12

Constraint Handling

12.1 Aims of this Chapter

In this chapter we consider the issue of constraint handling by evolutionary algorithms. This issue has great practical relevance because many practical problems are constrained. It is also a theoretically challenging subject since a great deal of intractable problems (NP-hard, NP-complete, etc.) are constrained. The presence of constraints has the effect that not all possible combinations of variable values represent valid solutions to the problem at hand. Unfortunately, constraint handling is not straightforward in an EA, because the variation operators (mutation and recombination) are typically "blind" to constraints. That is, there is no guarantee that even if the parents satisfy some constraints, the offspring will satisfy them as well. In this chapter we elaborate on the notion of constrained problems and distinguish two different types: constrained optimisation problems and constraint satisfaction problems. (This elaboration requires clarifying some basic notions, leading to definitions that implicitely have been used in earlier chapters.) Based on this classification of constrained problems, we discuss what constraint handling means from an EA perspective, and review the most commonly applied EA techniques to treat constraints. Analysing these techniques, we identify a number of common features and arrive at the conclusion that the presence of constraints is not harmful, but rather helpful in that it provides extra information that EAs can utilise.

12.2 Constrained Problems

To facilitate a clear discussion, let us have a look at the very notion of a constrained problem. For instance, consider the travelling salesman problem for n cities $C = \{c_1, \dots, c_n\}$ and the distance function d. Is this a constrained problem? The answer should be independent from the algorithm we are to apply to solve this problem, but at the first glance this is not the case. To illustrate this, assume that we chose for an iterative search algorithm that either:

1. Operates in the search space $S_1 = C^n$ and seeks a solution $s \in S_1$ that minimises the tour length $f(\bar{s}) = \sum_{i=1}^{n} d(s_i, s_{i+1})$ with s_{n+1} defined as s_1.
2. Operates in the search space $S_2 = \{permutations\ of\ c_1, \ldots, c_n\}$ and seeks a solution $s \in S_2$ that minimises the tour length $f(\bar{s})$ defined as above.

Note that in the first case we need a constraint requiring uniqueness of each city in a solution $\bar{s} \in S_1$, while in the second case we do not as every $\bar{s} \in S_2$ satisfies this condition by the definition of S_2. Thus, the notion of a constrained problem seems to depend on what we take as search space.

To clarify this matter we introduce some terminology. In the further discussion we assume that a problem is given in terms of its variables $v_i, \ldots, v_n$, each having its own domain $D_1, \ldots, D_n$, where the domains can be discrete or continuous.[1] We will call a Cartesian product of sets $S = D_1 \times \ldots \times D_n$ a **free search space**. The most important property of free search spaces is that testing the membership relation of such a space can be done independently on each coordinate, taking the conjunction of the results. Note that the usual mutation and recombination operators for bit, integer, and floating-point representation keep the offspring in the search space. In our perception requiring that a solution be within a free search space has nothing to do with constraints, instead it is merely the specification of the domains of the variables. In the further discussion we distinguish problems (over a free search space) by the presence or absence of

1. An objective function
2. Constraints

The resulting four categories are shown in Table 12.1. Next we will discuss these problem types more precisely.

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

Table 12.1. Problem types

[1] However, we might require that if D_i is continuous, then it is convex.

12.2.1 Free Optimisation Problems

A **free optimisation problem** (FOP) is a pair $\langle S, f \rangle$, where S is a free search space and f is a real-valued objective function on S, which has to be optimised (minimised or maximised). A **solution of a free optimisation problem** is an $\bar{s} \in S$ with an optimal f value.

Examples of FOP's abound in any literature related to optimisation, and some common examples using a variety of domains (binary, discrete, continuous) are given in Appendix B. FOPs do not pose specific challenges to EAs from our present perspective since EAs have a "basic instinct" to optimise. This is, of course, not to say that solving any FOP is easy with an EA, but the absence of constraints implies free search in the sense that the common variation operators do not generate values outside of the domains of the variables.

12.2.2 Constraint Satisfaction Problems

A **constraint satisfaction problem** (CSP) is a pair $\langle S, \phi \rangle$, where S is a free search space and ϕ is a formula (Boolean function on S). A **solution of a constraint satisfaction problem** is an $\bar{s} \in S$ with $\phi(\bar{s}) = true$. The formula ϕ is often called the **feasibility condition**, and it is typically a composed entity, derived from more elementary constraints. A **constraint** is a restriction on the possible value combinations of certain variables.

A well known CSP example is the graph three-colouring problem, where the nodes of a given graph $G = (N, E)$, $E \subseteq N \times N$ have to be coloured by three colours in such a way that no neighbouring nodes, i.e., nodes connected by an edge, have the same colour. This problem can be formalised by means of a CSP $\langle S, \phi \rangle$ as follows:

- $S = D^n$ with $D = \{1, 2, 3\}$ being the same domain for each of the $n = |N|$ variables.
- ϕ is composed of constraints that belong to edges. That is, for each edge $e \in E$, the corresponding constraint c_e is defined by $c_e(\langle s_1, \ldots, s_n \rangle) = true$ if an only if $e = (k, l)$ and $s_k \neq s_l$. Then the feasibility condition is the conjunction of all constraints $\phi(\bar{s}) = \bigwedge_{e \in E} c_e(\bar{s})$.

The main EA challenge in treating CSPs is the absence of an objective function that could be naturally used to define fitness. The feasibility condition imposes only a very simple landscape on candidate solutions having a large flat plateau at zero level (ϕ is false) with some singular peaks (ϕ is true). This is an extreme case of a needle in a haystack problem. The basis of all approaches to design an EA for a CSP is to transform constraints into optimisation objectives and rely on the optimisation power of EAs to achieve these objectives, and thereby to satisfy the constraints. If all constraints of a CSP are handled this way then such a transformation amounts to turning the CSP into an FOP. It should be noted, however, that this is not the only option.

Recall the eight-queens example from Section 2.4.1, which is clearly a CSP (see exercise at the end of this chapter). The evolutionary approach we have outlined uses permutations as chromosomes. These chromosomes represent board configurations where horizontal constraint violations (two queens on the same row) and vertical constraint violations (two queens on the same column) do not occur. Thus, if we can enforce that these constraints are respected at initialisation and remain maintained during the evolutionary search process then the only thing that remains to be achieved is the satisfaction of diagonal constraints. These have been treated by defining the number of such violations as an objective function to be minimised. With our present terminology we can describe this solution as transforming the given CSP into a problem that still has some explicitly present constraints to be satisfied (the horizontal and vertical constraints) together with some objectives to be achieved (number of diagonal constraint violations minimised).

12.2.3 Constrained Optimisation Problems

A **constrained optimisation problem** (COP) is combination of an FOP and a CSP. It is a triple $\langle S, f, \phi\rangle$, where S is a free search space, f is a real-valued objective function on S, and ϕ is a formula (Boolean function on S). A **solution of a constrained optimisation problem** is an $\bar{s} \in S$ with $\phi(\bar{s}) = \mathit{true}$ and an optimal f value.

To illustrate COPs we use the Travelling Salesman Problem for n cities $C = \{c_1, \dots, c_n\}$ which can be formalised by $\langle S, f, \phi\rangle$.

- The free search space is $S = C^n$.
- The objective function to be minimised is $f(\bar{s}) = \sum_{i=1}^{n} d(s_i, s_{i+1})$, with s_{n+1} defined as s_1.
- The feasibility condition $\phi = \phi_c \wedge \phi_u$ is the conjunction of the following two conditions:
  - $\phi_c(\bar{s}) = \mathit{true}$ if and only if for each $c \in C$ there is an $i \in \{1, \dots, n\}$ such that $c = s_i$ (completeness condition).
  - $\phi_u(\bar{s}) = \mathit{true}$ if and only if for each $k, l \in \{1, \dots, n\}$ $s_k \neq s_l$ (unicity condition).

With the aid of this framework we can now unambiguously classify the TSP as a constrained problem, a COP, since its definition involves a feasibility condition restricting the free search space.

Treating COPs by EAs poses very similar questions to those regarding CSPs, namely, the constraints must be handled. Transforming (some of) them into optimisation objectives is again a most straightforward option, although not as essential as for CSP, since a COP does have a natural fitness definition by the given f. If one is to leave constraints as constraints (i.e., not transforming them into optimisation objectives), then these constraints have to be treated explicitly in order to make sure that the candidate solutions satisfy them.

12.3 Two Main Types of Constraint Handling

Various technical options for constraint handling are discussed in Sect. 12.4. Without going into details yet, here we distinguish two conceptually different possibilities. If all constraints in a CSP or COP are replaced by optimisation objectives, then the given problem is transformed into an FOP. Formally we have $\langle S, \bullet, \phi \rangle \rightarrow \langle S, f, \bullet \rangle$, respectively $\langle S, f, \phi \rangle \rightarrow \langle S, g, \bullet \rangle$, where the $\bullet$ is a place-holder for the absent component of the given problem type. In these cases "constraint handling" means "constraint transformation" before running the EA. After the transformation is done, the EA can perform free search without paying any further attention to constraints. It is the algorithm designer's responsibility (and one of the main design guidelines) to ensure that solutions to the transformed FOP represent solutions to the original CSP or COP.

If not all constraints are replaced by optimisation objectives, then the problem that should be solved by the EA still has constraints – it is a COP. This can be the case if:

1. Some constraints of a CSP are not incorporated into the objective function, but left as constraints making up a new, weakened feasibility condition: $\langle S, \bullet, \phi \rangle \rightarrow \langle S, f, \psi \rangle$, which was the case in our eight-queens example.
2. The original COP is not transformed, so we are to design an EA for the given $\langle S, f, \phi \rangle$.
3. Some but not all constraints in a given COP are transformed into optimisation objectives, while the others are left as constraints making up a new, weakened feasibility condition: $\langle S, f, \phi \rangle \rightarrow \langle S, g, \psi \rangle$.

In cases 1 and 3 we have constraint handling in the sense of constraint transformation and in all of these cases we are facing "constraint handling" as "constraint enforcement" during the run of the EA because the (transformed) problem still has a feasibility condition.

Based on these observations we can eliminate an often-occurring ambiguity about constraint handling and identify the following two forms:

- In the case of **indirect constraint handling** constraints are transformed into optimisation objectives. After the transformation, they effectively practically disappear, and all we need to care about is optimising the resulting objective function. This type of constraint handling is done *before* the EA run.

- As an alternative to this option we distinguish **direct constraint handling**, meaning that the problem offered to the EA to solve has constraints (is a COP) that are enforced explicitly *during* the EA run.

It should be clear from the previous discussion that these options are not exclusive: for a given constrained problem (CSP or COP) some constraints might be treated directly and some others indirectly.

It is also important to note that even when all constraints are treated indirectly, so that we apply an EA for an FOP, this does not mean that the EA is necessarily ignoring the constraints. In theory one could fully rely on the general optimisation power of EAs and try to solve the given FOP without taking note of how f is obtained. However, it is also possible that one does take the specific origin of f into account, i.e., the fact that it is constructed from constraints. In this case one can try to make use of specific constraint-based information within the EA by, for instance, special mutation or crossover operators that explicitly aim at satisfying constraints by using some heuristics.

Finally, let us reiterate that indirect constraint handling is always part of the preparation of the problem before offering it to an EA to solve. However, direct constraint handling is an issue within the EA constituting methods that enforce satisfaction of the constraints.

12.4 Ways to Handle Constraints in EAs

In the discussion so far, we have not considered the nature of the domains of the variables. In this respect there are two extremes: they are all discrete or all continuous. Continuous CSPs are almost nonexistent, so by default a CSP is discrete. For COPs this is not the case as we have discrete COPs (**combinatorial optimisation** problems) and continuous COPs as well. Much of the evolutionary literature on constraint handling is restricted to either of these cases, but the ways for handling constraints are practically identical – at least at the conceptual level. In the following treatment of constraint handling methods we will be general, considering discrete and continuous cases together. The commonly shown list of available options is the following:

1. The use of penalty functions that reduce the fitness of infeasible solutions, preferably so that the fitness is reduced in proportion to the number of constraints violated, or to the distance from the feasible region.
2. The use of mechanisms that take infeasible solutions and "repair" them, i.e., return a feasible solution, which one would hope is close to the infeasible one.
3. The use of a specific alphabet for the problem representation, plus suitable initialisation, recombination, and mutation operators such that the feasibility of a solution is always ensured, and there is an unambiguous mapping from genotype to phenotype.

4. The use of "decoder" functions to manage the mapping from genotype to phenotype so that solutions (i.e., phenotypes) are guaranteed to be feasible. This approach differs from the previous one in that typically a number of potentially radically different genotypes may be mapped onto the same phenotype. It has the strong advantage of permitting the use of more standard variation operators.

Notice that the last option amounts to manipulating the search space S. In the discussion so far, within the $\langle S, f, \phi \rangle$ framework, we have only considered problem transformations regarding the f and the ϕ component. Using decoder functions means that a constrained problem (CSP $\langle S, \bullet, \phi \rangle$ or COP $\langle S, f, \phi \rangle$) is transformed into one with a different search space S'. Taking a COP–FOP transformation as an example, this is formally $\langle S, f, \phi \rangle \rightarrow \langle S', g, \bullet \rangle$, where the EA to solve $\langle S, f, \phi \rangle$ actually works on $\langle S', g, \bullet \rangle$. Here, constraint handling is neither transformation nor enforcement, but is carried out through the mapping (decoder) between S and S'. In a way one could say that in this case we do not handle constraints, instead we simply avoid the question by ensuring that a genotype in S' always mapped onto a feasible phenotype in S. For this reason we call this **mapping constraint handling**.

Within our framework we can arrange the above options as follows:

1. Indirect constraint handling (transformation) coincides with **penalising** constraint violations.
2. Direct constraint handling (enforcement) can be carried out in two different ways:
   a) Allowing the generation of candidate solutions that violate constraints: **repairing** infeasible candidates.
   b) Not allowing the generation of candidate solutions that violate constraints: **preserving** feasibility by suitable operators (and initialisation).
3. Mapping constraint handling is the same as **decoding**, i.e., transforming the search space into another one.

In general the presence of constraints will divide the space of potential solutions into two or more disjoint regions, the **feasible region** (or regions) $F \subset S$, containing those candidate solutions that satisfy the given feasibility condition, and the **infeasible region** containing those that do not. In practice, it is common to utilise as much domain-specific knowledge as possible, in order to reduce the amount of time spent generating infeasible solutions. As is pointed out in [277], the global optimum of a COP with continuous variables often lies on, or very near to, the boundary between the feasible and infeasible regions, and promising results are reported using algorithms that specifically search along that boundary. However, we concentrate here on the more general case, since the domain knowledge required to specify such operators may not be present.

In the following sections we briefly describe the above approaches, focusing on the facets that have implications for the applications of EAs in general.

For a fuller review of work in this area, the reader is referred to [112, 126, 275, 277, 352]. Furthermore, [85, 273, 276, 327] are especially recommended because they contain descriptions of problem instance generators for binary CSPs ([85]), continuous COPs ([273, 276]), or a large collection of continuous COP test landscapes [327]), together with detailed experimental results. One general point worth noting is that in [277] it was reported that for problems in the continuous domain, use of a real-valued rather than binary representation consistently gave better results.

12.4.1 Penalty Functions

Assuming a minimisation problem, the use of penalty functions constitutes a mapping from the objective function such that $f'(\bar{x}) = f(\bar{x}) + P(d(\bar{x}, F))$ where F is the feasible region as before, $d(\bar{x}, F))$ is a distance metric of the infeasible point to the feasible region (this might be simply a count of the number of constraints violated) and the penalty function P is monotonically increasing nonnegatively such that $P(0) = 0$.

It should be noted at the outset that this assumes that it is possible to evaluate an infeasible point; although for many problems this may be so (for example, the knapsack problem), for many others this is not the case. This discussion is also confined to *exterior* penalty functions, where the penalty is only applied to infeasible solutions, rather than *interior* penalty functions, where a penalty is applied to feasible solutions in accordance to their distance from the constraint boundary in order to encourage exploration of this region.

The conceptual simplicity of penalty function methods means that they are widely used, and they are especially suited to problems with disjoint feasible regions, or where the global optimum lies on (or near) the constraint boundary. However, their successful use depends on a balance between exploration of the infeasible region and not wasting time, which places a lot of emphasis on the form of the penalty function and the distance metric.

If the penalty function is too severe, then infeasible points near the constraint boundary will be discarded, which may delay, or even prevent, exploration of this region. Equally, if the penalty function is not sufficient in magnitude, then solutions in infeasible regions may dominate those in feasible regions, leading to the algorithm spending too much time in the infeasible regions and possibly stagnating there. In general, for a system with m constraints, the form of the penalty function is a weighted sum

$$P(d(\bar{x}, F)) = \sum_{i=1}^{m} w_i \cdot d_i^{\kappa}(\bar{x})$$

where κ is a user defined constant, often taking the value 1 or 2. The function $d_i(\bar{x})$ is a distance metric from the point $\bar{x}$ to the boundary for that constraint i, whose form depends on the nature of the constraint, but may be a simple binary value according to whether the constraint is satisfied, or a metric based on "cost of repair".

Many different approaches have been proposed, and a good review is given in in [346], where penalty functions are classified as *constant, static, dynamic,* or *adaptive.* This classification closely matches the options discussed in the example given in Sect. 8.3.2.

Static Penalty Functions

Three methods have commonly been used with static penalty functions, namely *extinctive* penalties (where all of the w_i are set so high as to prevent the use of infeasible solutions), binary penalties (where the value d_i is 1 if the constraint is violated, and zero otherwise), and distance-based penalties.

It has been reported that of these three the latter give the best results [172], and the literature contains many examples of this approach. This approach relies on the ability to specify a distance metric that accurately reflects the difficulty of repairing the solution, which is obviously problem dependent, and may also vary from constraint to constraint. The usual approach is to take the square of the Euclidean distance (i.e., set $\kappa = 2$) .

However, the main problem in using static penalty functions remains the setting of the values of w_i. In some situations it may be possible to find these by experimentation, using repeated runs and incorporating domain specific knowledge, but this is a time-consuming process that is not always possible.

Dynamic Penalty Functions

An alternative approach to setting fixed values of w_i by hand is to use dynamic values, which vary as a function of time. A typical approach is that of [217], in which the static values w_i were replaced with a simple function of the form $s_i(t) = (w_i t)^\alpha$, where it was found that for best performance $\alpha \in \{1, 2\}$. Although possibly less brittle as a result of not using fixed (possibly inappropriate) values for the w_i, this approach still requires the user to decide on the initial values.

An alternative approach, which can be seen as the logical extension of this approach, is the behavioural memory algorithm of [338, 387]. In this approach a population is evolved in a number of stages – the same number as there are constraints. In each stage i, the fitness function used to evaluate the population is a combination of the distance function for constraint i with a death penalty for all solutions violating constraints $j < i$. In the final stage all constraints are active, and the objective function is used as the fitness function. It should be noted that different results may be obtained, depending on the order in which the constraints are dealt with.

Adaptive Penalty Functions

Adaptive penalty functions represent an attempt to remove the danger of poor performance resulting from an inappropriate choice of values for the penalty weights w_i. An early approach described in [42, 188] was discussed in Sect. 8.3.2.

A second approach is that of [347, 393], in which adaptive scaling (based on population statistics of the best feasible and infeasible raw fitnesses yet discovered) is coupled with the distance metrics for each constraint based on the notion of "near feasible thresholds". These latter are scaling factors for each distance metric, which can vary with time.

The Stepwise Adaptation of Weights (SAW) algorithm of [130, 131, 132] can be seen as a population-level adaptation of the search space. In this method the weights w_i are adapted according to whether the best individual in the current population violates constraint i. In contrast to the mechanism of Bean and Hadj-Alouane above ([42, 188]), the updating function is much simpler. In this case a fixed penalty increment Δw is added to the penalty values for each of the constraints violated in the best individual of the generation at which the updating takes place. This algorithm was able to adapt weight values that were independent of the GA operators and the initial weight values, suggesting that this is a robust technique.

In the following two sections we discuss direct constraint handling methods. To this end, let us reiterate from Section 12.3 that indirect constraint handling, thus defining penalties, is always part of the preparation of the problem before offering it to an EA to solve. This can be part of solving a CSP or a COP. In contrast to this, direct constraint handling is an issue within the EA, constituting methods that enforce satisfaction of the constraints in the transformed problem. Therefore, the scope of the next two sections is handling COPs with EAs, where the COP might be the result of an earlier problem transformation CSP$\rightarrow$COP or COP$\rightarrow$COP.

12.4.2 Repair Functions

The use of repair algorithms for solving COPs with GAs can be seen as a special case of adding local search to the GA, where the aim of the local search in this case is to reduce (or remove) the constraint violation, rather than (as is usually the case) to simply improve the value of the fitness function. The use of local search has been intensively researched, with attention focusing on the benefits of so-called Baldwinian versus Lamarkian learning (Sect. 10.3.1). In either case, the repair algorithm works by taking an infeasible point and generating a feasible solution based on it. In the Baldwinian case, the fitness of the repaired solution is allocated to the infeasible point, which is kept, whereas with Lamarkian learning, the infeasible solution is overwritten with the new feasible point.

Although this debate has not been settled within unconstrained learning, many COP algorithms reach a compromise by introducing some stochasticity, for example Michalewicz's GENOCOP algorithm uses the repaired solution around 15% of the time [274].

To illustrate the use of repair functions we will first consider the binary knapsack problem described in Sect. 2.4.2. Although specifying a repair algorithm at first seems simple – simply change gene values from 1 to 0 until the weight constraint is satisfied – it raises some interesting questions. One of these is the replacement question just discussed; the second is whether the genes should be selected for altering in a predetermined order, or at random. In [271] it was reported that using a greedy deterministic repair algorithm gave the best results, and certainly the use of a nondeterministic repair algorithm will add noise to the evaluation of every individual, since the same potential solution may yield different fitnesses on separate evaluations. However, it has been found by some authors [362] that the addition of noise can assist the GA in avoiding premature convergence. In practice it is likely that the best method is not only dependent on the problem instance, but on the size of the population and the selection pressure.

The binary case above is relatively simple, however, in general defining a repair function may be as complex as solving the problem itself. One algorithm that eases this problem (and incidentally uses stochastic repair), is Michalewicz's GENOCOP III algorithm for optimisation in continuous domains [274].

This works by maintaining two populations, one P_s of so-called "search points" and one P_r of "reference points", with all of the latter being feasible. Points in P_r and feasible points from P_s are evaluated directly. When an infeasible point is generated in P_s it is "repaired" by picking a point in P_r and drawing a line segment from it to the infeasible point. This is then sampled until a "repaired" feasible point is found. If the new point is superior to that used from P_r, the new point replaces it. With a small probability (which represents the balance between Lamarkian and Baldwinian search) the new point replaces the infeasible point in P_s. It is worth noting that although two different methods are available for selecting the reference point used in the repair, both are stochastic, so the evaluation is necessarily noisy.

12.4.3 Restricting Search to the Feasible Region

In many COP applications it may be possible to construct a representation and operators so that the search is confined to the feasible region of the search space. In constructing such an algorithm, care must be taken in order to ensure that all of the feasible region is capable of being represented. It is equally desirable that any feasible solution can be reached from any other by (possibly repeated) applications of the mutation operator. The classic example of this is permutation problems. In Sect. 2.4.1 we showed an illustration for the eight-queens Problem and in Sects., 3.4.4 and 3.5.4 we described a number of variation operators that are guaranteed to deliver feasible offspring from feasible parents.

It should be noted that this approach to solving COP, although attractive, is not suitable for all types of constraints. In many cases it is difficult to find an existing or design a new operator that guarantees that the offspring are feasible. Although one possible option is simply to discard any infeasible points and reapply the operator until a feasible solution is generated, the process of checking that a solution is feasible may be so time consuming as to render this approach unsuitable. However, there remains a large class of problems where this approach is valid and with suitable choice of operators can be very successfully applied.

12.4.4 Decoder Functions

Decoder functions are a class of mappings from the genotype space S' to the feasible regions F of the solution space S that have the following properties:

- Every $z \in S'$ must map to a *single* solution $s \in F$.
- Every solution $s \in F$ must have at least one representation $s' \in S'$.
- Every $s \in F$ must have the same number of representations in S' (this need not be 1).

The use of decoder functions can be illustrated on the knapsack problem again. A simple approach here to sort the items by profit/weight ratio, and then represent a potential solution as a binary string of length l where a gene with allele value 1 is included in the subset. It can immediately be seen that this representation permits the creation of infeasible solutions from feasible ones if normal variation operators are used, and that constructing operators that guarantee feasible solutions is decidedly non-trivial. Therefore some form of pruning is needed.

One such decoder approach would start at the left hand end of the string and interpret a 1 as *take this item if possible....* Although a providing relatively simple way of using EAs for this type of problem, such decoder functions are not without their drawbacks. These are centred around the fact that they generally introduce a lot of redundancy into the genotype space. In the first example given, if the weight limit is reached after considering say 5 of 10 genes, then it is irrelevant what values the rest take, and so 2^5 strings all map onto the same solution.

In some cases it may be possible to devise a decoder function that permits the use of relatively standard representation and operators whilst preserving a one-to-one mapping between genotype and phenotype. One such example is the decoder for the TSP problem proposed by Grefenstette, and well described by Michalewicz in [272]. In this case a simple integer representation was used with each gene $a_i \in \{1, \ldots, l + 1 - i\}$. This representation permits the use of "standard" crossover operators and a bitwise mutation operator that randomly resets a gene value to one of its permitted allele values. The outcome of both of these operators is guaranteed to be valid. The decoder function works by considering an ordered list of cities, $\{ABCDE\}$, and using the genotype to index into this.

For example, with a genotype $\{4, 2, 3, 1, 1\}$ the first city in the constructed tour is the fourth item in the list, i.e., D. This city is then removed from the list and the second gene is considered, which in this case points to B. This process is continued until a complete tour is constructed: $\{4, 2, 3, 1, 1\} \rightarrow DBEAC$.

Although the one to one mapping means that there is no redundancy in the genotype space, and permits the use of straightforward crossover and mutation operators, the complexity of the mapping function means that a small mutation can have a large effect, e.g., $\{3, 2, 3, 1, 1\} \rightarrow CBDAE$. Equally, it can be easily shown that recombination operators no longer respect and propagate all features common to both solutions. Thus if the two solutions $\{1, 1, 1, 1, 1\} \rightarrow ABCDE$ and $\{5, 1, 2, 3, 1\} \rightarrow EACDB$, which share the common feature that C occurs in the third position and D in the fourth undergo 1-point crossover between the third and fourth loci, the solution $\{5, 1, 2, 1, 1\} \rightarrow EACBD$ is obtained, which does not possess this feature. If the crossover occurs in other positions, the edge CD may be preserved, but in a different position in the cycle.

In both of the examples given, the complexity of the genotype–phenotype mapping makes it very difficult to ensure locality and makes the fitness landscape associated with the search space highly complex, since the potential effects in fitness of changes at the left-hand end of the string are much bigger than those at the right-hand end [179]. Equally, it can become very difficult to specify exactly the common features the recombination operators are supposed to be preserving.

12.5 Application Example: Graph Three-Colouring

We illustrate the approaches outlined above via the description of two different ways of solving a well-known CSP problem, graph three-colouring as defined in Section 12.2.2.

12.5.1 Indirect Approach

In this section we take an indirect approach, transforming the problem from a CSP to a FOP by means of penalty functions. The most straightforward representation is using ternary strings of length n, where each variable stands for one node, and the integers 1, 2, and 3 denote the three colours. Using this standard GA representation has the advantage that all standard variation operators are immediately applicable. We now define two objective functions (penalty functions) that measure the amount of "incorrectness" of a chromosome. The first function is based on the number of "incorrect edges" that connect two nodes with the same colour, while the second relies on counting the "incorrect nodes" that have a neighbour with the same colour. For a formal description let us denote the constraints belonging to the edges as c_i ($i = \{1, \ldots, m\}$), and let C^i be the set of constraints involving variable v_i (edges connecting to node i). Then the penalties belonging to the two options above described can be expressed as follows:

$$f_1(\bar{s}) = \sum_{i=1}^{m} w_i \times \chi(\bar{s}, c_i),$$

where $\chi(\bar{s}, c_i) = \begin{cases} 1 & \text{if } \bar{s} \text{ violates } c_i, \\ 0 & \text{otherwise.} \end{cases}$

respectively,

$$f_2(\bar{s}) = \sum_{i=1}^{n} w_i \times \chi(\bar{s}, C^i),$$

where $\chi(\bar{s}, C^i) = \begin{cases} 1 & \text{if } \bar{s} \text{ violates at least one } c \in C^i, \\ 0 & \text{otherwise.} \end{cases}$

Note that both functions are correct transformations of the constraints in the sense that for each $\bar{s} \in S$ we have that $\phi(\bar{s}) = \mathit{true}$ if and only if $f_i(\bar{s}) = 0$ ($i = 1, 2$). The motivation to use weighted sums in this example, and in general, is that they provide the possibility of emphasising certain constraints (variables) by giving them a higher weight. This can be beneficial if some constraints are more important or known to be harder to satisfy. Assigning them a higher weight gives a higher reward to a chromosome, hence the EA naturally focuses on these. Setting the weights can be done manually by the user, but can also be done by the EA itself on-the-fly as in the stepwise adaptation of weights (SAW) mechanism [132].

Now the EA for the graph three-colouring problem can be composed from standard components. For instance, we can apply a steady state GA with population size 100, binary tournament selection and worst fitness deletion, using random resetting mutation with $p_m = 1/n$ and uniform crossover with $p_c = 0.8$. Notice that this EA really ignores constraints; it only tries to minimise the given objective function (penalty function).

12.5.2 Mixed Mapping – Direct Approach

We now present another EA for this problem, illustrating how constraints can be handled by a decoder. The main idea is to use permutations of the nodes as chromosomes. The phenotype (colouring) belonging to a genotype (permutation) is determined by a procedure that assigns colours to nodes in the order they occur in the given permutation, trying the colours in increasing order (1,2,3), and leaving the node uncoloured if all three colours would lead to a constraint violation. Formally, we shift from the search space $S = \{1, 2, 3\}^n$

to $S' = \{\bar{s} \in S \mid s_i \neq s_j \quad i,j = 1,\ldots,n\}$, and the colouring procedure (the decoder) is the mapping from S' to S. At the first glance this might not seem a good idea as we still have constraints in the transformed problem. However, we know from Chapter 3 that working in a permutation space using a direct approach is easy, as there are many suitable variation operators keeping the search in this space. In other words, we have various operators preserving the constraints defining this space.

An appropriate objective function for this representation can simply be defined as the number (weighted sum) of nodes that remain uncoloured after decoding. This function also has the property that an optimal value (0) implies that all constraints are satisfied, i.e., all nodes are coloured correctly. The rest of the EA can again use off-the-shelf components: a steady-state GA with population size 100, binary tournament selection and worst fitness deletion, using swap mutation with $p_m = 1/n$ and order crossover with $p_c = 0.8$.

Looking at this solution at a conceptual level we can note that there are two constraint handling issues. Primary constraint handling concerns handling the constraints of the original problem, the graph three-colouring CSP. This is done by the mapping approach via a decoder. However, the transformed search space S' in which the EA has to work in is not free, rather it is restricted by the constraints defining permutations. This constitutes the secondary constraint handling issue that is solved by a (direct) preserving approach using appropriate variation operators.

12.6 Exercises

1. Specify the eight-queens problem as a CSP $\langle S, \phi \rangle$.
2. Is it true that solving a COP with an EA always implies indirect constraint handling?
3. Is it true that solving a CSP with an EA never involves direct constraint handling?
4. Design an EA for solving a 3-SAT problem. In a propositional satisfiability problem (SAT) a propositional formula is given, and a truth assignment for its variables is sought that makes the formula true. Without loss of generality it can be assumed that the given formula is in *conjunctive normal form* (CNF), i.e., it is a conjunction of clauses where a clause is a disjunction of literals. In the 3-SAT version of this problem it is also assumed that the clauses consist of exactly three literals. In the common notation, a formula has l clauses $(L_1, \ldots, L_l)$ and n variables $(v_1, \ldots, v_n)$.

12.7 Recommended Reading for this Chapter

1. T. Bäck, D.B. Fogel, and Z. Michalewicz, editors. *Evolutionary Computation 2: Advanced Algorithms and Operators* Part II: Chapters 6–12,

pages 38–86. Institute of Physics Publishing, Bristol, 2000
A series of chapters providing comprehensive reviews of different EA approaches to constraint handling, written by experts in the field

2. B.G.W. Craenen, A.E. Eiben, and J.I. van Hemert. Comparing evolutionary algorithms on binary constraint satisfaction problems. *IEEE Transactions on Evolutionary Computation*, 2003 (in press)

3. A.E. Eiben. Evolutionary algorithms and constraint satisfaction: Definitions, survey, methodology, and research directions. In Kallel, Naudts, Rogers, Eds. [222], 2001
Clear definitions and a good overview of evolutionary constraint handling methods from the CSP point of view

4. J. Smith. *Handbook of Global Optimization Volume 2*, Chap. Genetic Algorithms, pages 275–362. Kluwer Academic Publishers, Boston, 2002
A good overview of constraint handling methods in GAs from the COP point of view

5. Z. Michalewicz and M. Schoenauer. Evolutionary algorithms for constrained parameter optimisation problems. *Evolutionary Computation*, 4:1 pp.1–32, 1996.