1 Problems to Be Solved

In this chapter we discuss problems to be solved, as encountered frequently by engineers, computer scientists, etc. We argue that problems and problem solvers can, and should, be distinguished, and observe that the field of evolutionary computing is primarily concerned with problem solvers. However, to characterise any problem solver it is useful to identify the kind of problems to which it can be applied. Therefore we start this book by discussing various classes of problems, and, in fact, even different ways of classifying problems.

In the following informal discussion, we introduce the concepts and the terminology needed for our purposes by examples, only using a formal treatment when it is necessary for a good understanding of the details. To avoid controversy, we are not concerned with social or political problems. The problems we have in mind are the typical ones with which artificial intelligence is associated: more akin to puzzles (e.g., the famous zebra puzzle), numerical problems (e.g., what is the shortest route from a northern city to a southern city), or pattern discovery (e.g., what will a new customer buy in our online book store, given their gender, age, address, etc).

1.1 Optimisation, Modelling, and Simulation Problems

The classification of problems used in this section is based on a black box model of computer systems. Informally, we can think of any computer-based system as follows. The system initially sits, awaiting some input from either a person, a sensor, or another computer. When input is provided, the system processes that input through some computational model, whose details are not specified in general (hence the name black box). The purpose of this model is to represent some aspects of the world relevant to the particular application. For instance, the model could be a formula that calculates the total route length from a list of consecutive locations, a statistical tool estimating the likelihood of rain given some meteorological input data, a mapping from real-time data regarding a car's speed to the level of acceleration necessary to

A.E. Eiben, J.E. Smith, *Introduction to Evolutionary Computing*,
Natural Computing Series, DOI 10.1007/978-3-662-44874-8_1

approach some prespecified target speed, or a complex series of rules that transform a series of keystrokes into an on screen version of the page you are reading now. After processing the input the system provides some outputs – which might be messages on screen, values written to a file, or commands sent to an actuator such as an engine. Depending on the application, there might be one or more inputs of different types, and the computational model might be simple, or very complex. Importantly, knowing the model means that we can compute the output for any input. To provide some concrete examples:

- When designing aircraft wings, the inputs might represent a description of a proposed wing shape. The model might contain equations of complex fluid dynamics to estimate the drag and lift coefficients of any wing shape. These estimates form the output of the system.
- A voice control system for smart homes takes as input the electrical signal produced when a user speaks into a microphone. Suitable outputs might be commands to be sent to the heating system, the TV set, or the lights. Thus in this case the model consists of a mapping from certain patterns in electrical waveforms coming from an audio input onto the outputs that would normally be created by key-presses on a keyboard.
- For a portable music player, the inputs might be a series of gestures and button presses – perhaps choosing a playlist that the user has created. Here the response of the model might involve selecting a series of mp3 files from a database and processing them in some way to provide the desired output for that sequence of gestures. In this case the output would be a fluctuating electrical signal fed to a pair of earphones that in turn produce the sound of the chosen songs.

In essence, the black box view of systems distinguishes three components, the input, the model, and the output. In the following we will describe three problem types, depending on which of these three is unknown.

1.1.1 Optimisation

In an **optimisation** problem the model is known, together with the desired output (or a description of the desired output), and the task is to find the input(s) leading to this output (Fig. 1.1).

For an example, let us consider the travelling salesman problem. This apparently rather abstract problem is popular in computer science, as there are many practical applications which can be reduced to this, such as organising delivery routes, plant layout, production schedules, and timetabling. In the abstract version we are given a set of cities and have to find the shortest tour which visits each city exactly once. For a given instance of this problem, we have a formula (the model) that for each given sequence of cities (the inputs) will compute the length of the tour (the output). The problem is to find an input with a desired output, that is, a sequence of cities with optimal (minimal) length. Note that in this example the desired output is defined implicitly.

That is, rather specifying the exact length, it is required that the tour should be shorter than all others, and we are looking for inputs realising this.

Another example is that of the eight-queens problem. Here we are given a chess board and eight queens that need to be placed on the board in such a way that no two queens can check each other, i.e., they must not share the same row, column, or diagonal. This problem can be captured by a computational system where an input is a certain configuration of all eight queens, the model calculates whether the queens in a given configuration check each other or not, and the output is the number of queens not being checked. As opposed to the travelling salesman problem, here the desired output is specified explicitly: the number of queens not being checked must be eight. An alternative system capturing this problem could have the same set of inputs, the same model, but the output can be a simple binary value, representing “OK” or “not OK”, referring to the configuration as a whole. In this case we are looking for an input that generates “OK” as output. Intuitively, this problem may not feel like real optimisation, because there is no graded measure of goodness. In Sect. 1.3 we will discuss this issue in more detail.

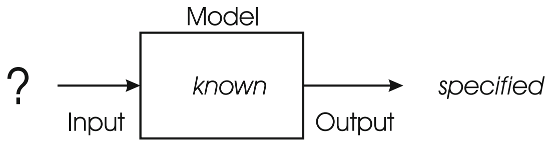

Fig. 1.1. Optimisation problems. These occur frequently in engineering and design. The label on the Output reads “specified”, instead of “known”, because the specific value of the optimum may not be known, only defined implicitly (e.g., the lowest of all possibilities).

1.1.2 Modelling

In a **modelling** or **system identification** problem, corresponding sets of inputs and outputs are known, and a model of the system is sought that delivers the correct output for each known input (Fig. 1.2). In terms of human learning this corresponds to finding a model of the world that matches our previous experience, and can hopefully generalise to as-yet unseen examples.

Let us take the stock exchange as an example, where some economic and societal indices (e.g., the unemployment rate, gold price, euro–dollar exchange rate, etc.) form the input, and the Dow Jones index is seen as output. The task is now to find a formula that links the known inputs to the known outputs, thereby representing a model of this economic system. If one can find a correct model for the known data (from the past), and if we have good reasons to believe that the relationships captured in this model remain true, then we have a prediction tool for the value of the Dow Jones index given new data.

As another example, let us take the task of identifying traffic signs in images – perhaps from video feeds in a smart car. In this case the system is composed of two elements. In a preprocessing stage, image processing routines take the electrical signals produced by the camera, divide these into regions of interest that might be traffic signs, and for each one they produce a set of numerical descriptors of the size, shape, brightness, contrast, etc. These values represent the image in a digital form and we consider the preprocessing component to be given for now. Then in the main system each input is a vector of numbers describing a possible sign, and the corresponding output is a label from a predefined set, e.g., "stop", "give-way", "50", etc. (the traffic sign). The model is then an algorithm which takes images as input and produces labels of traffic signs as output. The task here is to produce a model that responds with the appropriate traffic sign labels in every situation. In practice, the set of all possible situations would be represented by a large collection of images that are all labelled appropriately. Then the modelling problem is reduced to finding a model that gives a correct output for each image in the collection.

Also the voice control system for smart homes described in the beginning of this section includes a modelling problem. The set of all phrases pronounced by the user (inputs) must be correctly mapped onto the set of all control commands in the repertoire of the smart home.

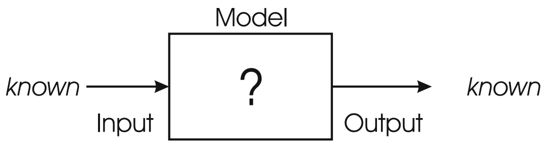

Fig. 1.2. Modelling or system identification problems. These occur frequently in data mining and machine learning

It is important to note that modelling problems can be transformed into optimisation problems. The general trick is to designate the error rate of a model as the quantity to be minimised or its hit rate to be maximised. As an example, let us take the traffic sign identification problem. This can be formulated as a modelling problem: that of finding the correct model m that maps each one of a collection of images onto the appropriate label(s) identifying the traffic signs in that image. The model m that solves the problem is unknown in advance, hence the question mark in Figure 1.2. In order to find a solution we need to start by choosing a technology. For instance, we may wish to have it as a decision tree, an artificial neural network, a piece of Java code, or a MATLAB expression. This choice allows us to specify the required form or syntax of m. Having done that, we can define the set of all possible solutions M for our chosen technology, being all correct expressions in the given syntax, e.g., all decision trees with the appropriate variables or all possible artificial neural networks with a given topology. Now we can define a related optimisation problem. The set of inputs is M and the output for a given $m \in M$ is an integer saying how many images were correctly labelled by m. It is clear that a solution of this optimisation problem with the maximum number of correctly labelled images is a solution to the original modelling problem.

1.1.3 Simulation

In a **simulation** problem we know the system model and some inputs, and need to compute the outputs corresponding to these inputs (Fig. 1.3). As an example, think of an electronic circuit, say, a filter cutting out low frequencies in a signal. Our model is a complex system of formulas (equations and inequalities) describing the working of the circuit. For any given input signal this model can compute the output signal. Using this model (for instance, to compare two circuit designs) is much cheaper than building the circuit and measuring its properties in the physical world. Another example is that of a weather forecast system. In this case, the inputs are the meteorological data regarding, temperature, wind, humidity, rainfall, etc., and the outputs are actually the same: temperature, wind, humidity, rainfall, etc., but at a different time. The model here is a temporal one to predict meteorological data.

Simulation problems occur in many contexts, and using simulators offers various advantages in different applications. For instance, simulation can be more economical than studying the real-world effects, e.g., for the electronic circuit designers. The real-world alternative may not be feasible at all, for instance, performing what-if analyses of various tax systems *in vivo* is practically impossible. And simulation can be the tool that allows us to look into the future, as in weather forecast systems.

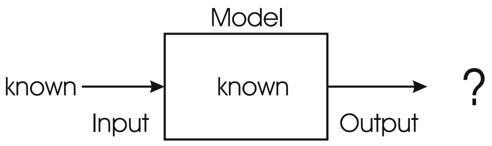

Fig. 1.3. Simulation problems. These occur frequently in design and in socio-economical contexts

1.2 Search Problems

A deeply rooted assumption behind the black box view of systems is that a computational model is directional: it computes from the inputs towards the outputs and it cannot be simply inverted. This implies that solving a simulation problem is different from solving an optimisation or a modelling problem. To solve a simulation problem, we only need to apply the model to some inputs and simply wait for the outcome.[1] However, solving an optimisation or a modelling problem requires the identification of a particular object in a space of possibilities. This space can be, and usually is, enormous. This leads us to the notion that the process of problem solving can be viewed as a search through a potentially huge set of possibilities to find the desired solution. Consequently, the problems that are to be solved this way can be seen as search problems. In terms of the classification of problems discussed in Section 1.1, optimisation and modelling problems can be naturally perceived as search problems, while this does not hold for simulation problems.

This view naturally leads to the concept of a **search space**, being the collection of all objects of interest including the solution we are seeking. Depending on the task at hand, the search space consists of all possible inputs to a model (optimisation problems), or all possible computational models that describe the phenomenon we study (modelling problems). Such search spaces can indeed be very large; for instance, the number of different tours through n cities is $(n-1)!$, and the number of decision trees with real-valued parameters is infinite. The specification of the search space is the first step in defining a search problem. The second step is the definition of a solution. For optimisation problems such a definition can be explicit, e.g., a board configuration where the number of checked queens is zero, or implicit, e.g., a tour that is the shortest of all tours. For modelling problems, a solution is defined by the property that it produces the correct output for every input. In practice, however, this is often relaxed, only requiring that the number of inputs for which the output is correct be maximal. Note that this approach transforms the modelling problem into an optimisation one, as illustrated in Section 1.1.2.

This notion of problem solving as search gives us an immediate benefit: we can draw a distinction between (search) problems – which define search spaces – and problem solvers – which are methods that tell us how to move through search spaces.

1.3 Optimisation Versus Constraint Satisfaction

The classification scheme discussed in this section is based on distinguishing between objective functions to be optimised and constraints to be satisfied. In general, we can consider an **objective function** to be some way of assigning a value to a possible solution that reflects its quality on a scale, whereas a **constraint** represents a binary evaluation telling us whether a given requirement holds or not. In the previous sections several objective functions were mentioned, including:

[1] The main challenge here is very often to build the simulator, which, in fact, amounts to solving a modelling problem.

(1) the number of unchecked queens on a chess board (to be maximised);
(2) the length of a tour visiting each city in a given set exactly once (to be minimised);
(3) the number of images in a collection that are labelled correctly by a given model m (to be maximised).

These examples illustrate that solutions to a problem can be identified in terms of optimality with respect to some objective function. Additionally, solutions can be subject to constraints phrased as criteria that must be satisfied. For instance:

(4) Find a configuration of eight queens on a chess board such that no two queens check each other.
(5) Find a tour with minimal length for a travelling salesman such that city X is visited after city Y.

There are a number of observations to be made about these examples. Example 2 refers to a problem whose solution is defined purely in terms of optimisation. On the other hand, example 4 illustrates the case where a solution is defined solely in terms of a constraint: a given configuration is either good or not. Note that this overall constraint regarding a whole configuration is actually composed from more elementary constraints concerning pairs of queens. A complete configuration is OK if all pairs of queens are OK. Example 5 is a mixture of these two basic types since it has an objective function (tour length) and a constraint (visit X after Y). Based on these observations we can set up another system for classifying problems, depending on the presence or absence of an objective function and constraints in the problem definition. The resulting four categories are shown in Table 1.1.

	Objective function	
Constraints	Yes	No
Yes	Constrained optimisation problem	Constraint satisfaction problem
No	Free optimisation problem	No problem

Table 1.1. Problem types distinguished by the presence or absence of an objective function and constraints

In these terms, the travelling salesman problem (item 2 above) is a **free optimisation problem (FOP)**, the eight-queens problem (item 4 above) is a **constraint satisfaction problem (CSP)**, and the problem shown in item 5 is a **constrained optimisation problem (COP)**. Comparing items 1 and 4 we can see that constraint satisfaction problems can be transformed into optimisation problems. The basic trick is the same as in transforming modelling problems into optimisation problems: rather than requiring perfection, we just count the number of satisfied constraints (e.g., non-checking pairs of queens) and introduce this as an objective function to be maximised. Obviously, an object (e.g., a board configuration) is a solution of the original constraint satisfaction problem if and only if it is a solution of this associated optimisation problem.

To underpin further interesting insights about problems, let us have a closer look at the eight-queens problem. Its original formulation is in natural language:

> Place eight queens on a chess board in such a way that no two queens check each other.

This problem definition is informal in the sense that it lacks any reference to the formal constructs we have introduced here, such as inputs/outputs, a search space, etc. In order to develop an algorithm for this problem, it needs to be formalised. As it happens, it can be formalised in different ways, and these lead to different types of formal problems describing it. The easiest way to illustrate a number of options is to take the search perspective.

FOP If we define search space S to be the set of all board configurations with eight queens, we can capture the original problem as a free optimisation problem with an objective function f that reports the number of free queens for a given configuration, and define a solution as a configuration $s \in S$ with $f(s) = 8$.

CSP Alternatively, we can formalise it as a constraint satisfaction problem with the same search space S and define a constraint ϕ such that $\phi(s) = true$ if and only if no two queens check each other for the configuration s.

COP Yet another formalisation is obtained if we take a different search space. This can be motivated by the observation that in any solution of the eight-queens problem the number of queens in each column must be exactly one. Obviously, the same holds for rows. So we could distinguish vertical constraints (for columns), horizontal constraints (for rows), and diagonal constraints, and decide to restrict ourselves to board configurations that satisfy the vertical and horizontal constraints already. This is a workable approach, since it is rather easy to find configurations with one queen in each column and in each row. These configurations are a subset of the original search space – let us call this S'. Formally, we can then define a constrained optimisation problem over S with a modified constraint ψ' such that $\psi'(s) = true$ if and only if all vertical and horizontal constraints are satisfied in s (i.e. $\phi'(s) = true$ if and only if s is in S') and a new function g that reports the number of pairs of queens in s that violate the diagonal constraints. It is easy to see that a board configuration is a solution of the eight-queens problem if and only if it is a solution of this constrained optimisation problem with $g(s) = 0$ and $\phi'(s) = true$.

These examples illustrate that the nature of a problem is less obvious than it may seem. In fact, it all depends on how we choose to formalise it. Which formalisation is to be preferred is a subject for discussion. It can be argued that some formalisations are more natural, or fit the problem better, than others. For instance, one may prefer to see the eight-queens problem as a constraint satisfaction problem by nature and consider all other formalisations as secondary transformations. Likewise, one can consider the traffic sign recognition problem as a modelling problem in the first place and transform it to an optimisation problem for practical purposes. Algorithmic considerations can also be a major influence here. If one has an algorithm that can solve free optimisation problems well, but cannot cope with constraints, then it is very sensible to formalise problems as free optimisation.

1.4 The Famous NP Problems

Up to this point we have discussed a number of different ways of categorising problems, and have deliberately stayed away from discussions about problem-solvers. Consequently, it is possible to classify a problem according to one of those schemes by only looking at the problem. In this section we discuss a classification scheme where this is not possible because the problem categories are defined through the properties of problem-solving algorithms. The motivation behind this approach is the intention to talk about problems in terms of their difficulty, for instance, being hard or easy to solve. Roughly speaking, the basic idea is to call a problem easy if there exists a fast solver for it, and hard otherwise. This notion of problem hardness leads to the study of computational complexity.

Before we proceed we need to make a further distinction among optimisation problems, depending on the type of objects in the corresponding search space. If the search space S is defined by continuous variables (i.e., real numbers), then we have a **numerical optimisation problem**. If S is defined by discrete variables (e.g., Booleans or integers), then we have a **combinatorial optimisation problem**. The various notions of problem hardness discussed further on are defined for combinatorial optimisation problems. Notice that discrete search spaces are always finite or, in the worst case, countably infinite.

We do not attempt to provide a complete overview of computational complexity as this is well covered in many books, such as [180, 330, 331, 318]. Rather, we provide a brief outline of some important concepts, their implications for problem-solving, and also of some very common misconceptions. Furthermore, we do not treat the subject with mathematical rigour as it would not be appropriate for this book. Thus, we do not give precise definitions of essential concepts, like algorithm, problem size, or run-time, but use such terms in an intuitive manner, explaining their meaning by examples if necessary.

The first key notion in computational complexity is that of **problem size**, which is grounded in the dimensionality of the problem at hand (i.e., the number of variables) and the number of different values for the problem variables. For the examples discussed before, the number of cities to visit, or the number of queens to place on the board could be sensible measures to indicate problem size. The second notion concerns algorithms, rather than problems. The **running-time** of an algorithm is the number of elementary steps, or operations, it takes to terminate. The general, although not always correct, intuition behind computational complexity is that larger problems need more time to solve. The best-known definitions of problem hardness relate the size of a problem to the (worst-case) running-time of an algorithm to solve it. This relationship is expressed by a formula that specifies an upper-bound for the worst-case running-time as a function of the problem size. To put it simply, this formula can be polynomial (considered to indicate relatively short running-times) or superpolynomial, e.g., exponential (indicating long running-times). The final notion is that of **problem reduction**, which is the idea that we can transform one problem into another via a suitable mapping. Note that the transformation might not be reversible. Although this idea of transforming or reducing problems is slightly complex, it is not entirely unfamiliar since we saw in the previous section that a given problem in the real world can often by formalised in different, but equivalent ways. The frequently used notions regarding problem hardness can now be phrased as follows.

A problem is said to belong to the **class P** if there exists an algorithm that can solve it in polynomial time. That is, if there exists an algorithm for it whose worst-case running-time for problem size n is less than $F(n)$ for some polynomial formula F. In common parlance, the set P contains the problems that can be easily solved, e.g., the Minimum Spanning Tree problem.

A problem is said to belong to the **class NP** if it can be solved by some algorithm (with no claims about its run-time) and any solution can be verified within polynomial time by some other algorithm.[2] Note that it follows that P is a subset of NP, since a polynomial solver can also be used to verify solutions in polynomial time. An example of an NP-problem is the subset-sum problem: given a set of integers, is there some set of one or more elements of that set which sum to zero? Clearly, giving a negative answer to this problem for a given set of numbers would require examining all possible subsets. Unfortunately, the number of the possible subsets is more than polynomial in the size of the set. However verifying that a solution is valid merely involves summing the contents of the subset discovered.

A problem is said to belong to the **class NP-complete** if it belongs to the class NP and any other problem in NP can be reduced to this problem by an algorithm which runs in polynomial time. In practice these represent difficult problems which crop up all the time. Several large lists of well-known examples of NP-complete problems can readily be found on the internet –

[2] For the sake of correctness, here we commit the most blatant oversimplification. We 'define' NP without any reference to non-deterministic Turing Machines, or restricting the notion to decision problems.

we will not attempt to summarise other than to say that the vast majority of interesting problems in computer science turn out to be NP-complete.

Finally a problem is said to belong to the **class NP-hard** if it is at least as hard as any problem in NP-complete (so all problems in NP-complete can be reduced to one in NP-hard), but where the solutions cannot necessarily be verified within polynomial time. One such example is the Halting Problem.

The existence of problems where a solution cannot be verified in polynomial time proves that the class P is not the same as the class NP-hard. What is unknown is whether the two classes P and NP are in fact the same. If this were to be the case then the implications would be enormous for computer science and mathematics as it would be known that fast algorithms must exist for problems which were previously thought to be difficult. Thus whether $P = NP$ is one of the grand challenges of complexity theory, and there is a million-dollar reward offered for any proof that $P = NP$ or $P \neq NP$. Notice, that while the latter is the subject of much complex mathematics, the former could simply be proved by the creation of a fast algorithm for any of the NP-complete problems, for instance, an algorithm for the travelling salesman problem whose worst-case running-time scaled polynomially with the number of cities. Figure 1.4 shows the classes of problem hardness depending on the equality of P and NP. If $P = NP$ then the sets $P = NP = NP$-complete but they are still a subset of NP-hard.

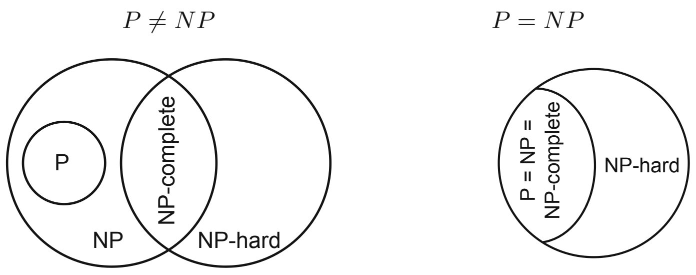

Fig. 1.4. Classes of problem hardness depending on the equality of P and NP

While this sounds rather theoretical, it has some very important implications for problem-solving. If a problem is NP-complete, then although we might be able to solve particular instances within polynomial time, we cannot say that we will be able to do so for all possible instances. Thus if we wish to apply problem-solving methods to those problems we must currently either accept that we can probably only solve very small (or otherwise easy) instances, or give up the idea of providing exact solutions and rely on approximation or metaheuristics to create good enough solutions. This is in contrast to problems which are known to be in P. Although the number of possible solutions for these problems may scale exponentially, algorithms exist which find solutions and whose running-times scale polynomially with the size of the instance.

To summarise this section, there are a huge number of practical problems which, on examination, turn out to be a variant of an abstract problem that is known to be in the class NP-complete. Although some instances of such a problem might be easy, most computer scientists believe that no polynomial-time algorithm exists for such problems, and certainly one has not yet been discovered. Therefore, if we wish to be able to create acceptable solutions for any instance of such a problem, we must turn to the use of approximation and metaheuristics and abandon the idea of definitely finding a solution which is provably the best for the instance.

For exercises and recommended reading for this chapter, please visit `www.evolutionarycomputation.org`.