# 2

# Combinatorial Optimization and Computational Complexity

Combinatorial optimization problems arise in several applications. Examples are the task of finding the shortest path from Paris to Rome in the road network of Europe or scheduling exams for given courses at a university. In this chapter, we give a basic introduction to the field of combinatorial optimization. Later on, we discuss how to measure the computational complexity of algorithms applied to these problems and point out some general limitations for solving difficult problems.

## 2.1 Combinatorial Optimization

Optimization problems can be divided naturally into two categories. The first category consists of problems with continuous variables. Such problems are well known from school courses on mathematics. A simple example consists of finding the minimum of the function $f : \mathbb{R} \to \mathbb{R}$ with $f(x) = x^2$. It is obvious that $x_0 = 0$ is the unique solution for this problem. More complicated problems are often tackled by computing the derivatives, using Newton methods or linear programming techniques. As this book deals with combinatorial optimization problems, we will not go into detail the different methods to tackle continuous optimization problems, and refer the interested reader to Nocedal and Wright (2000).

In the case of discrete variables we are dealing with discrete optimization. When speaking of combinatorial optimization problems, most people have "natural" discrete optimization problems in mind, such as computing shortest paths or scheduling different jobs on a set of available machines. In a combinatorial optimization problem, one aims at either minimizing or maximizing a given objective function under a given set of constraints.

A problem consists of a general question that has to be answered and is given by a set of input parameters. An instance of a problem is given by the problem together with a specified parameter setting. Formally, a combinatorial optimization problem can be defined as a triple $(S, f, \Omega)$, where $S$ is a given

F. Neumann, C. Witt, *Bioinspired Computation in Combinatorial Optimization*, Natural Computing Series,
DOI 10.1007/978-3-642-16544-3_2, 

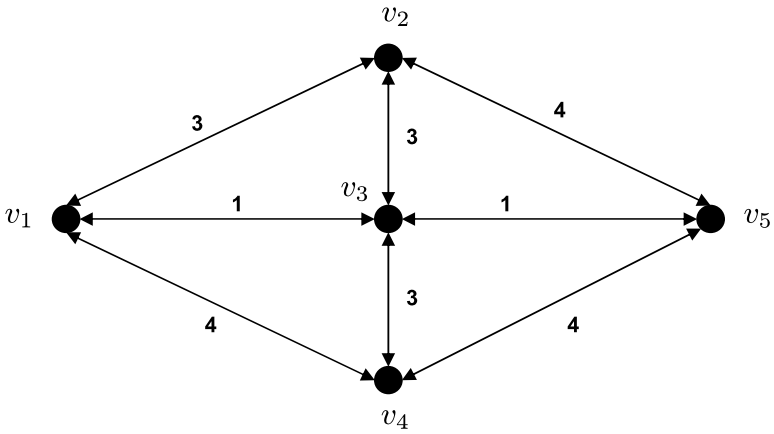


**Fig. 2.1.** Example graph $G$

search space, $f$ is the objective function, which should be either maximized or minimized, and $\Omega$ is the set of constraints that have to be fulfilled to obtain feasible solutions. The goal is to find a globally optimal solution, which is in the case of a maximization problem a solution $s^*$ with the highest objective value that fulfills all constraints. Similarly, in the case of minimization problems, one tries to achieve a smallest objective value under the condition that all constraints are fulfilled.

Throughout this book, we consider many combinatorial optimization problems on graphs. A directed graph $G$ is a pair $G = (V, E)$, where $V$ is a finite set and $E$ is a binary relation on $V$. The elements of $V$ are called vertices. $E$ is called the edge set of $G$ and its elements are called edges. For an illustration see Figure 2.1.

We use the notation $e = (u, v)$ for an edge in a directed graph. Note that self-loops that are edges of the kind $(u, u)$ are possible. In an undirected graph $G = (V, E)$, no self-loops are possible. The edge set $E$ consists of unordered pairs of vertices in this case, and an edge is a set $\{u, v\}$ consisting of two distinct vertices $u, v \in V$. Note that one can think of an undirected edge $\{u, v\}$ as two directed edges $(u, v)$ and $(v, u)$. If $(u, v)$ is an edge in a directed graph $G = (V, E)$ we say that $v$ is adjacent to vertex $u$. This leads to the representation of graphs by adjacency matrices, which will be discussed later in greater detail. A path of length $k$ from a vertex $v_0$ to a vertex $v_k$ in a graph $G = (V, E)$ is a sequence $v_0, v_1, \ldots, v_k$ of vertices such that $(v_{i-1}, v_i) \in E$, $1 \leq i \leq n$, holds. Note that a path implies a sequence of directed edges. Therefore, it is sometimes useful to denote a path $(v_0, v_1, \ldots, v_k)$ by its sequence of directed edges $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$.

The graph $G$ in Figure 2.1 consists of the vertex set

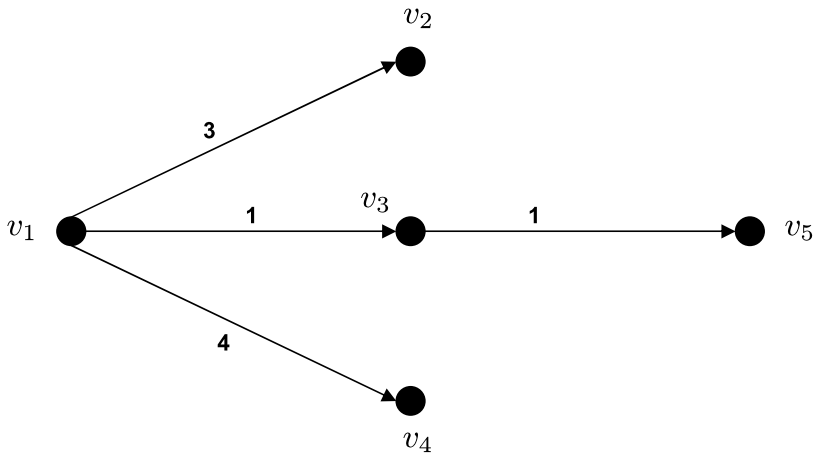


**Fig. 2.2.** Single source shortest path tree for $G$ and $s = v_1$

$$V = \{v_1, v_2, v_2, v_3, v_4, v_5\}$$

and the edge set

$$E = \{e_1, e_2, e_3, e_4, e_5, e_6, e_6, e_7, e_8\}$$

where $e_1 = \{v_1, v_2\}, e_2 = \{v_1, v_3\}, e_3 = \{v_1, v_4\}, e_4 = \{v_2, v_3\}, e_5 = \{v_2, v_5\}$, $e_6 = \{v_3, v_4\}, e_7 = \{v_3, v_5\}$, and $e_8 = \{v_4, v_5\}$. In addition, there is a weight function $w\colon E \to \mathbb{N}$ assigning weights to the edges, i.e., $w(e_1) = w(e_4) = w(e_6) = 3$, $w(e_2) = w(e_7) = 1$, and $w(e_3) = w(e_5) = w(e_8) = 4$. Clearly, $(v_1, v_2, v_3, v_5)$ is a path in $G$ whereas $(v_1, v_5, v_2)$ is not as there is no edge from $v_1$ to $v_5$.

There are many well-known combinatorial optimization problems on weighted graphs. We want to introduce two basic problems in the following. In the case of the single source shortest path problem, an undirected connected graph $G = (V, E)$ with positive weights on the edges is given. The goal is to compute from a designated vertex $s \in V$ the shortest paths to all other vertices of $V \setminus \{s\}$. The solution of this problem can be given by a tree rooted at $s$ which contains the shortest paths. Considering the graph $G$ of Figure 2.1 and $s = v_1$, a shortest path tree is shown in Figure 2.2. Another well-known combinatorial optimization problem on undirected connected graphs with positive weights is the minimum spanning tree problem. Here, one searches for a connected subgraph of the given graph $G$ that has minimal cost. As the edge weights are positive, such a graph does not contain cycles, i.e., it is a tree. Considering again the graph $G$ of Figure 2.1, a minimum spanning tree of $G$ is given in Figure 2.3.

Other important problems on graphs are covering problems. In the case of the so-called vertex cover problem for a given undirected graph $G = (V, E)$,

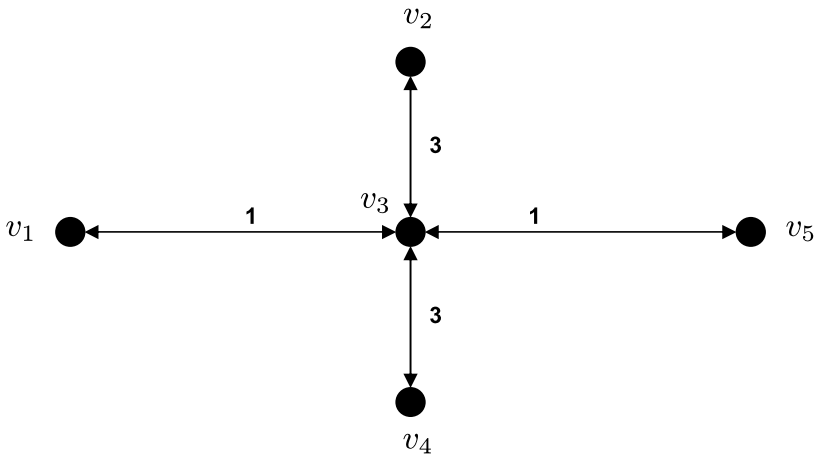


**Fig. 2.3.** Minimum spanning tree of $G$

one searches for a minimal subset of vertices $V' \subseteq V$ such that each edge $e \in E$ contains at least one vertex of $V$, i.e., $\forall e \in E \colon e \cap V' \neq \emptyset$ holds.

Another class of combinatorial optimization problems that has been widely examined in the literature is scheduling problems. Here, $n$ jobs are given that have to be processed on $m \geq 1$ machines. Associated with each job $j$, $1 \leq j \leq n$, is usually a processing time $p_j$. The processing time need not be the same for each machine. There are variants of scheduling problems where the processing time may depend on the machine by which it is processed. Often, also a specific due date for each job is given. Consider the following simple scheduling problem on two machines. Given are $n$ jobs and for each job $j$ a processing time $p_j$ which holds independently of the chosen machine. The goal is to find an assignment of the jobs to the two machines such that the overall completion time is minimized. Let $x \in \{0,1\}^n$ be a decision vector. Job $j$ is on machine 1 iff $x_j = 0$ holds and on machine 2 iff $x_j = 1$ holds. The goal is to minimize

$$\max \left\{ \sum_{i=1}^{n} p_j x_j, \sum_{i=1}^{n} p_j (1 - x_j) \right\}.$$

## 2.2 Computational Complexity

In contrast to the description of a problem, which is usually short, the search space is most of the time exponential in the problem dimension. In addition, for a lot of combinatorial optimization problems, one cannot hope to come up

with an algorithm that produces for all problem instances an optimal solution within a time bound that is polynomial in the problem dimension. The performance measure most widely used to analyze algorithms is the time an algorithm takes to present its final answer. Time is expressed in terms of number of elementary operations such as comparisons or branching instructions (Papadimitriou and Steiglitz, 1998). The time an algorithm needs to give the final answer is analyzed with respect to the input size. The input of a combinatorial optimization problem is often a graph or a set of integers. This input has to be represented as a sequence of symbols of a finite alphabet. The size of the input is the length of this sequence, that is, the number of symbols in it.

In this book, we are dealing with combinatorial optimization problems. Often we are considering a graph $G = (V, E)$ with $n$ vertices and $m$ edges and are searching for a subgraph $G' = (V', E')$ of the given one that fulfills given properties.

One approach to represent a graph is to do it by an adjacency matrix $A_G = [a_{ij}]$, where $a_{ij} = 1$ if $(v_i, v_j) \in E$ and $a_{ij} = 0$ otherwise. This matrix has $n^2$ entries, i.e., the number of entries is quadratic with respect to the number of vertices. An entry $a_{ij} = 1$ means that there is an edge from $v_i$ to $v_j$ and $a_{ij} = 0$ holds if this is not the case. Note that the adjacency matrix of a given undirected graph is symmetric. An undirected graph may have up to $\binom{n}{2} = \Theta(n^2)$ edges. However, if we are considering so-called sparse graphs, the number of edges is far less than $\binom{n}{2}$.

In the case of sparse graphs, it is better to represent a given graph by so-called adjacency lists. Here, for each vertex $v \in V$ we record a set $A(v) \subseteq V$ of vertices that are adjacent to it. The size of the representation is given by the sum of the length of lists. As each edge contributes 2 to this total length, we have to write down $2m$ elements. Another factor which effects the total length of the representation is how to encode the vertices. Our alphabet has finite size. Assume the alphabet is the set $\{0, 1\}$. Therefore we need $\Theta(\log n)$ bits to encode one single vertex. This implies that we need $\Theta(m \log n)$ bits (or symbols) to represent the graph $G$. In practice we say that a graph $G$ can be encoded in $\Theta(m)$ space, which seems to be a contradiction to the previous explanation. The reason is that computers treat all integers in their range the same. Here the same space is needed to store small integers such as 5 or large integers such as $3^{12}$. We assume that graphs are considered where the number of vertices is within the integer range of the computer. This range is in most cases 0 to $2^{31}$, which means that integers are represented by 32 bits. Therefore $\Theta(m)$ is a reasonable approximation of the size of a graph and analyzing graph algorithms with respect to $m$ is accepted in practice. In most cases both parameters $n$ and $m$ are taken into account when analyzing the complexity of a graph algorithm.

Considering graph algorithms where we can bound the runtime by a polynomial in $n$ and $m$, we obviously get a polynomial-time algorithm. We have to be careful when the input includes numbers. Let $N(I)$ be the largest integer

that appears in the input. An algorithm $A$ is called pseudo-polynomial if it is polynomial in the input size $|I|$ and $N(I)$. Note that $N(I)$ can be encoded by $\Theta(\log(N(I)))$ bits. Therefore a function that is polynomial in $|I|$ and $N(I)$ is not necessarily polynomial in the input size. Often the input consists of small integers. In the case where $N(I)$ is bounded by a polynomial in $|I|$, $A$ is a polynomial-time algorithm.

An important issue that comes up when considering combinatorial optimization problems is the classification of difficult problems (Papadimitriou and Steiglitz, 1998). To distinguish between easy and difficult problems, one considers the class of problems that are solvable by a deterministic Turing machine in polynomial time and problems that are solvable by a nondeterministic Turing machine in polynomial time. We do not want to formalize the characterization of the classes $P$ and $NP$ via Turing machines and prefer to outline the characteristics and notions connected with these classes at a more intuitive level. This leads to a straightforward definition to characterize problems that belong to $P$.

**Definition 2.1.** *A problem is in P iff it can be solved by an algorithm in polynomial time.*

Problems in $P$ can therefore be solved in polynomial time by using an appropriate algorithm. Examples of problems belonging to this class are the single source shortest path problem and the minimum spanning tree problem introduced in Section 2.1.

A class that is intuitively associated with hard problems is called $NP$. Typically, $NP$ is restricted to so-called decision problems, i.e., problems whose output is either YES or NO. This restriction has a technical background and captures the essentials of the problems without simplifying them too much.

**Definition 2.2.** *A decision problem is in NP iff any given solution of the problem can be verified in polynomial time.*

For problems in $NP$, it is therefore not necessary that a solution be computable in polynomial time. It is only necessary that we can verify the solution of the problem in polynomial time. Therefore $P \subseteq NP$ holds (slightly abusing notation by restricting $P$ to decision problems), and it is widely assumed that $P \neq NP$.

Consider the following decision variant of the vertex cover problem. The question is whether a given graph $G = (V, E)$ contains a vertex cover of at most $k$ vertices. Given a solution $x$ we can easily check whether each edge is covered by $x$. This can be done in linear time by examining each edge at most once. Additionally, we can count the number of vertices chosen by $x$ in linear time and therefore verify whether $x$ is a vertex cover with at most $k$ vertices in polynomial time.

Many optimization and decision problems, including the vertex cover problem, are at least as difficult as any problem in $NP$. Such problems are called

*NP*-hard. Showing that a problem is *NP*-hard is usually done by giving a polynomial-time reduction from an *NP*-hard problem to the considered problem. This reduction involves a transformation of the known *NP*-hard problem to the considered one, which has to be done in polynomial time. Such a reduction links the considered problem to the known *NP*-hard problem in such a way that iff the considered problem can be solved in polynomial time also the *NP*-hard problem to which it has been reduced can. We do not want to go into the details and refer the reader to a book on complexity theory (Wegener, 2005a) for further reading.

**Definition 2.3.** *A problem is called NP-hard iff it is at least as difficult as any problem in NP, i.e., each problem in NP can be reduced to it.*

As we are considering optimization problems in this book, we want to point out that many optimization problems are *NP*-hard but not in *NP*. We consider the vertex cover problem again, but at this time its optimization variant where the task is to compute a vertex cover of minimal size. Clearly, this optimization variant is at least as difficult as the problem of deciding whether a given graph contains a vertex cover of at most $k$ vertices. However, since the output of the optimization problem is a number, it is not a decision problem and, therefore, not in *NP*.

In summary, many optimization problems are at least as difficult as any problem in *NP*, i.e., *NP*-hard but not in *NP*. Problems that are *NP*-hard and also in *NP* are called *NP*-complete. This holds for many decision variants of *NP*-hard optimization problems.

**Definition 2.4.** *A problem is NP-complete iff it is NP-hard and in NP.*

The classical approach to deal with *NP*-hard problems is to search for good approximation algorithms (Hochbaum, 1997; Vazirani, 2001). These are algorithms that run in polynomial time but guarantee that the produced solution is within a given ratio of an optimal one. Such approximation algorithms can be totally different for different optimization problems. In the case of the *NP*-hard bin packing problem, even simple greedy heuristics work very well whereas in the case of more complicated scheduling problems often methods based on linear programming are used.

Another approach to solve *NP*-hard problems is to use sophisticated exact methods that have in the worst case an exponential runtime. The hope is that such algorithms produce good results for interesting problem instances in a small amount of time. A class of algorithms that tries to come up with exact solutions is branch and bound. Here the search space is shrunk during the optimization process by computing lower bounds on the value of an optimal solution in the case where we are considering maximization problems. The hope is to come up in a short period of time with a solution that matches such a lower bound. In this case an optimal solution has been obtained.

Related to this is the research on *parametrized complexity* (Downey and Fellows, 1999). Here, parametrized versions of given optimization problems are

studied. These are usually decision problems in the classical sense. Consider for example the decision variant of the vertex cover problem where we ask whether a given graph has a vertex cover of at most $k$ vertices. This question can be answered in time $O(1.2738^k + kn)$ (Chen, Kanj, and Xia, 2006), i.e., in polynomial time for any fixed $k$, and a corresponding solution with $k$ vertices can be computed within that time bound if it exists. Obviously, this approach can be turned into an optimization algorithm that is efficient iff the value of an optimal solution is small.

A crucial consideration in combinatorial optimization problems and stochastic search algorithms that search more or less locally is the neighborhood of the current search point. Let $s \in S$ be a search point in a given search space. The neighborhood is defined by a mapping $N\colon S \to 2^S$. In the case we are considering combinatorial optimization problems from the search space $\{0,1\}^n$, the neighborhood can be naturally defined by all solutions having at most Hamming distance $k$ from the current solution $s$. The parameter $k$ determines the size of the neighborhood from which the next solution is sampled. Choosing a small value $k$, e.g. $k = 1$, such a heuristic may get stuck in local optima. If the value of $k$ is large (in the extreme case $k = n$) and all search points of the neighborhood are chosen with the same probability, the next solution will be somehow independent of $s$. This leads to stochastic search algorithms that behave almost as if they were choosing in each step a search point uniformly at random from $\{0,1\}^n$. In this case the stochastic search algorithm does not take the previously sampled function values into account and the search cannot be directed into "good" regions of the considered search space.

## 2.3 Approximation Versus Exact Optimization

As already mentioned, *NP*-hard problems probably do not allow exact solutions in polynomial time, so good approximations of optimal solutions are desired. A formal definition of the quality of approximations is based on a fixed approximation algorithm and the worst case from the set of instances for the combinatorial optimization problem.

**Definition 2.5.** *Given an algorithm $A$ for the solution of a combinatorial optimization problem $(S, f, \Omega)$, let $s_A \in S$ denote a solution produced by $A$ and $f_A := f(s_A)$ its $f$-value. Given $f_{\text{opt}}$, the $f$-value of an optimal solution, the approximation ratio of $s_A$ is defined by $f_A/f_{\text{opt}}$ for minimization problems and by $f_{\text{opt}}/f_A$ for maximization problems.*

We say that an algorithm maintains a certain approximation ratio if it produces solutions of this approximation ratio on all instances of the underlying problem. In particular, we are interested in algorithms achieving a certain approximation ratio within polynomial time.

**Definition 2.6.** *A polynomial-time approximation algorithm with ratio $r$ to a combinatorial optimization problem is an algorithm that computes solutions of approximation ratio $r$ in polynomial time with respect to the input size.*

In the previous definition, $r$ might depend on the problem size, which is for example the case if the possible approximation ratios become worse for growing inputs. The special case of a *constant* approximation ratio is given if $r$ can be bounded independently of the problem size. Often, constant approximation ratios are obtainable even for *NP*-hard problems. An even stronger property is demanded by specifying the constant approximation ratio as a parameter of the approximation algorithm.

**Definition 2.7.** *A polynomial-time approximation scheme (PTAS) to a combinatorial optimization problem is an algorithm with parameter $\epsilon$ that computes solutions of approximation ratio $1+\epsilon$ in polynomial time with respect to the input size. If the time is also polynomial with respect to $1/\epsilon$, the algorithm is called fully polynomial-time approximation scheme (FPTAS).*

Definitions 2.6 and 2.7 require polynomial time with probability 1 and are more suitable for deterministic than for randomized algorithms. A natural relaxation of the definitions is to allow expected polynomial time, resulting in expected-polynomial-time approximation algorithms and schemes. However, it is more convenient to prescribe polynomial time with a certain success probability. This results in the following definition (Motwani and Raghavan, 1995).

**Definition 2.8.** *A polynomial-time randomized approximation scheme (PRAS) to a combinatorial optimization problem is an algorithm with parameter $\epsilon$ that with probability at least $3/4$ computes solutions of approximation ratio $1+\epsilon$ in polynomial time with respect to the input size.*

The somewhat mysterious bound 3/4 on the success probability goes back to applications of PRASs to a generalization of optimization problems, the so-called number problems. However, the exact value is not too significant. Any constant success probability can be boosted to at least 3/4 by running the approximation algorithm a constant number of times and taking the best solution out of the runs. In the domain of EAs, this is usually referred to as *multistart* schemes.

We will get to know characterizations of EAs as approximation algorithms in Chapter 12 and characterizations as PRASs in Chapters 6 and 7.

## 2.4 Multi-objective Optimization

Many problems in computer science ask for solutions with certain attributes or properties that can be expressed as functions mapping possible solutions

to scalar numeric values. The usual optimization approach is to take these attributes as constraints to determine the feasibility of a solution, while one of them is chosen as an objective function to determine the preference order of the feasible solutions. In the minimum spanning tree problem, as a simple example, constraints are imposed on the number of connected components (one) and the number of cycles (zero) of the chosen subgraph, while the total weight of its edges is the objective to be minimized.

A more general approach is multi-objective optimization (Ehrgott, 2005), where several attributes are employed as objective functions and used to define a partial preference order of the solutions, with respect to which the set of minimal (maximal) elements is sought. Most of the best known single-objective polynomial solvable problems like shortest path or minimum spanning tree become NP-hard when at least two weight functions have to be optimized at the same time. In this sense, multi-objective optimization is considered as more (at least as) difficult than (as) single-objective optimization.

In the case of multi-objective optimization, the objective function $f = (f_1, \ldots, f_k)$ is vector-valued, i.e., $f\colon S \to \mathbb{R}^k$. Since there is no canonical complete order on $\mathbb{R}^k$, one compares the quality of search points with respect to the canonical partial order on $\mathbb{R}^k$, namely $f(s) \leq f(s')$ iff $f_i(s) \leq f_i(s')$ for all $i \in \{1, \ldots, k\}$. A Pareto optimal search point $s$ is a search point such that (in the case of minimization problems) $f(s)$ is minimal with respect to this partial order and all $f(s'), s' \in S$. Again, there can be many Pareto optimal search points, but they do not necessarily have the same objective vector. The Pareto front, denoted by $F$, consists of all objective vectors $y = (y_1, \ldots, y_k)$ such that there exists a search point $s$ where $f(s) = y$ and $f(s') \leq f(s)$ implies $f(s') = f(s)$. The Pareto set consists of all solutions whose objective vector belongs to the Pareto front. The problem is to compute the Pareto front and for each element $y$ of the Pareto front one search point $s$ such that $f(s) = y$. We sometimes say that a search point $s$ belongs to the Pareto front, which means that its objective vector belongs to the Pareto front.

As in the case of optimization problems, one may be satisfied with approximate solutions. This can be formalized as follows. For each element $y$ of the Pareto front, we have to compute a solution $s$ such that $f(s)$ is close enough to $y$. Close enough is measured by an appropriate metric and an approximation parameter. In the single-objective case, one switches to the approximation variant if exact optimization is too difficult. The same reason may hold in the multi-objective case. There may be another reason. The size of the Pareto front may be too large for exact optimization.

The Pareto front $F$ may contain exponentially many objective vectors. Papadimitriou and Yannakakis (2000) have examined how to approximate the Pareto front for different multi-objective combinatorial optimization problems. W. l. o. g., they have considered the task of maximizing all objective functions. Given an instance $I$ and a parameter $\epsilon > 0$ they have examined how to obtain an $\epsilon$-approximate Pareto set. This is a set of solutions $X$ with the property that there is no solution $s'$ such that for all $s \in X$ $f_i(s') \geq (1 + \epsilon) \cdot f_i(s)$

holds for at least one $i$. Papadimitriou and Yannakakis (2000) showed that there exists an algorithm which constructs such a set $X$, which is polynomially bounded in $|I|$ and $1/\epsilon$ if and only if the corresponding gap problem problem can be solved. Given an instance $I$ of the considered problem and a vector $(b_1, \ldots, b_k)$, the gap problem consists of either presenting a solution $s$ with $f_i(s) \geq b_i$, $1 \leq i \leq k$, or answering that there is no solution $s'$ with $f_i(s') \geq (1+\epsilon) \cdot b_i$, $1 \leq i \leq k$. In the case of some multi-objective optimization problems (e.g., the multi-objective variants of the minimum spanning tree problem and the shortest path problem), such a set can also be computed within a time bound that is polynomial in $|I|$ and $1/\epsilon$. Algorithms with such properties constitute an FPTAS (Definition 2.7), which is the best we can hope for when dealing with NP-hard problems.

http://www.springer.com/978-3-642-16543-6

Bioinspired Computation in Combinatorial Optimization
Algorithms and Their Computational Complexity
Neumann, F.; Witt, C.
2010, XII, 216 p., Hardcover
ISBN: 978-3-642-16543-6