6

Popular Evolutionary Algorithm Variants

In this chapter we describe the most widely known evolutionary algorithm variants. This overview serves a twofold purpose: On the one hand, it introduces those historical EA variants without which no EC textbook would be complete together with some more recent versions that deserve their own place in the family tableau. On the other hand, it demonstrates the diversity of realisations of the same basic evolutionary algorithm concept.

6.1 Genetic Algorithms

The genetic algorithm (GA) is the most widely known type of evolutionary algorithm. It was initially conceived by Holland as a means of studying adaptive behaviour, as suggested by the title of the book describing his early research: *Adaptation in Natural and Artificial Systems* [220]. However, GAs have largely (if perhaps mistakenly – see [103]) been considered as function optimisation methods. This is perhaps partly due to the title of Goldberg's seminal book: *Genetic Algorithms in Search, Optimization and Machine Learning* [189] and some very high-profile early successes in solving optimisation problems. Together with De Jong's thesis [102] this work helped to define what has come to be considered as the classical genetic algorithm — commonly referred to as the 'canonical' or 'simple GA' (SGA). This has a binary representation, fitness proportionate selection, a low probability of mutation, and an emphasis on genetically inspired recombination as a means of generating new candidate solutions. It is summarised in Table 6.1. Perhaps because it is so widely used for teaching EAs, and is the first EA that many people encounter, it is worth re-iterating that many features that have been developed over the years are missing from the SGA — most obviously that of elitism.

While, the table does not indicate this, GAs traditionally have a fixed workflow: given a population of μ individuals, parent selection fills an intermediary population of μ, allowing duplicates. Then the intermediary population is shuffled to create random pairs and crossover is applied to each consecutive

A.E. Eiben, J.E. Smith, *Introduction to Evolutionary Computing*,
Natural Computing Series, DOI 10.1007/978-3-662-44874-8_6

pair with probability p_c and the children replace the parents immediately. The new intermediary population undergoes mutation individual by individual, where each of the l bits in an individual is modified by mutation with independent probability p_m. The resulting intermediary population forms the next generation replacing the previous one entirely. Note that in this new generation there might be pieces, perhaps complete individuals, from the previous one that survived crossover and mutation without being modified, but the likelihood of this is rather low (depending on the parameters μ, p_c, p_m).

Representation	Bit-strings
Recombination	1-Point crossover
Mutation	Bit flip
Parent selection	Fitness proportional - implemented by Roulette Wheel
Survival selection	Generational

Table 6.1. Sketch of the simple GA

In the early years of the field there was significant attention paid to trying to establish suitable values for GA parameters such as the population size, crossover and mutation probabilities. Recommendations were for mutation rates between $1/l$ and $1/\mu$, crossover probabilities around 0.6-0.8, and population sizes in the fifties or low hundreds, although to some extent these values reflect the computing power available in the 1980s and 1990s.

More recently it has been recognised that there are some flaws in the SGA. Factors such as elitism, and non-generational models were added to offer faster convergence if needed. As discussed in Chap. 5, SUS is preferred to roulette wheel implementation, and most commonly rank-based selection is used, implemented via tournament selection for simplicity and speed. Studying the biases in the interplay between representation and one-point crossover (e.g. [411]) led to the development of alternatives such as uniform crossover, and a stream of work through 'messy-GAs' [191] and 'Linkage Learning' [209, 395, 385, 83] to Estimation of Distribution Algorithms (see Sect. 6.8). Analysis and experience has recognised the need to use non-binary representations where more appropriate (as discussed in Chap. 4). Finally the problem of how to choose a suitable fixed mutation rate has largely been solved by adopting the idea of self-adaptation, where the rates are encoded as extra genes in an individuals representation and allowed to evolve [18, 17, 396, 383, 375].

Nevertheless, despite its simplicity, the SGA is still widely used, not just for teaching purposes, and for benchmarking new algorithms, but also for relatively straightforward problems in which binary representation is suitable. It has also been extensively modelled by theorists (see Chap. 16). Since it has provided so much inspiration and insight into the behaviour of evolutionary processes in combinatorial search spaces, it is fair to consider that if OneMax is the *Drosophilia* of combinatorial problems for researchers, then the SGA is the *Drosophilia* of evolutionary algorithms.

6.2 Evolution Strategies

Evolution strategies (ES) were invented in the early 1960s by Rechenberg and Schwefel, who were working at the Technical University of Berlin on an application concerning shape optimisation (see [54] for a brief history). The earliest ES's were simple two-membered algorithms denoted (1+1) ES's (pronounce: one plus one ES), working in a vector space. An offspring is generated by the addition of a random number independently to each to the elements of the parent vector and accepted if fitter. An alternative scheme, denoted as (1,1) ES (pronounce: one comma one ES) always replaces the parent by the offspring, thus forgetting the previous solutions by definition. The random numbers are drawn from a Gaussian distribution with mean zero and a standard deviation σ, where σ is called the mutation step size. One of the key early breakthroughs of ES research was to propose a simple mechanism for online adjustment of step sizes by the famous **1/5 success rule** of Rechenberg [352] as described in Sect. 8.2.1. In the 1970s the concept of multi-membered evolution strategies was introduced, with the naming convention based on μ individuals in the population and λ offspring generated in one cycle. The resulting $(\mu+\lambda)$ and (μ, λ) ES's gave rise to the possibility of more sophisticated forms of step-size control, and led to the development of a very useful feature in evolutionary computing: **self-adaptation** of strategy parameters, see Sect. 4.4.2. In general, self-adaptivity means that some parameters of the EA are varied during a run in a specific manner: the parameters are included in the chromosomes and coevolve with the solutions. Technically this means that an ES works with extended chromosomes $\langle \overline{x}, \overline{p} \rangle$, where $\overline{x} \in \mathbb{R}^n$ is a vector from the domain of the given objective function to be optimised, while $\overline{p}$ carries the algorithm parameters. Modern evolution strategies always self-adapt the mutation step sizes and sometimes their rotation angles. That is, since the procedure was detailed in 1977 [372] most ESs have been self-adaptive, and other EAs have increasingly adopted self-adaptivity. Recent forms of ES such as the CMA [207] are among the leading algorithms for optimisation of complex real-valued functions. A summary of ES is given in Table 6.2.

Representation	Real-valued vectors
Recombination	Discrete or intermediary
Mutation	Gaussian perturbation
Parent selection	Uniform random
Survivor selection	Deterministic elitist replacement by (μ, λ) or $(\mu + \lambda)$
Speciality	Self-adaptation of mutation step sizes

Table 6.2. Sketch of ES

The basic recombination scheme in evolution strategies involves two parents that create one child. To obtain λ offspring recombination is performed λ

times. There are two recombination variants distinguished by the manner of recombining parent alleles. Using **discrete recombination** one of the parent alleles is randomly chosen with equal chance for either parents. In **intermediate recombination** the values of the parent alleles are averaged. An extension of this scheme allows the use of more than two recombinants, because the two parents are drawn randomly for each position $i \in \{1, \ldots, n\}$ in the offspring anew. These drawings take the whole population of μ individuals into consideration, and the result is a recombination operator with possibly more than two individuals contributing to the offspring. The exact number of parents, however, cannot be defined in advance. This multiparent variant is called **global recombination**. To make terminology unambiguous, the original variant is called **local recombination**. Evolution strategies typically use global recombination. Interestingly, different recombination is used for the object variable part (discrete is recommended) and the strategy parameters part (intermediary is recommended). This scheme preserves diversity within the phenotype (solution) space, allowing the trial of very different combinations of values, whilst the averaging effect of intermediate recombination assures a more cautious adaptation of strategy parameters.

The selection scheme that is generally used in evolution strategies is (μ, λ) selection, which is preferred over $(\mu + \lambda)$ selection for the following reasons:

- The (μ, λ) discards all parents and so can in principle leave (small) local optima, which is advantageous for multimodal problems.
- If the fitness function changes over time, the $(\mu + \lambda)$ selection preserves outdated solutions, so is less able to follow the moving optimum.
- $(\mu + \lambda)$ selection hinders the self-adaptation, because misadapted strategy parameters may survive for a relatively large number of generations. For example, if an individual has relatively good object variables but poor strategy parameters, often all of its children will be bad. Thus they will be removed by an elitist policy, while the misadapted strategy parameters in the parent may survive for longer than desirable.

The selective pressure in evolution strategies is very high because λ is typically much higher than μ (traditionally a 1/7 ratio is recommended, although recently values around 1/4 seem to gain popularity). The **takeover time** τ^* of a given selection mechanism is defined as the number of generations it takes until the application of selection completely fills the population with copies of the best individual, given one copy initially. Goldberg and Deb [190] showed that

$$\tau^* = \frac{\ln \lambda}{\ln(\lambda/\mu)}.$$

For a typical evolution strategy with $\mu = 15$ and $\lambda = 100$, this results in $\tau^* \approx 2$. By way of contrast, for fitness proportional selection in a genetic algorithm with $\mu = \lambda = 100$ it is $\tau^* = \lambda \ln \lambda = 460$. This indicates that an ES is a more aggressive optimizer than a (simple) GA.

6.3 Evolutionary Programming

Evolutionary programming (EP) was originally developed by Fogel et al. in the 1960s to simulate evolution as a learning process with the aim of generating artificial intelligence [166, 174]. Intelligence, in turn, was viewed as the capability of a system to adapt its behaviour in order to meet some specified goals in a range of environments. Adaptive behaviour is the key term in this definition, and the capability to predict the environment was considered to be a prerequisite. The classic EP systems used finite state machines as individuals.

Nowadays EP frequently uses real-valued representations, and so has almost merged with ES. The principal differences lie perhaps in the biological inspiration: in EP each individual is seen as corresponding to a distinct *species*, and so there is no recombination. Furthermore, the selection mechanisms are different. In ES parents are selected stochastically, then the selection of the μ best from the union of $\mu + \lambda$ offspring is deterministic. By contrast, in EP each parent generates exactly one offspring (i.e., $\lambda = \mu$), but these parents and offspring populations are then merged and compete in stochastic round-robin tournaments for survival, as described in Sect. 5.3.2. The field now adopts a very open, pragmatic approach that the choice of representation, and hence mutation, should be driven by the problem; Table 6.3 is therefore a representative rather than a standard algorithm variant.

Representation	Real-valued vectors
Recombination	None
Mutation	Gaussian perturbation
Parent selection	Deterministic (each parent creates one offspring via mutation)
Survivor selection	Probabilistic $(\mu + \mu)$
Speciality	Self-adaptation of mutation step sizes (in meta-EP)

Table 6.3. Sketch of EP

The issue of the advantage of using a mutation-only algorithm versus a recombination and mutation variant has been intensively discussed since the 1990s. Fogel and Atmar [170] compared the results of EP algorithms with and without recombination on a series of linear functions with parameterisable interactions between genes. They concluded that improved performance was obtained from the version without recombination. This led to intensive periods of research in both the EP and the GA communities to try and establish the circumstances under which the availability of a recombination operator yielded improved performance [159, 171, 222, 408]. The current state of thinking has moved on to a stable middle ground. The latest results [232] confirm that the ability of both crossover or Gaussian mutation to produce new offspring of superior fitness to their parents depends greatly on the state of the

search process, with mutation better initially but crossover gaining in ability as evolution progresses. These conclusions agree with theoretical results developed elsewhere and discussed in more depth in Chap. 7. In particular it is stated that: "the traditional practice of setting operator probabilities at constant values, ... is quite limiting and may even prevent the successful discovery of suitable solutions." However, it is perhaps worth noting that even in these studies the authors did not detect a difference between the performance of different crossover operators, which they claim casts significant doubt on the building block hypothesis (Sect. 16.1), so we are not entirely without healthy scientific debate!

Since the 1990s EP variants for optimisation of real-valued parameter vectors have become more frequent and even positioned as 'standard' EP [22, 30]. During the history of EP a number of mutation schemes, such as one in which the step size is inversely related to the fitness of the solutions, have been proposed. Since the proposal of **meta-EP** [165, 166], self-adaptation of step sizes has become the norm, using the scheme in Eq. (4.4). A variety of schemes have been proposed, including mutation variables first then strategy parameters (which violates the rationale explained in Sect. 4.4.2). Tracing the literature on this issue, the paper by Gehlhaar and Fogel [182] seems to be a turning point. Here the authors explicitly compare the 'sigma first' and 'sigma last' strategies and conclude that the first one – the standard ES manner – offers a consistent general advantage over the second one. Notably in a paper and book [81, 168], Fogel uses the lognormal adaptation of n standard deviations σ_i, *followed* by the mutation of the object variables x_i themselves, suggesting that EP is practically merging with ES regarding this aspect. Other ideas from ES have also informed the development of EP algorithms, and a version with self-adaptation of covariance matrices, called **R-meta-EP** is also in use. Worthy of note is Yao's improved fast evolutionary programming algorithm (IFEP) [470], whereby two offspring are created from each parent, one using a Gaussian distribution to generate the random mutations, and the other using the Cauchy distribution. The latter has a fatter tail (i.e., more chance of generating a large mutation), which the authors suggest gives the overall algorithm greater chance of escaping from local minima, whilst the Gaussian distribution (if small step sizes evolve) gives greater ability to fine-tune the current parents.

6.4 Genetic Programming

Genetic programming is a relatively young member of the evolutionary algorithm family. It differs from other EA strands in its application area as well as the particular representation (using trees as chromosomes). While the EAs discussed so far are typically applied to optimisation problems, GP could instead be positioned in machine learning. In terms of the different problem types as discussed in Chapter 2, most other EAs are for finding some input

realising maximum payoff (Fig. 1.1), whereas GP is used to seek models with maximum fit (Fig. 1.2). Clearly, once maximisation is introduced, modelling problems can be seen as special cases of optimisation. This, in fact, is the basis of using evolution for such tasks: models — represented as parse trees — are treated as individuals, and their fitness is the model quality to be maximised. The summary of GP is given in Table 6.4.

Representation	Tree structures
Recombination	Exchange of subtrees
Mutation	Random change in trees
Parent selection	Fitness proportional
Survivor selection	Generational replacement

Table 6.4. Sketch of GP

The parse trees used by GP as chromosomes capture expressions in a given formal syntax. Depending on the problem at hand, and the users' perceptions on what the solutions must look like, this can be the syntax of arithmetic expressions, formulas in first-order predicate logic, or code written in a programming language, cf. Sect. 4.6. In particular, they can be envisioned as executable codes, that is, programs. The syntax of functional programming, e.g., the language LISP, very closely matches the so-called Polish notation of expressions. For instance, the formula in Eq. (4.8) can be rewritten in this Polish notation as

$$+(\cdot(2,\pi),-(+(x,3),/(y,+(5,1)))),$$

while the executable LISP code[1] looks like:

$$(+\ (\cdot\ 2\ \pi)\ (-\ (+\ x\ 3)\ (/\ y\ (+\ 5\ 1)))).$$

Based on this perception, GP can be positioned as the "programming of computers by means of natural selection" [252], or the "automatic evolution of computer programs" [37].

There are a few other issues that are specific to tree-based representations, and hence (but not exclusively) to genetic programming.

Initialisation can be carried out in different ways for trees. The most common method used in GP is the so-called **ramped half-and-half** method. In this method a maximum initial depth D_{max} of trees is chosen, and then each member of the initial population is created from the sets of functions F and terminals T using one of the two methods below with equal probability:

- *Full method*: here each branch of the tree has depth D_{max}. The contents of nodes at depth d are chosen from F if $d < D_{max}$ or from T if $d = D_{max}$.

[1] To be precise we should use PLUS, etc., for the operators.

- *Grow method*: here the branches of the tree may have different depths, up to the limit D_{max}. The tree is constructed beginning from the root, with the contents of a node being chosen stochastically from $F \cup T$ if $d < D_{max}$.

Stochastic Choice of a Single Variation Operator. In GP offspring are typically created by either recombination *or* mutation, rather than recombination *followed by* mutation, as is more common in other variants. This difference is illustrated in Fig. 6.1 (inspired by Koza [252]), which compares the loop for filling the next generation in a generational GA with that of GP.

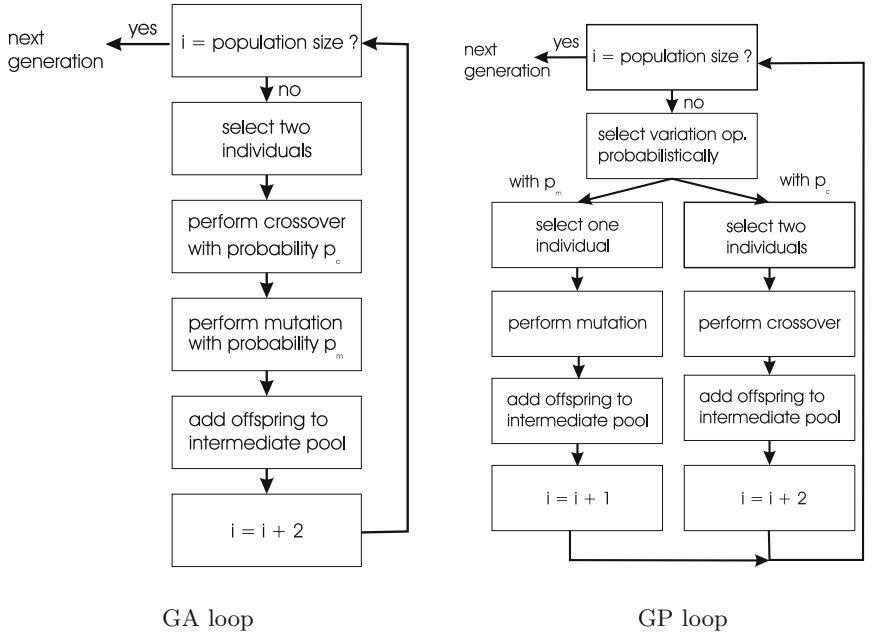

Fig. 6.1. GP flowchart versus GA flowchart. The two diagrams show two options for filling the intermediary population in a generational scheme. In a conventional GA mutation and crossover are used to produce the next offspring (left). Within GP, a new individual is created by either mutation or crossover (right).

Low or Zero Mutation Probabilities. Koza's classic book on GP from 1992 [252] advises users to set the mutation rate at 0, i.e., it suggests that GP works *without* mutation. More recently Banzhaf et al. recommended 5% [37]. In giving mutation such a limited role, GP differs from other EA streams. The reason for this is the generally shared view that crossover has a large shuffling effect, acting in some sense as a macromutation operator [9]. The current GP practice uses low mutation frequencies, even though some studies indicate that an (almost) pure crossover approach might be inferior [275].

Over-selection is often used to deal with the typically large population sizes (population sizes of several thousands are not unusual in GP). The method first ranks the population, then divides it into two groups, one containing the top $x\%$ and the other containing the other $(100-x)\%$. When parents are selected, 80% of the selection operations come from the first group, and the other 20% from the second group. The values of x used are found empirically by rule of thumb and depend on the population size with the aim that the number of individuals from which the majority of parents are chosen stays constant in the low hundreds, i.e., the selection pressure increases dramatically for larger populations.

Bloat (sometimes called the 'survival of the fattest') is a phenomenon observed in GP whereby average tree sizes tend to grow during the course of a run. There are many studies devoted to understanding why bloat occurs and to proposing countermeasures, see for instance [268, 407]. Although the results and discussions are not conclusive, one primary suspect is the sheer fact that we have chromosomes with variable length, meaning that the *possibility* for chromosome sizes to grow along the evolution already implies that they will *actually* do so. Probably the simplest way to prevent bloat is to introduce a maximum tree size and forbid a variation operator if the child(ren) resulting from its application would exceed this maximum size. In this case, this threshold can be seen as an additional parameter of mutation and recombination in GP. Several advanced techniques have also been proposed, but the only one that is widely acknowledged is that of **parsimony pressure**. Such a pressure towards parsimony (i.e., being 'stingy' or ungenerous) is achieved through introducing a penalty term in the fitness formula that reduces the fitness of large chromosomes [228, 406] or using multiobjective techniques [115].

6.5 Learning Classifier Systems

Learning Classifier Systems (LCS) represent an alternative evolutionary approach to model building based on the use of rule sets, rather than parse trees, to represent knowledge [270, 269]. LCS are used primarily in applications where the objective is to evolve a system that will respond to the current state of its environment (i.e., the inputs to the system) by suggesting a response that in some way maximises future reward from the environment.

An LCS is therefore a combination of a classifier system and a learning algorithm. The classifier system component is typically a set of rules, each mapping certain inputs to actions. The whole rule set therefore constitutes a model that covers the space of possible inputs and suggests the most appropriate actions for each. The learning algorithm component of an LCS is implemented by an evolutionary algorithm, whose population members either represent individual rules, or complete rule sets, known respectively as the Michigan and Pittsburgh approaches. The fitness driving the evolutionary process may be driven by many different forms of learning, here we restrict

ourselves to 'supervised' learning, where at each stage the system receives a training signal (reward) from the environment in response to the output it proposes. This helps emphasise the difference between the Michigan and Pittsburgh approaches. In the former, data items are presented to the system one-by-one and individual rules are rewarded according to their predictions. By contrast, in a Pittsburgh approach each individual represents a complete model, so the fitness would normally be calculated by presenting the entire data set and calculating the mean accuracy of the predictions.

The **Michigan-style LCS** was first described by Holland in 1976 as a framework for studying learning in condition/action rule-based systems, using genetic algorithms as the principal method for the discovery of new rules and the reinforcement of successful ones [219]. Typically each member of the population was a single rule representing a partial model – that is to say it might only cover a region of the decision space. Thus it is the entire population that together represents the learned model. Each rule is a tuple {`condition:action:payoff`}. The condition specifies a region of the space of possible inputs in which the rule applies. The condition parts of rules may contain wildcard, or 'don't-care' characters for certain variables, or may describe a set of values that a given variable may take – for example, a range of values for a continuous variable. Rules may be distinguished by the number of wildcards they contain, and one rule is said to be more specific than another if it contains fewer wildcards, or if the ranges for certain variables are smaller — in other words if it covers a smaller region of the input space. Given this flexibility, it is common for the condition parts of rules to overlap, so a given input may match a number of rules. In the terminology of LCS, the subset of rules whose condition matches the current inputs from the environment is known as the **match set**. These rules may prescribe different actions, of which one is chosen. The action specifies either the action to be taken (for example, if controlling robots or on-line trading agents) or the system's prediction (such as a class label or a numerical value). The subset of the match set advocating the chosen action is known as the **action set**. Holland's original framework maintained lists of which rules have been used, and when a reward was received from the environment a portion was passed back to recently used rules to provide information for the selection mechanism. The intended effect is that the strength of a rule predicts the value of the reward that the system will gain for undertaking the action. However the framework proved unwieldy and difficult to make work well in practice.

LCS research was reinvigorated in the mid-1990s by Wilson who removed the concept of memory and stripped out all but the essential components in his minimalist ZCS algorithm [464]. At the same time several authors were noting the conceptual similarity between LCS and reinforcement learning algorithms which attempt to learn, for each input state, an accurate mapping from possible actions to expected rewards. The XCS algorithm [465] firmly established this link by extending rule-tuples to {`condition:action:payoff,accuracy`}, where the accuracy value reflects the system's experience of how well the pre-

dicted payoff matches the reward received. Unlike ZCS, the EA is restricted at each cycle — originally to the match set, latterly to the action set, which increases the pressure to discover generalised conditions for each action. As per ZCS, a credit assignment mechanism is triggered by the receipt of rewards from the environment to update the predicted pay-offs for rules in the previous action set. However, the major difference is that these are not used directly to drive selection in the evolution process. Instead selection operates on the basis of accuracy, so the algorithm can in principle evolve a *complete* mapping from input space to actions.

Table 6.5 gives a simple overview of the major features of Michigan-style classifiers for a problem with a binary input and output space. The list below summarizes the main workflow of the algorithm.

1. A new set of inputs are received from the environment.
2. The rule base is examined to find the match-set of rules.
   - If the match set is empty, a 'cover operator' is invoked to generate one or more new matching rules with a random action.
3. The rules in the match-set are grouped according to their actions.
4. For each of these groups the mean accuracy of the rules is calculated.
5. An action is chosen, and its corresponding group noted as the action set.
   - If the system is an 'exploit' cycle, the action with the highest mean accuracy is chosen.
   - If the system is in an 'explore' cycle, an action is chosen randomly or via fitness-proportionate selection, acting on the mean accuracies.
6. The action is carried out and a reward is received from the environment.
7. The estimated accuracy and predicted payoffs are then updated for the rule in the current and previous action sets, based on the rewards received and the predicted pay-offs, using a Widrow–Hoff style update mechanism.
8. If the system is in an 'explore' cycle, an EA is run within the action-set, creating new rules (with pay-off and accuracies set to the mean of their parents), and deleting others.

Representation	tuple of {`condition:action:payoff,accuracy`} conditions use {0,1,#} alphabet
Recombination	One-point crossover on conditions/actions
Mutation	Binary/ternary resetting as appropriate on action/conditions
Parent selection	Fitness proportional with sharing within environmental niches
Survivor selection	Stochastic, inversely related to number of rules covering same environmental niche
Fitness	Each reward received updates predicted payoff and accuracy of rules in relevant action sets by reinforcement learning.

Table 6.5. Sketch of a Michigan-style LCS for a binary input and action space

The **Pittsburgh-style LCS** predates, but is similar to the better-known GP: each member of the evolutionary algorithm's population represents a complete model of the mapping from input to output spaces. Each gene in an individual typically represents a rule, and again a new input item may match more than one rule, in which case typically the first match is taken. This means that the representation should be viewed as an ordered list, and two individuals which contain the same rules, but in a different order on the genome, are effectively different models. Learning of appropriately complex models is typically facilitated by using a variable-length representation so that new rules can be added at any stage. This approach has several conceptual advantages — in particular, since fitness is awarded to complete rule sets, models can be learned for complex multi-step problems. The downside of this flexibility is that, like GP, Pittsburgh-style LCS suffers from bloat and the search space becomes potentially infinite. Nevertheless, given sufficient computational resources, and effective methods of parsimony to counteract bloat, Pittsburgh-style LCS has demonstrated state-of-the-art performance in several machine learning domains, especially for applications such as bioinformatics and medicine, where human-interpretability of the evolved models is vital and large data-sets are available so that the system can evolve off-line to minimise prediction error. Two recent examples winning Humies Awards for better than human performance are in the realms of prostate cancer detection [272] and protein structure prediction [16].

6.6 Differential Evolution

In this section we describe a young, but powerful member of the evolutionary algorithm family: differential evolution (DE). Its birth can be dated to 1995, when Storn and Price published a technical report describing the main concepts behind a "new heuristic approach for minimizing possibly nonlinear and nondifferentiable continuous space functions" [419]. The distinguishing feature that delivered the name of this approach is a twist to the usual reproduction operators in EC: the so-called **differential mutation**. Given a population of candidate solution vectors in $\mathbb{R}^n$ a new mutant vector $\overline{x}'$ is produced by adding a **perturbation vector** to an existing one,

$$\overline{x}' = \overline{x} + \overline{p},$$

where the perturbation vector $\overline{p}$ is the scaled vector difference of two other, randomly chosen population members

$$\overline{p} = F \cdot (\overline{y} - \overline{z}), \tag{6.1}$$

and the scaling factor $F > 0$ is a real number that controls the rate at which the population evolves. The other reproduction operator is the usual uniform crossover, subject to one parameter, the crossover probability $Cr \in [0, 1]$ that

defines the chance that for any position in the parents currently undergoing crossover, the allele of the first parent will be included in the child. (Remember that in GAs the crossover rate $p_c \in [0, 1]$ is defined for any given pair of individuals and it is the likelihood of actually executing crossover.) DE also has a slight twist to the crossover operator: at one randomly chosen position the child allele is taken from the first parent without making a random decision. This ensures that the child does not duplicate the second parent.

In the main DE workflow populations are lists, rather than (multi)sets, allowing references to the i-th individual by its position $i \in \{1, \ldots, \mu\}$ in this list. The order of individuals in such a population $P = \langle \overline{x}_1, \ldots, \overline{x}_i, \ldots, \overline{x}_\mu \rangle$ is not related to their fitness values. An evolutionary cycle starts with creating a mutant vector population $M = \langle \overline{v}_1, \ldots, \overline{v}_\mu \rangle$. For each new mutant $\overline{v}_i$ three vectors are chosen randomly from P, a base vector to be mutated and two others to define a perturbation vector. After making the mutant vector population, a so-called trial vector population $T = \langle \overline{u}_1, \ldots, \overline{u}_\mu \rangle$ is created, where $\overline{u}_i$ is the result of applying crossover to $\overline{v}_i$ and $\overline{x}_i$. (Note, that it is guaranteed that $\overline{u}_i$ does not duplicate $\overline{x}_i$.) In the last step deterministic selection is applied to each pair $\overline{x}_i$ and $\overline{u}_i$: the i-th individual in the next generation is $\overline{u}_i$ if $f(\overline{u}_i) \leq f(\overline{x}_i)$ and $\overline{x}_i$ otherwise.

Representation	Real-valued vectors
Recombination	Uniform crossover
Mutation	Differential mutation
Parent selection	Uniform random selection of the 3 necessary vectors
Survival selection	Deterministic elitist replacement (parent vs. child)

Table 6.6. Sketch of differential evolution

In general, a DE algorithm has three parameters, the scaling factor F, the population size μ (usually denoted by NP in the DE literature), and the crossover probability Cr. It is worth noting that despite mediating a crossover process, Cr can also be thought of as a mutation rate, i.e., the approximate probability that an allele will be inherited from a mutant [343]. The DE community also emphasises another aspect of uniform crossover: The number of inherited mutant alleles follows a binomial distribution, since allele origins are determined by a finite number of independent trials having two outcomes with constant probabilities.

Over the years, several DE variants have been invented and published. One of the modifications concerns the choice of the base vector when building the mutant population M. It can be randomly chosen for each v_i, as presented here, but it can be fixed, always using the best vector in the population and only varying the perturbation vectors. Another extension is obtained by allowing more than one difference vector to define the perturbation vector in the mutation operator. For example, using two difference vectors, Equation

6.1 becomes

$$\overline{p} = F \cdot (\overline{y} - \overline{z} + \overline{y}' - \overline{z}') \tag{6.2}$$

where $\overline{y}, \overline{z}, \overline{y}', \overline{z}'$ are four randomly chosen population members.

In order to classify the different variants, the notation DE/a/b/c has been introduced in the literature, where a specifies the base vector, e.g., "rand" or "best", b is the number of difference vectors used to define the perturbation vector, and c denotes the crossover scheme, e.g., "bin" stands for using uniform crossover (because of the binomial distribution of donor alleles it generates). Using this notation, the basic version described above is DE/rand/1/bin.

6.7 Particle Swarm Optimisation

The algorithm we describe here deviates somewhat from other evolutionary algorithms in that it is inspired by social behavior of bird flocking or fish schooling, while the name and the technical terminology are grounded in physical particles [340, 248]. Seemingly there is no evolution in a particle swarm optimizer, but algorithmically it does fit in the general EA framework. Particle swarm optimisation (PSO) was launched in 1995, when Kennedy and Eberhart published their seminal paper about a "concept for the optimization of nonlinear functions using particle swarm methodology" [247]. Similarly to DE, the distinguishing feature of PSO is a twist to the usual reproduction operators in EC: PSO does not use crossover and its mutation is defined through a vector addition. However, PSO differs from DE and most other EC dialects in that every candidate solution $\overline{x} \in \mathbb{R}^n$ carries its own perturbation vector $\overline{p} \in \mathbb{R}^n$. Technically, this makes them quite similar to evolution strategies that use the mutation step sizes in the perturbation vector parts, cf. Sect. 4.4.2 and Sect. 6.2. However, the PSO mindset and terminology is based on a spatial metaphor of particles with a location and velocity, rather than a biological one of individuals with a genotype and mutation.

To simplify the explanation and to emphasise the similarities to other evolutionary algorithms, we present PSOs in two steps. First we give a description that captures the essence of the system using the notion of perturbation vectors $\overline{p}$. Second, we provide the technical details in terms of vectors for velocity $\overline{v}$ and a personal best $\overline{b}$ in line with the PSO literature.

On a conceptual level every population member in a PSO can be considered as a pair $\langle \overline{x}, \overline{p} \rangle$, where $\overline{x} \in \mathbb{R}^n$ is a candidate solution vector and $\overline{p} \in \mathbb{R}^n$ is a perturbation vector that determines how the solution vector is changed to produce a new one. The main idea is that a new pair $\langle \overline{x}', \overline{p}' \rangle$ is produced from $\langle \overline{x}, \overline{p} \rangle$ by first calculating a new perturbation vector $\overline{p}'$ (using $\overline{p}$ and some additional information) and adding this to $\overline{x}$. That is,

$$\overline{x}' = \overline{x} + \overline{p}'$$

The core of the PSO perspective is to consider a population member as a point in space with a position and a velocity and use the latter to determine a new position (and a new velocity). Thus, looking under the hood of a PSO we find that a perturbation vector is a velocity vector $\overline{v}$ and a new velocity vector $\overline{v'}$ is defined as the weighted sum of three components: $\overline{v}$ and two vector differences. The first points from the current position $\overline{x}$ to the best position $\overline{y}$ the given population member ever had in the past, and the second points from $\overline{x}$ to the best position $\overline{z}$ the whole population ever had. Formally, we have

$$\overline{v}' = w \cdot \overline{v} + \phi_1 U_1 \cdot (\overline{y} - \overline{x}) + \phi_2 U_2 \cdot (\overline{z} - \overline{x})$$

where w and ϕ_i are the weights (w is called the inertia, ϕ_1 is the learning rate for the personal influence and ϕ_2 is the learning rate for the social influence), while U_1 and U_2 are randomizer matrices that multiply every coordinate of $\overline{y} - \overline{x}$ and $\overline{z} - \overline{x}$ by a number drawn from the uniform distribution.

It is worth noting that this mechanism requires some additional book keeping. In particular, the personal best $\overline{y}$ and the global best $\overline{z}$ must be kept in memory. This requires a unique identifier for population members that keeps the 'identity' of the given individuals and allows it to maintain the personal memory. To this end, PSO populations are lists, rather than (multi)sets, allowing references to the i-th individual. Similarly to DE, the order of individuals in such a population is not related to their fitness values. Furthermore, the perturbation vector $\overline{p}_i \in \mathbb{R}^n$ for any given $\overline{x}_i \in \mathbb{R}^n$ is not stored directly, as the notation $\langle \overline{x}, \overline{p} \rangle$ in the previous paragraph would indicate, but indirectly by the velocity vector $\overline{v}_i$ and the personal best $\overline{b}_i$ of the i-th population member. Thus, technically, the i-th individual is a triple $\langle \overline{x}_i, \overline{v}_i, \overline{b}_i \rangle$, where $\overline{x}_i$ is the solution vector (perceived as a position), $\overline{v}_i$ is its velocity vector, and $\overline{b}_i$ is its personal best. During an evolutionary cycle each triple $\langle \overline{x}_i, \overline{v}_i, \overline{b}_i \rangle$ is replaced by the mutant triple $\langle \overline{x}'_i, \overline{v}'_i, \overline{b}'_i \rangle$ using the following formulas

$$\overline{x}'_i = \overline{x} + \overline{v}'_i$$
$$\overline{v}'_i = w \cdot \overline{v}_i + \phi_1 U_1 \cdot (\overline{b}_i - \overline{x}_i) + \phi_2 U_2 \cdot (\overline{c} - \overline{x}_i)$$

where $\overline{c}$ denotes the population's global best (champion) and

$$\overline{b}'_i = \begin{cases} \overline{x}'_i & \text{if } f(\overline{x}'_i) < f(\overline{b}_i) \\ \overline{b}_i & \text{otherwise} \end{cases}$$

The rest of the basic PSO algorithm is actually quite simple, since parent selection and survivor selection are trivial. An overview is given in Table 6.7.

6.8 Estimation of Distribution Algorithms

Estimation of distribution algorithms (EDA) are based on the idea of replacing the creation of offspring by 'standard' variation operators (recombination

Representation	Real-valued vectors
Recombination	None
Mutation	Adding velocity vector
Parent selection	Deterministic (each parent creates one offspring via mutation)
Survival selection	Generational (offspring replace parents)

Table 6.7. Sketch of particle swarm optimisation

and mutation) by a three-step process. First a 'graphical model' is chosen to represent the current state of the search in terms of the dependencies between variables (genes) describing a candidate solution. Next the parameters of this model are estimated from the current population to create a conditional probability distribution over the variables. Finally, offspring are created by sampling this distribution.

Probabilistic Graphical Models (PGMs) have been used as models of uncertainty in artificial intelligence systems since the early 1980s. In this approach models are considered to be graphs $G = (V, E)$, where each vertex $v \in V$ represents a single variable, and each directed edge $e \in E$ represents a dependency between two variables. Thus, for example, the presence of an edge $e = \{i, j\}$ denotes that the probability of obtaining a particular value for variable j depends on the value of variable i. Usually graphs are restricted to be acyclic to avoid difficulties with infinite loops.

The pseudocode in Figure 6.2 illustrates the way that offspring are created via the processes of **model selection**, **model fitting** and **model sampling**.

```
BEGIN
  INITIALISE population P^0 with μ random candidate solutions;
  set t = 0;
  REPEAT UNTIL ( TERMINATION CONDITION is satisfied ) DO
    EVALUATE each candidate in P^t;
    SELECT subpopulation P_s^t to be used in the modeling steps;
    MODEL SELECTION creates graph G by dependencies in P_s^t;
    MODEL FITTING creates Bayesian Network BN with G and P_s^t;
    MODEL SAMPLING produces a set Sample(BN) of μ candidates;
    set t = t + 1;
    P^t = Sample(BN);
  OD
END
```

Fig. 6.2. Pseudocode for generic estimation of distribution algorithm

In principle, any standard selection method may be used to select P_s^t, but it is normal practice to use truncation selection and to take the fittest subset of the current population as the basis for the subsequent modelling process. Survivor selection in EDAs typically uses a generational model: newly generated offspring replaces the old population.

Model selection is the critical element in using a PGM approach to data modelling (in our case modelling a population in an EDA). In essence, it amounts to finding the appropriate structure to capture the conditional (in)dependencies between variables. In the context of genetics, and also of evolutionary computation, this process is also known as the "Linkage Learning" problem. Historically the pattern of research in this area has progressed via permitting increasing complexity for the types of structural dependencies examined. The earliest univariate algorithms such as PBIL [35] assumed variables behaved independently. The second generation of EDAs such as MIMIC [59] and BMDA [337] selected structures from the possible pairwise interactions, and current methods such as BOA [336] select structures from the set of all trees of some prespecified maximum size. Of course the number of possible combinations of variables expands extremely rapidly, meaning that some form of search is needed to identify good structural models. Broadly speaking there are two approaches to this. The first is direct estimation that has the reputation of being complex and in most cases impractical. More widely used is the "score + search" approach, which is effectively heuristic search in the space of possible graphical models. This relies heavily on the use of metrics which reflect how well a model, and hence the induced Bayesian Network, captures the underlying structure of the data. A variety of metrics have been proposed in the literature, mostly based on the Kullback–Liebler Divergence metric which is closely related to entropy. A full description of these different quality metrics is beyond the scope of this book.

The process of model-fitting may be characterised as per the pseudocode in Figure 6.3, where we use the notation $P(x, i, c)$ to denote the probability of obtaining allele value i for variable x given the set of conditions c. For the unconditional case we will use $P(x, i, -)$. It should be noted that this code is intended to illustrate the general concept, and that much more efficient implementations exist. For discrete data these parameters form a Bayesian Network, and for continuous data it is common to use a mixture of Gaussian models. Both of these processes are relatively straightforward.

The process of model sampling follows a similar pattern, but in this case within each subgraph we draw random variables to select the parent–node alleles, and then to select allele values for the other nodes using the probabilities from the appropriate partition.

Most of the discussion above has tacitly assumed discrete combinatorial problems and the models fitted have been Bayesian networks. For continuous variable problems a slightly different approach is needed to measure and describe probabilities, which is typically based on normal (Gaussian) distributions. There have been a number of developments, such as the **Iterated Den-**

```
BEGIN
  /*  Let P(x,i,c) denote the probability of generating */
  /*  allele value i for variable x given conditions c */
  /*  Let D denote the set of selected parents */
  /*  Let G denote the model selected */

  FOR EACH unconnected subgraph g ⊂ G DO
    FOR EACH node x ∈ g with no parents DO
      FOR EACH possible allele value i for variable x DO
        set P(x,i,−) = Frequency_In_Subpop(x,i,D);
      OD
    OD
    FOR EACH child node x ∈ g DO
      Partition D according to allele values in x's parents;
      FOR EACH partition c DO
        set P(c) = Sizeof(c)/Sizeof(D);
        FOR EACH possible allele value i for variable x DO
          set P(x,i,c) = Frequency_In_Subpop(x,i,c)/P(c);
        OD
      OD
    OD
  OD
END
```

Fig. 6.3. Pseudocode for generic model fitting

sity Estimation Algorithm [64], and also continuous versions for EDAs. Depending on the complexity of the models permitted, the result is either a univariate or multivariate normal distribution. These are very similar to the correlation matrix in evolution strategies, and a comparison of these approaches may be found in [206].

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