. But, for reasons of flexibility, a solution would only remain in the tabu list for a limited number of iterations, and then should be freed again. To be more specific the transition to the neighbor solution, i.e. a move, may be described by one or more attributes. These attributes (when properly chosen) can become the foundation for creating a so-called attribute based memory. For example, in a $0-1$ integer programming context the attributes may be the set of all possible value assignments (or changes in such assignments) for the binary variables. Then two attributes which denote that a certain binary variable is set to 1 or 0, may be called complementary to each other. A move may be considered as the assignment of the compliment attribute to the binary variable. That is, the complement of a move cancels the effect of the considered move. If a move and its complement are performed, the same solution is reached as without having performed both moves. Moves eventually leading to a previously visited solution may be stored in the tabu list and are hence forbidden or tabu. The tabu list may be derived from the running list (*RL*), which is an ordered list of all moves (or their attributes) performed throughout the search. That is, *RL* represents the trajectory of solutions encountered. Whenever the length of *RL* is limited the attribute based memory of tabu search based on exploring *RL* is structured to provide a short term memory function. Now, each iteration consist of two parts: The guiding or tabu process and the application process. The tabu process updates the tabu list hereby requiring the actual *RL*; the application process chooses the best move that is not tabu and updates *RL*. For faster computation or storage reduction both processes are often combined. The application process is a specification on, e.g., the neighborhood definition and has to be defined by the user. The tabu navigation method is a rather simple approach requiring one parameter l called *tabu list length*. The tabu navigation method disallows choosing any complement of the l most recent moves of the running list in order to establish the next move. Hence, the tabu list consists of a (complementary) copy of the last part of *RL*. Older moves are disregarded. The tabu status derived from the l most recent moves forces the algorithm to go l moves away from any explored solution before the first step backwards is allowed. Obviously, this approach may disallow more moves than necessary to avoid returning to a yet visited solution. This encourages the intention to keep l as small as possible without disregarding the principle aim of never exploring a solution twice. Consequently, if l is too small the algorithm probably will return to a local optimum just left. If a solution is revisited the same sequence of moves may be repeated consecutively until the algorithm eventually stops, i.e. the search process is cycling. Thus danger of cycling favors large values for l. An adequate value for l has to be adopted with respect to the problem structure, the cardinality of the considered problem instances (especially problem size), the objective, etc. The parameter l is usually fixed but could also be randomly or systematically varied after a certain number of iterations. The fact that the tabu navigation method disallows moves which are not necessarily tabu led to the development of a so called aspiration level criterion which may override the tabu status of a move. The basic form of the aspiration level criterion is to choose a move in spite of its tabu status if it leads to an objective function value better than the best obtained in all preceding iterations. Another possible implementation would be to create a tabu list $TL(y)$ for every solution y within the solution space S. After a transition from x to y, x would be placed in the list $TL(y)$, meaning that further transitions from y to x are forbidden (in effect, this amounts to deleting x from $\mathcal{N}(y)$). Here again, x should be dropped from $TL(y)$ after a number of transitions. For still other possible definitions of tabu lists, see e.g. [Glo86, Glo89, GG89, HJ90, HW90]. Tabu search encompasses many features beyond the possibility to avoid the trap of local optimality and the use of tabu lists. Even though we cannot discuss them all in the limited framework of this survey, we would like to mention two of them, which provide interesting links with artificial intelligence and with genetic algorithms. In order to guide the search, Glover suggests recording some of the salient characteristics of the best solutions found in some phase of the procedure (e.g., fixed values of the variables in all, or in a majority of those solutions, recurring relations between the values of the variables, etc.). In a subsequent phase, tabu search can then be restricted to the subset of feasible solutions presenting these characteristics. This enforces what Glover calls a "regional intensification" of the search in promising "regions" of the feasible set. An opposite idea may also be used to "diversify" the search. Namely, if all solutions discovered in an initial phase of the search procedure share some common features, this may indicate that other regions of the solution space have not been sufficiently explored. Identifying these unexplored regions may be helpful in providing new starting solutions for the search. Both ideas, of search intensification or diversification, require the capability of recognizing recurrent patterns within subsets of solutions. In many applications the aforementioned simple tabu search strategies are already very successful, cf. [GLTW93, PV95, OK96]. A brief outline of the tabu search algorithm can be presented as follows.

Algorithm 2.5.2 *Tabu search* [Glo89, Glo90a, Glo90b].

```
begin
Initialize ($x$, tabu list $TL$, running list $RL$, aspiration function $A(\Delta, k)$);
$x_{\text{best}} := x$;
$k := 1$;
Specify the tabu list length $l_k$ at iteration $k$;
$RL := \varnothing$;
$TL := \varnothing$;
$\alpha := \infty$;
repeat
  repeat
    Generate neighbor $y \in \mathcal{N}(x)$;
    $\Delta := \gamma(x) - \gamma(y)$;
```

```
        Calculate the aspiration value $A(\Delta, k)$;
    until $A(\Delta, k) < \alpha$ or $\Delta = \max\{\gamma(x) - \gamma(y) \mid y$ is not tabu$\}$;
    Update $RL$, i.e. $RL := RL \cup \{$some attributes of $y\}$;
    $TL := \{$the last $l_k$ non-complimentary entries of $RL\}$;
    if $A(\Delta, k) < \alpha$ then $\alpha := A(\Delta, k)$;
    $x := y$;
    if $\gamma(y) < \gamma(x_{\text{best}})$ then $x_{\text{best}} := y$;
    $k := k+1$;
until some stopping criterion is met
end;
```

As mentioned above, tabu search may be applied in a more advanced way to incorporate different means for solid theoretical foundations. Other concepts have been developed like the reverse elimination method or the reactive tabu search incorporating a memory employing simple reactive mechanisms that are activated when repetitions of solutions are discovered throughout the search, see e.g. [GL97].

Ejection Chains

Variable depth methods, whose terminology was popularized by Papadimitriou and Steiglitz [PS82], have had an important role in heuristic procedures for optimization problems. The origins of such methods go back to prototypes in network and graph theory methods of the 1950s and 1960s. A class of these procedures called *ejection chain* methods has proved highly effective in a variety of applications, see [LK73] which is a special instance of an ejection chain on the TSP, and [Glo91, Glo96, DP94, Pes94, PG97, Reg98].

Ejection chain methods extend ideas exemplified by certain types of shortest path and alternating path constructions. The basic moves for a transition from one solution to another are compound moves composed of a sequence of paired steps. The first component of each paired step in an ejection chain approach introduces a change that creates a dislocation (i.e., an inducement for further change), while the second component creates a change designed to restore the system. The dislocation of the first component may involve a form of unfeasibility, or may be heuristically defined to create conditions that can be usefully exploited by the second component. Typically, the restoration of the second component may not be complete, and hence in general it is necessary to link the paired steps into a chain that ultimately achieves a desired outcome. The ejection terminology comes from the typical graph theory setting where each of the paired steps begins by introducing an element (such as a node, edge or path) that disrupts the graph's preferred structure, and then is followed by ejecting a corresponding element, in a way that recovers a critical portion of the structure. A chain of such steps is controlled to assure the preferred structure eventually will be fully recovered (and preferably, fully recovered at various intermediate stages by means of trial solutions). The candidate element to be ejected in such instances may not be unique, but normally comes from a limited set of alternatives. The alternating path construction [Ber62] gives a simple illustration. Here, the preferred graph structure requires a degree constraint to be satisfied at each node (bounding the number of edges allowed to enter the node). The first component of a paired step introduces an edge that violates such a degree constraint, causing too many edges to enter a particular node, and thus is followed by a second component that ejects one of the current edges at the node so that the indicated constraint may again be satisfied. The restoration may be incomplete, since the ejected edge may leave another node with too few edges, and thus the chain is induced to continue. A construction called a reference structure becomes highly useful for controlling such a process, in order to restore imbalances at each step by means of special trial solution moves, see [Glo91, Glo96, PG97]. Loosely speaking, a reference structure is a representation of a (sometimes several) feasible solution such that, however, a very small number of constraints may be violated. Finding a feasible solution from a reference structure must be a trivial task which should be performable in constant time. Ejection chain processes of course are not limited to graph constructions. For example, they can be based on successively triggered changes in values of variables, as illustrated by a linked sequence of zero-one exchanges in multiple choice integer programming applications or by linked "bound escalations" in more general integer programs. The approach can readily be embedded in a complete tabu search implementation, or in a genetic algorithm or simulated annealing implementation. Such a method can also be used as a stand-alone heuristic, which terminates when it is unable to find an improved solution at the conclusion of any of its constructive passes. (This follows the customary format of a variable depth procedure.) As our construction proceeds, we therefore note the trial solutions (e.g. feasible tours in case of a TSP) that would result by applying these feasibility-recovering transformations after each step, keeping track of the best. At the conclusion of the construction we simply select this best trial solution to replace the current solution, provided it yields an improvement. In this process, the moves at each level cannot be obtained by a collection of independent and non-intersecting moves of previous levels. The list of forbidden (tabu) moves grows dynamically during variable depth search iteration and is reset at the beginning of the next iteration. In the subsequent algorithmic description we designate the lists of variables (in the basis of a corresponding LP solution) locked in and out of the solution by the names tabu-to-drop and tabu-to-add, where the former contains variables added by the current construction (hence which must be prevented from being dropped) and the latter contains variables dropped by the current construction (hence which must be prevented from being added). The resulting ejection chain procedure is shown in Algorithm 2.5.3. We denote the cost of a solution x by $\gamma(x)$. The cost difference of a solution x' and x, i.e. $\gamma(x) - \gamma(x')$, where x' results from x by replacing variable i by variable j will be defined by γ_{ij}. The reference structure that results by performing d ejection steps, is denoted by $x(d)$, where d is the "depth" of the ejection chain (hence $x = x(0)$ for a given starting solution x).

Algorithm 2.5.3 *Ejection chain* [PG97].

```
begin
Start with an initial solution $x_{\text{start}}$;
$x := x_{\text{start}}$;  $x^* := x_{\text{start}}$;
Let $s$ be any variable in $x$;      -- $s$ is the root
$k^* := s$;
repeat
    $d := 0$;       -- $d$ is the current search depth
    while there are variables in $x(d)$ that are not tabu-to-drop
          and variables outside of $x(d)$ that are not tabu-to-add  do
        begin
        $i := k^*$;
        $d := d+1$;
        Find the best component move that maintains the reference structure,
            where this 'best' is given by the variable pair $i, j$ for which the gain
            $\gamma_{i^*j^*} = \max\{\gamma_{ij} \mid j$ is not a variable in $x(d-1)$ and $i$ is a variable
                        in $x(d-1)$; $j$ is not tabu-to-add; $i$ is not tabu-to-drop$\}$;
        Perform this move, i.e. introduce variable $j^*$ and remove variable $i^*$
            thus obtaining $x(d)$ as a new reference structure at search depth $d$;
        $j^*$ becomes tabu-to-drop and $i^*$ becomes tabu-to-add;
        end;
    Let $d^*$ denote the search depth at which the best solution $x^*(d^*)$with
        $\gamma(x^*(d^*)) = \min\{\gamma(x^*(d)) \mid 0 < d \leq n\}$ has been found;
    if $d^* > 0$ then $x^* := x^*(d^*)$; $x := x^*$;
until $d^* = 0$;
end;
```

The above procedure describes in its inner **repeat ... until** loop one iteration of an ejection chain search. The **while ... do** describes one component move. Starting with an initially best solution $x^*(0)$, the procedure executes a construction that maintains the reference structure for a certain number of component moves. The new currently best trial solution $x^*(d^*)$, encountered at depth d^*, becomes the starting point for the next ejection chain iteration. The iterations are repeated as long as an improvement is possible. The maximum depth of the construction is reached if all variables in the current solution x are set tabu-to-drop. The step leading from a solution x to a new solution consists of a varying number d^* of component moves, hence motivating the "variable depth" terminology. A continuously growing tabu list avoids cycling of the search procedure. As an extension of the algorithm (not shown here), the whole **repeat**

`... until` part could easily be embedded in yet another control loop leading to a multi-level (parallel) search algorithm, see [Glo96].

Genetic Algorithms

As the name suggests, *genetic algorithms* are motivated by the theory of evolution; they date back to the early work described in [Rec73, Hol75, Sch77], see also [Gol89] and [Mic97]. They have been designed as general search strategies and optimization methods working on populations of feasible solutions. Working with populations permits to identify and explore properties which good solutions have in common (this is similar to the regional intensification idea mentioned in our discussion of tabu search). Solutions are encoded as strings consisting of elements chosen from a finite alphabet. Roughly speaking, a genetic algorithm aims at producing near-optimal solutions by letting a set of strings, representing random solutions, undergo a sequence of unary and binary transformations governed by a selection scheme biased towards high-quality solutions. Therefore, the quality or *fitness* value of an individual in the population, i.e. a string, has to be defined. Usually it is the value of the objective function or some scaled version of it. The transformations on the individuals of a population constitute the *recombination* steps of a genetic algorithm and are performed by three simple operators. The effect of the operators is that implicitly good properties are identified and combined into a new population which hopefully has the property that the value of the best individual (representing the best solution in the population) and the average value of the individuals are better than in previous populations. The process is then repeated until some stopping criteria are met. It can be shown that the process converges to an optimal solution with probability one (cf. [EAH91]). The three basic operators of a classical genetic algorithm when a new population is constructed are *reproduction, crossover* and *mutation*.

Via reproduction a new temporary population is generated where each member is a replica of a member of the old population. A copy of an individual is produced with probability proportional to its fitness value, i.e. better strings probably get more copies. The intended effect of this operation is to improve the quality of the population as a whole. However, no genuinely new solutions and hence no new information are created in the process. The generation of such new strings is handled by the crossover operator.

In order to apply the crossover operator the population is randomly partitioned into pairs. Next, for each pair, the crossover operator is applied with a certain probability by randomly choosing a position in the string and exchanging the tails (defined as the substring starting at the chosen position) of the two strings (this is the simplest version of a crossover). The effect of the crossover is that certain properties of the individuals are combined to new ones or other properties are destroyed. The construction of a crossover operator should also take into consideration that fitness values of offspring are not too far from those of their parents, and that offspring should be genetically closely related to their parents.

The mutation operator which makes random changes to single elements of the string only plays a secondary role in genetic algorithms. Mutation serves to maintain diversity in the population (see the previous section on tabu search).

Besides unary and binary recombination operators, one may also introduce operators of higher arities such as consensus operators, that fix variable values common to most solutions represented in the current population. Selection of individuals during the reproduction step can be realized in a number of ways: one could adopt the scenario of [Gol89] or use deterministic ranking. Further it matters whether the newly recombined offspring compete with the parent solutions or simply replace them.

The traditional genetic algorithm, based on a binary string representation of solutions, is often unsuitable for combinatorial optimization problems because it is very difficult to represent a solution in such a way that sub-strings have a meaningful interpretation. Nevertheless, the number of publications on genetic algorithm applications to sequencing and scheduling problems exploded.

Problems from combinatorial optimization are well within the scope of genetic algorithms and early attempts closely followed the scheme of what Goldberg [Gol89] calls a *simple genetic algorithm*. Compared to standard heuristics, genetic algorithms are not well suited for fine-tuning structures which are very close to optimal solutions. Therefore, it is essential, if a competitive genetic algorithm is desired, to compensate for this drawback by incorporating (local search) improvement operators into the basic scheme. The resulting algorithm has then been called *genetic local search heuristic* or *genetic enumeration* (cf. [Joh90b, UAB+91, Pes94, DP95]). Each individual of the population is then replaced by a locally improved one or an individual representing a locally optimal solution, i.e. an improvement procedure is applied to each individual either partially (to a certain number of iterations, [KP94]) or completely. Some type of improvement heuristic may also be incorporated into the crossover operator, cf. [KP94].

Putting things into a more general framework, a solution of a combinatorial optimization problem may be considered as resolution of a sequence of local decisions (such as priority rules or even more complicated ones). In an enumeration tree of all possible decision sequences the solutions of the problem are represented as a path corresponding to the different decisions from the root of the tree to a leaf (hence the name genetic enumeration). While a branch and bound algorithm learns to find those decisions leading to an optimal solution (with respect to the space of all decision sequences) genetics can guide the search process in order to learn to find the most promising decision combinations within a reasonable amount of time, see [Pes94, DP95]. Hence, instead of (implicitly) enumerating all decision sequences a rudimentary search tree will be established. Only a polynomial number of branches can be considered where population genetics drives the search process into those regions which more likely contain optimal solutions. The scheme of a genetic enumeration algorithm is subsequently described; it requires further refinement in order to design a successful genetic algorithm.

Algorithm 2.5.4 *Genetic enumeration* [DP95, Pes94].

```
begin
Initialization: Construct an initial population of individuals each of which is a
    string of local decision rules;
Assessment / Improvement: Assess each individual in the current population
    introducing problem specific knowledge by special purpose heuristics (such as
    local search) which are guided by the sequence of local decisions;
if special purpose heuristics lead to a new string of local decision rules
then
    replace each individual by the new one, for instance, a locally optimal one;
repeat
    Recombination: Extend the current population by adding individuals obtained
        by unary and binary transformations (crossover, mutation) on one or two
        individuals in the current population;
    Assessment / Improvement: Assess each individual in the current population
        introducing problem specific knowledge by special purpose heuristics
        (such as local search) which are guided by the sequence of local decisions;
    if special purpose heuristics lead to a new string of local decision rules
    then
        replace each individual by the new one, for instance, a locally optimal one;
until some stopping criterion is met
end;
```

It is an easy exercise to recognize that the simple genetic algorithm as well as genetic local search fits into the provided framework.

For a successful genetic algorithm in combinatorial optimization a genetic meta-strategy is indispensable in order to guide the operation of good special purpose heuristics and to incorporate problem-specific knowledge. An older concept of a population based search technique which dates back in its origins beyond the early days of genetic algorithms is introduced in [Glo95] and called *scatter search*. The idea is to solve 0−1 programming problems departing from a solution of a linear programming relaxation. A set of reference points is created by perturbing the values of the variables in this solution. Then new points are defined as selected convex combinations of reference points that constitute good solutions obtained from previous solution efforts. Non-integer values of these points are rounded and then heuristically converted into candidate solutions for the integer programming problem. The idea parallels and extends the idea basic to the genetic algorithm design, namely, combining parent solutions in some way in order to obtain new offspring solutions. One of the issues that differentiates scatter search from the early genetic algorithm paradigm is the fact that the former creates new points strategically rather than randomly. Scatter search does not prespecify the number of points it will generate to retain. This can be adaptively established by considering the solution quality during the generation process. The "data perturbation idea" meanwhile has gained considerable attention within the

GA community. In [LB95] it is transferred as a tool for solving resource constrained project scheduling problems with different objective functions. The basic idea of their approach may be referred to as a "data perturbation" methodology which makes use of so-called problem space based neighborhoods. Given a well-known concept for deriving feasible solutions (e.g. a priority rule), a search approach is employed on account of the problem data and respective perturbations. By modifying (i.e. introducing some noise or perturbation) the problem data used for the priority values of activities, further solutions within a certain neighborhood of the original data are generated.

The ideas mentioned above are paving the way in order to do some steps into the direction of machine learning. This is in particular true if learning is considered to be a right combination of employing inference on memory. Thus, local search in terms of tabu search and genetic algorithms emphasize such a unified approach in all successful applications. This probably resembles most the human way of thinking and learning.

References

Agi66	N. Agin, Optimum seeking with branch and bound, *Manage. Sci.* 13, 1966, B176-185.
AGP95	E. J. Anderson, C. A. Glass, C. N. Potts, Local search in combinatorial optimization: applications in machine scheduling, Working paper, University of Southampton, 1995.
AHU74	A. V. Aho, J. E. Hopcroft, J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974.
AK89	E. H. L. Aarts, J. Korst, *Simulated Annealing and Boltzmann Machines,* J. Wiley, Chichester, 1989.
AL97	E. H. L. Aarts, J. K. Lenstra (eds.), *Local Search in Combinatorial Optimization*, J. Wiley, New York, 1997.
AMO93	R. K. Ahuja, T. L. Magnanti, J. B. Orlin, *Network Flows*, Prentice-Hall, Englewood Cliffs, N.J., 1993.
Bak74	K. Baker, *Introduction to Sequencing and Scheduling*, J. Wiley, New York, 1974.
BD62	R. Bellman, S. E. Dreyfus, *Applied Dynamic Programming*, Princeton University Press, Princeton, N.J., 1962.
Bel57	R. Bellman, *Dynamic Programming*, Princeton University Press, Princeton, N.J., 1957.
Ber62	C. Berge, *Theory of Graphs and its Applications*, Methuen, London, 1962.
Ber73	C. Berge, *Graphs and Hypergraphs*, North-Holland, Amsterdam, 1973.
CEG88	N. E. Collins, R. W. Eglese, B. L. Golden, Simulated annealing - an annotated bibliography, *American Journal of Mathematical and Management Sciences* 8, 1988, 209-307.

Cer85 V. Cerny, Thermodynamical approach to the traveling salesman problem: an efficient simulation algorithm, *J. Optim. Theory Appl.* 45, 1985, 41-51.

Che77 B. V. Cherkassky, An algorithm for building a max-flow in a network, running in time $O(V^2E^{1/2})$, *Mathematical Methods for Solving of Economic Problems* 7, 1977, 117-126 (in Russian).

Che80 T.-Y. Cheung, Computational comparison of eight methods for the maximum network flow problem, *ACM Trans. Math. Softw.* 6, 1980, 1-16.

CHW87 M. Chams, A. Hertz, D. de Werra, Some experiments with simulated annealing for colouring graphs, *Eur. J. Oper. Res.* 32, 1987, 260-266.

CKP95 Y. Crama, A. Kolen, E. Pesch, Local search in combinatorial optimization, *Lect. Notes Comput. Sc.* 931, 1995, 157-174.

Cof76 E. G. Coffman, Jr. (ed.), *Scheduling in Computer and Job Shop Systems*, J. Wiley, New York, 1976.

Coo71 S. A. Cook, The complexity of theorem proving procedures, *Proceedings of the 3rd ACM Symposium on Theory of Computing*, 1971, 151-158.

Den82 E. V. Denardo, *Dynamic Programming: Models and Applications*, Prentice-Hall, Englewood Cliffs, N.J., 1982.

Din70 E. A. Dinic, An algorithm for the solution of the problem of maximal flow with polynomial estimation, *Dokl. Akad. Nauk SSSR* 194, 1970, 754-757 (in Russian).

DL79 S. E. Dreyfus, A. M. Law, *The Art and Theory of Dynamic Programming*, Academic Press, New York, 1979.

DP94 U. Dorndorf, E. Pesch, Fast clustering algorithms, *ORSA Journal on Computing* 6, 1994, 141-153.

DP95 U. Dorndorf, E. Pesch, Evolution based learning in a job shop scheduling environment, *Comput. Oper. Res.* 22,1995, 25-40.

DS90 G. Dueck, T. Scheuer, Threshold accepting: a general purpose optimization algorithm appearing superior to simulated annealing, *J. Comput. Phys.* 90, 1990, 161-175.

EAH91 A. E. Eiben, E. H. L. Aarts, K. H. van Hee, Global convergence of genetic algorithms: a Markov chain analysis, *Lect. Notes Comput. Sc.* 496, 1991, 4-9.

Edm65 J. Edmonds, Paths, trees and flowers, *Can. J. Math.-J. Can. Math.* 17, 1965, 449-467.

EK72 J. Edmonds, R. M. Karp, Theoretical improvement in algorithmic efficiency for network flow problem, *J. ACM* 19, 1972, 248-264.

Eve79 S. Even, *Graph Algorithms*, Computer Science Press, New York, 1979.

FF62 L. R. Ford, Jr., D. R. Fulkerson, *Flows in Networks*, Princeton University Press, Princeton, N.J., 1962.

Fis70 P. C. Fishburn, Intransitive indifference in preference theory: a survey, *Oper. Res.* 18, 1970, 207-228

GG89 F. Glover, H. J. Greenberg, New approaches for heuristic search: a bilateral linkage with artificial intelligence, *Eur. J. Oper. Res.* 13, 1989, 563-573.

GJ78 M. R. Garey, D. S. Johnson, Strong NP-completeness results: motivation, examples, and implications, *J. ACM* 25, 1978, 499-508.

GJ79 M. R. Garey, D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, San Francisco, 1979.

GL97 F. Glover, M. Laguna, *Tabu Search*, Kluwer Academic Publishers, Boston, 1997.

Glo77 F. Glover, Heuristic for integer programming using surrogate constraints, *Decis. Sci.* 8, 1977, 156-160.

Glo86 F. Glover, Future paths for integer programming and links to artificial intelligence, *Comput. Oper. Res.* 13,1986, 533-549.

Glo89 F. Glover, Tabu search - Part I, *ORSA Journal on Computing* 1, 1989, 190-206.

Glo90a F. Glover, Tabu search - Part II, *ORSA Journal on Computing* 2, 1990, 4-32.

Glo90b F. Glover, Tabu search: a tutorial, *Interfaces* 20, 1990, 74-94.

Glo91 F. Glover, Multilevel tabu search and embedded search neighborhoods for the traveling salesman problem, Working paper, University of Colorado, Boulder, 1991.

Glo96 F. Glover, Ejection chains, reference structures and alternating path methods for traveling salesman problems, *Discret Appl. Math*. 65, 1996, 223-253.

Glo95 F. Glover, Scatter search and star-paths: beyond the genetic metaphor, *OR Spektrum* 17, 1995, 125-137.

GLTW93 F. Glover, M. Laguna, E. Taillard, D. de Werra (eds.), Tabu Search, *Ann. Oper. Res.* 41, Baltzer, Basel, 1993.

GM86 F. Glover, C. McMillan, The general employee scheduling problem: an integration of MS and AI, *Comput. Oper. Res.* 13, 1986, 563-573.

Gol89 D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley, Reading, Mass., 1989.

HJ90 P. Hansen, B. Jaumard, Algorithms for the maximum satisfiability problem, *Computing* 44, 1990, 279-303.

Hol75 J. H. Holland, *Adaptation in Natural and Artificial Systems*, The University of Michigan Press, Ann Arbor, 1975.

How69 R. A. Howard, *Dynamic Programming and Markov Processes*, MIT Press, Cambridge, Mass., 1969.

HW90 A. Hertz, D. de Werra, The tabu search metaheuristic: how we use it, *Ann. Math. Artif. Intell.* 1, 1990, 111-121.

JAMS89 D. S. Johnson, C. R. Aragon, L. A. McGeoch, C. Schevon, Optimization by simulated annealing: an experimental evaluation; Part I, Graph partitioning, *Oper. Res.* 37, 1989, 865-892.

JAMS91 D. S. Johnson, C. R. Aragon, L. A. McGeoch, C. Schevon, Optimization by simulated annealing: an experimental evaluation; Part II, Graph coloring and number partitioning, *Oper. Res.* 39, 1991, 378-406.

Joh90a D. S. Johnson, A catalog of complexity classes, in: J. van Leeuwen (ed.), *Handbook of Theoretical Computer Science*, Elsevier, New York, 1990, 69-161.

Joh90b D. S. Johnson, Local optimization and the traveling salesman problem, *Lect. Notes Comput. Sc.* 443, 1990, 446-461.

Jon90 K. de Jong, Genetic-algorithm-based learning, in: Y. Kodratoff, R. Michalski (eds.) *Machine Learning*, Vol. III, Morgan Kaufmann, San Mateo, 1990, 611-638.

JPY88 D. S. Johnson, C. H. Papadimitriou, M. Yannakakis, How easy is local search? *J. Comput. Syst. Sci.* 37, 1988,79-100.

Kar72 R. M. Karp, Reducibility among combinatorial problems, in: R. E. Miller, J. W. Thatcher (eds.), *Complexity of Computer Computation*, Plenum Press, New York, 1972, 85-103.

Kar74 A. W. Karzanov, Determining the maximum flow in a network by the method of preflows, *Dokl. Akad. Nauk SSSR* 215, 1974, 434-437 (in Russian).

KGV83 S. Kirkpatrick, C. D. Gelatt Jr., M. P. Vecchi, Optimization by simulated annealing, *Science* 220, 1983, 671-680.

KP94 A. Kolen, E. Pesch, Genetic local search in combinatorial optimization, *Discret Appl. Math.* 48, 1994, 273-284.

Kub87 M. Kubale, The complexity of scheduling independent two-processor tasks on dedicated processors, *Inf. Process. Lett.* 24, 1987, 141-147.

LA87 P. J. M. van Laarhoven, E. H. L. Aarts, *Simulated Annealing: Theory and Applications*, Reidel, Dordrecht, 1987.

Law76 E. L. Lawler, *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart and Winston, New York, 1976.

LB95 V. J. Leon, R. Balakrishnan, Strength and adaptability of problem-space based neighborhoods for resource constrained scheduling, *OR Spektrum* 17, 1995, 173-182.

Len77 J. K. Lenstra, *Sequencing by Enumerative Methods*, Mathematical Centre Tracts 69, Amsterdam, 1977.

LK73 S. Lin, B. W. Kernighan, An effective heuristic algorithm for the traveling salesman problem, *Oper. Res.* 21, 1973, 498-516.

LRKB77 J. K. Lenstra, A. H. G. Rinnooy Kan, P. Brucker, Complexity of machine scheduling problems, *Annals of Discrete Mathematics* 1, 1977, 343-362.

LW66 E. L. Lawler, D. E. Wood, Branch and bound methods: a survey, *Oper. Res.* 14, 1966, 699-719.

Mic97 Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs*, Springer, Berlin, 1997.

Mit70 L. G. Mitten, Branch-and-bound methods: general formulation and properties, *Oper. Res.* 18, 1970, 24-34.

MRR+53 M. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, E. Teller, Equation of state calculations by fast computing machines, *J. Chem. Phys.* 21, 1953, 1087-1092.

OK96 I. H. Osman, J. P. Kelly, *Meta-Heuristics: Theory and Applications*, Kluwer, Dordrecht, 1996.

Pap94 C. H. Papadimitriou, *Computational Complexity*, Addison-Wesley, Reading, Mass., 1994.

Pes94 E. Pesch, *Learning in Automated Manufacturing,* Physica, Heidelberg, 1994.

PG97 E. Pesch, F. Glover, TSP ejection chains, *Discret Appl. Math.* 76, 1997, 165-181.

PS82 C. H. Papadimitriou, K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, N.J., 1982.

PV95 E. Pesch, S. Voß (eds.), Applied Local Search, *OR Spektrum* 17, 1995.

Rec73 I. Rechenberg, *Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*, Problemata, Frommann-Holzboog, 1973.

Ree93 C. Reeves (ed.), *Modern Heuristic Techniques for Combinatorial Problems*, Blackwell Scientific Publishing, 1993.

Reg98 C. Rego, A subpath ejection method for the vehicle routing problem, *Manage. Sci.* 44, 1998, 1447-1459.

Rin76a A. H. G. Rinnooy Kan, *Machine Scheduling Problems: Classification, Complexity and Computations*, Martinus Nijhoff, The Hague, 1976.

Rin76b A. H. G. Rinnooy Kan, On Mitten's axiom for branch and bound, *Oper. Res.* 24, 1976, 1176-1178.

Rin87 A. H. G. Rinnooy Kan, Probabilistic analysis of approximation algorithms, *Annals of Discrete Mathematics* 31, 1987, 365-384.

Sch77 H.-P. Schwefel, *Numerische Optimierung von Computer-Modellen mittels der Evolutionsstrategie*, Birkhäuser, Basel, 1977.

SVW80 E. A. Silver, R. V. Vidal, D. de Werra, A tutorial on heuristic methods, *Eur. J. Oper. Res.* 5, 1980, 153-162.

SW07 P. Schuurman, G. J. Woeginger, Approximation schemes - a tutorial, in: R. H. Moehring, C. N. Potts, A. S. Schulz, G. J. Woeginger, L. A. Wolsey (eds.), *Lectures on Scheduling*, 2007, to appear, http://www.win.tue.nl/~gwoegi/papers/ptas.pdf.

UAB+91 N. L. J. Ulder, E. H. L. Aarts, H.-J. Bandelt, P. J. M. van Laarhoven, E. Pesch, Genetic local search algorithms for the traveling salesman problem, *Lect. Notes Comput. Sc.* 496, 1991, 109-116.

VAL96 R. J. M. Vaessens, E. H. L. Aarts, J. K. Lenstra, Job shop scheduling by local search, *ORSA Journal on Computing* 13, 1996, 302-317.

VTL82 J. Valdes, R. E. Tarjan, E. L. Lawler, The recognition of series parallel digraphs, *SIAM J. Comput.* 11, 1982, 298-313.

WH89 D. de Werra, A. Hertz, Tabu search techniques: a tutorial and an application to neural networks, *OR Spektrum* 11, 1989, 131-141.

Yan90 M. Yannakakis, The analysis of local search problems and their heuristics, *Lect. Notes Comput. Sc.* 415, 1990, 298-311.