

Chapter 2

Optimization Problems

Optimization problems are common in many disciplines and various domains. In optimization problems, we have to find solutions which are optimal or near-optimal with respect to some goals. Usually, we are not able to solve problems in one step, but we follow some process which guides us through problem solving. Often, the solution process is separated into different steps which are executed one after the other. Commonly used steps are recognizing and defining problems, constructing and solving models, and evaluating and implementing solutions.

Combinatorial optimization problems are concerned with the efficient allocation of limited resources to meet desired objectives. The decision variables can take values from bounded, discrete sets and additional constraints on basic resources, such as labor, supplies, or capital, restrict the possible alternatives that are considered feasible. Usually, there are many possible alternatives to consider and a goal determines which of these alternatives is best. The situation is different for continuous optimization problems which are concerned with the optimal setting of parameters or continuous decision variables. Here, no limited number of alternatives exist but optimal values for continuous variables have to be determined.

The purpose of this chapter is to set the stage and give an overview of properties of optimization problems that are relevant for modern heuristics. We describe the process of how to create a problem model that can be solved by optimization methods, and what can go wrong during this process. Furthermore, we look at important properties of optimization models. The most important one is how difficult it is to find optimal solutions. For some well-studied problems, we can give upper and lower bounds on problem difficulty. Other relevant properties of optimization problems are their locality and decomposability. The locality of a problem is exploited by local search methods, whereas the decomposability is exploited by recombination-based search methods. Consequently, we discuss the locality and decomposability of a problem and how it affects the performance of modern heuristics.

The chapter is structured as follows: Sect. 2.1 describes the process of solving optimization problems. In Sect. 2.2, we discuss problems and problem instances. Relevant definitions and properties of optimization models are discussed in Sect. 2.3. We describe common metrics that can be defined on a search space, resulting neigh-

borhoods, and the concept of a fitness landscape. Finally, Sect. 2.4 deals with properties of problems. We review complexity theory as a tool for formulating upper and lower bounds on problem difficulty. Furthermore, we study the locality and decomposability of a problem and their importance for local and recombination-based search, respectively.

2.1 Solution Process

Researchers, users, and organizations like companies or public institutions are confronted in their daily life with a large number of planning and optimization problems. In such problems, different decision alternatives exist and a user or an organization has to select one of these. Selecting one of the available alternatives has some impact on the user or the organization which can be measured by some kind of evaluation criteria. Evaluation criteria are selected such that they describe the (expected) impact of choosing one of the different decision alternatives. In optimization problems, users and organizations are interested in choosing the alternative that either maximizes or minimizes an evaluation function which is defined on the selected evaluation criteria.

Usually, users and organizations cannot freely choose from all available decision alternatives but there are constraints that restrict the number of available alternatives. Common restrictions come from law, technical limitations, or interpersonal relations between humans. In summary, optimization problems have the following characteristics:

- Different decision alternatives are available.
- Additional constraints limit the number of available decision alternatives.
- Each decision alternative can have a different effect on the evaluation criteria.
- An evaluation function defined on the decision alternatives describes the effect of the different decision alternatives.

For optimization problems, a decision alternative should be chosen that considers all available constraints and maximizes/minimizes the evaluation function. For planning problems, a rational, goal-oriented planning process should be used that systematically selects one of the available decision alternatives. Therefore, planning describes the process of generating and comparing different courses of action and then choosing one prior to action.

Planning processes to solve planning or optimization problems have been of major interest in operations research (OR) (Taha, 2002; Hillier and Lieberman, 2002; Domschke and Drexl, 2005). Planning is viewed as a systematic, rational, and theory-guided process to analyze and solve planning and optimization problems. The planning process consists of several steps:

1. Recognizing the problem,
2. defining the problem,
3. constructing a model for the problem,

4. solving the model,
5. validating the obtained solutions, and
6. implementing one solution.

The following sections discuss the different process steps in detail.

2.1.1 Recognizing Problems

In the very first step, it must be recognized that there is a planning or optimization problem. This is probably the most difficult step as users or institutions often quickly get used to a currently used approach of doing business. They appreciate the current situation and are not aware that there are many different ways to do their business or to organize a task. Users or institutions are often not aware that there might be more than one alternative they can choose from.

The first step in problem recognition is that users or institutions become aware that there are different alternatives (for example using a new technology or organizing the current business in a different way). Such an analysis of the existing situation often occurs as a result of external pressure or changes in the environment. If everything goes well, users and companies do not question the currently chosen decision alternatives. However, when running into economic problems (for example accumulating losses or losing market share), companies have to think about re-structuring their processes or re-shaping their businesses. Usually, a re-design of business processes is done with respect to some goals. Designing the proper (optimal) structure of the business processes is an optimization problem.

A problem has been recognized if users or institutions have realized that there are other alternatives and that selecting from these alternatives affects their business. Often, problem recognizing is the most difficult step as users or institutions have to abandon the current way of doing business and accept that there are other (and perhaps better) ways.

2.1.2 Defining Problems

After we have identified a problem, we can describe and define it. For this purpose, we must formulate the different decision alternatives, study whether there are any additional constraints that must be considered, select evaluation criteria which are affected by choosing different alternatives, and determine what are the goals of the planning process. Usually, there is not only one possible goal but we have to choose from a variety of different goals. Possible goals of a planning or optimization process are either to find an optimal solution for the problem or to find a solution that is better than some predefined threshold (for example the current solution).

An important aspect of problem definition is the selection of relevant decision alternatives. There is a trade-off between the number of decision alternatives and

the difficulty of the resulting problem. The more decision alternatives we have to consider, the more difficult it is to choose a proper alternative. In principle, we can consider all possible decision alternatives (independently of whether they are relevant for the problem, or not) and try to solve the resulting optimization problem. However, since such problems can not be solved in a reasonable way, usually only decision alternatives are considered that are relevant and which affect the evaluation criteria. All aspects that have no direct impact on the goal of the planning process are neglected. Therefore, we have to focus on carefully selected parts of the overall problem and find the right level of abstraction.

It is important to define the problem large enough to ensure that solving the problem yields some benefits and small enough to be able to solve the problem. The resulting problem definition is often a simplified problem description.

2.1.3 Constructing Models

In this step, we construct a *model* of the problem which represents its essence. Therefore, a model is a (usually simplified) representative of the real world. Mathematical models describe reality by extracting the most relevant relationships and properties of a problem and formulating them using mathematical symbols and expressions. Therefore, when constructing a model, there are always aspects of reality that are idealized or neglected. We want to give an example. In classical mechanics, the energy E of a moving object can be calculated as $E = \frac{1}{2}mv^2$, where m is the object's mass and v its velocity. This model describes the energy of an object well if v is much lower than the speed of light c ($v \ll c$) but it becomes inaccurate for $v \rightarrow c$. Then, other models based on the special theory of relativity are necessary. This example illustrates that the model used is always a representation of the real world.

When formulating a model for optimization problems, the different decision alternatives are usually described by using a set of *decision variables* $\{x_1, \dots, x_n\}$. The use of decision variables allows modeling of the different alternatives that can be chosen. For example, if somebody can choose between two decision alternatives, a possible decision variable would be $x \in \{0, 1\}$, where $x = 0$ represents the first alternative and $x = 1$ represents the second one. Usually, more than one decision variable is used to model different decision alternatives (for choosing proper decision variables see Sect. 2.3.2). Restrictions that hold for the different decision variables can be expressed by constraints. Representative examples are relationships between different decision variables (e.g. $x_1 + x_2 \leq 2$). The *objective function* assigns an *objective value* to each possible decision alternative and measures the quality of the different alternatives (e.g. $f(x) = 2x_1 + 4x_2^2$). One possible decision alternative, which is represented by different values for the decision variables, is called a *solution* of a problem.

To construct a model with an appropriate level of abstraction is a difficult task (Schneeweiß, 2003). Often, we start with a realistic but unsolvable problem model

and then iteratively simplify the model until it can be solved by existing optimization methods. There is a basic trade-off between the ability of optimization methods to solve a model (*tractability*) and the similarity between the model and the underlying real-world problem (*validity*). A step-wise simplification of a model by iteratively neglecting some properties of the real-world problem makes the model easier to solve and more tractable but reduces the relevance of the model.

Often, a model is chosen such that it can be solved by using existing optimization approaches. This especially holds for classical optimization methods like the Simplex method or branch-and-bound-techniques which guarantee finding the optimal solution. In contrast, the use of modern heuristics allow us to reduce the gap between reality and model and to solve more relevant problem models. However, we have to pay a price since such methods often find good solutions but we have no guarantee that the solutions found are optimal.

Two other relevant aspects of model construction are the availability of relevant data and the testing of the resulting model. For most problems, is it not sufficient to describe the decision variables, the relationships between the decision variables, and the structure of the evaluation function, but additional parameters are necessary. These parameters are often not easily accessible and have to be determined by using simulation and other predictive techniques. An example problem is assigning jobs to different agents. Relevant for the objective value of an assignment is the order of jobs. To be able to compare the duration of different assignments (each specific assignment is a possible decision alternative), parameters like the duration of one work step, the time that is necessary to transfer a job to a different agent, or the setup times of the different agents are relevant. These additional parameters can be determined by analyzing or simulating real-world processes.

Finally, a model is available which should be a representative of the real problem but is usually idealized and simplified in comparison to the real problem. Before continuing with this model, we must ensure that the model is a valid representative of the real world and really represents what we originally wanted to model. A proper criterion for judging the correctness of a model is whether different decision alternatives are modeled with sufficient accuracy and lead to the expected results. Often, the relevance of a model is evaluated by examining the relative differences of the objective values resulting from different decision alternatives.

2.1.4 Solving Models

After we have defined a model of the original problem, the model can be solved by some kind of *algorithm* (usually an optimization algorithm). An algorithm is a procedure (a finite set of well-defined instructions) for accomplishing some task. An algorithm starts in an initial state and terminates in a defined end-state. The concept of an algorithm was formalized by Turing (1936) and Church (1936) and is at the core of computers and computer science. In optimization, the goal of an algorithm

is to find a solution (either specific values for the decision variables or one specific decision alternative) with minimal or maximal evaluation value.

Practitioners sometimes view solving a model as simple, as the outcome of the model construction step is already a model that can be solved by some kind of optimization method. Often, they are not aware that the effort to solve a model can be high and only small problem instances can be solved with reasonable effort. They believe that solving a model is just applying a *black-box optimization method* to the problem at hand. An algorithm is called a black-box algorithm if it can be used without any further problem-specific adjustments.

However, we have a trade-off between tractability and specificity of optimization methods. If optimization methods are to perform well for the problem at hand, they usually need to be adapted to the problem. This is typical for modern heuristics but also holds for classical optimization methods like branch-and-bound approaches. Modern heuristics can easily be applied to problems that are very realistic and near to real-world problems but usually do not guarantee finding an optimal solution. Modern heuristics should not be applied out of the box as black-box optimization algorithms but adapted to the problem at hand. To design high-quality heuristics is an art as they are problem-specific and exploit properties of the model.

Comparing classical OR methods like the Simplex method with modern heuristics reveals that for classical methods constructing a valid model of the real problem is demanding and needs the designer's intuition. Model solution is simple as existing algorithms can be used which yield optimal solutions (Ackoff, 1973). The situation is different for modern heuristics, where formulating a model is often a relatively simple step as modern heuristics can also be applied to models that are close to the real world. However, model solution is difficult as standard variants of modern heuristics usually show limited performance and only problem-specific and model-specific variants yield high-quality solutions (Droste and Wiesmann, 2002; Puchta and Gottlieb, 2002; Bonissone et al, 2006).

2.1.5 Validating Solutions

After finding optimal or near-optimal solutions, we have to evaluate them. Often, a *sensitivity analysis* is performed which studies how the optimal solution depends on variations of the model (for example using different parameters). The use of a sensitivity analysis is necessary to ensure that slight changes of the problem, model, or model parameters do not result in large changes in the resulting optimal solution.

Another possibility is to perform *retrospective tests*. Such tests use historical data and measure how well the model and the resulting solution would have performed if they had been used in the past. Retrospective tests can be used to validate the solutions, to evaluate the expected gains from new solutions, and to identify problems of the underlying model. For the validation of solutions, we must also consider that the variables that are used as input for the model are often based on historical data. In general, we have no guarantee that past behavior will correctly forecast future

behavior. A representative example is the prediction of stock indexes. Usually, prediction methods are designed such that they well predict historical developments of stock indexes. However, as the variables that influence stock indexes are continuously changing, accurate predictions of future developments with unforeseen events are very difficult, if not impossible.

2.1.6 Implementing Solutions

Validated solutions have to be implemented. There are two possibilities: First, a validated solution is implemented only once. The outcome of the planning or optimization process is a solution that usually replaces an existing, inferior, solution. The solution is implemented by establishing the new solution. After establishing the new solution, the planning process is finished. An example is the redesign of a company's distribution center. The solution is a new design of the processes in the distribution center. After establishing the new design, the process terminates.

Second, the model is used and solved repeatedly. Then, we have to install a well-documented system that allows the users to continuously apply the planning process. The system includes the model, the solution algorithm, and procedures for implementation. An example is a system for finding optimal routes for deliveries and pick-ups of trucks. Since the problem continuously changes (there are different customers, loads, trucks), we must continuously determine high-quality solutions and are not satisfied with a one-time solution.

We can distinguish between two types of systems. Automatic systems run in the background and no user-interaction is necessary for the planning process. In contrast, *decision support systems* determine proper solutions for a problem and present the solution to the user. Then, the user is free to modify the proposed solution and to decide.

This book focuses on the design of modern heuristics. Therefore, defining and solving a model are of special interest. Although the steps before and afterwards in the process are of equal (or even higher) importance, we refrain from studying them in detail but refer the interested reader to other literature (Turban et al, 2004; Power, 2002; Evans, 2006).

2.2 Problem Instances

We have seen in the previous section how the construction of a model is embedded in the solution process. When building a model, we can represent different decision alternatives using a vector $\mathbf{x} = (x_1, \dots, x_n)$ of n decision variables. We denote an assignment of specific values to \mathbf{x} as a solution. All solutions together form a set X of solutions, where $\mathbf{x} \in X$.

Decision variables can be either continuous ($\mathbf{x} \in \mathbb{R}^n$) or discrete ($\mathbf{x} \in \mathbb{Z}^n$). Consequently, optimization models are either continuous where all decision variables are real numbers, combinatorial where the decision variables are from a finite, discrete set, or mixed where some decision variables are real and some are discrete. The focus of this book is on models for combinatorial optimization problems. Typical sets of solutions used for combinatorial optimization models are integers, permutations, sets, or graphs.

We have seen in Sect. 2.1 that it is important to carefully distinguish between an optimization problem and possible optimization models which are more or less accurate representations of the underlying optimization problem. However, in optimization literature, this distinction is not consistently made and often models are denoted as problems. A representative example is the traveling salesman problem which in most cases denotes a problem model and not the underlying problem. We follow this convention throughout this book and usually talk about problems meaning the underlying model of the problem and, if no confusion can occur, we do not explicitly distinguish between problem and model.

We want to distinguish between problems and *problem instances*. An instance of a problem is a pair (X, f) , where X is a set of feasible solutions $x \in X$ and $f : X \rightarrow \mathbb{R}$ is an evaluation function that assigns a real value to every element x of the search space. A solution is feasible if it satisfies all constraints. The problem is to find an $x^* \in X$ for which

$$f(x^*) \geq f(x) \quad \text{for all } x \in X \quad (\text{maximization problem}) \quad (2.1)$$

$$f(x^*) \leq f(x) \quad \text{for all } x \in X \quad (\text{minimization problem}), \quad (2.2)$$

x^* is called a globally optimal solution (or optimal solution if no confusion can occur) to the given problem instance.

An *optimization problem* is defined as a set I of instances of a problem. A problem instance is a concrete realization of an optimization problem and an optimization problem can be viewed as a collection of problem instances with the same properties and which are generated in a similar way. Most users of optimization methods are usually dealing with problem instances as they want to have a better solution for a particular problem instance. Users can obtain a solution for a problem instance since all parameters are usually available. Therefore, it is also possible to compare the quality of different solutions for problem instances by evaluating them using the evaluation function f .

(2.1) and (2.2) are examples of definitions of optimization problems. However, it is expensive to list all possible $x \in X$ and to define the evaluation function f separately for each x . A more elegant way is to use standardized model formulations. A representative formulation for optimization models that is understood by people working with optimization problems as well as computer software is:

$$\begin{aligned}
& \text{minimize} && z = f(x), \\
& \text{subject to} && \\
& && g_i(x) \geq 0, \quad i \in \{1, \dots, m\}, \\
& && h_i(x) = 0, \quad i \in \{1, \dots, p\}, \\
& && x \in W_1 \times W_2 \times \dots \times W_n, \quad W_i \in \{\mathbb{R}, \mathbb{Z}, \mathbb{B}\}, \quad i \in \{1, \dots, n\},
\end{aligned} \tag{2.3}$$

where x is a vector of n decision variables x_1, \dots, x_n , $f(x)$ is the objective function that is used to evaluate different solutions, and $g(x)$ and $h(x)$ are inequality and equality constraints on the variables x_i . \mathbb{B} indicates the set of binary values $\{0, 1\}$.

By using such a formulation, we can model optimization problems with inequality and equality constraints. We cannot describe models with other types of constraints or where the evaluation function f or the constraints g_i and h_j cannot be formulated in an algorithmic way. Also not possible are multi-criteria optimization problems where more than one evaluation criterion exists (Deb, 2001; Coello Coello et al, 2007; Collette and Siarry, 2004). To formulate models in a standard way allow us to easily recognize relevant structures of the model, to easily add details of the model (e.g. additional constraints), and to feed it directly into computer programs (problem solvers) that can compute optimal or good solutions.

We see that in standard optimization problems, there are decision alternatives, restrictions on the decision alternatives, and an evaluation function. Generally, the decision alternatives are modeled as a vector of variables. These variables are used to construct the restrictions and the objective criteria as a mathematical function. By formulating them, we get a mathematical model relating the variables, constraints, and objective function. Solving this model yields the values of the decision variables that optimize (maximize or minimize) values of the objective function while satisfying all constraints. The resulting solution is referred to as an optimal feasible solution.

2.3 Search Spaces

In optimization models, a search space X is implicitly defined by the definition of the decision variables $x \in X$. This section defines important aspects of search spaces. Section 2.3.1 introduces metrics that can be used for measuring similarities between solutions in metric search spaces. In Sect. 2.3.2, neighborhoods in a search space are defined based on the metric used. Finally, Sects. 2.3.3 and 2.3.4 introduce fitness landscapes and discuss differences between locally and globally optimal solutions.

2.3.1 Metrics

To formulate an optimization model, we need to define a *search space*. Intuitively, a search space contains the set of feasible solutions of an optimization problem. Furthermore, a search space can define relationships (for example distances) between solutions.

Very generally, a search space can be defined as a *topological space*. A topological space is a generalization of metric search spaces (as well as other types of search spaces) and describes similarities between solutions not by defining distances between solutions but by relationships between sets of solutions. A topological space is an ordered pair (X, T) , where X is a set of solutions (points) and T is a collection of subsets of X called open sets. A set Y is in X (denoted $Y \subseteq X$) if every element $x \in Y$ is also in X ($x \in Y \Rightarrow x \in X$). A topological space (X, T) has the following properties

1. the empty set \emptyset and whole space X are in T ,
2. the intersection of two elements of T is again in T , and
3. the union of an arbitrary number of elements of T is again in T .

We can define different topologies (search spaces) by combining X with different T . For a given X , the most simple search space is $T = \{\emptyset, X\}$, which is called the *trivial topology* or indiscrete topology. The trivial topology can be used for describing search spaces where no useful metrics between the different decision alternatives are known or can be defined. For the definition of a topological space, we need no definition of similarity between the different elements in the search space but the definition of relationships between different subsets is sufficient. We give examples. Given $X = \{a, b, c\}$, we can define the trivial topology with only two subsets $T = \{\{\}, \{a, b, c\}\}$. More complex topologies can be defined by different T . For example, defining four subsets $T = \{\{\}, \{a\}, \{a, b\}, \{a, b, c\}\}$ results in a different topology. For more information on topological spaces, we refer to Buskes and van Rooij (1997) and Bredon (1993).

Metric search spaces are a specialized form of topological spaces where the similarities between solutions are measured by a distance. Therefore, in metric search spaces, we have a set X of solutions and a real-valued distance function (also called a metric)

$$d : X \times X \rightarrow \mathbb{R}$$

that assigns a real-valued distance to any combination of two elements $x, y \in X$. In metric search spaces, the following properties must hold:

$$\begin{aligned} d(x, y) &\geq 0, \\ d(x, x) &= 0, \\ d(x, y) &= d(y, x), \\ d(x, z) &\leq d(x, y) + d(y, z), \end{aligned}$$

where $x, y, z \in X$.

An example of a metric space is the set of real numbers \mathbb{R} . Here, a metric can be defined by $d(x, y) := |x - y|$. Therefore, the distance between any solutions $x, y \in \mathbb{R}$ is just the absolute value of their differences. Extending this definition to 2-dimensional spaces \mathbb{R}^2 , we get the *city-block metric* (also known as taxicab metric or Manhattan distance). It is defined for 2-dimensional spaces as

$$d(x, y) := |x_1 - y_1| + |x_2 - y_2|, \quad (2.4)$$

where $x = (x_1, x_2)$ and $y = (y_1, y_2)$. This metric is named the city-block metric as it describes the distance between two points on a 2-dimensional plane in a city like Manhattan or Mannheim with a rectangular ground plan. On n -dimensional search spaces \mathbb{R}^n , the city-block metric becomes

$$d(x, y) := \sum_{i=1}^n |x_i - y_i|, \quad (2.5)$$

where $x, y \in \mathbb{R}^n$.

Another example of a metric that can be defined on \mathbb{R}^n is the *Euclidean metric*. In Euclidean spaces, a solution $x = (x_1, \dots, x_n)$ is a vector of continuous values ($x_i \in \mathbb{R}$). The Euclidean distance between two solutions x and y is defined as

$$d(x, y) := \sqrt{\sum_{i=1}^n (x_i - y_i)^2}. \quad (2.6)$$

For $n = 1$, the Euclidean metric coincides with the city-block metric. For $n = 2$, we have a standard 2-dimensional search space and the distance between two elements $x, y \in \mathbb{R}^2$ is just a direct line between two points on a 2-dimensional plane.

If we assume that we have a binary space ($x \in \{0, 1\}^n$), a commonly used metric is the binary *Hamming metric* (Hamming, 1980)

$$d(x, y) = \sum_{i=1}^n |x_i - y_i|, \quad (2.7)$$

where $d(x, y) \in \{0, \dots, n\}$. The binary Hamming distance between two binary vectors x and y of length n is just the number of binary decision variables on which x and y differ. It can be extended to continuous and discrete decision variables:

$$d(x, y) = \sum_{i=1}^n z_i, \quad (2.8)$$

where

$$z_i = \begin{cases} 0, & \text{for } x_i = y_i, \\ 1, & \text{for } x_i \neq y_i. \end{cases}$$

In general, the Hamming distance measures the number of decision variables on which x and y differ.

2.3.2 Neighborhoods

The definition of a *neighborhood* is important for optimization problems as it determines which solutions are similar to each other. A neighborhood is a mapping

$$N(x) : X \rightarrow 2^X, \quad (2.9)$$

where X is the search space containing all possible solutions to the problem. 2^X stands for the set of all possible subsets of X and N is a mapping that assigns to each element $x \in X$ a set of elements $y \in X$. A neighborhood definition can be viewed as a mapping that assigns to each solution $x \in X$ a set of solutions y that are neighbors of x . Usually, the neighborhood $N(x)$ defines a set of solutions y which are in some sense similar to x .

The definition of a topological space (X, T) already defines a neighborhood as it introduces an abstract structure of space in the set X . Given a topological space (X, T) , a subset N of X is a neighborhood of a point $x \in X$ if N contains an open set $U \subset T$ containing the point x . We want to give examples. For the trivial topology $(\{a, b, c\}, \{\{\}, \{a, b, c\}\})$, the points in the search space cannot be distinguished by topological means and either all or no points are neighbors to each other. For $(\{a, b, c\}, \{\{\}, \{a\}, \{a, b\}, \{a, b, c\}\})$, the points a and b are neighbors.

Many optimization models use metric search spaces. A metric search space is a topological space where a metric between the elements of the set X is defined. Therefore, we can define similarities between solutions based on the distance d . Given a metric search space, we can use balls to define a neighborhood. For $x \in X$, an (open) ball around x of radius ε is defined as the set

$$B_\varepsilon := \{y \in X \mid d(x, y) < \varepsilon\}.$$

The ε -neighborhood of a point $x \in X$ is the open set consisting of all points whose distance from x is less than ε . This means that all solutions $y \in X$ whose distance d from x is lower than ε are neighbors of x . By using balls we can define a neighborhood function $N(x)$. Such a function defines for each x a set of solutions similar to x .

Figure 2.1 illustrates the definition of a neighborhood in a 2-dimensional continuous search space \mathbb{R}^2 for Euclidean distances (Fig. 2.1(a)) and Manhattan distances (Fig. 2.1(b)). Using an open ball, all solutions y where $d(x, y) < \varepsilon$ are neighboring solutions to x . For Euclidean distances, we use $d(x, y) := \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}$ and neighboring solutions are all solutions that can be found inside of a circle around x with radius ε . For city-block distances, we use $d(x, y) := |x_1 - y_1| + |x_2 - y_2|$ and all solutions inside a rhombus with the vertices $(x_1 - \varepsilon, y_1), (x_1, y_1 + \varepsilon), (x_1 + \varepsilon, y_1), (x_1, y_1 - \varepsilon)$ are neighboring solutions.

It is problematic to apply metric search spaces to problems where no meaningful similarities between different decision alternatives can be defined or do not exist. For such problems, the only option is to define a trivial topology (see p. 16) which assumes that all solutions are neighbors and no meaningful structure on the search

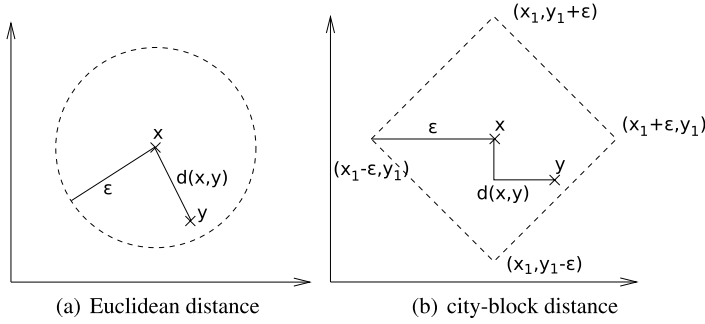


Fig. 2.1 Neighborhoods on a 2-dimensional Euclidean space using different metrics

space exists. However, practitioners (as well as users) are used to metric search spaces and often seek to apply them also to problems where no meaningful similarities between different decision alternatives can be defined. This is a mistake as a metric search space does not model such a problem in an appropriate way.

We want to give an example. We assume a search space containing four different fruits (apple (a), banana (b), pear (p), and orange (o)). This search space forms a trivial topology $(\{a, b, p, o\}, \{\emptyset, \{a, b, p, o\}\})$ as no meaningful distances between the four fruits exist and all solutions are neighbors of each other. Nevertheless, we can define a metric search space $X = \{0, 1\}^2$ for the problem. Each solution $((0, 0), (0, 1), (1, 0), \text{ and } (1, 1))$ represents one fruit. Although the original problem defines no similarities, the use of a metric space induces that the solution $(0, 0)$ is more similar to $(0, 1)$ than to $(1, 1)$ (using Hamming distance (2.7)). Therefore, a metric space is inappropriate for the problem definition as it defines similarities where no similarities exist. A more appropriate model would be $x \in \{0, \dots, 3\}$ and using Hamming distance (2.8). Then, all distances between the different solutions are equal and all solutions are neighbors.

A different problem can occur if the metric used is not appropriate for the problem and existing “similarities” between different decision alternatives do not fit the similarities between different solutions described by the model. The metric defined for the problem model is a result of the choice of the decision variables. Any choice of decision variables $\{x_1, \dots, x_n\}$ allows the definition of a metric space and, thus, defines similarities between different solutions. However, if the metric induced by the use of the decision variables does not fit the metric of the problem description, the problem model is inappropriate.

Table 2.1 illustrates this situation. We assume that there are $s = 9$ different decision alternatives $\{a, b, c, d, e, f, g, h, i\}$. We assume that the decision alternatives form a metric space (using the Hamming metric (2.8)), where the distances between all elements are equal. Therefore, all decision alternatives are neighbors (for $\varepsilon > 1$). In the first problem model (model 1), we use a metric space $X = \{0, 1, 2\}^2$ and Hamming metric (2.8). Therefore, each decision alternative is represented by

(x_1, x_2) with $x_i \in \{0, 1, 2\}$. For the Hamming metric, each solution has four neighbors. For example, decision alternative $(1, 1)$ is a neighbor of $(1, 2)$ but not of $(2, 2)$. Model 1 results in a different neighborhood in comparison to the original decision alternatives. Model 2 is an example of a different metric space. In this model, we use binary variables x_{ij} and the search space is defined as $X = x_{ij}$, where $x_{ij} \in \{0, 1\}$. We have an additional restriction, $\sum_j x_{ij} = 1$, where $i \in \{1, 2\}$ and $j \in \{1, 2, 3\}$. Again, Hamming distance (2.8) can be used. For $\varepsilon = 1.1$, no neighboring solutions exist. For $\varepsilon = 2.1$, each solution has four neighbors. We see that different models for the same problem result in different neighborhoods which do not necessarily coincide with the neighborhoods of the original problem.

Table 2.1 Two different search spaces for a problem

| decision alternatives | model 1 (x_1, x_2) | model 2 $\begin{pmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \end{pmatrix}$ |
|---------------------------------|--|---|
| $\{a, b, c, d, e, f, g, h, i\}$ | $\{(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)\}$ | $\left\{ \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix} \right\}$ |

The examples illustrate that selecting an appropriate model is important. For the same problem, different models are possible which can result in different neighborhoods. We must select the model such that the metric induced by the model fits well the metric that exists for the decision alternatives. Although, in the problem description no neighborhood needs to be defined, usually a notion of neighborhoods exist. Users that formulate a model description often know which decision alternatives are similar to each other as they have a feeling about which decision alternatives result in the same outcome. When constructing the model, we must ensure that the neighborhood induced by the model fits well the (often intuitive) neighborhood formulated by the user.

Relevant aspects which determine the resulting neighborhood of a model are the type and number of decision variables. The types of decision variables should be determined by the properties of the decision alternatives. If decision alternatives are continuous (for example, choosing the right amount of crushed ice for a drink), the use of continuous decision variables in the model is useful and discrete decision variables should not be used. Analogously, for discrete decision alternatives, discrete decision variables and combinatorial models should be preferred. For example, the number of ice cubes in a drink should be modelled using integers and not continuous variables.

The number of decision variables used in the model also affects the resulting neighborhood. For discrete models, there are two extremes: first, we can model s different decision alternatives by using only one decision variable that can take s

different values. Second, we can use $l = \log_2(s)$ binary decision variables $x_i \in \{0, 1\}$ ($i \in \{1, \dots, l\}$). If we use Hamming distance (2.8) and define neighboring solutions as $d(x, y) \leq 1$, then all possible solutions are neighbors if we use only one decision variable. In contrast, each solution $x \in \{0, 1\}^l$ has only l neighbors $y \in \{0, 1\}^l$, where $d(x, y) \leq 1$. We see that using different numbers of decision variables for modeling the decision alternatives results in completely different neighborhoods. In general, a high-quality model is a model where the neighborhoods defined in the model fit well the neighborhoods that exist in the problem.

2.3.3 Fitness Landscapes

For combinatorial search spaces where a metric is defined, we can introduce the concept of *fitness landscape* (Wright, 1932). A fitness landscape (X, f, d) of a problem instance consists of a set of solutions $x \in X$, an objective function f that measures the quality of each solution, and a distance measure d . Figure 2.2 is an example of a one-dimensional fitness landscape.

We denote $d_{\min} = \min_{x, y \in X} (d(x, y))$, where $x \neq y$, as the minimum distance between any two elements x and y of a search space. Two solutions x and y are denoted as neighbors if $d(x, y) = d_{\min}$. Often, d can be normalized to $d_{\min} = 1$. A fitness landscape can be described using a graph G_L with a vertex set $V = X$ and an edge set $E = \{(x, y) \in X \times X \mid d(x, y) = d_{\min}\}$ (Reeves, 1999a; Merz and Freisleben, 2000b). The objective function assigns an objective value to each vertex. We assume that each solution has at least one neighbor and the resulting graph is connected. Therefore, an edge exists between neighboring solutions. The distance between two solutions $x, y \in X$ is proportional to the number of nodes that are on the path of minimal length between x and y in the graph G_L . The maximum distance $d_{\max} = \max_{x, y \in X} (d(x, y))$ between any two solutions $x, y \in X$ is called the diameter $\text{diam } G_L$ of the landscape.

We want to give an example: We use the search space defined by model 1 in Table 2.1 and Hamming distance (2.8). Then, all solutions where only one decision variable is different are neighboring solutions ($d(x, y) = d_{\min}$). The maximum distance $d_{\max} = 2$. More details on fitness landscapes can be found in Reeves and Rowe (2003, Chap. 9) or Deb et al (1997).

2.3.4 Optimal Solutions

A globally optimal solution for an optimization problem is defined as the solution $x^* \in X$, where $f(x^*) \leq f(x)$ for all $x \in X$ (minimization problem). For the definition of a globally optimal solution, it is not necessary to define the structure of the search space, a metric, or a neighborhood.

Given a problem instance (X, f) and a neighborhood function N , a feasible solution $x' \in X$ is called locally optimal (minimization problem) with respect to N if

$$f(x') \leq f(x) \quad \text{for all } x \in N(x'). \quad (2.10)$$

Therefore, locally optimal solutions do not exist if no neighborhood is defined. Furthermore, the existence of local optima is determined by the neighborhood definition used as different neighborhoods can result in different locally optimal solutions.

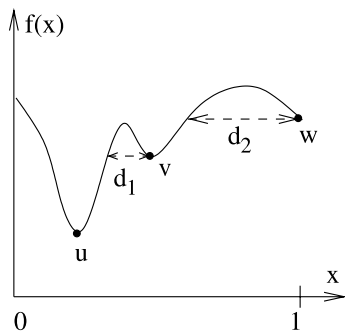


Fig. 2.2 Locally and globally optimal solutions

Figure 2.2 illustrates the differences between locally and globally optimal solutions and shows how local optima depend on the definition of N . We have a one-dimensional minimization problem with $x \in [0, 1] \in \mathbb{R}$. We assume an objective function f that assigns objective values to all $x \in X$. Independently of the neighborhood used, u is always the globally optimal solution. If we use the 1-dimensional Euclidean distance (2.6) as metric and define a neighborhood around x as $N(x) = \{y \mid y \in X \text{ and } d(x, y) \leq \varepsilon\}$ the solution v is a locally optimal solution if $\varepsilon < d_1$. Analogously, w is locally optimal for all neighborhoods with $\varepsilon < d_2$. For $\varepsilon \geq d_2$, the only locally optimal solution is the globally optimal solution u .

The modality of a problem describes the number of local optima in the problem. Unimodal problems have only one local optimum (which is also the global optimum) whereas multi-modal problems have multiple local optima. In general, multi-modal problems are more difficult for guided search methods to solve than unimodal problems.

2.4 Properties of Optimization Problems

The purpose of optimization algorithms is to find high-quality solutions for a problem. If possible, they should identify either optimal solutions x^* , near-optimal solutions $x \in X$, where $f(x) - f(x^*)$ is small, or at least locally optimal solutions.

Problem difficulty describes how difficult it is to find an optimal solution for a specific problem or problem instance. Problem difficulty is defined independently

of the optimization method used. Determining the difficulty of a problem is often a difficult task as we have to prove that there are no optimization methods that can better solve the problem. Statements about the difficulty of a problem are method-independent as they must hold for all possible optimization methods.

We know that different types of optimization methods lead to different search performance. Often, optimization methods perform better if they exploit some characteristics of an optimization problem. In contrast, methods that do not exploit any problem characteristics, like black-box optimization techniques, usually show low performance. For an example, we have a look at *random search*. In random search, solutions $x \in X$ are iteratively chosen in a random order. We want to assume that each solution is considered only once by random search. When random search is stopped, it returns the best solution found. During random search, new solutions are chosen randomly and no problem-specific information about the structure of the problem or previous search steps is used. The number of evaluations needed by random search is the number of elements drawn from X , which is independent of the problem itself (if we assume a unique optimal solution). Consequently, a distinction between “easy” and “difficult” problems is meaningless when random search is the only available optimization method.

The following sections study properties of optimization problems. Section 2.4.1 starts with complexity classes which allow us to formulate bounds on the performance of algorithmic methods. This allows us to make statements about the difficulty of a problem. Then, we continue with properties of optimization problems that can be exploited by modern heuristics. Section 2.4.2 introduces the locality of a problem and presents corresponding measurements. The locality of a problem is exploited by guided search methods which perform well if problem locality is high. Important for high locality is a proper definition of a metric on the search space. Finally, Sect. 2.4.3 discusses the decomposability of a problem and how recombination-based optimization methods exploit a problem’s decomposability.

2.4.1 Problem Difficulty

The complexity of an algorithm is the effort (usually time or memory) that is necessary to solve a particular problem. The effort depends on the input size, which is equal to the size n of the problem to be solved. The difficulty or complexity of a problem is the lowest possible effort that is necessary to solve the problem.

Therefore, problem difficulty is closely related to the complexity of algorithms. Based on the complexity of algorithms, we are able to find upper and lower bounds on the problem difficulty. If we know that an algorithm can solve a problem, we automatically have an upper bound on the difficulty of the problem, which is just the complexity of the algorithm. For example, we study the problem of finding a friend’s telephone number in the telephone book. The most straightforward approach is to search through the whole book starting from “A”. The effort for doing this increases linearly with the number of names in the book. Therefore, we have an upper bound

on the difficulty of the problem (problem has at most linear complexity) as we know a linear algorithm that can solve the problem. A more effective way to solve this problem is bisection or binary search which iteratively splits the entries of the book into halves. With n entries, we only need $\log(n)$ search steps to find the number. So, we have a new, improved, upper bound on problem difficulty.

Finding lower bounds on the problem difficulty is more difficult as we have to show that no algorithm exists that needs less effort to solve the problem. Our problem of finding a friend's name in a telephone book is equivalent to the problem of searching an ordered list. Binary search which searches by iteratively splitting the list into halves is optimal and there is no method with lower effort (Knuth, 1998). Therefore, we have a lower bound and there is no algorithm that needs less than $\log(n)$ steps to find an address in a phone book with n entries. A problem is called *closed* if the upper and lower bounds on its problem difficulty are identical. Consequently, the problem of searching an ordered list is closed.

This section illustrates how bounds on the difficulty of problems can be derived by studying the effort of optimization algorithms that are used to solve the problems. As a result, we are able to classify problems as easy or difficult with respect to the performance of the best-performing algorithm that can solve the problem.

The following paragraphs give an overview of the Landau notation which is an instrument for formulating upper and lower bounds on the effort of optimization algorithms. Thus, we can also use it for describing problem difficulty. Then, we illustrate that each optimization problem can also be modeled as a decision problem of the same difficulty. Finally, we illustrate different complexity classes (P, NP, NP-hard, and NP-complete) and discuss the tractability of decision and optimization problems.

2.4.1.1 Landau Notation

The Landau notation (which was introduced by Bachmann (1894) and made popular by the work of Landau (1974)) can be used to compare the asymptotic growth of functions and is helpful when measuring the complexity of problems or algorithms. It allows us to formulate asymptotic upper and lower bounds on function values. For example, Landau notation can be used to determine the minimal amount of memory or time that is necessary to solve a specific problem. With $n \in \mathbb{N}$, $c \in \mathbb{R}$, and $f, g : \mathbb{N} \rightarrow \mathbb{R}$ the following bounds can be described using the Landau symbols:

- **asymptotic upper bound** (“big O notation”):
 $f \in O(g) \Leftrightarrow \exists c > 0 \exists n_0 > 0 \forall n \geq n_0 : |f(n)| \leq c|g(n)|$: f is dominated by g .
- **asymptotically negligible** (“little o notation”):
 $f \in o(g) \Leftrightarrow \forall c > 0 \exists n_0 > 0 \forall n \geq n_0 : |f(n)| < c|g(n)|$: f grows slower than g .
- **asymptotic lower bound**:
 $f \in \Omega(g) \Leftrightarrow g \in O(f)$: f grows at least as fast as g .
- **asymptotically dominant**:
 $f \in \omega(g) \Leftrightarrow g \in o(f)$: f grows faster than g .

- **asymptotically tight bound:**

$f \in \Theta(g) \Leftrightarrow g \in O(f) \wedge f \in O(g)$: g and f grow at the same rate.

Using this notation, it is easy to compare the difficulty of different problems: using $O(g)$ and $\Omega(g)$, it is possible to give an upper, respectively lower bound for the asymptotic running time f of algorithms that are used to solve a problem. It is important to have in mind that lower bounds on problem difficulty hold for all possible optimization problems for a specific problem. In contrast, upper bounds only indicate that there is at least one algorithm that can solve the problem with this effort but there are also other algorithms where a higher effort is necessary. The Landau notation does not consider constant factors since these mainly depend on the computer or implementation used to solve the problem. Therefore, constants do not directly influence the difficulty of a problem.

We give three small examples. In problem 1, we want to find the smallest number in an unordered list of n numbers. The complexity of this problem is $O(n)$ (it increases linearly with n) when using linear search and examining all possible elements in the list. As it is not possible to solve this problem faster than linear, there is no gap between the lower bound $\Omega(n)$ and upper bound $O(n)$. In problem 2, we want to find an element in an ordered list with n items (for example finding a telephone number in the telephone book). Binary search (bisection) iteratively splits the list into two halves and can find any item in $\log(n)$ search steps. Therefore, the upper bound on the complexity of this problem is $O(\log(n))$. Again, the lower bound is equal to the upper bound (Harel and Rosner, 1992; Cormen et al, 2001) and the complexity of the problem is $\Theta(\log(n))$. Finally, in problem 3 we want to sort an array of n arbitrary elements. Using standard sorting algorithms like merge sort (Cormen et al, 2001) it can be solved in $O(n \log(n))$. As the lower bound is $\Omega(n \log(n))$, the difficulty of this problem is $\Theta(n \log(n))$.

2.4.1.2 Optimization Problems, Evaluation Problems, and Decision Problems

To derive upper and lower bounds on problem difficulty, we need a formal description of the problem. Thus, at least the solutions $x \in X$ and the objective function $f : X \rightarrow \mathbb{R}$ must be defined. The search space can be very trivial (e.g. have a trivial topology) as the definition of a neighborhood structure is not necessary. Developing bounds is difficult for problems where the objective function does not systematically assign objective values to each solution. Although describing X and f is sufficient for formulating a problem model, in most problems of practical relevance the size $|X|$ of the search space is large and, thus, the direct assignment of objective values to each possible solution is often very time-consuming and not appropriate for building an optimization model that should be solved by a computer.

To overcome this problem, we can define each optimization problem implicitly using two algorithms \mathcal{A}_X and \mathcal{A}_f , and two sets of parameters S_X and S_f . Given a set of parameters S_X , the algorithm $\mathcal{A}_X(x, S_X)$ decides whether the solution x is an element of X , i.e. whether x is a feasible solution. Given a set of parameters S_f , the algorithm $\mathcal{A}_f(x, S_f)$ calculates the objective value of a solution x . Therefore,

$\mathcal{A}_f(x, S_f)$ is equivalent to the objective function $f(x)$. For given \mathcal{A}_X and \mathcal{A}_f , we can define different instances of a combinatorial optimization problem by assigning different values to the parameters in S_X and S_f .

We want to give an example. We have a set of s different items and want to find a subset of n items with minimal weight. Then, the algorithm \mathcal{A}_X checks whether a solution x is feasible. The set S_X contains only one parameter which is the number n of items that should be found. If x contains exactly n items, $\mathcal{A}_X(x, S_X)$ indicates that x is a feasible solution. The parameters S_f are the weights w_i of the different items. Algorithm \mathcal{A}_f calculates the objective value of a feasible solution x by summing up the weights of the n items ($f(x, w) = \sum w_i$) using the solution x and the weights w_i . Using these definitions, we can formulate an *optimization problem* as:

Given are the two algorithms \mathcal{A}_X and \mathcal{A}_f and representations of the parameters S_X and S_f .
The goal is to find the optimal feasible solution.

This formulation of an optimization problem is equivalent to (2.3) (p. 15). The algorithm \mathcal{A}_X checks whether all constraints are met and \mathcal{A}_f calculates the objective value of x . Analogously, the *evaluation version* of an optimization problem can be defined as:

Given are the two algorithms \mathcal{A}_X and \mathcal{A}_f and representations of the parameters S_X and S_f .
The goal is to find the objective value of the optimal solution.

Finally, the *decision version* (also known as recognition version) of an optimization problem can be defined as:

Given are the algorithms \mathcal{A}_X and \mathcal{A}_f , representations of the parameters S_X and S_f , and an integer L . The goal is to decide whether there is a feasible solution $x \in X$ such that $f(x) \leq L$.

The first two versions are problems where an optimal solution has to be found, whereas in the decision version of an optimization problem a question has to be answered either by *yes* or *no*. We denote feasible solutions x whose objective value $f(x) \leq L$ as *yes-solutions*. We can solve the decision version of the optimization problem by solving the original optimization problem, calculating the objective value $f(x^*)$ of the optimal solution x^* , and deciding whether $f(x^*)$ is greater than L . Therefore, the difficulty of the three versions is roughly the same if we assume that $f(x^*)$, respectively $\mathcal{A}_f(x^*, S_f)$, is easy to compute (Papadimitriou and Steiglitz, 1982, Chap. 15.2). If this is the case, all three versions are equivalent.

We may ask why we want to formulate an optimization as a decision problem which is much less intuitive? The reason is that in computational complexity theory, many statements on problem difficulty are formulated for decision (and not optimization) problems. By formulating optimization problems as decision problems, we can apply all these results to optimization problems. Therefore, complexity classes which can be used to categorize decision problems in classes with different difficulty (compare the following paragraphs) can also be used for categorizing optimization problems. For more information on the differences between optimization, evaluation, and decision versions of an optimization problem, we refer the interested reader to Papadimitriou and Steiglitz (1982, Chap. 15) or Harel and Rosner (1992).

2.4.1.3 Complexity Classes

Computational complexity theory ((Hartmanis and Stearns, 1965; Cook, 1971; Garey and Johnson, 1979; Papadimitriou and Yannakakis, 1991; Papadimitriou, 1994; Arora and Barak, 2009) allows us to categorize decision problems in different groups based on their difficulty. The difficulty of a problem is defined with respect to the amount of computational resources that are at least necessary to solve the problem.

In general, the effort (amount of computational resources) that is necessary to solve an optimization problem of size n is determined by its time and space complexity. *Time complexity* describes how many iterations or number of search steps are necessary to solve a problem. Problems are more difficult if more time is necessary. *Space complexity* describes the amount of space (usually memory on a computer) that is necessary to solve a problem. As for time, problem difficulty increases with higher space complexity. Usually, time and space complexity depend on the input size n and we can use the Landau notation to describe upper and lower bounds on them. A *complexity class* is a set of computational problems where the amount of computational resources that are necessary to solve the problem shows the same asymptotic behavior. For all problems that are contained in one complexity class, we can give bounds on the computational complexity (in general, time and space complexity). Usually, the bounds depend on the size n of the problem, which is also called its *input size*. Usually, n is much smaller than the size $|X|$ of the search space. Typical bounds are asymptotic lower or upper bounds on the time that is necessary to solve a particular problem.

Complexity Class P

The complexity class P (P stands for polynomial) is defined as the set of decision problems that can be solved by an algorithm with worst-case polynomial time complexity. The time that is necessary to solve a decision problem in P is asymptotically bounded (for $n > n_0$) by a polynomial function $O(n^k)$. For all problems in P, an algorithm exists that can solve any instance of the problem in time that is $O(n^k)$, for some k . Therefore, all problems in P can be solved effectively in the worst case. As we showed in the previous section that all optimization problems can be formulated as decision problems, the class P can be used to categorize optimization problems.

Complexity Class NP

The class NP (which stands for non-deterministic polynomial time) describes the set of decision problems where a *yes* solution of a problem can be verified in polynomial time. Therefore, both the formal representation of a solution x and the time it takes to check its validity (to check whether it is a *yes* solution) must be polynomial or polynomially-bounded.

Therefore, all problems in NP have the property that their *yes* solutions can be checked effectively. The definition of NP says nothing about the time necessary for verifying *no* solutions and a problem in NP can not necessarily be solved in polynomial time. Informally, the class NP consists of all “reasonable” problems of practical importance where a *yes* solution can be verified in polynomial time: this means the objective value of the optimal solution can be calculated fast. For problems not in NP, even verifying that a solution is valid (is a *yes* answer) can be extremely difficult (needs exponential time).

An alternative definition of NP is based on the notion of *non-deterministic algorithms*. Non-deterministic algorithms are algorithms which have the additional ability to guess any verifiable intermediate result in a single step. If we assume that we find a *yes* solution for a decision problem by iteratively assigning values to the decision variables, a non-deterministic algorithm always selects the value (possibility) that leads to a *yes* answer, if a *yes* answer exists for the problem. Therefore, we can view a non-deterministic algorithm as an algorithm that always guesses the right possibility whenever the correctness can be checked in polynomial time. The class NP is the set of all decision problems that can be solved by a non-deterministic algorithm in worst-case polynomial time. The two definitions of NP are equivalent to each other. Although non-deterministic algorithms cannot be executed directly by conventional computers, this concept is important and helpful for the analysis of the computational complexity of problems.

All problems that are in P also belong to the class NP. Therefore, $P \subseteq NP$. An important question in computational complexity is whether P is a proper subset of NP ($P \subset NP$) or whether NP is equal to P ($P = NP$). So far, this question is not finally answered (Fortnow, 2009) but most researchers assume that $P \neq NP$ and there are problems that are in NP, but not in P.

In addition to the classes P and NP, there are also problems where *yes* solutions cannot be verified in polynomial time. Such problems are very difficult to solve and are, so far, of only little practical relevance.

Tractable and Intractable Problems

When solving an optimization problem, we are interested in the running time of the algorithm that is able to solve the problem. In general, we can distinguish between polynomial running time and exponential running time. Problems that can be solved using a polynomial-time algorithm (there is an upper bound $O(n^k)$ on the running time of the algorithm, where k is constant) are *tractable*. Usually, tractable problems are easy to solve as running time increases relatively slowly with larger input size n . For example, finding the lowest element in an unordered list of size n is tractable as there are algorithms with time complexity that is $O(n)$. Spending twice as much effort solving the problem allows us to solve problems twice as large.

In contrast, problems are *intractable* if they cannot be solved by a polynomial-time algorithm and there is a lower bound on the running time which is $\Omega(n^k)$, where $k > 1$ is a constant and n is the problem size (input size). For example, guessing the

correct number for a digital door lock with n digits is an intractable problem, as the time necessary for finding the correct key is $\Omega(10^n)$. Using a lock with one more digit increases the number of required search steps by a factor of 10. For this problem, the size of the problem is n , whereas the size of the search space is $|X| = 10^n$. The effort to find the correct key depends on n and increases at the same rate as the size of the search space. Table 2.2 lists the growth rate of some common functions ordered by how fast they grow.

Table 2.2 Polynomial (top) and exponential (bottom) functions

| | |
|----------------------------|-----------------|
| constant | $O(1)$ |
| logarithmic | $O(\log n)$ |
| linear | $O(n)$ |
| quasilinear | $O(n \log n)$ |
| quadratic | $O(n^2)$ |
| polynomial (of order c) | $O(n^c), c > 1$ |
| exponential | $O(k^n)$ |
| factorial | $O(n!)$ |
| super-exponential | $O(n^n)$ |

- We can identify three different types of problems with different difficulty.
- Tractable problems with known polynomial-time algorithms. These are easy problems. All tractable problems are in P.
 - Provably intractable problems, where we know that there is no polynomial-time algorithm. These are difficult problems.
 - Problems where no polynomial-time algorithm is known but intractability has not yet been shown. These problems are also difficult.

NP-Hard and NP-Complete

All decision problems that are in P are tractable and thus can be easily solved using the “right” algorithm. If we assume that $P \neq NP$, then there are also problems that are in NP but not in P. These problems are difficult as no polynomial-time algorithms exist for them.

Among the decision problems in NP, there are problems where no polynomial algorithm is available and which can be transformed into each other with polynomial effort. Consequently, a problem is denoted *NP-hard* if an algorithm for solving this problem is polynomial-time reducible to an algorithm that is able to solve *any* problem in NP. A problem A is polynomial-time reducible to a different problem B if and only if there is a transformation that transforms any arbitrary solution x of A into a solution x' of B in polynomial time such that x is a *yes* instance for A if and only if x' is a *yes* instance for B . Informally, a problem A is reducible to some other problem B if problem B either has the same difficulty or is harder than problem A . Therefore, NP-hard problems are at least as hard as any other problem in NP, although they might be harder. Therefore, NP-hard problems are not necessarily in NP.

Cook (1971) introduced the set of *NP-complete* problems as a subset of NP. A decision problem A is denoted NP-complete if

- A is in NP and
- A is NP-hard.

Therefore, no other problem in NP is more than a polynomial factor harder than any NP-complete problem. Informally, NP-complete problems are the most difficult problems that are in NP.

All NP-complete problems form one set as all NP-complete problems have the same complexity. However, it is as yet unclear whether NP-complete problems are tractable. If we are able to find a polynomial-time algorithm for any one of the NP-complete problems, then every NP-complete problem can be solved in polynomial time. Then, all other problems in NP can also be solved in polynomial time (are tractable) and thus $P = NP$. On the other hand, if it can be shown that one NP-complete problem is intractable, then all NP-complete problems are intractable and $P \neq NP$.

Summarizing our discussion, we can categorize optimization problems with respect to the computational effort that is necessary for solving them. Problems that are in P are usually easy as algorithms are known that solve such problems in polynomial time. Problems that are NP-complete are difficult as no polynomial-time algorithms are known. Decision problems that are not in NP are even more difficult as we could not evaluate in polynomial time whether a particular solution for such a problem is feasible. To be able to calculate upper and lower bounds on problem complexity, usually well-defined problems are necessary that can be formulated in functional form.

2.4.2 Locality

In general, the *locality* of a problem describes how well the distances $d(x, y)$ between any two solutions $x, y \in X$ correspond to the difference of the objective values $|f(x) - f(y)|$ (Lohmann, 1993; Rechenberg, 1994; Rothlauf, 2006). The locality of a problem is high if neighboring solutions have similar objective values. In contrast, the locality of a problem is low if low distances do not correspond to low differences of the objective values. Relevant determinants for the locality of a problem are the metric defined on the search space and the objective function f .

In the heuristic literature, there are a number of studies on locality for discrete decision variables (Weicker and Weicker, 1998; Rothlauf and Goldberg, 1999, 2000; Gottlieb and Raidl, 2000; Gottlieb et al, 2001; Whitley and Rowe, 2005; Caminiti and Petreschi, 2005; Raidl and Gottlieb, 2005; Paulden and Smith, 2006) as well as for continuous decision variables (Rechenberg, 1994; Igel, 1998; Sendhoff et al, 1997b,a). For continuous decision variables, locality is also known as *causality*. High and low locality correspond to strong and weak causality, respectively. Although causality and locality describe the same concept and causality is the older

one, we refer to the concept as *locality* as it is currently more often used in the literature.

Guided search methods are optimization approaches that iteratively sample solutions and use the objective values of previously sampled solutions to guide the future search process. In contrast to random search which samples solutions randomly and uses no information about previously sampled solutions, guided search methods differentiate between promising (for maximization problems these are solutions with high objective values) and non-promising (solutions with low objective values) areas in the fitness landscape. New solutions are usually generated in the neighborhood of promising solutions with high objective values. A prominent example of guided search is greedy search (see Sect. 3.4.1).

The locality of optimization problems has a strong impact on the performance of guided search methods. Problems with high locality allow guided search methods to find high-quality solutions in the neighborhood of already found good solutions. Furthermore, the underlying idea of guided search methods to move in the search space from low-quality solutions to high-quality solutions works well if the problem has high locality. In contrast, if a problem has low locality, guided search methods cannot make use of previous search steps to extract information that can be used for guiding the search. Then, for problems with low locality, guided search methods behave like random search.

One of the first approaches to the question of what makes problems difficult for guided search methods, was the study of deceptive problems by Goldberg (1987) which was based on the work of Bethke (1981). In deceptive problems, the objective values are assigned in such a way to the solutions that guided search methods are led away from the global optimal solution. Therefore, based on the structure of the fitness landscape (Weinberger, 1990; Manderick et al, 1991; Deb et al, 1997), the correlation between the fitness of solutions can be used to describe how difficult a specific problem is to solve for guided search methods. For an overview of correlation measurements and problem difficulty we refer to Bäck et al (1997, Chap. B2.7) or Reeves and Rowe (2003).

The following paragraphs present approaches that try to determine what makes a problem difficult for guided search. Their general idea is to measure how well the metric defined on the search space fits the structure of the objective function. A high fit between metric and structure of the fitness function makes a problem easy for guided search methods.

2.4.2.1 Fitness-Distance Correlation

A straightforward approach for measuring the difficulty of problems for guided search methods has been presented in Jones and Forrest (1995). They assumed that the difficulty of an optimization problem is determined by how the objective values are assigned to the solutions $x \in X$ and what metric is defined on X . Combining both aspects, problem difficulty can be measured by the *fitness-distance correlation coefficient*

$$\rho_{FDC} = \frac{c_{fd}}{\sigma(f)\sigma(d_{opt})}, \quad (2.11)$$

where

$$c_{fd} = \frac{1}{m} \sum_{i=1}^m (f_i - \langle f \rangle)(d_{i,opt} - \langle d_{opt} \rangle)$$

is the covariance of f and d_{opt} . $\langle f \rangle$, $\langle d_{opt} \rangle$, $\sigma(f)$, and $\sigma(d_{opt})$ are the means and standard deviations of the fitness f and the distance d_{opt} to the optimal solution x^* , respectively (Jones, 1995a; Jones and Forrest, 1995; Altenberg, 1997). $d_{i,opt}$ is the distance of solution i to the optimal solution x^* . The fitness-distance correlation coefficient $\rho_{FDC} \in [-1, 1]$ measures the linear correlation between the fitness of search points and their distances to the global optimum x^* .

As ρ_{FDC} represents a summary statistic of f and d_{opt} , it works well if f and d_{opt} follow a bivariate normal distribution. For problems where f and d_{opt} do not follow a normal distribution, using the correlation as a measure of problem difficulty for guided search methods will not yield meaningful results (Jones and Forrest, 1995).

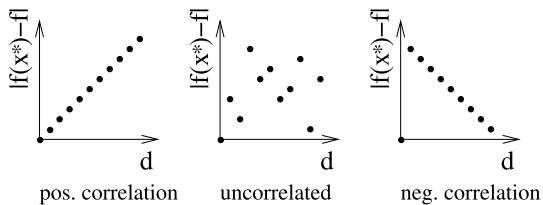
Using the fitness-distance correlation coefficient, Jones and Forrest (1995) classified fitness landscapes (for maximization problems) into three classes, straightforward ($\rho_{FDC} \leq -0.15$), difficult ($-0.15 < \rho_{FDC} < 0.15$), and misleading ($\rho_{FDC} \geq 0.15$):

1. Straightforward: For such problems, the fitness of a solution is correlated with the distance to the optimal solution. With lower distance, the fitness difference to the optimal solution decreases. As the structure of the search space guides search methods towards the optimal solution such problems are usually easy for guided search methods.
2. Difficult: There is no correlation between the fitness difference and the distance to the optimal solution. The fitness values of neighboring solutions are uncorrelated and the structure of the search space provides no information about which solutions should be sampled next by the search method.
3. Misleading: The fitness difference is negatively correlated to the distance to the optimal solution. Therefore, the structure of the search space misleads a guided search method to sub-optimal solutions.

For minimization problems, the situation is reversed as problems are straightforward for $\rho_{FDC} \geq 0.15$, difficult for $-0.15 < \rho_{FDC} < 0.15$, and misleading for $\rho_{FDC} \leq -0.15$. The three different classes of problem difficulty are illustrated in Fig. 2.3. We show how the fitness difference $|f(x^*) - f|$ depends on the distance d_{opt} to the optimal solution x^* . In the following paragraphs, we want to discuss these three classes in some more detail.

Problems are easy for guided search methods if there is a positive correlation between a solution's distance to the optimal solution and the difference between its fitness and the fitness of the optimal solution. An example is a one-dimensional problem where the fitness of a solution is equal to the distance to the optimal solution ($f(x) = d(x, x^*)$). Then, $\rho_{FDC} = 1$ (for a minimization problem) and the problem can easily be solved using guided search methods.

Fig. 2.3 Different classes of problem difficulty



Problems become more difficult if there is no correlation between the fitness difference and the distance to the optimal solution. The locality of such problems is low, as no meaningful relationship exists between the distances d between different solutions and their objective values. Thus, the fitness landscape cannot guide guided search methods to optimal solutions. Optimization methods cannot use information about a problem which was collected in prior search steps to determine the next search step. Therefore, all search algorithms show the same performance as no useful information (information that indicates where the optimal solution can be found) is available for the problem. Because all search strategies are equivalent, also random search is an appropriate search method for such problems. Random search uses no information and performs as well as other search methods on these types of problems.

We want to give two examples. In the first example, we have a discrete search space X with n elements $x \in X$. A deterministic random number generator assigns a random number to each x . Again, the optimization problem is to find x^* , where $f(x^*) \leq f(x)$ for all $x \in X$. Although we can define neighborhoods and similarities between different solutions, all possible optimization algorithms show the same behavior. All elements of the search space must be evaluated to find the globally optimal solution.

The second example is the needle-in-a-haystack (NIH) problem. Following its name, the goal is to find a needle in a haystack. In this problem, a metric exists defining distances between solutions, but there is no meaningful relationship between the metric and the objective value (needle found or not) of different solutions. When physically searching in a haystack for a needle, there is no good strategy for choosing promising areas of the haystack that should be searched in the next search step. The NIH problem can be formalized by assuming a discrete search space X and the objective function

$$f(x) = \begin{cases} 0 & \text{for } x \neq x^{opt} \\ 1 & \text{for } x = x^{opt}. \end{cases} \quad (2.12)$$

Figure 2.4(a) illustrates the problem. The NIH problem is equivalent to the problem of finding the largest number in an unordered list of numbers. The effort to solve such problems is high and increases linearly with the size $|X|$ of the search space. Therefore, the difficulty of the NIH problem is $\Theta(n)$.

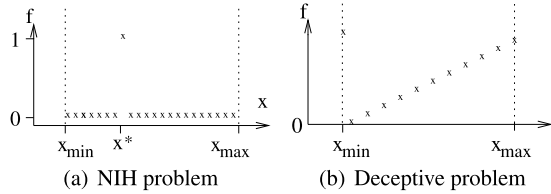


Fig. 2.4 Different types of problems

Guided search methods perform worst for problems where the fitness landscape leads the search method away from the optimal solution. Then, the distance to the optimal solution is negatively correlated to the fitness difference between a solution and the optimal solution. The locality of such problems is relatively high as most neighboring solutions have similar fitness. However, since guided search finds the optimal solution by performing iterated small steps in the direction of better solutions, all guided search approaches must fail as they are misled. All other search methods that use information about the fitness landscape also fail. More effective search methods for such problems are those that do not use information about the structure of the search space but search randomly, like random search. The most prominent example of such types of problems are deceptive traps (see Figure 2.4(b)). For this problem, the optimal solution is $x^* = x_{\min}$. The solution x_{\max} is a deceptive attractor and guided search methods that search in the direction of solutions with higher objective function always find x_{\max} , which is not the optimal solution.

A common tool for studying the fitness-distance correlation of problems is fitness distance plots. Usually, such plots are more meaningful than just calculating c_{fd} . Fitness distance problems show how the fitness of randomly sampled solutions depends on their distance to the optimal solution. Examples can be found in Kauffman (1989) (NK-landscapes), Boese (1995) (traveling salesman problem), Reeves (1999b) (flow-shop scheduling problems), Inayoshi and Manderick (1994) and Merz and Freisleben (2000b) (graph bipartitioning problem), Merz and Freisleben (2000a) (quadratic assignment problem), or Mendes et al (2002) (single machine scheduling problem).

2.4.2.2 Ruggedness

For studying the fitness-distance correlation of problems, it is necessary to know the optimal solution. However, for real-world problems the optimal solution is not a priori known and other approaches are necessary that describe how well the metric fits the structure of the objective function.

The performance of guided search methods depends on the properties of the fitness landscape like the number of local optima or peaks in the landscape, the distribution of the peaks in the search space, and the height of the different peaks. Consequently, correlation functions have been proposed to measure the ruggedness of a fitness landscape (Kauffman and Levin, 1987; Kauffman, 1989; Weinberger, 1990; Kauffman, 1993). Like in fitness-distance correlation, the idea is to consider the ob-

jective values as random variables and to obtain statistical properties on how the distribution of the objective values depends on the distances between solutions. The *autocorrelation function* (which is interchangeable with the autocovariance function if the normalization factor $\langle f^2 \rangle - \langle f \rangle^2$ is dropped) of a fitness landscape is defined as (Merz and Freisleben, 2000b)

$$\rho(d) = \frac{\langle f(x)f(y) \rangle_{d(x,y)=d} - \langle f \rangle^2}{\langle f^2 \rangle - \langle f \rangle^2}, \quad (2.13)$$

where $\langle f \rangle$ denotes the average value of f over all $x \in X$ and $\langle f(x)f(y) \rangle_{d(x,y)=d}$ is the average value of $f(x)f(y)$ for all pairs $(x,y) \in S \times S$, where $d(x,y) = d$. The autocorrelation function has the attractive property of being in the range $[-1, 1]$. An autocorrelation value of 1 indicates perfect correlation (positive correlation) and -1 indicates perfect anti-correlation (negative correlation). For a fixed distance d , ρ is the correlation between the objective values of all solutions that have a distance of d . Weinberger recognized that landscapes with exponentially decaying autocovariance functions are often easy to solve for guided search methods (Weinberger, 1990).

To calculate the autocorrelation function is demanding for optimization problems as it requires evaluating all solutions of the search space. Therefore, Weinberger used random walks through the fitness landscape to approximate the autocorrelation function. A random walk is an iterative procedure where in each search step a random neighboring solution is created. The *random walk correlation function* (Weinberger, 1990; Stadler, 1995, 1996; Reidys and Stadler, 2002) is defined as

$$r(s) = \frac{\langle f(x_i)f(x_{i+s}) \rangle - \langle f \rangle^2}{\langle f^2 \rangle - \langle f \rangle^2}, \quad (2.14)$$

where x_i is the solution examined in the i th step of the random walk. s is the number of steps between two solutions in the search space. For a fixed s , r defines the correlation of two solutions that are reached by a random walk in s steps, where $s \geq d_{\min}$. For a random walk with a large number of steps, $r(s)$ is a good estimate for $\rho(d)$.

Correlation functions have some nice properties and can be used to measure the difficulty of a problem for guided search methods. If we assume that we have a completely random problem, where random objective values are assigned to all $x \in X$, then the autocorrelation function will have a peak at $d = s = 0$ and will be close to zero for all other d and s . In general, for all possible problems the autocorrelation function reaches its peak at the origin $d = s = 0$. Thus it holds $|r(s)| \leq r(0)$ for all $0 < s \leq d_{\max}$.

When assuming that the distance between two neighboring solutions x and y is equal to one ($d(x,y) = 1$), $r(1)$ measures the correlation between the objective values of all neighboring solutions. The *correlation length* l_{corr} of a landscape (Stadler, 1992, 1996) is defined as

$$l_{\text{corr}} = -\frac{1}{\ln(|r(1)|)} = -\frac{1}{\ln(|\rho(1)|)}$$

for $r(1), \rho(1) \neq 0$.

The ruggedness of a fitness landscape depends on the correlation between neighboring solutions. If the correlation (the correlation length) is high, neighboring solutions have similar objective values and the fitness landscape is smooth and not rugged. For optimization problems where the autocorrelation function indicates strong correlation ($\rho(d) \approx 1$), guided search methods are a good choice as the structure of the search space defined by the metric fits well the structure of the objective function. High-quality solutions are grouped together in the search space and the probability of finding a good solution is higher in the neighborhood of a high-quality solution than in the neighborhood of a low-quality solution.

Correlation functions give us meaningful estimates on how difficult a problem is for guided search only if the search space possesses *regularity*. Regularity means that all elements of the landscape are visited by a random walk with equal probability (Weinberger, 1990). Many optimization problems like the traveling salesman problem (Kirkpatrick and Toulouse, 1985; Stadler and Schnabl, 1992), the quadratic assignment problem (Taillard, 1995; Merz and Freisleben, 2000a), and the flow-shop scheduling problem (Reeves, 1999b) possess regular search spaces. However, other problems like job-shop scheduling problems do not possess this regularity as random walks through the search space are biased (Bierwirth et al, 2004). Such a bias affects random walks and directed stochastic local search algorithms.

In the literature, there are various examples of how the correlation length can be used to study the difficulty of optimization problems for guided search methods (Kauffman and Levin, 1987; Kauffman, 1989; Huynen et al, 1996; Kolarov, 1997; Barnett, 1998; Angel and Zissimopoulos, 1998a, 2000, 1998b, 2001, 2002; Grahl et al, 2007).

2.4.3 Decomposability

The *decomposability* of a problem describes how well the problem can be decomposed into several, smaller subproblems that are independent of each other (Polya, 1945; Holland, 1975; Goldberg, 1989c). The decomposability of a problem is high if the structure of the objective function is such that not all decision variables must be simultaneously considered for calculating the objective function but there are groups of decision variables that can be set independently of each other. It is low if it is not possible to decompose a problem into subproblems with few interdependencies between the groups of variables.

When dealing with decomposable problems, it is important to choose the type and number of decision variables such that they fit the properties of the problem. The fit is high if the variables used result in a problem model where groups of decision variables can be solved independently or, at least, where the interactions between groups of decision variables are low. Given the set of decision variables $D = \{x_1, \dots, x_l\}$, a problem can be decomposed into several subproblems if the objective value of a solution x is calculated as $f(x) = \sum_{D_s} f(\{x_i | x_i \in D_s\})$, where D_s

are non-intersecting and proper subsets of D ($D_s \subsetneq D, \cup D_s = D$) and $i \in \{1, \dots, l\}$. Instead of summing the objective values for the subproblems also other functions (e.g. multiplication resulting in $f = \prod_{D_s} f(\{x_i | x_i \in D_s\})$) can be used.

In the previous Sect. 2.4.2, we have studied various measures of locality and discussed how the locality of a problem affects the performance of guided search methods. This section focuses on the decomposability of problems and how it affects the performance of *recombination-based search methods*. Recombination-based search methods solve problems by trying different decompositions of the problem, solving the resulting subproblems, and putting together the obtained solutions for these subproblems to get a solution for the overall problem. High decomposability of a problem usually leads to high performance of recombination-based search methods because solving a larger number of smaller subproblems is usually easier than solving the larger, original problem. Consequently, it is important for effective recombination-based search methods to identify proper subsets of variables such that there are no strong interactions between the variables of the different subsets.

We discussed in Sect. 2.3.2 that the type and number of decision variables influences the resulting neighborhood structure. In the case of recombination-based search methods, we must choose the decision variables such that the problem can be easily decomposed by the search method. We want to give an example of how the decomposition of a problem can make a problem easier for recombination-based search methods. Imagine you have to design the color and material of a chair. For each of the two design variables, there are three different options. The quality of a design is evaluated by marketing experts that assign an objective value to each combination of color and material. Overall, there are $3 \times 3 = 9$ possible chair designs. If the problem cannot be decomposed, the experts have to evaluate all nine different solutions to find the optimal design. If we assume that the color and material are independent of each other, we (or a recombination-based search method) can try to decompose the problem and separately solve the decomposed subproblems. If the experts separately decide about color and material, the problem becomes easier as only $3 + 3 = 6$ designs have to be evaluated.

Therefore, the use of recombination-based optimization methods suggests that we should define the decision variables of a problem model such that they allow a decomposition of the problem. The variables should be chosen such that there are no (or at least few) interdependencies between different sets of variables. We can study the importance of choosing proper decision variables for the chair example. The first variant assumes no decomposition of the problem. We define one decision variable $x \in X$, where $|X| = 9$. There are nine different solutions and non-recombining optimization methods have to evaluate all possible solutions to find the optimal one. In the second variant, we know that the objective function of the problem can be decomposed. Therefore, we choose two decision variables $x_1 \in X_1 = \{y, b, g\}$ (yellow, blue, and green) and $x_2 \in X_2 = \{w, m, p\}$ (wooden, metal, or plastics), where $|X_1| = |X_2| = 3$. A possible decomposition for the example problem is $f = f_1(x_1) + f_2(x_2)$ (see Table 2.3). Decomposing the problem in such a way results in two subproblems f_1 and f_2 of size $|X_1| = |X_2| = 3$. Comparing the two problem formulations shows that the resulting objective values f of different solu-

tions are the same for both formulations. However, the size of the resulting search space is lower for the second variant. Therefore, the problem becomes easier to solve for recombination-based search methods as the assumed problem decomposition ($f = f_1 + f_2$) fits well the properties of the problem.

| without problem decomposition $f = f(x_1, x_2)$ | additive problem decomposition $f = f_1(x_1) + f_2(x_2)$ |
|--|---|
| $f(y, w) = 3, f(y, m) = 2, f(y, p) = 1,$ | $f_1(y) = 0, f_2(w) = 3,$ |
| $f(b, w) = 4, f(b, m) = 3, f(b, p) = 2,$ | $f_1(b) = 1, f_2(m) = 2,$ |
| $f(g, w) = 5, f(g, m) = 4, f(g, p) = 3.$ | $f_1(g) = 2, f_2(p) = 1.$ |

Table 2.3 Two different problem formulations

We want to give another example and study two different problems with l binary decision variables $x_i \in \{0, 1\}$ ($|X| = 2^l$). In the first problem, a random objective value is assigned to each $x \in X$. This problem cannot be decomposed. In the second problem, the objective value of a solution is calculated as $f = \sum_{i=1}^l x_i$. This example problem can be decomposed. Using recombination-based search methods for the first example is not helpful as no decomposition of the problem is possible. Therefore, all efforts of recombination-based search methods to find proper decompositions of the problem are useless. The situation is different for the second example. Recombination-based methods should be able to correctly decompose the problem and to solve the l subproblems. If the decomposition is done properly by the recombination-based search method, only $2l$ different solutions need to be evaluated and the problem becomes much easier to solve once the correct decomposition of the problem is found. However, usually there is additional effort necessary for finding the correct decomposition.

We see that the choice of proper decision variables is important for the decomposability of an optimization problem. In principle, there are two extremes for combinatorial optimization problems. The one extreme is to encode all possible solutions $x \in X$ using only one decision variable x_1 , where $x_1 \in \{1, \dots, |X|\}$. Using such a problem model, no decomposition is possible as only one decision variable exists. At the other extreme, we could use $\log_2 |X|$ binary decision variables encoding the $|X|$ different solutions. Then, the number of possible decompositions becomes maximal (there are $2^{\log |X|}$ possible decompositions of the problem). Proper decision variables for an optimization model should be chosen such that they allow a high decomposition of the problem. Problem decomposition is problem-specific and depends on the properties of f . We should have in mind that using a different number of decision variables not only influences problem decomposition but also results in a different neighborhood (see also Sect. 2.3.2).

In the following paragraphs, we discuss different approaches developed in the literature to estimate how well a problem can be solved using recombination-based search methods. All approaches assume that search performance is higher if a problem can be decomposed into smaller subproblems. Section 2.4.3.1 presents polynomial problem decomposition and Sect. 2.4.3.2 illustrates the Walsh decomposition

of a problem. Finally, Sect. 2.4.3.3 discusses schemata and building blocks and how they affect the performance of recombination-based search methods.

2.4.3.1 Polynomial Decomposition

The *linearity of an optimization problem* (which is also known as *epistasis*) can be measured by its polynomial decomposition. Epistasis is low if the linear separability of a problem is high. Epistasis measures the interference between decision variables and describes how well a problem can be decomposed into smaller sub-problems (Holland, 1975; Davidor, 1989, 1991; Naudts et al, 1997). For binary decision variables, any objective function f defined on l decision variables $x_i \in \{0, 1\}$ can be decomposed into

$$f(x) = \sum_{N \subset \{1, \dots, l\}} \alpha_N \prod_{n \in N} \mathbf{e}_n^T x,$$

where the vector \mathbf{e}_n contains 1 in the n th column and 0 elsewhere, T denotes transpose, and the α_N are the coefficients (Liepins and Vose, 1991). Regarding $x = (x_1, \dots, x_l)$, we may view f as a polynomial in the variables x_1, \dots, x_l . The coefficients α_N describe the non-linearity of the problem. If there are high order coefficients, the problem function is non-linear. If a decomposed problem has only order 1 coefficients, then the problem is linear decomposable. It is possible to determine the maximum non-linearity of $f(x)$ by its highest polynomial coefficients. The higher the order of the α_N , the more non-linear the problem is.

There is some correlation between the non-linearity of a problem and its difficulty for recombination-based search methods (Mason, 1995). However, as illustrated in the following example, there could be high order α_N although the problem can still easily be solved by recombination-based search methods. The function

$$f(x) = \begin{cases} 1 & \text{for } x_1 = x_2 = 0, \\ 2 & \text{for } x_1 = 0, x_2 = 1, \\ 4 & \text{for } x_1 = 1, x_2 = 0, \\ 10 & \text{for } x_1 = x_2 = 1, \end{cases} \quad (2.15)$$

can be decomposed into $f(x) = \alpha_1 + \alpha_2 x_1 + \alpha_3 x_2 + \alpha_4 x_1 x_2 = 1 + 3x_1 + x_2 + 5x_1 x_2$. The problem is decomposable and, thus, easy for recombination-based search methods as each of the two decision variables can be solved independently of each other. However, as the problem is non-linear and high order coefficients exist, the polynomial decomposition wrongly classifies the problem as difficult. This misclassification is due to the fact that the polynomial decomposition assumes a linear decomposition and cannot appropriately describe non-linear dependencies.

2.4.3.2 Walsh Decomposition

Instead of decomposing an objective function into its polynomial coefficients, binary optimization problems can also be decomposed into the corresponding Walsh coefficients. The Walsh transformation is analogous to the discrete Fourier transformation but for functions with binary decision variables. Every real-valued function $f : \{0, 1\}^l \rightarrow \mathbb{R}$ over l binary decision variables x_i can be expressed as:

$$f(x) = \sum_{j=0}^{2^l-1} w_j \psi_j(x).$$

The *Walsh functions* $\psi_j : \{0, 1\}^l \rightarrow \{-1, 1\}$ form a set of 2^l orthogonal functions. The weights $w_j \in \mathbb{R}$ are called *Walsh coefficients*. The indices j are binary strings of length l representing the integers ranging from 0 to $2^l - 1$. The j th Walsh function is defined as:

$$\psi_j(x) = (-1)^{bc(j \wedge x)},$$

with x, j are binary strings and elements of $\{0, 1\}^l$, \wedge denotes the bitwise logical AND, and $bc(x)$ is the number of 1 bits in x (Goldberg, 1989a,b; Vose and Wright, 1998a,b). The Walsh coefficients can be computed by the Walsh transformation:

$$w_j = \frac{1}{2^l} \sum_{k=0}^{2^l-1} f(k) \psi_j(k),$$

where k is a binary string of length l . The coefficients w_j measure the contribution to the objective function by the interaction of the binary decision variables x_i indicated by the positions of the 1's in j . With increasing number of 1's in the binary string j , we have more interactions between the binary decision variables x_i . For example, w_{100} and w_{010} measure the linear contribution to f associated with the decision variable x_0 and x_1 , respectively. Analogously, w_{001} is the linear contribution of decision variable x_2 . w_{111} measures the nonlinear interaction between all three decision variables x_0, x_1 , and x_2 . Any function f over a discrete $\{0, 1\}^l$ search space can be represented as a weighted sum of all possible 2^l Walsh functions ψ_j .

Walsh coefficients are sometimes used to estimate the expected performance of recombination-based search algorithms (Goldberg, 1989a; Oei, 1992; Goldberg, 1992; Reeves and Wright, 1994; Heckendorn et al, 1996). It is known that problems are easy for recombination-based search algorithms like genetic algorithms (see Sect. 5.2.1, p. 147) if a problem has only Walsh coefficients of order 1. Furthermore, difficult problems tend to have higher order Walsh coefficients. However, analogously to the linear polynomial decomposition, the highest order of a coefficient w_i does not allow us an accurate prediction of problem difficulty. This behavior is expected as Walsh functions are polynomials (Goldberg, 1989a,b; Liepins and Vose, 1991).

The insufficient measurement of problem difficulty for recombination-based search methods can be illustrated by the example (2.15). The Walsh coefficients

are $w = (4.25, -1.75, -2.75, 1.25)$. Although the problem is easy to solve for recombination-based search methods (x_1 and x_2 can be set independently of each other), there are high-order Walsh coefficients ($w_{11} = 1.25$) which indicate high problem difficulty.

Walsh analysis not only overestimates problem difficulty but also underestimates it. For example, MAX-SAT problems (see Sect. 4.4, p. 126) are difficult (APX-hard, see Sect. 3.4.2), but have only low-order Walsh coefficients (Rana et al, 1998). For example, the MAX-3SAT problem has no coefficients of higher order than 3 and the number of non-zero coefficients of order 3 is low (Rana et al, 1998). Although, Walsh coefficients indicate that the problem is easy, recombination-based search methods cannot perform well for this difficult optimization problem (Rana et al, 1998; Rana and Whitley, 1998; Heckendorn et al, 1996, 1999).

2.4.3.3 Schemata Analysis and Building Blocks

Schemata analysis is an approach developed and commonly used for measuring the difficulty of problems with respect to genetic algorithms (GA, Sect. 5.2.1). As the main search operator of GAs is recombination, GAs are a representative example of recombination-based search methods. Schemata are usually defined for binary search spaces and thus schemata analysis is mainly useful for problems with binary decision variables. However, the idea of building blocks is also applicable to other search spaces (Goldberg, 2002). In the following paragraphs, we introduce schemata and building blocks and describe how these concepts can be used for estimating the difficulty of problems for recombination-based search methods.

Schemata

Schemata were first proposed by Holland (1975) to model the ability of GAs to process similarities between binary decision variables. When using l binary decision variables $x_i \in \{0, 1\}$, a schema $H = [h_1, h_2, \dots, h_l]$ is a sequence of symbols of length l , where $h_i \in \{0, 1, *\}$. $*$ denotes the “don’t care” symbol and tells us that a decision variable is not fixed. A schema stands for the set of solutions which match the schema at all the defined positions, i.e., those positions having either a 0 or a 1. Schemata of this form allow for coarse graining (Stephens and Waelbroeck, 1999; Contreras et al, 2003), where whole sets of strings can be treated as a single entity.

A position in a schema is fixed if there is either a 0 or a 1 at this position. The size or order $o(H)$ of a schema H is defined as the number of fixed positions (0’s or 1’s) in the schema string. The defining length $\delta(H)$ of a schema H is defined as the distance between (meaning the number of bits that are between) the two outermost fixed bits. The fitness $f_s(H)$ of a schema is defined as the average fitness of all instances of this schema and can be calculated as

$$f_s(H) = \frac{1}{||H||} \sum_{\mathbf{x} \in H} f(\mathbf{x}),$$

where $||H||$ is the number of solutions $\mathbf{x} \in \{0, 1\}^l$ that are instances of the schema H . The instances of a schema H are all solutions $\mathbf{x} \in H$. For example, $\mathbf{x} = (0, 1, 1, 0, 1)$ and $\mathbf{y} = (0, 1, 1, 0, 0)$ are instances of $H = [0 * 1 * *]$. The number of solutions that are instances of a schema H can be calculated as $2^{l-o(H)}$. For a more detailed discussion of schemata in the context of GA, we refer to Holland (1975), Goldberg (1989c), Altenberg (1994) or Radcliffe (1997).

Building Blocks

Based on schemata, Goldberg (1989c, p. 20 and p. 41) defined *building blocks* (BB) as “highly fit, short-defining-length schemata”. Although BBs are commonly used (especially in the GA literature) they are rarely defined. We can describe a BB as a solution to a subproblem that can be expressed as a schema. Such a schema has high fitness and its size is smaller than the length l of the binary solution. By combining BBs of lower order, recombination-based search methods like GAs can form high-quality over-all solutions.

We can interpret BBs also from a biological perspective and view them as genes. A gene consists of one or more alleles and can be described as a schema with high fitness. Often, genes do not strongly interact with each other and determine specific properties of individuals like hair or eye color.

BBs can be helpful for estimating the performance of recombination-based search algorithms. If the sub-solutions to a problem (the BBs) are short (low $\delta(H)$) and of low order (low $o(H)$), then the problem is assumed to be easy for recombination-based search.

BB-Based Problem Difficulty

Goldberg (2002) presented an approach for problem difficulty based on schemata and BBs. He decomposed problem difficulty for recombination-based search methods like genetic algorithms into

- difficulty within a building block (intra-BB difficulty),
- difficulty between building blocks (inter-BB difficulty), and
- difficulty outside of building blocks (extra-BB difficulty).

This decomposition of problem difficulty assumes that difficult problems are challenging for methods based on building blocks. In the following paragraphs, we briefly discuss these three aspects of BB-difficulty.

If we count the number of schemata of order $o(H) = k$ that have the same fixed positions, there are 2^k different schemata. Viewing a BB of size k as a subproblem, there are 2^k different solutions to this subproblem. Such subproblems cannot be

decomposed any more and usually guided or random search methods are applied to find the correct solution for the decomposed subproblems.

Therefore, *intra-BB difficulty* depends on the locality of the subproblem. As discussed in Sect. 2.4.2, (sub)problems are most difficult to solve if the structure of the fitness landscape leads guided search methods away from the optimal solution. Consequently, the deceptiveness (Goldberg, 1987) of a subproblem (for an example of a deceptive problem, see Fig. 2.4(b), p. 34) is at the core of intra-BB difficulty. We can define the deceptiveness of a problem not only by the correlation between objective function and distance (as we have done in Sect. 2.4.2) but also by using the notion of BBs. A problem is said to be deceptive of order k_{max} if for $k < k_{max}$ all schemata that contain parts of the best solution have lower fitness than their competitors (Deb and Goldberg, 1994). Schemata are competitors if they have the same fixed positions. An example of four competing schemata of size $k = 2$ for a binary problem of length $l = 4$ are $H_1 = [0*0*]$, $H_2 = [0*1*]$, $H_3 = [1*0*]$, and $H_4 = [1*1*]$. Therefore, the highest order k_{max} of the schemata that are not misleading determines the intra-BB difficulty of a problem. The higher the maximum order k_{max} of the schemata, the higher is the intra-BB difficulty.

Table 2.4 Average schema fitness for example described by (2.15)

| order | 2 | 1 | 0 |
|---------|-----------|------------|-------------|
| schema | 11 | 1* *1 | ** |
| fitness | 10 | 7 6 | 4.25 |
| schema | 01 10 00 | 0* *0 | |
| fitness | 2 4 1 | 1.5 2.5 | |

Table 2.4 shows the average fitness of the schemata for the example from (2.15). All schemata that contain a part of the optimal solution are above average (printed bold) and better than their competitors. Calculating the deceptiveness of the problem based on the fitness of the schemata correctly classifies this problem as very easy.

When using this concept of BB-difficulty for estimating the difficulty of a problem for recombination-based search methods, the most natural and direct way to measure problem difficulty is to analyze the size and length of the BB in the problem. The intra-BB difficulty of a problem can be measured by the maximum length $\delta(H)$ and size $k = o(H)$ of its BBs H (Goldberg, 1989c). Representative examples of use of these concepts to estimate problem difficulty can be found in Goldberg (1992), Radcliffe (1993), or Horn (1995).

Recombination-based search methods solve problems by trying different problem decompositions and solving the resulting subproblems. If a problem is correctly decomposed, optimal solutions (BBs) of the subproblems can be determined independently of each other. Often, the contributions of different subproblems to the overall objective value of a solution is non-uniform. Non-uniform contributions of subproblems to the objective value of a solution determine *inter-BB difficulty*. Problems become more difficult if some BBs have a lower contribution to the objective value of a solution. Furthermore, problems often cannot be decomposed into completely separated and independent sub-problems, but have some interdependencies between subproblems which are an additional source of inter-BB difficulty.

Sources of *extra-BB difficulty* for recombination-based search methods are factors like noise. Non-deterministic noise can randomly modify the objective values of solutions and make the problem more difficult for recombination-based search methods as no accurate decisions can be made on the optimal solutions for the different subproblems. A similar problem occurs if the evaluation of the solutions is non-stationary. Non-stationary environments result in solutions that have different evaluation values at different moments in time.



<http://www.springer.com/978-3-540-72961-7>

Design of Modern Heuristics

Principles and Application

Rothlauf, F.

2011, XI, 267 p., Hardcover

ISBN: 978-3-540-72961-7