Summary Introduction to Evolutionary Computing

Evolutionary Computing (Vrije Universiteit Amsterdam)

Summary Evolutionary Computing

Babiche de Jong

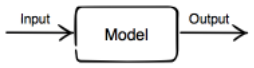
October 2015

1 Problems to be solved

In this chapter we discuss problems to be solved, as encountered frequently by engineers, computer scientists, etc. The field of evolutionary computing is primarily concerned with problem solvers. There are various classes of problems to solve, and in fact, these problems can be classified in different ways.

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. Computer system compose of three elements; inputs, model, outputs.



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.

Examples of a model
- *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 approach some prespecified target speed*
- *a complex series of rules that transform a series of keystrokes into an on screen version of the page you are reading now.*

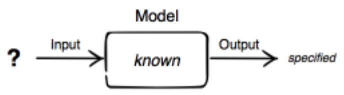
After processing the input the system provide some outputs.

Examples of outputs
- *messages on screen*
- *values written to a file*
- *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.

1.1.1 Optimisation

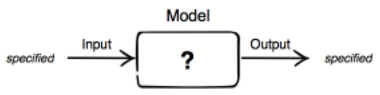
In an optimisation problem the model is known, together with the desired output (or a description of the desired output), and the task is to find the input(s) leading to this output.



Examples
- *Travelling salesman problem*
- *Eight-queens problem*

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. 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.

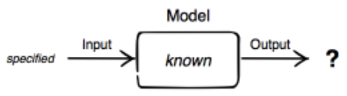


It is important to note that modelling problems can be transformed into optimisation problems by designating the error rate of a model as the quantity to be minimised or its hit rate to be maximised.

Examples
- *Stock exchange*
- *Task of identifying traffic signs in images - perhaps from videa feeds in a smart car*

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.



Advantages of simulators:

- Simulation can be more economical than studying the real-world effects. The real-world alternative may not be feasible at all
- Simulation can be the tool that allows us to look into the future

Examples
- *Electronic circuit, say, a filter cutting out low frequencies in a signal*
- *Weather forecast system*

1.2 Search problems

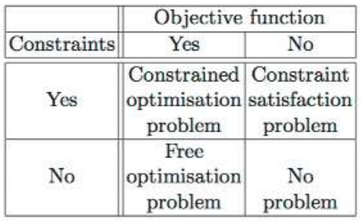
A deeply rooted assumption behind the black box view of systems is that a computational model is directional: it computes from the inputs towards the outputs and it cannot be simply inverted. The process of problem solving can be viewed as a search through a potentially huge set of possibilities to find the desired solution. Consequently, the problems that are to be solved this way can be seen as search problems. 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).
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. 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.
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

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. 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.
Examples of objection functions
1. *the number of checked 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)*

Examples of solutions subject to constraints
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*

Four categories of problem types distinguished by the presence or absence of an objective function and constraint are shown below



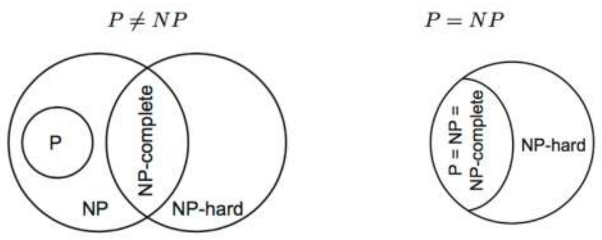
	Objective function	
Constraints	Yes	No
Yes	Constrained optimisation problem	Constraint satisfaction problem
No	Free optimisation problem	No problem

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 item 1 and 4 we can see that constraint satisfaction problems can be transformed into optimisation problems.

1.4 The famous NP problems

In this section we discuss a classification scheme where it is not possible to classify a problem according to one of the previous schemes 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 (e.g. being hard or easy to solve). 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.
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. Notice that the discrete search spaces are always finite or, in the worst case, countably finite.
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.
The second notion concerns algorithms. The running-time of an algorithm is the number of elementary steps, or operations, it takes to terminate. *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.*
The final notion is that of problem reduction, which is the idea that we can transform one problem into another via a suitable mapping.

A problem is said to belong to class P if there exists an algorithm that can solve it in polynomial time (easily solved).
A problem is said to belong to 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.
A problem is said to belong to 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.
A problem is said to belong to class NP-hard if it is at least as hard as any problem in NP-complete, but where the solution cannot necessarily be verified within polynomial time (e.g., Halting problem).



If a problem is NP-complete, then although we might be able to solve particular instances within polynomial time, we can not 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 and abandon the idea of definitely finding a solution which is provably the best for the instance.

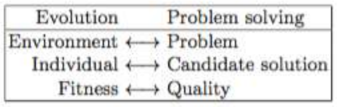
2 Evolutionary computing: The origins

This chapter provides the reader with the basics for studying evolutionary computing.

2.1 The main evolutionary computing metaphor

Evolutionary computing is a research area within computer science, which draws inspiration from the process of natural evolution. The power of evolution in nature is evident in the diverse species that make up our world, each tailored to survive well in its own niche. The fundamental metaphor of evolutionary computing relates this powerful natural evolution to a particular style of problem solving - that of trial-and-error.

Consider natural evolution as:
A given environment is filled with a population of individuals that strive for survival and reproduction. The fitness of these individuals is determined by the environment, and relates to how well they succeed in achieving their goals.In other words, it represents their chances of survival and of multiplying. Meanwhile, in the context of stochastic trial-and-error style problem solving process, we have a collection of candidate solutions. Their quality (how well they solve the problem) determines the chance that they will be kept and used as seeds for constructing further candidate solutions.



Evolution		Problem solving
Environment	⟷	Problem
Individual	⟷	Candidate solution
Fitness	⟷	Quality

2.2 Brief history

1948 Turing proposed ”genetical or evolutionary search”
1962 Bremermann had actually executed computer experiments on ”optimization through evolution and recombination”
1960s Three different implementations of the basic idea were developed.

- Fogel, Owens and Walsh introduced evolutionary programming (USA)
- Holland called his method genetic algorithm (USA)
- Rechenberg and Schwefel invented evolution strategies (Germany)

For about 15 years these areas developed separately; but since the early 1990s they have been viewed as different representatives (’dialects’) of one technology that has come to be known as evolutionary computing
Early 1990s Fourth stream emerged – genetic programming

The algorithms involved are termed evolutionary algorithms, and it considers evolutionary programming, evolution strategies, genetic algorithms, and genetic programming as subareas belonging to the corresponding algorithm variants.

2.3 The inspiration from biology

2.3.1 Darwinian evolution

In Darwin’s theory of evolution natural selection plays a central role. Given an environment that can host only a limited number of individuals, and the basic instinct of individuals to reproduce, selection becomes inevitable if the population size is not to grow exponentially. Natural selection favours those individuals that compete for the given resources most effectively, in other words, those that are adapted or fit to the environmental conditions best – survival of the fittest. Competition-based selection is one of the two cornerstones of evolutionary progress. The other primary force identified by Darwin results from phenotypic variations among members of the population. Phenotypic traits are those behavioural and physical features of an individual that directly affect its response to the environment, thus determining its fitness.
Summarising this basic model:
A population consists of a number of individuals. These individuals are the units of selection, that is to say that their reproductive success depends on how well they are adapted to their environment relative to the rest of the population. As the more successful individuals reproduce, occasional mutations give rise to new

individuals to be tested. Thus, as time passes, there is a change in the constitution of the population, i.e., the population is the unit of evolution.
This process is well captured by the intuitive metaphor of an adaptive landscape. On this landscape the height dimension belongs to fitness: high altitude stands for high fitness. The other two (or more, in the general case) dimensions correspond to biological traits. A given population can be plotted as a set of points on this landscape, where each dot is one individual realising a possible trait combination. Evolution is then the process of gradual advances of the population to high-altitude areas, powered by variation and natural selection. This leads to the concept of multimodal problems, in which there are a number of points that are better than all their neighbouring solutions. We call each of these points a local optimum and denote the highest of these as the global optimum. A problem in which there is only one local optimum is known as unimodal.
Evolution is not a unidirectional uphill process. Because the population has a finite size, and random choices are made in the selection and variation operators, it is common to observe the phenomenon of genetic drift, whereby highly fit individuals may be lost from the population, or the population may suffer from a loss of variety concerning some traits. **Effect:** populations 'melt down' the hill, and enter low-fitness valleys. The combined global effects of drift and selection enable populations to move uphill as well as downhill, and of course there is no guarantee that the population will climb back up the same hill.

2.3.2 Genetics

The fundamental observation from genetics is that each individual is a dual entity: its phenotypic properties (outside) are represented at a genotypic level (inside). In other words, an individual's geno- type encodes its phenotype. Genes are the functional units of inheritance encoding phenotypic characteristics. It is important to distinguish genes and alleles. An allele is one of the possible values that a gene can have - so its relationship to a gene is just like that of a specific value to a variable in mathematics. It is important to understand that all variations (mutation and recombination) happen at the genotypic level, while selection is based on actual performance in a given environment, that is, at the phenotypic level. In natural systems the genetic encoding is not one-to-one: one gene might affect more phenotypic traits (pleitropy), and in turn one phenotypic trait can be determined by more than one gene (polygeny). Phenotypic variations are always caused by genotypic variations, which in turn are the consequences of mutations of genes, or recombination of genes by sexual reproduction.
The term genome stands for the complete genetic information of a living being containing its total building plan. This genetic material is arranged in several chromosomes. Higher life forms (many plants and animals) contain a double complement of chromosomes in most of their cells, and such cells – and the host organisms – are called diploid. Gametes (i.e., sperm and egg cells) contain only one single complement of chromosomes and are called haploid. The combination of paternal and maternal features in the offspring of diploid organisms is a consequence of fertilisation by a fusion of such gametes. The new organism develops by the process named ontogenesis, which does not change the genetic information of the cells.
In evolutionary computing, the combination of features from two individuals in offspring is often called crossover. Crossing-over is not a process during mating and fertilisation, but rather happens during the formation of gametes, a process called meiosis. Meiosis is a special type of cell division that ensures that gametes contain only one copy of each chromosome.

2.3.3 Putting it together

Any living being is a dual entity with an invisible code (its genotype) and observable traits (its phenotype). Its success in surviving and reproducing is determined by its phenotypical properties. In other words, the forces known as natural selection and sexual selection act on phenotype level. Obviously, selection also affects the genotype level, albeit implicitly. The key here is reproduction. New individuals may have one single parent (asexual reproduction) or two parents (sexual reproduction). In either case, the genome of the new individual is not identical to that of the parent(s), because of small reproductive variations and because the combina- tion of two parents will differ from both. In this way genotype variations are created, which in turn translate to phenotype variations2 and thus are subject to selection. Hence, at a second level, genes are also subject to the game of survival and reproduction.

2.4 Evolutionary computing: why?

- Computerisation in the second half of the 20^{th} century created a growing demand for problem-solving automation. There is a need for algorithms that are applicable to a wide range of problems, do not need much tailoring for specific problems, and deliver good (not necessarily optimal) solutions within acceptable time. Evolutionary algorithms do all this, and so provide an answer to the challenge of deploying automated solution methods for more and more problems, which are ever more complex, in less and less time
- To understand how evolution works. From this perspective, evolutionary computing represents the possibility of performing experiments differently from traditional biology. Evolutionary processes can be simulated in a computer, where millions of generations can be executed in a matter of hours or days and repeated under various circumstances

3 What is an evolutionary algorithm?

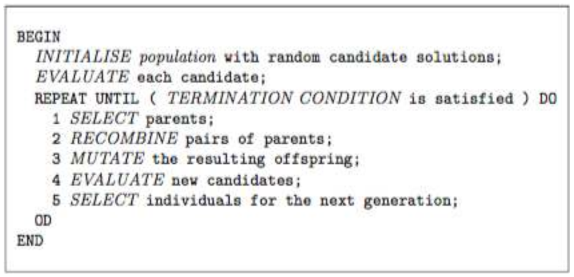
The most important aim of this chapter is to describe what an evolutionary algorithm is. This chapter presents a general scheme that forms the common basis for all the different variants of evolutionary algorithms.

3.1 What is an evolutionary algorithm?

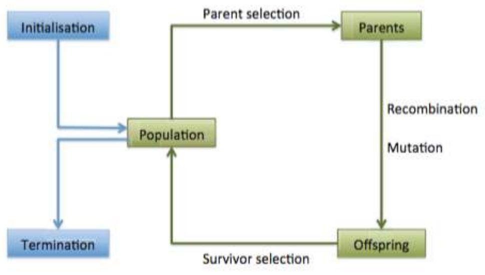
Given a population of individuals within some environment that has limited resources, competition for those resources causes natural selection (survival of the fittest). This in turn causes a rise in the fitness of the population. Given a quality function to be maximised, we can randomly create a set of candidate solutions, i.e., elements of the function's domain. We then apply the quality function to these as an abstract fitness measure – the higher the better. On the basis of these fitness values some of the better candidates are chosen to seed the next generation. This is done by applying recombination and/or mutation to them. Recombination is an operator that is applied to two or more selected candidates (the so-called parents), producing one or more new candidates (the children). Mutation is applied to one candidate and results in one new candidate. Therefore executing the operations of re- combination and mutation on the parents leads to the creation of a set of new candidates (the offspring). These have their fitness evaluated and then compete – based on their fitness (and possibly age) – with the old ones for a place in the next generation. This process can be iterated until a candidate with sufficient quality (a solution) is found or a previously set computational limit is reached.
Two main forces that form the basis of evolutionary systems:

- Variation operators (recombination and mutation); create diversity within the population, and thereby facilitate novelty
- Selection; acts as a force increasing the mean quality of solutions in the population

The combined application of variation and selection generally leads to improving fitness values in consecutive populations. It is easy to view this process as if evolution is optimising the fitness function. An alternative view is that evolution may be seen as a process of adaptation. From this perspective, the fitness is not seen as an objective function to be optimised, but as an expression of environmental requirements. Matching these requirements more closely implies an increased viability, which is reflected in a higher number of offspring. The evolutionary process results in a population which is increasingly better adapted to the environment.
It should be noted that many components of such an evolutionary process are stochastic. The general scheme of an evolutionary algorithm is given in pseudocode and is shown as a flowchart.



```
BEGIN
  INITIALISE population with random candidate solutions;
  EVALUATE each candidate;
  REPEAT UNTIL ( TERMINATION CONDITION is satisfied ) DO
    1 SELECT parents;
    2 RECOMBINE pairs of parents;
    3 MUTATE the resulting offspring;
    4 EVALUATE new candidates;
    5 SELECT individuals for the next generation;
  OD
END
```



Evolutionary algorithms possess a number of features that can help position them within the family of generate-and-test methods:

- EAs are population based, i.e., they process a whole collection of candidate solutions simultaneously
- Most EAs use recombination, mixing information from two or more candidate solutions to create a new one
- EAs are stochastic

The various dialects of evolutionary computing we have mentioned previously all follow these general outlines, differing only by the representation of a candidate solution – that is to say the data structures used to encode candidates. Typically this has the form of strings over a finite alphabet in genetic algorithms (GAs), real-valued vectors in evolution strategies (ESs), finite state machines in classical evolutionary programming (EP), and trees in genetic programming (GP). Technically, one representation might be preferable to others if it matches the given problem better; that is, it makes the encoding of candidate solutions easier or more natural. It is important to note two points. First, the recombination and mutation operators working on candidates must match the given representation. Second, in contrast to variation operators, the selection process only takes fitness information into account, and so it works independently from the choice of representation.

3.2 Components of evolutionary algorithms

There are a number of components, procedures, or operators that must be specified in order to define a particular EA. To create a complete, runnable algorithm, it is necessary to specify each component and to define the initialisation procedure. If we wish the algorithm to stop at some stage, we must also provide a termination condition.

3.2.1 Representation (Definition of individuals)

The first design step is called representation; mapping from the phenotypes (possible solutions within the original problem context) onto a set of genotypes (encoding of the phenotypes, individuals within the EA) that are said to represent them. It is important to understand that the phenotype space can be very different from the genotype space, and that the whole evolutionary search takes place in the genotype space.
It should be noted that the word 'representation' is used in two slightly different ways.
- Sometimes it stands for the mapping from the phenotype to the genotype space (=encoding). The inverse mapping from genotypes to phenotypes is usually called decoding, and it is necessary that the representation should be invertible so that for each genotype there is at most one corresponding phenotype.
- It can also be used, where the emphasis is not on the mapping itself, but on the data structure of the genotype space.

3.2.2 Evaluation function (Fitness function)

The role of the evaluation function is to represent the requirements the population should adapt to meet. It forms the basis for selection, and so it facilitates improvements; it defines what improvement means.
The evaluation function is commonly called the fitness function in EC.

3.2.3 Population

The role of the population is to hold (the representation of) possible solutions. A population is a multiset of genotypes. The population forms the unit of evolution. Individuals are static objects that do not change or adapt; it is the population that does. Given a representation, defining a population may be as simple as specifying how many individuals are in it, that is, setting the population size.
The population size is constant, which produces the limited resources need to create competition. The selection operators (parent selection and survivor selection) work at the population level. In general, they take the whole current population into account, and choices are always made relative to what is currently present. This population level activity is in contrast to variation operators, which act on one or more parent individuals.
The diversity of a population is a measure of the number of different solutions present. Note that the presence of only one fitness value in a population does not necessarily imply that only one phenotype is present, since many phenotypes may have the same fitness. Equally, the presence of only one phenotype does not necessarily imply only one genotype. However, if only one genotype is present then this implies only one phenotype and fitness value are present.

3.2.4 Parent selection mechanism

The role of parent selection or mate selection is to distinguish among individuals based on their quality, and, in particular, to allow the better individuals to become parents of the next generation. An individual is a parent if it has been selected to undergo variation in order to create offspring. Together with the survivor selection mechanism, parent selection is responsible for pushing quality improvements. In EC, parent selection is typically probabilistic. Thus, high-quality individuals have more chance of becoming parents than those with low quality.

3.2.5 Variation operators (Mutation and recombination)

The role of variation operators is to create new individuals from old ones.
Mutation
A unary variation operator is commonly called mutation. It is applied to one genotype and delivers a (slightly) modified mutant, the child or offspring. A mutation operator is always stochastic: its output − the child − depends on the outcomes of a series of random choices.
Variation operators form the evolutionary implementation of elementary (search) steps, giving the search space its topological structure.
Recombination
A binary variation operator is called recombination or crossover. Such an operator merges information from two parent genotypes into one or two offspring genotypes. Like mutation, recombination is a stochastic operator: the choices of what parts of each parent are combined, and how this is done, depend on random drawings.
The principle behind recombination is simple − by mating two individuals with different but desirable features, we can produce an offspring that combines both of those features. Recombination operators in EAs are usually applied probabilistically, that is, with a nonzero chance of not being performed.
It is important to remember that variation operators are representation dependent.

3.2.6 Survivor selection mechanism (Replacement)

The role of survivor selection or environmental selection is to distinguish among individuals based on their quality. The survivor selection mechanism is called after the creation of the offspring from the selected parents. Survivor selection is often deterministic.

3.2.7 Initialisation

Initialisation is kept simple in most EA applications; the first population is seeded by randomly generated individuals.

3.2.8 Termination condition

We can distinguish two cases of a suitable termination condition. If the problem has a known optimal fitness level, probably coming from a known optimum of the given objective function, then in an ideal world our stopping condition would be the discovery of a solution with this fitness. If we know that our model of the real-world problem contains necessary simplifications, or may contain noise, we may accept a solution that reaches the optimal fitness to within a given precision $\epsilon > 0$. However, EAs are stochastic and mostly there are no guarantees of reaching such an optimum, so this condition might never get satisfied, and the algorithm may never stop. Therefore we must extend this condition with one that certainly stops the algorithm.

1. The maximally allowed CPU time elapses
2. The total number of fitness evaluations reaches a given limit
3. The fitness improvement remains under a threshold value for a given period of time
4. The population diversity drops under a given threshold

3.3 An evolutionary cycle by hand

To illustrate the working of an EA, we show the details of one selection–reproduction cycle on a simple problem, that of maximising the values of x^2 for integers in the range 0–31. To execute a full evolutionary cycle, we must make design decisions regarding the EA components representation, parent selection, recombination, mutation, and survivor selection.

- Representation; simple five-bit binary encoding mapping integers (phenotypes) to bit-strings (genotypes)
- Parent selection; fitness proportional mechanism, where the probability p_i that an individual i in population P is chosen to be a parent is $p_i = \frac{f(i)}{\sum_{j \in P} f(j)}$.
- Survivor selection; all existing individuals are removed from the population and all new individuals are added to it without comparing fitness values
- Mutation; generating a random number (from a uniform distribution over the range [0, 1]) in each bit position, and comparing it to a fixed threshold, usually called the mutation rate. If the random number is below that rate, the value of the gene in the corresponding position is flipped
- Recombination; one-point crossover. This operator is applied to two parents and produces two children by choosing a random crossover-point along the strings and swapping the bits of the parents after this point

String no.	Initial population	x Value	Fitness $f(x) = x^2$	$Prob_i$	Expected count	Actual count
1	0 1 1 0 1	13	169	0.14	0.58	1
2	1 1 0 0 0	24	576	0.49	1.97	2
3	0 1 0 0 0	8	64	0.06	0.22	0
4	1 0 0 1 1	19	361	0.31	1.23	1
Sum			1170	1.00	4.00	4
Average			293	0.25	1.00	1
Max			576	0.49	1.97	2

Table 3.1. The x^2 example, 1: initialisation, evaluation, and parent selection

After having made the essential design decisions, we can execute a full selection–reproduction cycle. Table 3.1 shows a random initial population of four genotypes, the corresponding phenotypes, and their fitness values. The cycle then starts with selecting the parents to seed the next generation. The fourth column of Table 3.1 shows the expected number of copies of each individual after parent selection, being $f_i/\bar{f}$, where $\bar{f}$ denotes the average fitness (displayed values are rounded up). As can be seen, these numbers are not integers; rather they represent a probability distribution, and the mating pool is created by making random choices to sample from this distribution. The column "Actual count" stands for the number of copies in the mating pool, i.e., it shows one possible outcome.

String no.	Mating pool	Crossover point	Offspring after xover	x Value	Fitness $f(x) = x^2$
1	0 1 1 0 | 1	4	0 1 1 0 0	12	144
2	1 1 0 0 | 0	4	1 1 0 0 1	25	625
2	1 1 | 0 0 0	2	1 1 0 1 1	27	729
4	1 0 | 0 1 1	2	1 0 0 0 0	16	256
Sum					1754
Average					439
Max					729

Table 3.2. The x^2 example, 2: crossover and offspring evaluation

String no.	Offspring after xover	Offspring after mutation	x Value	Fitness $f(x) = x^2$
1	0 1 1 0 0	1 1 1 0 0	26	676
2	1 1 0 0 1	1 1 0 0 1	25	625
2	1 1 0 1 1	1 1 0 1 1	27	729
4	1 0 0 0 0	1 0 1 0 0	18	324
Sum				2354
Average				588.5
Max				729

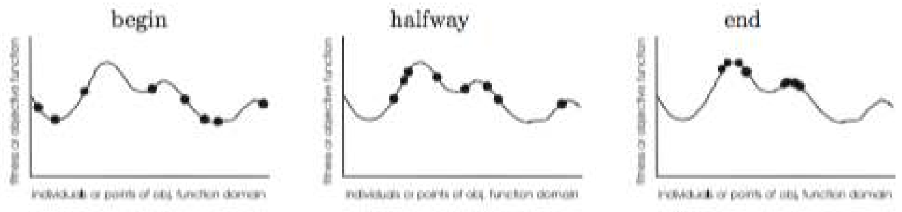
Table 3.3. The x^2 example, 3: mutation and offspring evaluation

Next the selected individuals are paired at random, and for each pair a random point along the string is chosen. Table 3.2 shows the results of crossover on the given mating pool for crossover points after the fourth and second genes, respectively, together with the corresponding fitness values. Mutation is applied to the offspring delivered by crossover. Once again, we show one possible outcome of the random drawings, and Table 3.3 shows the hand-made 'mutants'. In this case, the mutations shown happen to have caused positive changes in fitness, but we should emphasise that in later generations, as the number of 1's in the population rises, mutation will be on average (but not always) deleterious.

3.4 The operation of an evolutionary algorithm

Evolutionary algorithms have some rather general properties concerning how they work. To illustrate how an EA typically works, we will assume a one- dimensional objective function to be maximised.
Three stages of evolutionary search:

- In the first stage directly after initialisation, the individuals are randomly spread over the whole search space
- After only a few generations this distribution changes: because of selection and variation operators the population abandons low-fitness regions and starts to climb the hills
- Yet later (close to the end of the search, if the termination condition is set appropriately), the whole population is concentrated around a few peaks, some of which may be suboptimal



Exploration is the generation of new individuals in as-yet untested regions of the search space, while exploitation means the concentration of the search in the vicinity of known good solutions. Evolutionary search processes are often referred to in terms of a trade-off between exploration and exploitation. Premature convergence is the well-known effect of losing population diversity too quickly, and getting trapped in a local optimum.

3.5 Natural vs artificial evolution

It could be argued that evolutionary algorithms are not faithful models of natural evolution. However, they certainly are a form of evolution. As phrased by Dennett: If you have variation, heredity, and selection, then you must get evolution. In the table below, we compare natural evolution and artificial evolution as used in contemporary evolutionary algorithms.

	Natural evolution	Artificial evolution
Fitness	Observed quantity: *a posteriori* effect of selection ('in the eye of the observer').	Predefined *a priori* quantity that drives selection.
Selection	Complex multifactor force based on environmental conditions, other individuals of the same species and other species (e.g., predators). Viability is tested continually; reproducibility is tested at discrete times.	Randomized operator with selection probabilities based on given fitness values. Parent selection and survivor selection both happen at discrete times.
Genotype-phenotype mapping	Highly complex biochemical process influenced by the environment.	Relatively simple mathematical transformation or parameterised procedure.
Variation	Offspring created from one (asexual reproduction) or two parents (sexual reproduction).	Offspring may be generated from one, two, or many parents.
Execution	Parallel, decentralized execution; birth and death events are not synchronised.	Typically centralized with synchronised birth and death.
Population	Spatial embedding implies structured populations. Population size varies according to the relative number of death and birth events.	Typically unstructured and panmictic (all individuals are potential partners). Population size is kept constant by synchronising time and number of birth and death events.

3.6 Evolutionary computing, global optimisation, and other search algorithms

In an ideal world, we would possess the technology and algorithms that could provide a provably optimal solution to any problem that we could suit- ably pose to the system.
Decades of computer science research have taught us that many real-world problems can be reduced in their essence to well-known abstract forms, for which the number of potential solutions grows very quickly with the number of variables considered. For some of these abstract problems exact methods are known whose time complexity scales linearly (or at least polynomially) with the number of variables. However, it is widely accepted that for many types of problems encountered, no such algorithms exist.
The term global optimisation refers to the process of attempting to find the solution with the optimal value for some fitness function.
Examples of algorithms that will find a solution to maximisation problems:

- *Deterministic algorithms; complete enumeration or branch and bound*
- *Heuristics; simulated annealing or combinatorial optimisation*
- *Local-search algorithms*

The ability of EAs to maintain a diverse set of points provides not only a means of escaping from local optima, but also a means of coping with large and discontinuous search spaces.

4 Representation, mutation and recombination

There are two fundamental forces that form the basis of evolutionary systems: variation and selection. In this chapter we discuss the EA components behind the first one.

4.1 Representation and the roles of variation operators

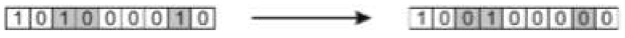
The first stage of building any evolutionary algorithm is to decide on a genetic representation of a candidate solution to the problem. This involves defining the genotype and the mapping from genotype to phenotype. Mutation is the generic name given to those variation operators that use only one parent and create one child by applying some kind of randomised change to the representation (genotype). The form taken depends on the choice of encoding used, as does the meaning of the associated parameter, which is often introduced to regulate the intensity or magnitude of mutation.
Recombination (crossover) is the process whereby a new individual solution is created from the information contained within two (or more) parent solutions. Crossover tends to refer to the most common two-parent case. Recombination operators are usually applied probabilistically according to a crossover rate p_c. Usually two parents are selected and two offspring are created via recombination of the two parents with probability p_c; or by simply copying the parents, with probability $1 - p_c$.

4.2 Binary representation

The genotype consists simply of a string of binary digits - a bit-string.
For a particular application we have to decide how long the string should be, and how we will interpret it to produce a phenotype. In choosing the genotype–phenotype mapping for a specific problem, one has to make sure that the encoding allows that all possible bit strings denote a valid solution to the given problem and that, vice versa, all possible solutions can be represented.
One of the problems of coding numbers in binary is that different bits have different significance, and so the effect of a single bit mutation is very variable.

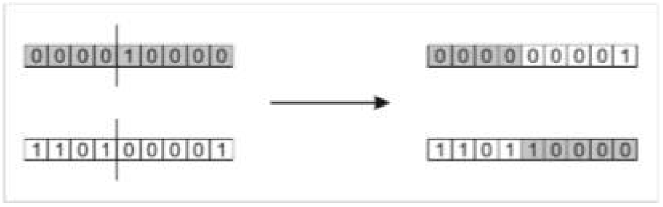
4.2.1 Mutation for binary representation

Bit-flip considers each gene separately and allows each bit to flip (i.e., from 1 to 0 or 0 to 1) with a small probability p_m. The actual number of values changed is thus not fixed, but depends on the sequence of random numbers drawn.

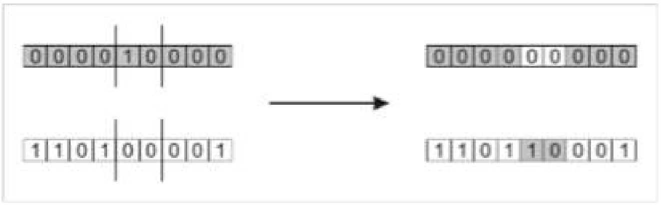


4.2.2 Recombination for binary representation

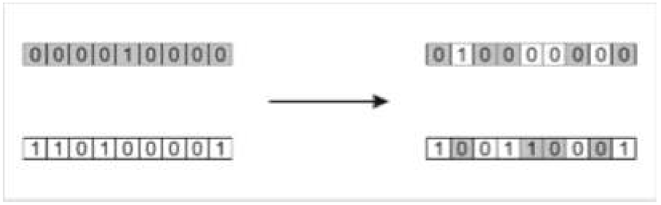
They all start from two parents and create two children.
One-point crossover works by choosing a random number r in the range $[1, l-1]$ (with l the length of the encoding), and then splitting both parents at this point and creating the two children by exchanging the tails.



***n*-point crossover** the chromosome is broken into more than two segments of contiguous genes, and the offspring are created by taking alter- native segments from the parents.



Uniform crossover works by treating each gene independently and making a random choice as to which parent it should be inherited from. This is implemented by generating a string of l random variables from a uniform distribution over [0,1]. In each position, if the value is below a parameter p (usually 0.5), the gene is inherited from the first parent; otherwise from the second. The second offspring is created using the inverse mapping.



4.3 Integer representation

The genotype consists of a string of interger values.
When designing the encoding and variation operators, it is worth considering whether there are any natural relations between the possible values that an attribute can take. This might be obvious for ordinal attributes such as integers (2 is more like 3 than it is 389), but for cardinal attributes such as the compass points, there may not be a natural ordering.

4.3.1 Mutation for integer representation

Both methods mutate each gene independently with user-defined probability p_m.
Random resetting the bit-flipping mutation of binary encodings is extended to random resetting: in each position independently, with probability p_m, a new value is chosen at random from the set of permissible values (designed for cardinal attributes). **Creep mutation** works by adding a small (positive or negative) value to each gene with probability p. Usually these values are sampled randomly for each position, from a distribution that is symmetric about zero, and is more likely to generate small changes than large ones (designed for ordinal atrributes).

4.3.2 Recombination for integer representation

For representations where each gene has a finite number of possible allele values (such as integers) it is normal to use the same set of operators as for binary representations.

4.4 Real-valued or floating-point representation

Genotype consists of a string of real values, this occurs when the values that we want to represent as genes come from a continuous rather than a discrete distribution. The genotype for a solution with k genes is now a vector $\langle x_1, ..., x_k \rangle$ with $x_i \in \mathbb{R}$.

4.4.1 Mutation for real-valued representation

Change the allele value of each gene randomly within its domain given by a lower L_i and upper U_i bound, resulting in the following transformation:

$$\langle x_1, ..., x_n \rangle \rightarrow \langle x_1', ..., x_n' \rangle, \text{where } x_i, x_i' \in [L_i, U_i]$$

Uniform mutation the values of x_i' are drawn uniformly randomly from $[L_i, U_i]$. Normally a positionwise mutation probability is used.
Nonuniform mutation works by adding to the current gene value an amount drawn randomly from a Gaussian distribution with mean zero and user-specified standard deviation, and then curtailing the resulting value to the range $[L_i, U_i]$ if necessary.

4.4.2 Self-adaptive mutation for real-valued representation

The concept of self-adaptation represents a solution to the problem of how to adapt the step-sizes. The essential feature is that the step sizes are also included in the chromosomes and they themselves undergo variation and selection.
The key concept is that the mutation step sizes are not set by the user; rather the σ co-evolves with the solutions (the $\bar{x}$ part). In order to achieve this behaviour it is essential to modify the value of σ first, and then mutate the x_i values with the new σ value.
Uncorrelated mutation with one step size the same distribution is used to mutate each x_i, therefore we only have one strategy parameter σ in each individual. This σ is mutated each time step by multiplying it by a term e^{Γ}, with Γ a random variable drawn each time from a normal distribution with mean 0 and standard deviation τ (can be interpreted as a learning rate). Since N(0,τ) = τ · N(0,1), the mutation mechanism is thus specified by the following formulas:

$$\sigma' = \sigma \times e^{\tau \times N(0,1)},$$
$$x_i' = x_i + \sigma' \times N_i(0,1)$$

Furthermore, since standard deviations very close to zero are unwanted (they will have on average a negligible effect), the following boundary rule is used to force step sizes to be no smaller than a threshold:

$$\sigma' < \varepsilon_0 \Rightarrow \sigma' = \varepsilon_0$$

Uncorrelated mutation with n step size The motivation behind using n step sizes is the wish to treat dimensions differently. In particular, we want to be able to use different step sizes for different dimensions $i \in \{1, ..., n\}$. The reason for this is the trivial observation that the fitness landscape can have a different slope in one direction (along axis i) than in another direction (along axis j). The mutation mechanism is now specified as follows:

$$\sigma_i' = \sigma_i \times e^{\tau' \times N(0,1) + \tau \times N_i(0,1)},$$
$$x_i' = x_i + \sigma_i' \times N_i(0,1)$$

Once again a boundary rule is applied to prevent standard deviations very close to zero.

$$\sigma_i' < \varepsilon_0 \Rightarrow \sigma_i' = \varepsilon_0$$

Correlated mutation The rationale behind correlated mutations is to allow the ellipses to have any orientation by rotating them with a rotation (covariance) matrix C.

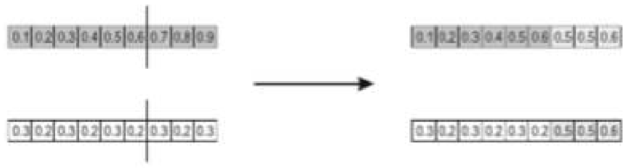
Theoretical and experimental results agree on the fact that for a successful run the σ values must decrease over time. The intuitive explanation for this is that in the beginning of a search process a large part of the search space has to be sampled in an explorative fashion to locate promising regions (with good fitness values). Therefore, large mutations are appropriate in this phase. As the search proceeds and optimal values are approached, only fine tuning of the given individuals is needed; thus smaller mutations are required.

4.4.3 Recombination operators for real-valued representation

First, using an analogous operator to those used for bit-strings, but now split between floats - discrete recombination; if we are creating an offspring z from parents x and y, then the allele value for gene i is given by $z_i = x_i$ or y_i with equal likelihood.
Second, using an operator that, in each gene position, creates a new allele value in the offspring that lies between those of the parents - intermediate or arithmetic recombination; $z_i = \alpha x_i + (1 - \alpha)y_i$.
Third, using an operator that in each position creates a new allele value in the offspring which is close to that of one of the parents, but may lie outside them - blend recombination.
Simple arithmetic recombination First pick a recombination point k. Then, for child 1, take the first k floats of parent 1 and put them into the child. The rest is the arithmetic average of parent 1 and 2:

$$\text{Child 1: } \langle x_1, ..., x_k, \alpha \times y_{k+1} + (1 - \alpha) \times x_{k+1}, ..., \alpha \times y_n + (1 - \alpha) \times x_n \rangle$$

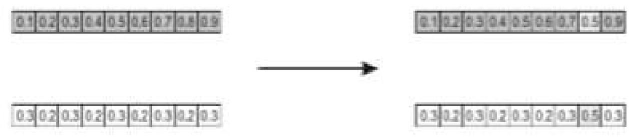
Child 2 is analogous, with x and y reversed.



Single arithmetic recombination Pick a random allele k. At that position, take the arithmetic average of the two parents. The other points are the points from the parents, i.e.:

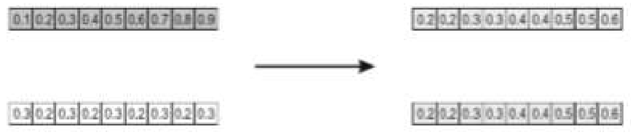
$$\text{Child 1: } \langle x_1, ..., x_{k-1}, \alpha \times y_k + (1 - \alpha) \times x_k, x_{k+1}, ..., x_n \rangle$$

The second child is created in the same way with x and y reversed.



Whole arithmetic recombination works by taking the weighted sum of the two parental alleles for each gene, i.e.:

$$\text{Child 1: } \alpha \times \bar{x} + (1 - \alpha) * \bar{y},$$
$$\text{Child 2: } \alpha \times \bar{y} + (1 - \alpha) * \bar{x}$$



Blend crossover If we have two parents x and y and assume that in position i the value $x_i < y_i$ then the difference $d_i = y_i - x - i$ and the range for the ith value in the child z is $[x_i - \alpha \times d_i, x_i + \alpha \times d_i]$. To create a child we can sample a random number u uniformly from [0, 1], calculate $\gamma = (1 - 2\alpha)u - \alpha$, and set:

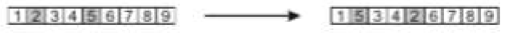
$$z_i = (1 - \gamma)x_i + \gamma y_i$$

4.5 Permutation representation

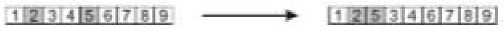
Many problems naturally take the form of deciding on the order in which a sequence of events should occur, the most natural representation of such problems is as a permutation of a fixed set of values that can be represented as integers.
Important property: each possible allele value occurs exactly once in the solution.
There are two classes of problems that are represented by permutations. In the first of these, the order in which events occur is important. Another type of problem depends on adjacency.
There are two possible ways to encode a permutation. In the first (most commonly used) of these the ith element of the representation denotes the event that happens in that place in the sequence (or the ith destination visited). In the second, the value of the ith element denotes the position in the sequence in which the ith event happens.

4.5.1 Mutation for permutation representation

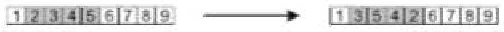
Dinding legal mutations is a matter of moving alleles around in the genome. This has the immediate consequence that the mutation parameter is interpreted as the probability that the chromosome undergoes mutation, rather than that a single gene in the chromosome is altered.
Swap mutation Two positions (genes) in the chromosome are selected at random and their allele values swapped



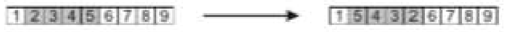
Insert mutation Two alleles are selected at random and the second moved next to the first, shuffling along the others to make room



Scramble mutation Here the entire chromosome, or some randomly chosen subset of values within it, have their positions scrambled



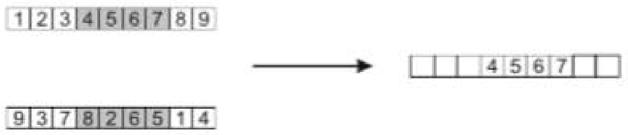
Inversion mutation works by randomly selecting two positions in the chromosome and reversing the order in which the values appear between those positions



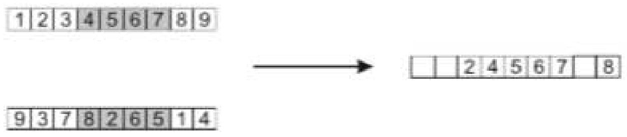
4.5.2 Recombination for permutation representation

Partially mapped crossover

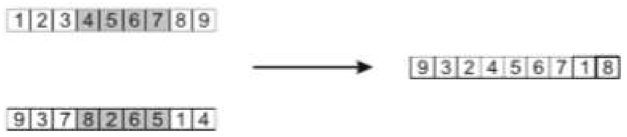
1. Choose two crossover points at random, and copy the segment between them from the first parent (P1) into the first offspring



2. Starting from the first crossover point look for elements in that segment of the second parent (P2) that have not been copied.
3. For each of these (say i), look in the offspring to see what element (say j) has been copied in its place from P1.
4. Place i into the position occupied by j in P2, since we know that we will not be putting j there (as we already have it in our string).

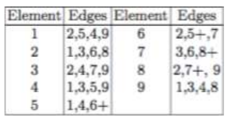


5. If the place occupied by j in P2 has already been filled in the offspring by an element k, put i in the position occupied by k in P2.
6. Having dealt with the elements from the crossover segment, the remaining positions in this offspring can be filled from P2, and the second child is created analogously with the parental roles reversed.

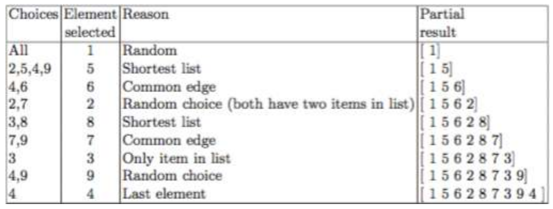


Edge crossover is based on the idea that offspring should be created as far as possible using only edges that are present in (one of) the parents. In order to achieve this, an edge table (also known as an adjacency list) is constructed, which for each element lists the other elements that are linked to it in the two parents. A '+' in the table indicates that the edge is present in both parents. The operator works as follows:

1. Construct the edge table
2. Pick an initial element at random and put it in the offspring
3. Set the variable current element = entry
4. Remove all references to current element from the table
5. Examine list for current element
   - If there is a common edge, pick that to be the next element
   - Otherwise pick the entry in the list which itself has the shortest list
   - Ties are split at random
6. In the case of reaching an empty list, the other end of the offspring is examined for extension; otherwise a new element is chosen at random



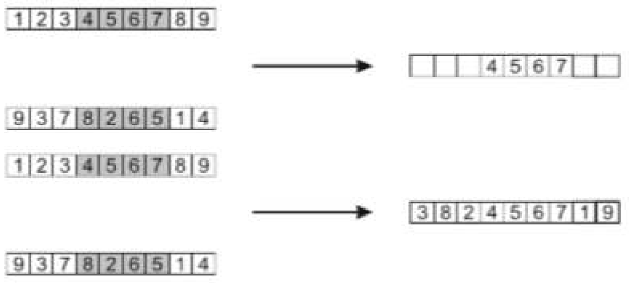
Element	Edges	Element	Edges
1	2,5,4,9	6	2,5+,7
2	1,3,6,8	7	3,6,8+
3	2,4,7,9	8	2,7+, 9
4	1,3,5,9	9	1,3,4,8
5	1,4,6+		



Choices	Element selected	Reason	Partial result
All	1	Random	[1]
2,5,4,9	5	Shortest list	[1 5]
4,6	6	Common edge	[1 5 6]
2,7	2	Random choice (both have two items in list)	[1 5 6 2]
3,8	8	Shortest list	[1 5 6 2 8]
7,9	7	Common edge	[1 5 6 2 8 7]
3	3	Only item in list	[1 5 6 2 8 7 3]
4,9	9	Random choice	[1 5 6 2 8 7 3 9]
4	4	Last element	[1 5 6 2 8 7 3 9 4]

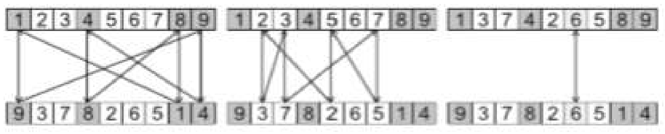
Order crossover begins by copying a randomly chosen segment of the first parent into the offspring. Proceeds as follows:

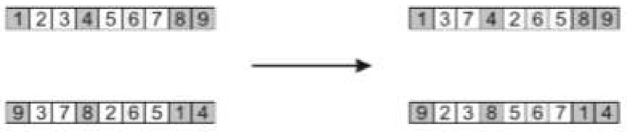
1. Choose two crossover points at random, and copy the segment between them from the first parent (P1) into the first offspring.
2. Starting from the second crossover point in the second parent, copy the remaining unused numbers into the first child in the order that they appear in the second parent, wrapping around at the end of the list.
3. Create the second offspring in an analogous manner, with the parent roles reversed.



Cycle crossover is concerned with preserving as much information as possible about the absolute position in which elements occur. The operator works by dividing the elements into cycles. A cycle is a subset of elements that has the property that each element always occurs paired with another element of the same cycle when the two parents are aligned. Having divided the permutation into cycles, the offspring are created by selecting alternate cycles from each parent. The procedure for constructing cycles is as follows:

1. Start with the first unused position and allele of P1
2. Look at the allele in the same position in P2
3. Go to the position with the same allele in P1
4. Add this allele to the cycle
5. Repeat steps 2 through 4 until you arrive at the first allele of P1



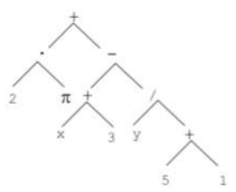


4.6 Tree representation

Trees are among the most general structures for representing objects in computing, and form the basis for the branch of evolutionary algorithms known as genetic programming (GP).

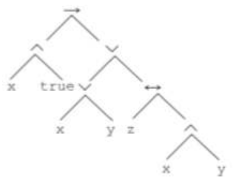
An arithmetic formula:

$$2 \times \pi + ((x + 3) - \frac{y}{5 + 1})$$



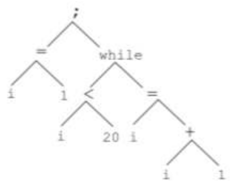
A logical formula:

$$(x \wedge true) \rightarrow ((x \vee y) \vee (z \leftrightarrow (x \wedge y)))$$



The following program:

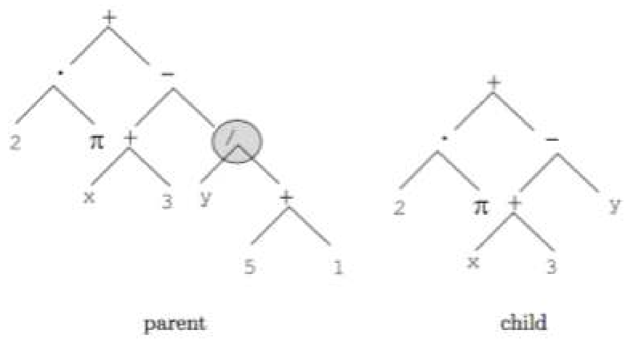
```
           i = 1;
while (i <20) {
        i = i+1; }
```



The specification of how to represent individuals boils down to defining the syntax of the trees, or equivalently the syntax of the symbolic expressions (s-expressions) they represent. This is commonly done by defining a function set and a terminal set. Elements of the terminal set are allowed as leaves, while symbols from the function set are internal nodes.

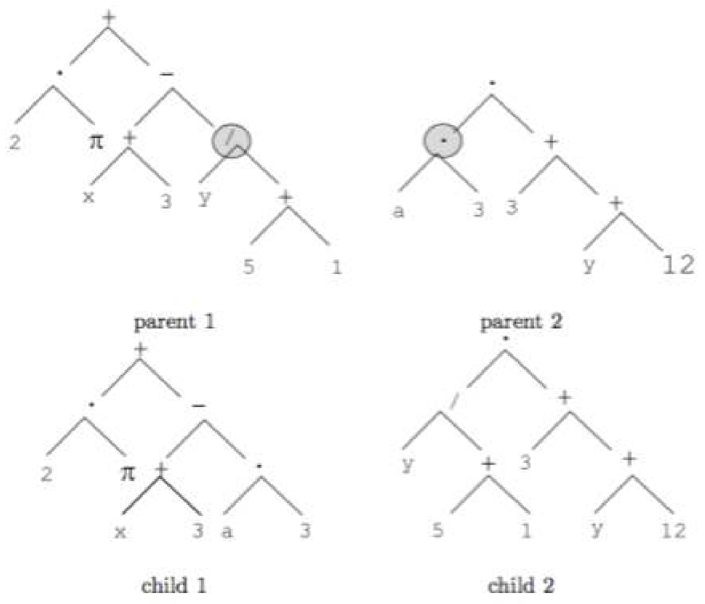
4.6.1 Mutation for tree representation

Tree-based mutation works by selecting a node at random from the tree, and replacing the sub-tree starting there with a randomly generated tree. This newly created sub-tree is usually generated the same way as in the initial population, and so is subject to conditions on maximum depth and width. Tree-based mutation has two parameters: the probability of choosing mutation at the junction with recombination, and the probability of choosing an internal point within the parent as the root of the sub-tree to be replaced.



4.6.2 Recombination for tree representation

Tree-based recombination creates offspring by swapping genetic material among the selected parents. In technical terms, it is a binary operator creating two child trees from two parent trees. The most common implementation is sub-tree crossover, which works by interchanging the sub-trees starting at two randomly selected nodes in the given parents. Note that the size (tree depth) of the children can exceed that of the parent trees. Tree-based recombination has two parameters: the probability of choosing recombination at the junction with mutation, and the probability of choosing internal nodes as crossover points.



5 Fitness, selection, and population management

There are two fundamental forces that form the basis of evolutionary systems: variation and selection. In this chapter we discuss the EA components behind the second one.

5.1 Population management models

Two different models of population management are found in the literature: the generational model and the steady-state model.
Generational model
In each generation we begin with a population of size μ, from which a mating pool of parents is selected. Every member of the pool is a copy of something in the population, but the proportions will probably differ, with (usually) more copies of the 'better' parents. Next, λ offspring are created from the mating pool by the application of variation operators, and evaluated. After each generation, the whole population is replaced by μ individuals selected from its offspring, which is called the next generation.
Steady-state model
The entire population is not changed at once, but rather a part of it. In this case, λ ($<\mu$) old individuals are replaced by λ new ones, the offspring. The proportion of the population that is replaced is called the generational gap, and is equal to λ/μ.

The operators that are responsible for this competitive element of population management work on the basis of an individual's fitness. As a direct consequence, these selection and replacement operators work independently of the problem representation chosen.

5.2 Parent selection

5.2.1 Fitness proportional selection

Fitness proportional selection (FPS); for each choice, the probability that an individual i is selected for mating depends on its absolute fitness value compared to the absolute fitness values of the rest of the population.
Some problems with this selection mechanism:

- Outstanding individuals take over the entire population very quickly. This phenomenon is often observed in early generations, when many of the randomly created individuals will have low fitness, and is known as premature convergence
- When fitness values are all very close together, there is almost no selection pressure, so selection is almost uniformly random, and having a slightly better fitness is not very 'useful' to an individual. Therefore, later in a run, when some convergence has taken place and the worst individuals are gone, it is typically observed that the mean population fitness only increases very slowly
- The mechanism behaves differently if the fitness function is transposed

Transposing the fitness function changes the selection probabilities, while the shape of the fitness landscape, and hence the location of the optimum, remains the same.
To avoid the second two problems with FPS, a procedure known as windowing is often used. Under this scheme, fitness differentials are maintained by subtracted from the raw fitness $f(x)$ a value β^t, which depends in some way on the recent search history, and so can change over time.
Another well-known approach is sigma scaling, which incorporates information about the mean $\bar{f}$ and standard deviationσ_f of fitnesses in the population:

$$f'(x) = max(f(x) - (\bar{f} - c \times \sigma_f), 0),$$

where c is a constant value, usually set to 2.

5.2.2 Ranking selection

Rank-based selection preserves a constant selection pressure by sorting the population on the basis of fitness, and then allocating selection probabilities to individuals according to their rank, rather than according to their actual fitness values.
The usual formula for calculating the selection probability for linear ranking schemes is parameterised by a value s (1<s<2). The selection probability for an individual of rank i is:

$$P_{lin-rank}(i) = \frac{(2-s)}{\mu} + \frac{2i(s-1)}{\mu(\mu-1)}$$

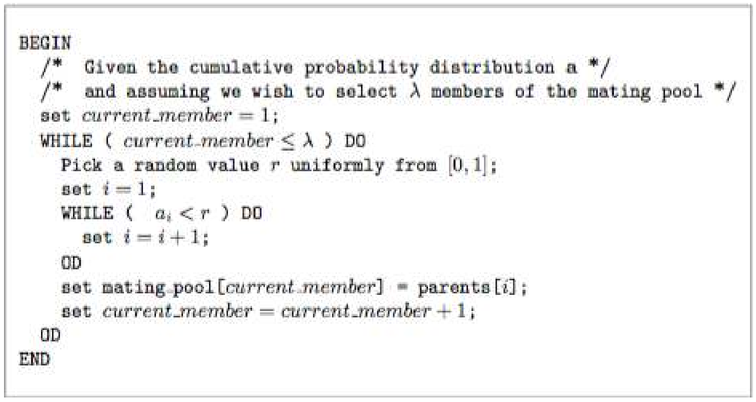
Note that the first term will be constant for all individuals (it is there to ensure the probabilities add to one). Since the second term will be zero for the worst individual (with rank $i = 0$), it can be thought of as the 'baseline' probability of selecting that individual.
When the mapping from rank to selection probabilities is linear, only limited selection pressure can be applied. This arises from the assumption that, on average, an individual of median fitness should have one chance to be reproduced, which in turn imposes a maximum value of $s = 2$. If a higher selection pressure is required, i.e., more emphasis on selecting individuals of above-average fitness, an exponential ranking scheme is often used, of the form:

$$P_{exp-rank}(i) = \frac{1-e^{-i}}{c}$$

The normalisation factor c is chosen so that the sum of the probabilities is unity, i.e., it is a function of the population size.

5.2.3 Implementing selection probabilities

In an ideal world, the mating pool of parents taking part in recombination would have exactly the same proportions as this selection probability distribution. This would mean that the number of any given individual would be given by its selection probability, multiplied by the size of the mating pool. However, in practice this is not possible be- cause of the finite size of the population, i.e., when we do this multiplication, we find typically that some individuals have an expected number of copies which is non-integer – whereas of course in practice we need to select complete individuals. In other words, the mating pool of parents is sampled from the selection probability distribution, but will not in general accurately reflect it.
The simplest way of achieving this sampling is by the roulette wheel algorithm. In general, the algorithm can be applied to select λ members from the set of μ parents into a mating pool. See figure below for the outlines of the algorithm.



```
BEGIN
  /*  Given the cumulative probability distribution a */
  /*  and assuming we wish to select $\lambda$ members of the mating pool */
  set $current\_member = 1$;
  WHILE ( $current\_member \leq \lambda$ ) DO
    Pick a random value $r$ uniformly from $[0,1]$;
    set $i = 1$;
    WHILE (  $a_i < r$ ) DO
      set $i = i + 1$;
    OD
    set mating_pool[$current\_member$] = parents[$i$];
    set $current\_member = current\_member + 1$;
  OD
END
```

Despite its inherent simplicity, it has been recognised that the roulette wheel algorithm does not in fact give a particularly good sample of the required distribution. Whenever more than one sample is to be drawn from the distribution – for instance λ – the use of the stochastic universal sampling (SUS) algorithm is preferred.

```
BEGIN
  /* Given the cumulative probability distribution a */
  /* and assuming we wish to select $\lambda$ members of the mating pool */
  set $current\_member = i = 1$;
  Pick a random value $r$ uniformly from $[0, 1/\lambda]$;
  WHILE ( $current\_member \le \lambda$ ) DO
    WHILE ( $r \le a[i]$ ) DO
      set mating_pool[$current\_member$] = parents[$i$];
      set $r = r + 1/\lambda$;
      set $current\_member = current\_member + 1$;
    OD
    set $i = i + 1$;
  OD
END
```

5.2.4 Tournament selection

Tournament selection is an operator with the useful property that it does not require any global knowledge of the population, nor a quantifiable measure of quality. Instead it only relies on an ordering relation that can compare and rank any two individuals.

```
BEGIN
  /* Assume we wish to select $\lambda$ members of a pool of $\mu$ individuals */
  set $current\_member = 1$;
  WHILE ( $current\_member \le \lambda$ ) DO
    Pick $k$ individuals randomly, with or without replacement;
    Compare these $k$ individuals and select the best of them;
    Denote this individual as $i$;
    set mating_pool[$current\_member$] = $i$;
    set $current\_member = current\_member + 1$;
  OD
END
```

Because tournament selection looks at relative rather than absolute fitness, it has the same properties as ranking schemes in terms of invariance to translation and transposition of the fitness function. The probability that an individual will be selected as the result of a tournament depends on four factors, namely:

- Its rank in the population
- The tournament size k. The larger the tournament, the greater the chance that it will contain members of above-average fitness, and the less that it will consist entirely of low-fitness members. Hence we say that increasing k increases the selection pressure
- The probability p that the most fit member of the tournament is selected. Usually this is 1 (deterministic tournaments), but stochastic versions are also used with p <1. Since this makes it more likely that a less-fit member will be selected, decreasing p will decrease the selection pressure
- Whether individuals are chosen with or without replacement

5.2.5 Uniform parent selection

In some dialects of EC it is common to use mechanisms such that each individual has the same chance to be selected.

5.2.6 Over-selection for large populations

In some cases it may be desirable to work with extremely large populations.
Regardless of the implementation details, if the potential search space is enormous it might be a good idea to

use a large population to avoid 'missing' promising regions in the initial random generation, and thereafter to maintain the diversity needed to support exploration. Often a method called over-selection is used for population sizes of 1000 and above.
In this method, the population is first ranked by fitness and then divided into two groups, the top x% in one and the remaining (100 - x)% in the other. When parents are selected, 80% of the selection operations choose from the first group, and the other 20% from the second.

5.3 Survivor selection

The survivor selection mechanism is responsible for managing the process of reducing the working memory of the EA from a set of μ parents and λ offspring to a set of μ individuals forming the next generation. In principle, any of the mechanisms introduced for parent selection could be also used for selecting survivors. This step in the main evolutionary cycle is also called replacement. Replacement strategies can be categorised according to whether they discriminate on the basis of the fitness or the age of individuals.

5.3.1 Age-based replacement

The basis of these schemes is that the fitness of individuals is not taken into account during the selection of which individuals to replace in the population. Instead, they are designed so that each individual exists in the population for the same number of EA iterations.
Since the number of offspring produced is the same as the number of parents ($\mu = \lambda$), each individual exists for just one cycle, and the parents are simply discarded, to be replaced by the entire set of offspring. This is the generational model, but in fact this replacement strategy can also be implemented in a steady-state with overlapping populations ($\lambda < \mu$), right to the other extreme where a single offspring is created and inserted in the population in each cycle. In this case the strategy takes the form of a first-in-first-out (FIFO) queue. An alternative method of age-based replacement is to randomly select a parent for replacement.

5.3.2 Fitness-based replacement

Replace worst (GENITOR)
In this scheme the worst λ members of the population are selected for replacement. Although this can lead to very rapid improvements in the mean population fitness, it can also lead to premature convergence as the population tends to rapidly focus on the fittest member currently present.
Elitism
This scheme is commonly used in conjunction with age-based and stochastic fitness-based replacement schemes, to prevent the loss of the current fittest member of the population. In essence a trace is kept of the current fittest member, and it is always kept in the population. Thus if it is chosen in the group to be replaced, and none of the offspring being inserted into the population has equal or better fitness, then it is kept and one of the offspring is discarded.
Round-Robin tournament
The method works by holding pairwise tournament competitions in round-robin format, where each individual is evaluated against q others randomly chosen from the merged parent and offspring populations. For each comparison, a "win" is assigned if the individual is better than its opponent. After finishing all tournaments, the μ individuals with the greatest number of wins are selected.
($\mu + \lambda$) Selection
In general, it refers to the case where the set of offspring and parents are merged and ranked according to (estimated) fitness, then the top μ are kept to form the next generation.

(μ, λ) Selection
This method works on a mixture of age and fitness. The age component means that all the parents are discarded, so no individual is kept for more than one generation (although of course copies of it might exist later). The fitness component comes from the fact that the λ offspring are ranked according to the fitness, and the best μ form the next generation.

5.4 Selection pressure

As selection pressure increases, so fitter solutions are more likely to survive, or be chosen as parents, and less-fit solutions are correspondingly less likely.
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.

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

Other measures of selection pressure have been proposed, including the 'Expected Loss of Diversity', which is the expected change in the number of diverse solutions after μ selection events; and from theoretical biology, the 'Selection Intensity', which is the expected relative increase in mean population fitness after applying a selection operator.
While these measures can help in understanding the effect of different strate- gies, they can also be rather misleading since they consider selection alone, rather than in the context of variation operators providing diversity.

5.5 Multimodal problems, selection and the need for diversity

5.5.1 Multimodal problems

Multimodality is a typical aspect of the type of problems for which EAs are often employed, either in attempt to locate the global optimum (particularly when a local optimum has the largest basin of attraction), or to identify a number of high–fitness solutions corresponding to various local optima. The latter situation can often arise, for example, when the fitness function used by the EA does not completely specify the underlying problem. In this situation it is valuable to be able to examine a number of possible options, first so as to permit room for human aesthetic judgements, and second because it is probably desirable to use solutions from niches with broader peaks rather than from a sharp peak (may be overfitted).
The population-based nature of EAs holds out much promise for identifying multiple optima, however, in practice the finite population size, when coupled with recombination between any parents (known as panmictic mixing) leads to the phenomenon known as genetic drift and eventual convergence around one optimum.

5.5.2 Characterising Selection and Population Management Approaches for Preserving Diversity

A number of mechanisms have been proposed to aid the use of EAs on multimodal problems. These can be broadly separated into two camps: explicit approaches, in which specific changes are made to operators in order to preserve diversity, and implicit approaches, in which a framework is used that permits, but does not guarantee, the preservation of diverse solutions.
Define a number of spaces within which the evolutionary algorithm operate:

Genotype space
We may perceive the set of representable solutions as a genotype space and define some distance metrics. This can be a natural distance metrics in that space (e.g., the Manhattan distance) or based on some fundamental move operator. Typical move operators include a single bit-flip for binary spaces, a single inversion for adjacency-based permutation problems and a single swap for order-based permutations problems
Phenotype space
This is the end result: a search space whose structure is based on distance metrics between solutions. The neighbourhood structure in this space may bear little relationship to that in the genotype space according to the complexity of the representation–solution mapping
Algorithmic space
This is the equivalent of the geographical space on which life on Earth has evolved. Effectively we are considering that the working memory of the EA, that is, the population of candidate solutions, can be structured in some way. This spatial structure could be either a conceptual division, or real
- Explicit approaches: Fitness Sharing, Crowding, and Speciation; all of which work by affecting the probability distributions used by selection
- Implicit approaches: Island Model EAs and Cellular EAs

5.5.3 Fitness sharing

This scheme is based upon the idea that the number of individuals within a given niche is controlled by sharing their fitness immediately prior to selection, in an attempt to allocate individuals to niches in proportion to the niche fitness. Each possible pairing of individuals i and j within the population (including i with itself) is considered and a distance $d(i, j)$ between them is calculated according to some distance metric. The fitness F of each individual i is then adjusted according to the number of individuals falling within some prespecified distance σ_{share} using a power-law distribution:

$$F'(i) = \frac{F(i)}{\sum_j sh(d(i,j))},$$

where the sharing function $sh(d)$ is a function of the distance d, given by

$$sh(d) = \begin{cases} 1 - (d/\sigma_{share})^{\alpha} & \text{if } d \leq \sigma_{share} \\ 0 & \text{otherwise} \end{cases}$$

5.5.4 Crowding

The crowding algorithm was suggested as a way of preserving diversity by ensuring that new individuals replaced similar members of the population.
Deterministic crowding
This algorithm relies on the fact that offspring are likely to be similar to their parents as follows:
1. The parent population is randomly paired
2. Each pair produces two offspring via recombination
3. These offspring are mutated and then evaluated
4. The four pairwise distances between offspring and parents are calculated
5. Each offspring then competes for survival in a tournament with one parent, so that the inter-competition distances are minimised. In other words, denoting the parents as p, the offspring as o, and using the subscript to indicate tournament pairing, $d(p_1, o_1) + d(p_2, o_2) < d(p_1, o_2) + d(p_2, o_1)$

The net result of all this is that offspring tend to compete for survival with the most similar parent, so subpopulations are preserved in niches but their size does not depend on fitness; rather it is equally distributed amongst the peaks available.

5.5.5 Automatic Speciation Using Mating Restrictions

The automatic speciation approach imposes mating restrictions based on some aspect of the candidate solutions (or their genotypes) defining them as belonging to different species. The population contains multiple species, and during parent selection for recombination individuals will only mate with others from the same (or similar) species.
Schemes to implement speciation can be divided into two main approaches. In the first speciation is based on the solution (or its representation), e.g., Deb's phenotype (genotype)-restricted mating. The alternative approach is to add some elements such as tags to the genotype that code for the individual's species, rather than representing part of the solution. Common to both approaches is the idea that once an individual has been selected to be a parent, then the choice of mate involves the use of a pairwise distance metric (in phenotype or genotype space as appropriate), with potential mates being rejected beyond a certain distance.

5.5.6 Running Multiple Populations in Tandem: Island Model EAs

The idea of evolving multiple populations in tandem is also known as island model EAs, parallel EA, and, more precisely coarse-grain parallel EAs.
The essential idea is to run multiple populations in parallel, in some kind of communication structure. The communication structure is usually a ring or a torus. After a (usually fixed) number of generations (known as an epoch), a number of individuals are selected from each population to be exchanged with others from neighbouring populations – this can be thought of as migration.

- How often to exchange individuals? The essential problem here is that if the communication occurs too frequently, then all sub-populations will converge to the same solution. Equally if it is done too infrequently, and one or more sub-populations has converged quickly in the vicinity of a peak, then significant amounts of computational effort may be wasted
- How many, and which individuals to exchange? Many authors have found that in order to prevent too rapid convergence to the same solution, it is better to exchange a small number of solutions between sub-populations. Which individuals are selected from each population to be exchanged can be done either by some fitness-based selection mechanism or at random. The choices of how many and which individuals to exchange will evidently affect the tendency of the sub-populations to converge to the same solution. Random, rather than fitness-based, selection strategy is less likely to lead to takeover of one population by a new high-fitness migrant, and ex- changing more solutions also leads to faster mixing and possible takeover
- How to divide the population into sub-populations? The general rule here appears to be that provided a certain (problem-dependent) minimum sub-population size is respected, then more sub-populations usually gives better results

Finally, it is worth mentioning that it is perfectly possible to use different algorithmic parameters on different islands. Thus in the injection island models the sub-populations are arranged hierarchically with each level operating at a different granularity of representation.

5.5.7 Spatial Distribution Within One Population: Cellular EAs

A single population is considered to be split into a larger number of smaller overlap- ping sub-populations (demes) by being distributed within algorithmic space.
Each member of the population exists on a different point on a grid, and only recombination and selection with neighbours is permitted, hence the common names of parallel EAs, fine-grain parallel EAs, diffusion model EA, distributed EAs and, more commonly nowadays cellular EAs.

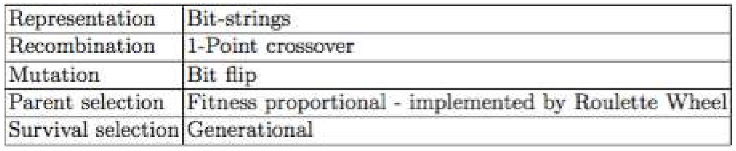
The outline of the algorithm is as follows:

1. The current population is conceptually distributed on a (usually toroidal) grid, with one individual per node
2. For each node we have defined a deme (neighbourhood). This is usually the same for all nodes, e.g., for a neighbourhood size of nine on a square lattice, we take the node and all of its immediate neighbours
3. In each generation we consider each deme in turn and perform the following operations within it:
    - Select two solutions from the nodes in the deme that will act as parents
    - Generate an offspring via recombination
    - Mutate, then evaluate the offspring
    - Select one solution residing on a node in the deme and replace it with the new offspring

6 Popular evolutionary algorithm variant

In this chapter we describe the most widely known evolutionary algorithm variants.

6.1 Genetic algorithms

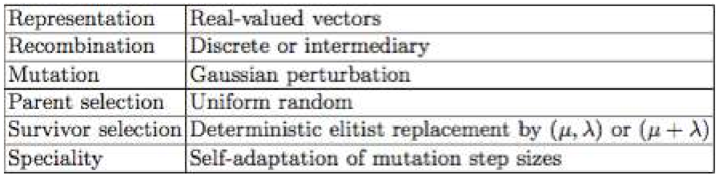


Representation	Bit-strings
Recombination	1-Point crossover
Mutation	Bit flip
Parent selection	Fitness proportional - implemented by Roulette Wheel
Survival selection	Generational

The genetic algorithm (GA) is the most widely known type of evolutionary algorithm. 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.
GAs traditionally have a fixed work-flow: 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 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).

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
- 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

6.2 Evolution strategies



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

Evolution strategies (ES) were invented in the early 1960s by Rechenberg and Schwefel. One of the key early breakthroughs of ES research was to propose a simple mechanism for on-line adjustment of step sizes by the famous 1/5 success rule of Rechenberg. 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. 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.

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

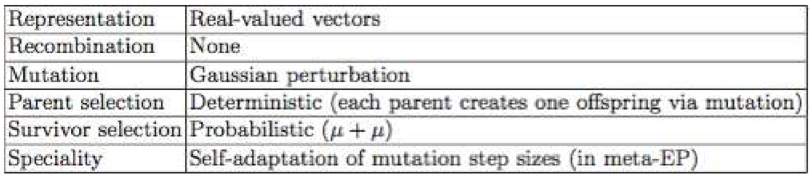
Two recombination variants:
- Discrete recombination; one of the parent alleles is randomly chosen with equal chance for either parents
- Intermediate recombination; the values of the parent alleles are averaged

An extension of this scheme allows the use of more than two recombinants, because the two parents are drawn randomly for each position $i \in 1, ..., n$ in the offspring anew. This multiparent variant is called global recombination (or local recombination).

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

6.3 Evolutionary programming

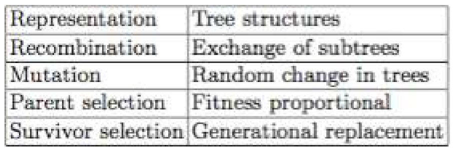


Representation	Real-valued vectors
Recombination	None
Mutation	Gaussian perturbation
Parent selection	Deterministic (each parent creates one offspring via mutation)
Survivor selection	Probabilistic $(\mu + \mu)$
Speciality	Self-adaptation of mutation step sizes (in meta-EP)

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. 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.

EP 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.

6.4 Genetic programming

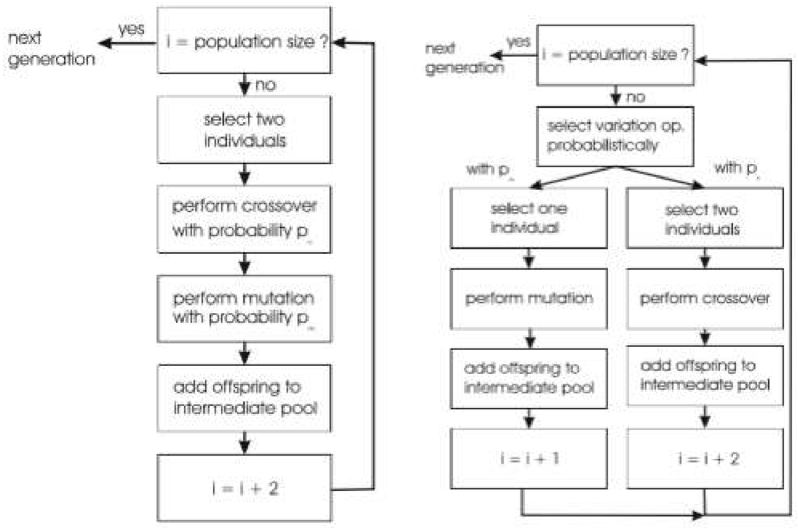


Representation	Tree structures
Recombination	Exchange of subtrees
Mutation	Random change in trees
Parent selection	Fitness proportional
Survivor selection	Generational replacement

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. Most other EAs are for finding some input realising maximum payoff, whereas GP is used to seek models with maximum fit.

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.
There are a few 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 –>most common method: 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 $d = D_{max}$
- 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 $d < D_{max}$



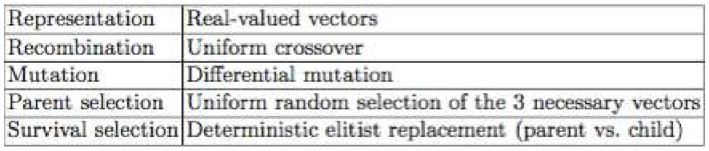
GA loop GP loop

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.
Low or Zero Mutation Probabilities
GP works without mutation (or very small mutation). The reason for this is the generally shared view that crossover has a large shuffling effect, acting in some sense as a macromutation operator.
Overselection
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'. A phenomenon observed in GP whereby average tree sizes tend to grow during the course of a run. 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. 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.

6.5 Differential evolution



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)

Differential evolution (DE) is a a young, but powerful member of the evolutionary algorithm family. 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 $\bar{x}'$ is produced by adding a perturbation vector to an existing one,

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

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

$$\bar{p} = F \times (\bar{y} - \bar{z}),$$

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. 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 work-flow populations are lists, rather than (multi)sets, allowing references to the i-th individual by its position $i \in \{1, ..., \mu\}$ in this list. The order of individuals in such a population $P = \langle \bar{x}_1, ..., \bar{x}_i, ..., \bar{x}_\mu \rangle$ is not related to their fitness values. An evolutionary cycle starts with creating a mutant vector population $M = \langle \bar{v}_1, ..., \bar{v}_\mu \rangle$. For each new mutant $\bar{v}_i$ 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 $T = \langle \bar{u}_1, ..., \bar{u}_\mu \rangle$ is created, where $\bar{u}_i$ is the result of applying crossover to $\bar{v}_i$ and $\bar{x}_i$. (Note, that it is guaranteed that $\bar{u}_i$ does not duplicate $\bar{x}_i$) In the last step deterministic selection is applied to each pair $\bar{x}_i$ and $\bar{u}_i$: the i-th individual in the next generation is $\bar{u}_i$ if $f(\bar{u}_i) \leq f(\bar{x}_i)$ and $\bar{x}_i$ otherwise.

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. 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.

6.6 Particle swarm optimisation

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)

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. 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 $\bar{x} \in \mathbb{R}^n$ carries its own perturbation vector $\bar{p} \in \mathbb{R}^n$. 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.
On a conceptual level every population member in a PSO can be considered as a pair $\langle \bar{x}, \bar{p} \rangle$, where $\bar{x} \in \mathbb{R}^n$ is a candidate solution vector and $\bar{p} \in \mathbb{R}^n$ 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 $\langle \bar{x}', \bar{p}' \rangle$ is produced from $\langle \bar{x}, \bar{p} \rangle$ by first calculating a new perturbation vector $\bar{p}'$ (using $\bar{p}$ and some additional information) and adding this to $\bar{x}$. That is,

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

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 $\bar{v}$ and a new velocity vector $\bar{x}'$ is defined as the weighted sum of three components: $\bar{v}$ and two vector differences. The first points from the current position $\bar{x}$ to the best position $\bar{y}$ the given population member ever had in the past, and the second points from $\bar{x}$ to the best position $\bar{z}$ the whole population ever had. Formally, we have

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

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 $\bar{y} - \bar{x}$ and $\bar{z} - \bar{x}$ 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 $\bar{y}$ and the global best $\bar{z}$ must be kept in memory. This requires a unique identifier for population members. To this end, PSO populations are lists, rather than (multi)sets, allowing references to the i-th individual. Thus, technically, the i-th individual is a triple $\langle \bar{x}_i, \bar{v}_i, \bar{b}_i \rangle$, where $\bar{x}_i$ is the solution vector (perceived as a position), $\bar{v}_i$ is its velocity vector, and $\bar{b}_i$ is its personal best. During an evolutionary cycle each triple $\langle \bar{x}_i, \bar{v}_i, \bar{b}_i \rangle$ is replaced by the mutant triple $\langle \bar{x}'_i, \bar{v}'_i, \bar{b}'_i \rangle$ using the following formulas

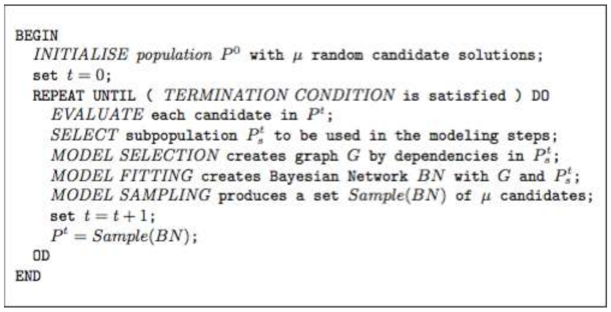
$$\begin{aligned} \bar{x}'_i &= \bar{x} + \bar{v}'_i \\ \bar{v}'_i &= w \times \bar{v}_i + \phi_1 U_1 \times (\bar{b}_i - \bar{x}_i) + \phi_2 U_2 \times (\bar{c} - \bar{x}_i) \end{aligned}$$

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

$$\bar{b}'_i = \begin{cases} \bar{x}'_i & iff(\bar{x}'_i) < f(\bar{b}_i) \\ \bar{b}_i & otherwise \end{cases}$$

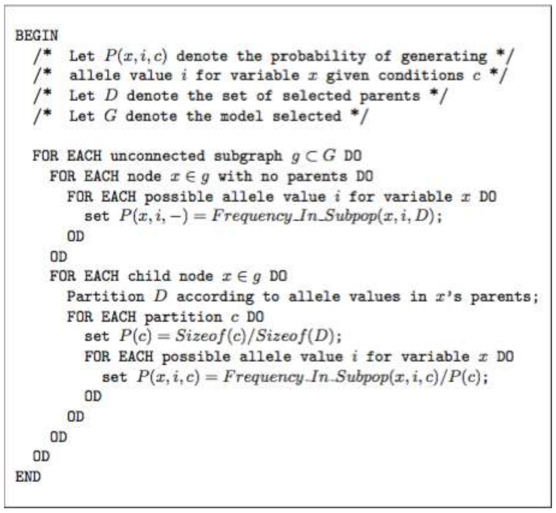
6.7 Estimation of distribution algorithms

Estimation of distribution algorithms (EDA) are based on the idea of replacing the creation of offspring by 'standard' variation operators (recombination 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.
The pseudocode in below illustrates the way that offspring are created via the processes of model selection, model fitting and model sampling.



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

In principle, any standard selection method may be used to select P^t_s, 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.
The process of model-fitting may be characterised as per the pseudocode below, 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 $P(x, i, -)$.



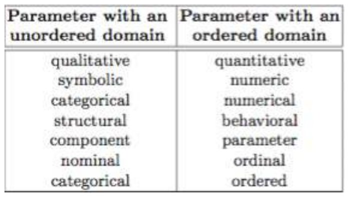
```
BEGIN
  /*  Let $P(x,i,c)$ denote the probability of generating */
  /*  allele value $i$ for variable $x$ given conditions $c$ */
  /*  Let $D$ denote the set of selected parents */
  /*  Let $G$ denote the model selected */

  FOR EACH unconnected subgraph $g \subset G$ DO
    FOR EACH node $x \in g$ 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 $x \in g$ 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
```

7 Parameters and parameter tuning

A decision to use an evolutionary algorithm implies that the user adopts the main design decisions behind this framework. Thus, the main algorithm setup follows automatically: the algorithm is based on a population of candidate solutions that is manipulated by selection, recombination, and mutation operators. To obtain a concrete, executable EA, the user only needs to specify a few details. In this chapter we have a closer look at these details, named parameters.

7.1 Evolutionary algorithm parameters

For detailed discussion of the notion of EA parameters, let us consider a simple GA. This is a well-established algorithm with a few degrees of freedom, including the parameters *crossoveroperator*, *crossoverrate*, and *populationsize*. To obtain a fully specified, executable version we must provide specific values for these parameters, for instance, setting the parameter *crossoveroperator* to *onepoint*, the parameter *crossoverrate* to 0.5, and the parameter *populationsize* to 100. We can distinguish parameters by their domains. The parameter *crossoveroperator* has a finite domain with no sensible distance metric or ordering, e.g., {*onepoint, uniform, averaging*}, whereas the domain of the parameter $p_c \in [0,1]$ is a subset of the real numbers $\mathbb{R}$ with the natural structure for real numbers. This difference is essential for searchability. For parameters with a domain that has a distance metric, or is at least partially ordered, one can use heuristic search and optimization methods to find optimal values. For the other type of parameters this is not possible because the domain has no exploitable structure. The only option in this case is sampling.

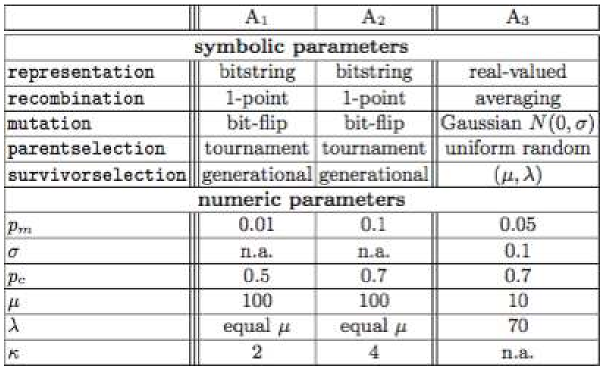


Parameter with an unordered domain	Parameter with an ordered domain
qualitative	quantitative
symbolic	numeric
categorical	numerical
structural	behavioral
component	parameter
nominal	ordinal
categorical	ordered

Two important terms: symbolic parameter and numeric parameter. For both types of parameters the elements of the parameter's domain are called parameter values and we instantiate a parameter by allocating a value to it.
It is important to note that, depending on particular design choices, one might obtain different numbers of parameters for an EA. There can also be a hierarchy among parameters. Namely, symbolic parameters may have numeric parameters 'under them'.

7.2 EAs and EA instances

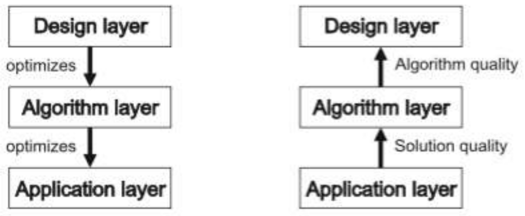
The distinction between symbolic and numeric parameters naturally supports a distinction between EAs and EA instances. To be specific, we can consider symbolic parameters as high-level ones that define the essence of an evolutionary algorithm, and look at numeric parameters as low-level ones that define a specific variant of this EA. Following this naming convention, an evolutionary algorithm is a partially specified algorithm fitting a specific framework, where the values to instantiate symbolic parameters are defined, but the numeric parameters are not. Hence, we consider two EAs to be different if they differ in one or more of their symbolic parameters. If the values are specified for all parameters, including the numeric ones then we obtain an evolutionary algorithm instance. If two EA instances differ only in some of the values of their numeric parameters (e.g., the mutation rate and the tournament size), then we consider them as two variants of the same EA.
In the figure below A_1 and A_2 are just variants of the same EA. And A_3 is an example of a different EA, because it has different symbolic parameter values.



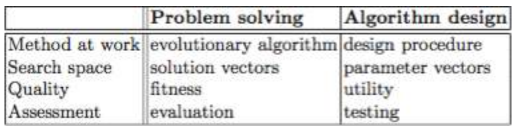
	A₁	A₂	A₃
symbolic parameters			
`representation`	bitstring	bitstring	real-valued
`recombination`	1-point	1-point	averaging
`mutation`	bit-flip	bit-flip	Gaussian $N(0,\sigma)$
`parentselection`	tournament	tournament	uniform random
`survivorselection`	generational	generational	(μ,λ)
numeric parameters			
p_m	0.01	0.1	0.05
σ	n.a.	n.a.	0.1
p_c	0.5	0.7	0.7
μ	100	100	10
λ	equal μ	equal μ	70
κ	2	4	n.a.

7.3 Designing evolutionary algorithms

In the broad sense, algorithm design includes all the decisions needed to specify an algorithm (instance) to solve a given problem (instance). The principal challenge for evolutionary algorithm designers is that the design details, i.e., parameter values, have such a large influence on the performance of the algorithm. Hence, the design of algorithms in general, and EAs in particular, is an optimization problem itself.
To understand this issue, we distinguish three layers: application, algorithm, and design, as shown in the figure. The whole scheme can be divided into two optimization problems that we refer to as problem solving (the lower part) and algorithm design (the upper part). The lower part consists of an EA instance at the algorithm layer that is trying to find an optimal solution for the given problem instance at the application layer. The upper part contains a design method - the intuition and heuristics of a human user or an automated design strategy - that is trying to find optimal parameter values for the given EA at the algorithm layer.



In keeping with the usual EC terminology we use the term fitness at the application layer, and the term utility at the algorithm layer. In the same spirit, we use the term evaluation only in relation to fitness, cf. fitness evaluation, and testing in relation to utility. With this nomenclature, the problem to be solved by the algorithm designer can be seen as an optimization problem in the space of parameter vectors given some utility function. Solutions of the EA design problem are therefore EA parameter vectors with maximum utility.



	Problem solving	**Algorithm design**
Method at work	evolutionary algorithm	design procedure
Search space	solution vectors	parameter vectors
Quality	fitness	utility
Assessment	evaluation	testing

Now we can define the utility landscape as an abstract landscape where the locations are the parameter vectors of an EA and the height reflects utility. It is obvious that fitness landscapes, commonly used in EC, have a lot in common with utility landscapes as introduced here. However, despite the obvious analogies, there are some differences. First of all, fitness values are typically deterministic for most problems. However, utility values are always stochastic, because they reflect the performance of an EA which is a stochastic search method.

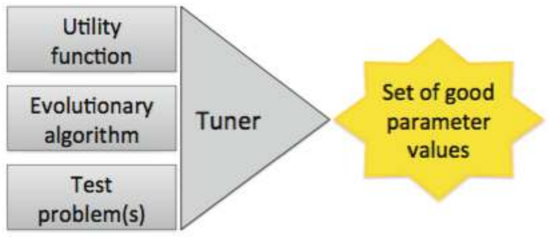
Second, the notion of fitness is usually strongly related to the objective function of the problem in the application layer, and differences between suitable fitness functions mostly concern arithmetic details. In contrast, the notion of utility depends on the performance metrics used, which reflect the preferences of the user and the context in which the EA is being used.

7.4 The tuning problem

Parameter tuning is a commonly practised approach to algorithm design, where the values for the parameters are established before the run of the algorithm and they remain fixed during the run.
The common way to solve the tuning problem is based on conventions ("mutation rate should be low"), ad hoc choices ("why not use population size 100") and limited experimentation with different values. The problems with the experimentation-based approach are traditionally summarised as follows:

- Parameter effects interact (for example, diversity can be created by re- combination or mutation), hence they cannot be optimised one by one
- Trying all different combinations systematically is extremely time consuming
- For a numerical parameter, the best parameter values may not even be among the ones we selected for testing. This is because an optimal parameter value could very well lie between the points in the grid we were testing. Increasing the resolution by increasing the number of grid points increases the number of runs exponentially

Generic scheme of parameter tuning in graphical form.



7.5 Algorithm quality: Performance and robustness

In general, there are two basic performance measures for EAs, one regarding solution quality and one regarding algorithm speed. Solution quality can be naturally expressed by the fitness function. As for algorithm speed, time or search effort needs to be measured. There are three basic combinations of solution quality and computing time that can be used to define algorithm performance in a single run: fix time and measure quality; fix quality and measure time; or fix both and measure completion. For instance:

- Given a maximum running time (computational effort), performance is defined as the best fitness at termination
- Given a minimum fitness level, performance is defined as the running time (computational effort) needed to reach it
- Given a maximum running time (computational effort) and a minimum fitness level, performance is defined through the Boolean notion of success: a run is deemed successful if the given fitness is reached within the given time

Because of the stochastic nature of EAs, a good estimation of performance requires multiple runs on the same problem with the same parameter values and some statistical aggregation of the measures defined for single runs. The following three measures gives us the performance metrics commonly used in evolutionary computing:

- MBF (mean best fitness)
- AES (average number of evaluations to a solution)
- SR (success rate)

The choice of performance metrics determines the utility landscape, and therefore which parameter vector is best.
There are different interpretations regarding robustness. The existing (informal) definitions do agree that robustness is related to the variance of an algorithm's performance across some dimension, but they differ in what this dimension is. There are indeed more options here, given the fact that the performance of an EA (instance) depends on (1) the problem instance it is solving, (2) the parameter vector it uses, and (3) effects from the random number generator. Therefore, the variance of performance can be considered along three different dimensions: parameter values, problem instances, and random seeds, leading to three different types of robustness.
The first type of robustness is encountered if we are tuning an evolutionary algorithm A on a test suite consisting of many problem instances or test functions. In this case robustness is defined for EA instances, not EAs.
Another interpretation of algorithm robustness is related to performance variations caused by different parameter values. This notion of robustness is defined for EAs. EAs can be compared by their robustness.

7.6 Tuning methods

In essence, all tuning algorithms work by the GENERATE-and-TEST principle, i.e., by generating parameter vectors and testing them to establish their utility. Considering the GENERATE step, tuners can be then divided into two main categories: non-iterative and iterative tuners. All non-iterative tuners execute the GENERATE step only once, during initialization, thus creating a fixed set of vectors. One could say that non-iterative tuners follow the INITIALIZE-and-TEST template. In contrast, iterative tuners do not fix the set of vectors during initialization, but start with a small initial set and create new vectors iteratively during execution.
Considering the TEST step, we can again distinguish two types of tuners: single-stage and multi-stage procedures. Single-stage procedures perform the same number of tests for each given vector, while multi-stage procedures use a more sophisticated strategy. In general, they augment the TEST step by adding a SELECT step, where only promising vectors are selected for further testing, deliberately ignoring those with a low performance.
A further useful distinction between tuning methods can be made by their use of meta-models of the utility landscape. From this perspective tuners can be divided into two major classes: model-free and model-based approaches. Model-free approaches are 'simply' optimising the given utility landscape, trying to find parameter vectors that maximise the performance of the EA to be tuned. Model-based approaches create a model that estimates the performance of an EA for any given parameter vector during the tuning process. In other words, they use meta models or surrogate models that have two advantages. First, meta models reduce the number of expensive utility tests by replacing some of the real tests by model estimates that can be calculated very quickly. Second, they capture information about parameters and their utility for algorithm analysis.

8 Parameter control

In this chapter we dis- cuss how to do this during a run of an EA, in other words, we elaborate on controlling EA parameters on-the-fly. This has the potential of adjusting the algorithm to the problem while solving the problem.

8.1 Introduction

Parameter tuning can greatly increase the performance of EAs. However, the tuning approach has an inherent drawback: parameter values are specified before the run of the EA and these values remain fixed during the run. But a run of an EA is an intrinsically dynamic, adaptive process. Additionally, it is intuitively obvious that different values of parameters might be optimal at different stages of the evolutionary process.
A straightforward way to overcome the limitations of static parameters is by replacing a parameter p by a function $p(t)$, where t is the generation counter (or any other measure of elapsed time). Designing optimal dynamic parameters (that is, functions for $p(t)$) may be really difficult. Another drawback to this approach is that the parameter value $p(t)$ changes are caused by a 'blind' deterministic rule triggered by the progress of time t, unaware of the current state of the search.

8.2 Examples of changing parameters

Consider a numerical optimisation problem of minimising

$$f(\bar{x}) = f(x_1, ..., x_n),$$

subject to some inequality and equality constraints

$$g_i(\bar{x}) \leq 0, i = 1, ..., q,$$

and

$$h_j(\bar{x}) = 0, j = q+1, ..., m,$$

where the domains of the variables are given by lower and upper bounds $l_i \leq x_i \leq u_i$ for $1 \leq i \leq n$.For such a problem we might design an EA based on a floating-point representation, where each individual $\bar{x}$ in the population is represented as a vector of floating-point numbers $\bar{x} = \langle x_1, ..., x_n \rangle$.

8.2.1 Changing the mutation step size

Let us assume that offspring are produced by arithmetic crossover followed by Gaussian mutation that replaces components of the vector $\bar{x}$ by

$$x'_i = x_i + N(0, \sigma)$$

To adjust σ over time we use a function $\sigma(t)$ defined by some heuristic rule and a given measure of time t. For example,

$$\sigma(t) = 1 - 0.9 \times \frac{t}{T},$$

where t varies from 0 to T, the maximum generation number. In this approach, the value of the given parameter changes according to a fully deterministic scheme. The user thus has full control of the parameter, and its value at a given time t is completely determined and predictable.
Second, it is possible to incorporate feedback from the search process, still using the same σ for all vectors in the population and for all variables of each vector. For example, Rechenberg's 1/5 success rule states that

the ratio of successful mutations to all mutations should be 1/5. The rule is executed at periodic intervals, for instance, after k iterations each σ is reset by

$$\sigma' = \begin{cases} \sigma/c & \text{if } p_s > 1/5, \\ \sigma \times c & \text{if } p_s < 1/5, \\ \sigma & \text{if } p_s = 1/5, \end{cases}$$

where p_s is the relative frequency of successful mutations, measured over a number of trials, and the parameter c should be $0.817 \leq c \leq 1$. Using this mechanism, changes in the parameter values are now based on feedback from the search. The influence of the user is much less direct here.
Third, we can assign an individual mutation step size to each solution and make these co-evolve with the values encoding the candidate solutions. To this end we extend the representation of individuals to length $n+1$ as $\langle x_1, ..., x_n, \sigma\rangle$ and apply some variation operators (e.g., Gaussian mutation and arithmetic crossover) to the values of x_i as well as to the σ value of an individual. In this way, not only the solution vector values (x_i) but also the mutation step size of an individual undergo evolution.

$$\sigma' = \sigma \times e^{\tau \times N(0,1)},$$
$$x_i' = x_i + \sigma \times N_i(0,1)$$

Within this self-adaptive scheme the heuristic character of the mechanism resetting the parameter values is eliminated, and a certain value of σ acts on all values of a single individual.
Finally, we can use a separate σ_i for each x_i, extend the representation to $\langle x_1, ..., x_n, \sigma_1, ..., \sigma_n\rangle$. The resulting system is the same as the previous one, except the granularity, here we are co-evolving n parameters of the EA instead of 1.

8.2.2 Changing the penalty coefficients

The evaluation function (and consequently the fitness function) can also be parameterised and varied over time.
When dealing with constrained optimisation problems, penalty functions are often used. A common technique is the method of static penalties, which requires penalty parameters within the evaluation function as follows:

$$eval(\bar{x}) = f(\bar{x}) + W \times penalty(\bar{x}),$$

where f is the objective function, $penalty(\bar{x})$ is zero if no violation occurs and is positive otherwise, and W is a user-defined weight prescribing how severely constraint violations are weighted. For instance, a set of functions f_j ($1 \leq j \leq m$) can be used to construct the penalty, where the function f_j measures the violation of the jth constraint:

$$f_j(\bar{x}) = \begin{cases} max\{0, g_j(\bar{x})\} & \text{if } 1 \leq j \leq q, \\ |h_j(\bar{x})| & \text{if } q+1 \leq j \leq m, \end{cases}$$

To adjust the evaluation function over time, we can replace the static parameter W by a function $W(t)$. A second option is to utilise feedback from the search process. Third, we could allow self-adaptation of the weight parameter, similarly to the mutation step sizes in the previous section.
It is important to note the crucial difference between self-adapting mutation step sizes and constraint weights. Even if the mutation step sizes are encoded in the chromosomes, the evaluation of a chromosome is independent from the actual $'\sigma$ values. That is,

$$eval(\langle \bar{x}, \bar{\sigma}\rangle) = f(\bar{x}),$$

for any chromosome $\langle \bar{x}, \bar{\sigma}\rangle$. In contrast, if constraint weights are encoded in the chromosomes, then we have

$$eval(\langle \bar{x}, \bar{w}\rangle) = f_{\bar{w}}(\bar{x}),$$

for any chromosome $\langle \bar{x}, \bar{w}\rangle$. This could enable the evolution to ‘cheat’ in the sense of making improvements by minimising the weights instead of optimising f and satisfying the constraints.

8.3 Classification of control techniques

In classifying parameter control techniques of an evolutionary algorithm, many aspects can be taken into account. For example:

1. what is changed (e.g., representation, evaluation function, operators, selection process, mutation rate, population size, and so on)
2. how the change is made (i.e., deterministic heuristic, feedback-based heuristic, or self-adaptive)
3. he evidence upon which the change is carried out (e.g., monitoring performance of operators, diversity of the population, and so on)
4. the scope/level of change (e.g., population-level, individual-level, and so forth)

8.3.1 What is changed?

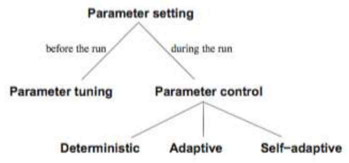
To classify parameter control techniques from the perspective of what component or parameter is changed, it is necessary to agree on a list of all major components of an evolutionary algorithm.

- representation of individuals
- evaluation function
- variation operators and their probabilities
- selection operator (parent selection or mating selection)
- replacement operator (survival selection or environmental selection)
- population (size, topology, etc.)

Note that each component can be parameterised, and that the number of parameters is not clearly defined.

8.3.2 How are changes made?

Methods for changing the value of a parameter (i.e., the 'how-aspect') can be classified into one of three categories. **Deterministic parameter control**
Here the value of a parameter is altered by some deterministic in rule predetermined (i.e., user-specified) manner without using any feedback from the search
Adaptive parameter control
Here some form of feedback from the search serves as input to a mechanism that determines the change. Updating the parameter values may involve credit assignment, based on the quality of solutions discovered by different operators/parameters, so that the updating mechanism can distinguish between the merits of competing strategies. The important point to note here is that the updating mechanism used to control parameter values is externally supplied, rather than being part of the usual evolutionary cycle
Self-adaptive parameter control
Here the evolution of evolution is used to implement the self-adaptation of parameters. The parameters to be adapted are encoded into the chromosomes and undergo mutation and recombination. The better values of these lead to better individuals, which in turn are more likely to survive, produce offspring and hence propagate these better parameter values. This is an important distinction between adaptive and self-adaptive schemes: in the latter the mechanisms for the credit assignment and updating of different strategy parameters are entirely implicit, i.e., they are the selection and variation operators of the evolutionary cycle itself



8.3.3 What evidence informs the change?

The third criterion for classification concerns the evidence used for determining the change of parameter value. Most commonly, the progress of the search is monitored by looking at the performance of operators, the diversity of the population, and so on, and the information gathered is used as feedback for adjusting the parameters. From this perspective, we can further distinguish between the following two cases: **Absolute evidence**
We speak of absolute evidence when the rule to change the value of a parameter is applied when a predefined event occurs. As opposed to deterministic parameter control, where a rule fires by a deterministic trigger (e.g., time elapsed), here feedback from the search is used. Such mechanisms require that the user has a clear intuition about how to steer the given parameter into a certain direction in cases that can be specified in advance (e.g., they determine the threshold values for triggering rule activation). This intuition relies on the implicit assumption that changes that were appropriate in another run on another problem are applicable to this run on this problem
Relative evidence
In the case of using relative evidence, parameter values within the same run are compared according to the positive/negative effects they produce, and the better values get rewarded. The direction and/or magnitude of the change of the parameter is not specified deterministically, but relative to the performance of other values, i.e., it is necessary to have more than one value present at any given time.

8.3.4 What is the scope of the change?

Any change within any component of an EA may affect a gene (parameter), whole chromosomes (individuals), the entire population, another component (e.g., selection), or even the evaluation function. This is the aspect of the scope or level of adaptation. Note, however, that the scope or level is not an independent dimension, as it usually depends on the component of the EA where the change takes place.

8.3.5 Summary?

In conclusion, the main criteria for classifying methods that change the values of the strategy parameters of an algorithm during its execution are:

1. what component/parameter is changed?
2. how is the change made?
3. what evidence is used to make the change?

Our classification is thus three-dimensional. The component dimension consists of six categories: representation, evaluation function, variation operators (mutation and recombination), selection, replacement, and population. The other dimensions have respectively three (deterministic, adaptive, self-adaptive) and two categories (absolute, relative). Their possible combinations are given in the table below. As the table indicates, deterministic parameter control with relative evidence is impossible by definition, and so is self-adaptive parameter control with absolute evidence. Within the adaptive scheme both options are possible and are indeed used in practice.

	Deterministic	Adaptive	Self-adaptive
Absolute	+	+	–
Relative	–	+	+

8.4 Examples of varying EA parameters

8.4.1 Representation

```
BEGIN
  /* given a starting population and genotype-phenotype encoding */
  WHILE ( $HD > 1$ ) DO
    RUN_GA with $k$ bits per object variable;
  OD
  REPEAT UNTIL ( global termination is satisfied ) DO
    save best solution as INTERIM;
    reinitialise population with new coding;
    /* k-1 bits as the distance $\delta$ to the object value in */
    /* INTERIM and one sign bit */
    WHILE ( $HD > 1$ ) DO
      RUN_GA with this encoding;
    OD
  OD
END
```

We illustrate variable representations with the delta coding algorithm of Mathias and Whitley, which effectively modifies the encoding of the function parameters. The motivation behind this algorithm is to maintain a good balance between fast search and sustaining diversity. This is an adaptive adjustment of the representation based on absolute evidence. The GA is used with multiple restarts; the first run is used to find an interim solution, and subsequent runs decode the genes as distances (delta values) from the last interim solution. This way each restart forms a new hypercube with the interim solution at its origin. The restarts are triggered when population diversity (measured by the Hamming distance between the best and worst strings of the current population) is not greater than 1.

8.4.2 Evaluation function

Evaluation functions are typically not varied in an EA because they are often considered as part of the problem to be solved and not as part of the problem-solving algorithm. In fact, an evaluation function forms the bridge between the two, so both views are at least partially true. In many EAs the evaluation function is derived from the (optimisation) problem at hand with a simple transformation of the objective function. In the class of constraint satisfaction problems, however, there is no objective function in the problem definition. One possible approach here is based on penalties. Let us assume that we have m constraints c_i $(i \in \{1, ..., m\})$ and n variables v_j $(j \in \{1, ..., n\})$ with the same domain S. Then the penalties can be defined as follows:

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

where w_i is the weight associated with violating c_i, and

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

Obviously, the setting of these weights has a large impact on the EA performance, and ideally w_i should reflect how hard c_i is to satisfy.
The stepwise adaptation of weights (SAW) mechanism provides a simple and effective way to set these weights. The basic idea is that constraints that are not satisfied after a certain number of steps (fitness evaluations) must be difficult, and thus must be given a high weight (penalty).

8.4.3 Mutation

The 1/5 rule of Rechenberg constitutes a classic example for adaptive mutation step size control in ES. Furthermore, self-adaptive control of mutation step sizes is traditional in ES.

8.4.4 Crossover

The classic example for adapting crossover rates in GAs is Davis's adaptive operator fitness. The method adapts the rates of crossover operators by rewarding those that are successful in creating better offspring. This method is adaptive based on relative evidence.

8.4.5 Selection

Most existing mechanisms for varying the selection pressure are based on the so-called Boltzmann selection mechanism, which changes the selection pressure during evolution according to a predefined cooling schedule. We illustrate variable selection pressure in the survivor selection (replacement) step by simulated annealing (SA). SA is a generate-and-test search technique based on a physical, rather than a biological, analogy.

```
BEGIN
  /* given a current solution $i \in S$ */
  /* given a function to generate the set of neighbours $N_i$ of $i$ */
  generate $j \in N_i$;
  IF ($f(i) < f(j)$) THEN
    set $i = j$;
    ELSE
      IF ( $\exp\left(\frac{f(i)-f(j)}{c_k}\right) > \text{random}[0,1)$) THEN
        set $i = j$;
      FI
    ESLE
  FI
END
```

In this mechanism the parameter c_k, the temperature, decreases according to a predefined scheme as a function of time, making the probability of accepting inferior solutions smaller and smaller (for minimisation problems).

8.4.6 Population

An innovative way to control the population size is offered by Arabas et al. in their GA with variable population size (GAVaPS). In fact, the population size parameter is removed entirely from GAVaPS, rather than adjusted on-the-fly. Certainly, in an evolutionary algorithm the population always has a size, but in GAVaPS this size is a derived measure, not a controllable parameter. The main idea is to assign a lifetime to each individual when it is created, and then to reduce its remaining lifetime by one in each consecutive generation. When the remaining lifetime becomes zero, the individual is removed from the population. The lifetime allocated to a newborn individual is biased by its fitness and the expected number of offspring of an individual is proportional to the number of generations it survives.

8.4.7 Varying several parameters simultaneously

Mutation, crossover, and population size are all controlled on-the-fly in the GA "without parameters" of Back et al. Here a new self-adaptive technique is invented for regulating the crossover rates of the individuals, and the GAVaPS lifetime idea is adjusted for a steady-state GA model. The crossover rates are included in the chromosomes, much like the mutation rates. If a pair of individuals is selected for reproduction, then their individual crossover rates are compared with a random number $r \in [0, 1]$, and an individual is seen as ready to mate if its $p_c > r$. Then there are three possibilities:

1. if both individuals are ready to mate then uniform crossover is applied, and the resulting offspring is mutated
2. If neither is ready to mate then both create a child by mutation only
3. If exactly one of them is ready to mate, then the one not ready creates a child by mutation only (which is inserted into the population immediately through the steady-state replacement), the other is put on hold, and the next parent selection round picks only one other parent

9 Working with evolutionary algorithms

n this chapter we discuss the practical aspects of using EAs.

9.1 What do you want an EA to do?

A good first step is to examine the given problem context. We can roughly distinguish two main types of problems:

1. design (one-off) problems
2. repetitive problems, including on-line control problems as special cases

An example of a design problem, let us consider the optimisation of extensions to an existing road network to meet new demands. This is a highly complex multiobjective optimisation problem, subject to many constraints. It requires an algorithm that creates one excellent solution at least once. In this context the quality of the solution is of utmost importance, and other aspects of algorithm performance are secondary. Thus, the algorithm does not have to be fast. The algorithm does not need to be generally applicable either.
Repetitive problems form a counterpoint to design problems. Consider a domestic transportation firm, having dozens of trucks and drivers that need to be given a daily schedule every morning. The schedule should contain a pick-up and delivery plan, plus the corresponding route description for each truck and driver. For each of them, this is just a TSP problem (probably with time windows), but the optimisation criteria and the constraints must be taken across the whole firm, together making the actual problem very complex. The algorithm must be able to find good solutions quickly and be able to do this repeatedly for different instances of the problem. It is important that the algorithm performance is stable. Finally, for repetitive problems the widescale applicability of the algorithm is also important as the system will be used under various circumstances. In other words, the algorithm will be run on different problem instances.
On-line control problems can also be seen as repetitive problems with ex- tremely tight time constraints.

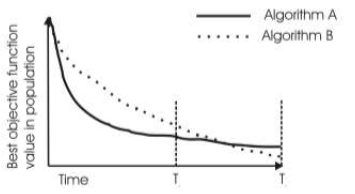
9.2 Performance measures

Assessing the quality of an evolutionary algorithm commonly implies experimental comparisons between the given EA and other evolutionary or traditional algorithms. Such comparisons always assume the use of some algorithm performance measures.
Because EAs are stochastic, performance measures are statistical in nature, meaning that a number of experiments need to be conducted to gain sufficient experimental data. There are three basic performance measures: **Success rate**, **Effectiveness (solution quality**, **Efficiency**.

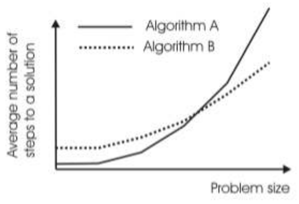
9.2.1 Different performance measures

In quite a few cases, experimental research concerns problems where either the optimal solution can be recognised, or a criterion for sufficient solution quality can be given. In these cases one can easily define a success criterion: finding a solution of the required quality, and the success rate (SR) measure can be defined as the percentage of runs where this happens. For problems where the optimal solutions cannot be recognised, the SR measure cannot be used in theory. Nevertheless, a success criterion in the practical sense can often be given even in these cases. Practical success criteria can also be used even in cases when the theoretical optimum is known, but the user does not require this optimum.
The mean best fitness measure (MBF) can be defined for any problem that is tackled with an EA - at least for any EA using an explicit fitness measure. For each run of a given EA, we record the fitness of the best individual at termination. The MBF is the average of these values over all runs.
The difference between the two measures is rather obvious: SR cannot be defined for some problems, while the MBF is always a valid measure. Furthermore, all possible combinations of low or high SR and MBF values can occur. Low SR and high MBF is possible and indicates a good approximizer algorithm: it gets close consistently, but seldom really makes it. Solution: increasing the length of the runs, hoping that this

allows the algorithm to finish the search. High SR and low MBF is also possible, indicating a 'Murphy algorithm': if it goes wrong, it goes very wrong. That is, those few runs that terminate without an (optimal) solution end in a disaster, with a very bad best fitness value deteriorating MBF.



In this figure we are comparing algorithms A and B after terminating at time T1 and T2 (for a minimisation problem). Algorithm A clearly wins in the first case, while B is better in the second one.
SR and MBF are performance measures for an algorithm's effectiveness, indicating how far it can get within a given computational limit.
The complementary approach is to specify when a candidate solution is satisfactory and measure the amount of computing needed to achieve this solution quality. This is the issue of algorithm efficiency or speed. Speed is often measured in elapsed computer time, CPU time, or user time. However, these measures depend on the specific hardware, operating system, compiler, network load, and so on. A common way around this problem is to count the number of points visited in the search space. Since EAs immediately evaluate each newly generated candidate solution, this measure is usually expressed as the number of fitness evaluations. Of necessity, because of the stochastic nature of EAs, this is always measured over a number of independent runs, and the average number of evaluations to a solution (AES) is used. The average is only taken over the successful runs (that is, 'to a solution').
An additional problem with AES is that it can be difficult to apply for comparing an EA with search algorithms that do not work in the same search space, in the same fashion. A possible treatment for this is to compare the scale-up behaviour of the algorithms. This requires a problem that is scalable, i.e., its size can be changed. The number of variables is a natural scaling parameter for many problems. Two different types of methods can then be compared by plotting their own speed measure figures against the problem size.



A great advantage of this comparison is that it can also be applied to plain running times (e.g., CPU times), not only to the number of abstract search steps.
Success percentages and run lengths can be meaningfully combined into a measure expressing the amount of processing required to solve a problem with a given probability.
Another alternative to AES, especially in cases where one cannot specify satisfactory solution quality in advance, is the pace of progress to indicate algorithm speed. Here the best (or alternatively the worst or average) fitness value of the consecutive populations is plotted against a time axis - typically the number of generations or fitness evaluations.

9.2.2 Peak versus average performance

For some, but not all, performance measures, there is an additional question of whether one is interested in peak performance, or average performance, considered over all these experiments. In evolutionary computing it is typical to suggest that algorithm A is better than algorithm B if its average performance is better. In general, if there is time for more runs on the given problem and the final solution can be selected from the best solutions of these runs, then peak performance is more relevant than average performance.

9.3 Test problems for experimental comparisons

In addition to the issue of performance measures, experimental comparisons between algorithms require a choice of benchmark problems and problem in- stances. We distinguish three different approaches:

1. Using problem instances from an academic benchmark repository
2. Using problem instances created by a problem instance generator
3. Using real-life problem instances

9.3.1 Using predefined problem instances

Bäck and Michalewicz gave some general guidelines for composing test suites for EC research in [29]. Below we reproduce the main points from their recommendations. The test suite should contain:

1. A few unimodal functions for comparisons of convergence velocity (efficiency), e.g., AES
2. Several multimodal functions with a large number of local optima (e.g., a number growing exponentially with n, the search space dimension)
3. A test function with randomly perturbed objective function values models a typical characteristic of numerous real-world applications and helps to investigate the robustness of the algorithm with respect to noise
4. Constrained problems,since real-world problems are typically constrained, and constraint handling is a topic of active research
5. High-dimensional objective functions, because these are representative of real-world applications. Most useful are test functions that are scalable with respect to n, i.e., which can be used for arbitrary dimensions.

9.3.2 Using problem instance generators

An alternative to such test landscapes is formed by problem instances of a certain (larger) class, for instance, operations research problems, constrained problems or machine-learning problems. The advantage of such collections is that the problem instances are interesting in the sense that many other researchers have investigated and evaluated them already.
Over the last few years there has been a growing research interest in using problem instance generators. Using such a generator, which could of course come from an archive, means that problem instances are produced on-the-spot. The advantage of this approach is that the characteristics of the problem instances can be tuned by the generator's parameters.

9.3.3 Using real-world problems

Testing on real data has the advantages that results can be considered as very relevant viewed from the application domain (data supplier). However, it also has some disadvantages. Namely, practical problems can be overcomplicated. Furthermore, there can be few available sets of real data, and these data may be commercially sensitive and therefore difficult to publish and to allow others to compare. Last, but not least, there might be so many application-specific aspects involved that the results are hard to generalise.

10 Hybridisation with other techniques: Memetic algorithms

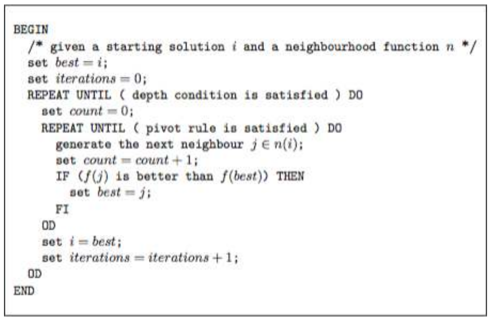
In this chapter we turn our attention to systems in which, rather than existing as stand-alone algorithms, EA-based approaches are either incorporated within larger systems, or alternatively have other methods or data structures incorporated within them. This category of algorithms is very successful in practice and forms a rapidly growing research area with great potential. This area and the algorithms that form its subject of study are named memetic algorithms (MA).

10.1 Motivation for hybridising EAs

There are a number of factors that motivate the hybridization of evolutionary algorithms with other techniques. Many complex problems can be decomposed into a number of parts, for some of which exact methods, or very good heuristics, may already be available. In these cases it makes sense to use a combination of the most appropriate methods for different subproblems.
In practice we frequently apply an evolutionary algorithm to a problem where there is a considerable amount of hard-won user experience and knowledge available. In such cases performance benefits can often arise from utilising this information in the form of specialist operators and/or good solutions, provided that care is taken not to bias the search too much away from the generation of novel solutions. In these cases it is commonly experienced that the combination of an evolutionary and a heuristic method - a hybdrid EA - performs better than either of its ‘parent’ algorithms alone.
A final concept, which is often used as a motivation by researchers in this field, is Dawkins’ idea of memes. These can be viewed as units of cultural transmission, in the same way that genes are the units of biological transmis- sion. They are selected for replication according to their perceived utility or popularity, and then copied and transmitted via interperson communication.
Memetic algorithm (MA); evolutionary search is augmented by the addition of one or more phases of local search, or by the use of problem-specific information.

10.2 A brief introduction to local search

Local search is an iterative process of examining the set of points in the neighbourhood of the current solution, and replacing it with a better neighbour if one exists. In this section we give a brief introduction to local search in the context of memetic algorithms.



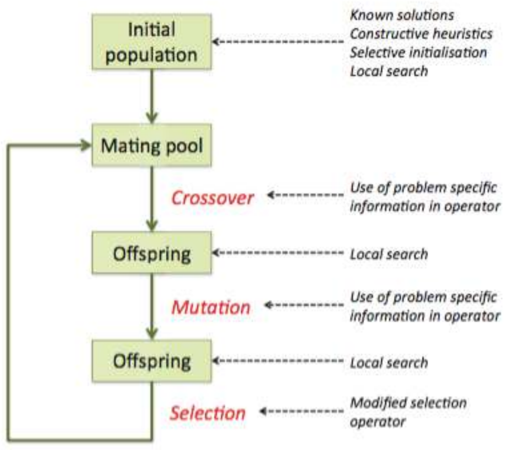
```
BEGIN
  /* given a starting solution $i$ and a neighbourhood function $n$ */
  set $best = i$;
  set $iterations = 0$;
  REPEAT UNTIL ( depth condition is satisfied ) DO
    set $count = 0$;
    REPEAT UNTIL ( pivot rule is satisfied ) DO
      generate the next neighbour $j \in n(i)$;
      set $count = count + 1$;
      IF ($f(j)$ is better than $f(best)$) THEN
        set $best = j$;
      FI
    OD
    set $i = best$;
    set $iterations = iterations + 1$;
  OD
END
```

There are three principal components that affect the workings of this local search algorithm.
- pivot rule; can be steepest ascent or greedy ascent. In the former, the condition for terminating the inner loop is that the entire neighbourhood $n(i)$ has been searched; whereas in the latter the termination condition is as soon as an improvement is found
- depth of the local search, i.e., the termination condition for the outer loop
- the choice of neighbourhood generating function (in some cases, domain-specific information may be used to guide the choice of neighbourhood structure within the local search algorithms)

10.2.1 Lamarckianism and the Baldwin effect

Within a memetic algorithm, we can consider the local search stage to occur as an improvement or developmental learning phase within the evolutionary cycle, and (taking our cue from biology) we should consider whether the changes made to the individual (acquired traits) should be kept in the genotype, or whether the resulting improved fitness should be awarded to the original (pre-local search) member of the population. The issue of whether acquired traits could be inherited by an individual's offspring was a major issue in the nineteenth century, with Lamarck arguing in favour. By contrast, the Baldwin effect suggests a mechanism whereby evolutionary progress can be guided towards favourable adaptation without the changes in individuals' fitness arising from learning or development being reflected in changed genetic characteristics.
In general, MAs are referred to as Lamarckian if the result of the local search stage replaces the individual in the population, and Baldwinian if the original member is kept, but has as its fitness the value belonging to the outcome of the local search process.

10.3 Structure of a memetic algorithm



There are a number of ways in which an EA can be used in conjunction with other operators and/or domain-specific knowledge as illustrated in the picture.

10.3.1 Heuristic or intelligent initialisation

The most obvious way in which existing knowledge about the structure of a problem or potential solutions can be incorporated into an EA is in the initialisation phase. Starting the EA by using existing solutions can offer interesting benefits:

1. It is possible to avoid reinventing the wheel by using existing solutions
2. A nonrandom initial population can direct the search into particular regions of the search space that contain good solutions
3. All in all, a given total amount of computational effort divided over heuris- tic initialisation and evolutionary search might deliver better results than spending it all on 'pure' evolutionary search, or an equivalent multistart heuristic

There are a number of possible ways in which the initialisation function can be changed from simple random creation, such as:
Seeding
Seed the population with one or more previously known good solutions arising from other techniques

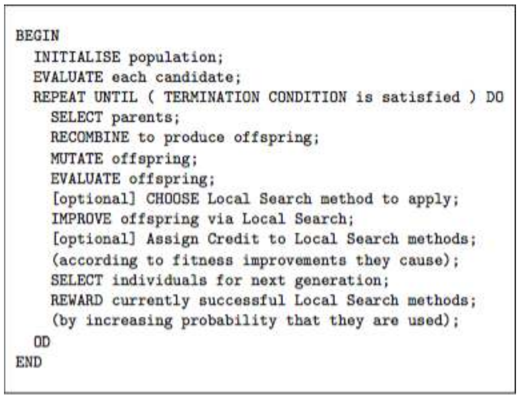
Selective initialisation
A large number of random solutions are created and then the initial population is selected from these
Local search with specific start point
Performing a local search starting from each member of the initial pop- ulation, so that the initial population consists of a set of points that are locally optimal with respect to some move operator
Using one or more of the above
To identify one (or possibly more) good solutions, and then cloning them and applying mutation at a high rate (mass mutation) to produce a number of individuals in the vicinity of the start point

10.3.2 Hybridisation within variation operators: intelligent crossover and mutation

Intelligent variation operators incorporate problem- or instance-specific knowledge, introducing bias into the operators.
At the other end of the scale, at their most complex, the operators can be modified to incorporate highly specific heuristics, which make use of instance- specific knowledge.

10.3.3 Local search acting on the output from variation operators

The most common use of hybridisation within EAs, and that which fits best with Dawkins' concept of the meme, is via the application of one or more phases of improvement to individual members of the population during the EA cycle, i.e., local search acting on whole solutions created by mutation or recombination. As is suggested from the previous picture, this can occur in different places in cycle i.e., before or after selection or after crossover and/or mutation, but a typical implementation might take the form given below.



```
BEGIN
  INITIALISE population;
  EVALUATE each candidate;
  REPEAT UNTIL ( TERMINATION CONDITION is satisfied ) DO
    SELECT parents;
    RECOMBINE to produce offspring;
    MUTATE offspring;
    EVALUATE offspring;
    [optional] CHOOSE Local Search method to apply;
    IMPROVE offspring via Local Search;
    [optional] Assign Credit to Local Search methods;
    (according to fitness improvements they cause);
    SELECT individuals for next generation;
    REWARD currently successful Local Search methods;
    (by increasing probability that they are used);
  OD
END
```

10.3.4 Hybridisation during genotype to phenotype mapping

A widely used hybridisation of memetic algorithms with other heuristics is during the genotype–phenotype mapping prior to evaluation. A good example of this is the use of instance-specific knowledge within a decoder or a repair function.

10.4 Adaptive memetic algorithms

Probably the most important factor in the design of a memetic algorithm incorporating local search or heuristic improvement is the choice of improving heuristic or local search move operator, that is to say, the way that the set of neighbouring points - to be examined when looking for an improved solution - is generated.
'Multimeme' algorithms, the evolutionary algorithm is coupled with not one, but several, local search methods, together with some mechanism for choosing between them according to their perceived usefulness at

any given stage of search. In this case they used self- adaptation, so that each candidate solution carried a gene which indicated which local search method to use.

10.5 Design issues for memetic algorithms

MAs are not some 'magic solution' to optimisation problems, and care must be taken in their implementation.
Preservation of diversity
The problem of premature convergence, whereby the population converges around some suboptimal point, is recognised within EAs but is exacerbated in MAs by the effect of local search. If the local search phase continues until each point has been moved to a local optimum, then this leads to an inevitable loss of diversity within the population. A number of approaches have been developed to combat this problem such as:

- When initialising the population with known good individuals, only using a relatively small proportion of them
- Using recombination operators which are designed to preserve diversity
- Modifying the selection operator to prevent duplicates
- Modifying the selection operator or local search acceptance criteria to use a Boltzmann method so as to preserve diversity

Use of knowledge
A final point that might be taken into consideration when designing a new memetic algorithm concerns the use and reuse of knowledge gained during the optimisation process. To a certain extent this is done automatically by recombination, but, generally speaking, explicit mechanisms are not used.
One possible hybridisation that explicitly uses knowledge about points already searched to guide optimisation is with tabu search. In this algorithm a "tabu" list of visited points is maintained, to which the algorithm is forbidden to return.

11 Nonstationary and noisy function optimisation

There are many EA applications in environments featuring change or noise when solutions are evaluated. In these nonstationary situations the search algorithm has to be designed so that it can compensate for the unpredictable environment by monitoring its performance and altering some aspects of its behaviour. An objective of the resulting adaptation is not to find a single optimum, but rather to select a sequence of values over time that maximise or minimise some measure of the evaluations, such as the average or worst.

11.1 Characterisation of nonstationary problems

At this stage we must consider some basic facts about the process of going from a representation of a solution (genotype) x, to measuring the quality of a candidate solution for the task at hand, $f(x)$. The quality recorded for a given solution may be unpredictable for one or more of the following reasons.

The genotype to phenotype mapping is not exact and one-to-one
In terms of a search landscape, the fitness observed may be that of a point in the region of x: $f_{observed}(x) = f(x + \delta x)$.
The act of measurement itself is prone to error or uncertainty
This might arise from, for example, human error, small random fluctuations in shape of physical objects as their molecules vibrate, randomness in the movement of electrons through a sensor or wire, or the collected randomness of more complex organisms such as people, markets, packets on computer network or physical traffic. Quality function: $f_{observed}(x) = f_{mean}(x) + f_{noise}(x)$. Here the first component represents the average that we would find if we measured fitness many times, and the second noise component is typically modelled as a random drawing from a normal distribution $N(0, \sigma)$.
The environment changes over time
This may be because the external environment is inherently volatile, or it may be that the very act of evaluating solutions affects subsequent fitness. Search landscape: the locations of the optima are now time-dependent, i.e., $f_{observed}(x) = f(x, t)$.

11.2 The effect of different sources of uncertainty

Algorithms are typically compared by running them for a fixed period and calculating two time-averaged metrics, which correspond to different types of real-world applications.
The first of these is the on-line measure and is simply the average of all calls to the evaluation function during the run of the algorithm. The second metric considered is the off-line performance and is the time-averaged value of the best-performing member of the current population. Unlike the on-line metric, offline performance is unaffected by the occasional generation of individuals with very low fitness, and so is more suitable for problems where the testing of such individuals is not penalised.
If we use the time-dependent notation for the fitness function as $f(x, t)$ and denote the best individual in a population $P(t)$ at time t by $best(P(t))$, then we can formalise the two metrics over a period T as follows:

$$online = \frac{1}{T} \times \sum_{t=1}^{T} \frac{1}{|P(t)|} \sum_{x \in P(t)} f(x, t),$$

$$offline = \frac{1}{T} \times \sum_{t=1}^{T} f(best(P(t)), t)$$

Finally, let us note that in some cases it may be appropriate to consider both metrics in a multiobjective approach, since optimising the mean fitness may be the principle desiderata, but evaluating low-fitness solutions might be catastrophic.
The three different sources of uncertainty identified in the previous section affect the performance of the EA in different ways. Considering errors in the genotype–phenotype mapping, we can note that using the

average of n repeated fitness evaluations $\frac{1}{n} \times f(x + \delta x)$ to determine the fitness of any given x means using a sample of n points from the neighbourhood around x. However, the sampled neighbourhoods of adjacent solutions can overlap, that is, $x + \delta x$ can coincide with $y + \delta y$. Hence, fine-grained features of the fitness landscape will be smoothened out possibly removing local optima in the process.
Considering noise in the act of measurement itself, the average of n repeated measurements is $f(x) + \frac{1}{n} \times \sum_n N(0, \sigma)$ and the second term will approximate zero as n is increased.
Regarding the third situation of nonstationary fitness functions, Cobb defines two ways of classifying these:

- Switching versus continuous, which is based on time-scale of change with respect to the rate of evaluation - the former providing sudden and the latter more gradual changes. Continuous changes might be cyclical (e.g., related to seasonal effects) or reflect a more uniform movement of land- scape features
- Markovian versus state dependent. In the first of these, the environment at the next time step is purely derived from the current one, whereas in the latter far more complex dynamics may play out

11.3 Algorithmic approaches

11.3.1 Approaches that increase robustness or reduce noise

The only viable approach for reducing the effect of noise, whether in the fitness function evaluation, or in the genotype-to-phenotype mapping, is to repeatedly re-evaluate solutions and take an average. This might be done either explicitly, or implicitly via the population management processes of parent and survivor selection. The principle question that arises in explicit approaches is, how many times should the fitness be sampled? Bearing in mind that EAs naturally contain some randomness in their processes anyway, the key issue from the perspective of evolution is being able to reliably distinguish between good and bad members of the population. Thus, a common approach is to monitor the degree of variation present, and re-sample when this is greater than the range of estimated fitnesses in the population. This reasoning suggests that the rate of re-sampling would increase as the population converges towards high-quality solutions.
When calculating how large a sample to take, there is also the law of diminishing returns: in general, the standard deviation observed decreases only as fast as the square root of the number of measurements taken. Finally, it is worth mentioning that it is often worth the extra book-keeping of making re-sampling decisions independently for each solution, since the amount of noise will often not be uniform across the search space.

11.3.2 Pure evolutionary approaches to dynamic environments

The distributed nature of the genetic search provides a natural source of power for exploring in changing environments. As long as sufficient diversity remains in the population, the EA can respond to a changing search landscape by reallocating future trials. However, the tendency of EAs, especially of GAs, to converge rapidly results in the population becoming homogeneous, which reduces the ability of the EA to identify regions of the search space that might become more attractive as the environment changes. In such cases it is necessary to complement the standard EA with a mechanism for maintaining a healthy exploration of the search space.

11.3.3 Memory-based approaches for switching or cyclic environments

The first strategy expands the memory of the EA in order to build up a repertoire of ready responses for various environmental conditions. The main examples of this approach are the GA with diploid representation and the structured GA.

11.3.4 Explicitly increasing diversity in dynamic environments

The second modification strategy effectively increases diversity in the population directly (i.e., without extending the EA memory) in order to compensate for changes encountered in the environment. Examples of this strategy involve the GA with a hypermutation operator, the random immigrants GA, the GA with a variable local search (VLS) operator, and the thermodynamic GA.

12 Multiobjective evolutionary algorithms

In this chapter we describe the application of evolutionary techniques to a particular class of problems, namely multiobjective optimisation.

12.1 Multiobjective optimisation problems

A class of problems that are currently receiving a lot of interest within the optimisation community, and in practical appli- cations are the so-called multiobjective problems (MOPs), where the quality of a solution is defined by its performance in relation to several, possibly conflicting, objectives. In practice it turns out that a great many applications that have traditionally been tackled by defining a single objective function (quality function) have at their heart a multiobjective problem that has been transformed into a single-objective function in order to make optimisation tractable.
A feature that is common to multiobjective problems is that it is desirable to present the user with a diverse set of possible solutions, representing a range of different trade-offs between objectives.
The alternative is to assign a numerical quality function to each objective, and then combine these scores into a single fitness score using some weighting (often called scalarisation). Drawbacks of this approach:

- the use of a weighting function implicitly assumes that we can capture all of the user's preferences, even before we know what range of possible solutions exist
- for applications where we are repeatedly solving different instances of the same problem, the use of a weighting function assumes that the user's preferences remain static, unless we explicitly seek a new weighting every time

12.2 Dominance and Pareto optimality

The concept of dominance is a simple one: given two solutions, both of which have scores according to some set of objective values, one solution is said to dominate the other if its score is at least as high for all objectives, and is strictly higher for at least one. We can represent the scores that a solution A gets for n objectives as an n-dimensional vector $\bar{a}$. Using the $\succeq$ symbol to indicate domination, we can define $A \succeq B$ formally as:

$$A \succeq B \iff \forall i \in \{1, ..., n\} a_i \geq b_i, \text{ and } \exists i \in \{1, ..., n\}, a_i > b_i$$

For conflicting objectives, there exists no single solution that dominates all others, and we will call a solution nondominated if it is not dominated by any other. All nondominated solutions possess the attribute that their quality cannot be increased with respect to any of the objective functions without detrimentally affecting one of the others. In the presence of constraints, such solutions usually lie on the edge of the feasible regions of the search space. The set of all nondominated solutions is called the Pareto set or the Pareto front.

12.3 EA approaches to multiobjective optimisation

There have been many approaches to multiobjective optimisation using EAs, beginning with Schaffer's vector-evaluated genetic algorithm (VEGA) in 1984. In this algorithm the population was randomly divided into subpopulations that were then each assigned a fitness (and subject to selection) according to a different objective function, but parent selection and recombination were performed globally.
Subsequent to this, Goldberg suggested the use of fitness based on dominance rather than on absolute objective scores, coupled with niching and/or speciation methods to preserve diversity, and this breakthrough triggered a dramatic increase in research activity in this area.

12.3.1 Non-elitist approaches

Fonseca and Fleming's multiobjective genetic algorithm (MOGA) assigns a raw fitness to each solution equal to the number of members of the current population that it dominates, plus one. It uses fitness

sharing amongst solutions of the same rank, coupled with fitness-proportionate selection to help promote diversity.
Srinivas and Deb's nondominated sorting genetic algorithm (NSGA) works in a similar way, but assigns fitness based on dividing the population into a number of fronts of equal domination. To achieve this, the algorithm iteratively seeks all the nondominated points in the population that have not been labelled as belonging to a previous front. It then labels the new set as belonging to the current front, and increments the front count, repeating until all solutions have been labelled. Each point in a given front gets as its raw fitness the count of all solutions in inferior fronts. Again fitness sharing is implemented to promote diversity, but this time it is calculated considering only members from that individual's front.
Horn et al.'s niched Pareto genetic algorithm (NPGA) differs in that it uses a modified version of tournament selection rather than fitness pro- portionate with sharing. The tournament operator works by comparing two solutions first on the basis of whether they dominate each other, and then second on the number of similar solutions already in the new population.
These three algorithms share two common features. The first of these is that the performance they achieve is heavily dependent on a suitable choice of parameters in the sharing/niching procedures. The second is that they can potentially lose good solutions.

12.3.2 Elitist approaches

Deb and coworkers proposed the revised NSGA-II [112], which still uses the idea of non-dominated fronts, but incorporates the following changes:

- A crowding distance metric is defined for each point as the average side length of the cuboid defined by its nearest neighbours in the same front. The larger this value, the fewer solutions reside in the vicinity of the point
- A $(\mu + \lambda)$ survivor selection strategy is used (with $\mu = \lambda$). The two populations are merged and fronts assigned. The new population is obtained by accepting individuals from progressively inferior fronts until it is full. If not all of the individuals in the last front considered can be accepted, they are chosen on the basis of their crowding distance
- Parent selection uses a modified tournament operator that considers first dominance rank then crowding distance

Two other prominent algorithms, the strength Pareto evolutionary algorithm (SPEA-2) and the Pareto archived evolutionary strategy (PAES), both achieve the elitist effect in a slightly different way by using an archive containing a fixed number of nondominated points discovered during the search process.

12.3.3 Diversity maintenance in MOEAs

To finish our discussion on MOEAs it is appropriate to consider how sets of diverse solutions can be maintained during evolution. It should be clear from the descriptions of the MOEAs above that all of them use explicit methods to enforce preservation of diversity, rather than relying simply on implicit measures such as parallelism (in one form or another) or artificial speciation.
The aim of MOEAs is to attempt to distribute the population evenly along the current approximation to the Pareto front.

12.3.4 Decomposition-based approaches

An unavoidable problem of trying to evenly represent an approximation of the Pareto front is that this approach does not scale well as the dimensionality of the solution space increases above 5–10 objectives. One recent method that has gathered a lot of attention is the decomposition approach taken by Zhang's MOEA-D algorithm which shares features from the single-objective weighted sum approach and the population-based approaches.

13 Constraint handling

In this chapter we return to problems have constraints associated with them. This means that not all possible combinations of variable values represent valid solutions to the problem at hand, and we examine how this impacts on the design of an evolutionary algorithm. This issue has great practical relevance because many real-world problems are constrained. Unfortunately, constraint handling is not straightforward in an EA, because the variation operators (mutation and recombination) are typically 'blind' to constraints. This means that even if the parents satisfy some constraints, there is no guarantee their offspring will.

13.1 Two main types of constraint handling

Before discussing how constraints may be dealt with, we first briefly recap our classification, based on whether problems contained two features: constraints on the form that solutions were allowed to take; and a quality, or fitness function. If we have:

- neither feature then we do not have a problem
- a fitness function but no constraints then we have a Free Optimisation Problem (FOP)
- constraints that a candidate solution must meet but no other fitness criteria then we have a Constraint Satisfaction Problem (CSP)
- both a fitness function and constraints then we have a Constrained Optimisation Problem (COP)

Various techniques for constraint handling are discussed in Sect. 13.2. Before going into details, let us distinguish two conceptually different possibilities.

- In the case of **indirect constraint handling**, constraints are transformed into optimisation objectives. After the transformation, they effectively 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.
- In **direct constraint handling**, the problem offered to the EA to solve has constraints (is a COP) that are enforced explicitly during the EA run.

These options are not exclusive: for a given constrained problem (CSP or COP) some constraints might be treated directly and some others indirectly.

13.2 Approaches to handling constraints

Consider the nature of the domains of the variables. In this respect there are two extremes: they are all discrete or all continuous. 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. General treatment of constraint handling methods, the presence of constraints will divide the space of potential solutions **S** into two or more dis- joint regions, the **feasible region** (or regions) **F** containing those candidate solutions that satisfy the given constraints, and **U**, the **infeasible region** containing those that do not.
We distinguish between approaches by considering how they modify one or more facets of the search: the genotype space, the phenotype space **S**, the mapping from genotype to phenotype, or the fitness function.

- Indirect approaches
    1. Penalty functions modify the fitness function. For feasible solutions in **F** the objective function values are used, but for those in **U** an extra penalty value is applied. Preferably this is designed so that the fitness is reduced in proportion to the number of constraints violated, or to the distance from the feasible region.

- Direct approaches
  1. Specialised representations, together with initialisation, and reproduction operators reduce the genotype space to ensure all candidate solutions are feasible. The mapping, phenotype space and fitness functions are left unchanged.
  2. Repair mechanisms modify the original mapping. For feasible solutions in **F** the mapping is left unchanged, but for those in **U** an extra stage is added to turn an infeasible solution into a feasible one, hopefully close to the infeasible one. Note that this assumes that we can in some sense evaluate a solution to see if it violates constraints.
  3. Decoder functions that replace the original mapping from genotype to phenotype so that all solutions (i.e., phenotypes) are guaranteed to be feasible. The genotype space and fitness functions are left unchanged, and standard evolutionary operators may be applied. Unlike repair functions, which work on whole solutions, decoder functions typically take constraints into account as a solution is constructed from partial components.

13.2.1 Penalty functions

Penalty functions modify the original fitness function $f(\bar{x})$ applied to a candidate solution $\bar{x}$ such that $f'(\bar{x}) = f(\bar{x}) + P(d(\bar{x}, F))$, where $d(\bar{x}, F)$ is a distance metric of the infeasible point to the feasible region F (this might be simply a count of the number of constraints violated). The penalty function P is zero for feasible solutions, and it increases with distance from the feasible region (for minimisation problems).
Penalty function methods are especially suited to problems with disjoint feasible regions, or where the global optimum lies on (or near) the constraint boundary. And penalty functions can be classified as static, dynamic or adaptive.
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.
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.
An alternative, which can be seen as the logical extension of this approach, is the behavioural memory algorithm. Here 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. **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.

13.2.2 Repair functions

The use of repair algorithms for solving COPs with EAs can be seen as a special case of adding local search to the EA. In this case the aim of the local search is to reduce (or remove) the constraint violation, rather than to simply improve the value of the fitness function, as is usually the case.

13.2.3 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.

13.2.4 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)

Decoder functions generally introduce a lot of redundancy into the original genotype space. This arises when the new mapping is many- to-one, meaning that a number of potentially radically different genotypes may be mapped onto the same phenotype, and only a subset of the phenotype space can be reached.

14 Interactive evolutionary algorithms

This chapter discusses the topic of interactive evolution, where the measure of a solution's fitness is provided by a human's subjective judgement, rather than by some predefined model of a problem. Applications of Interactive Evolutionary Algorithms (IEAs) range from capturing aesthetics in art and design, to the personalisation of artifacts such as medical devices.

14.1 Characteristics of interactive evolution

The defining feature of IEAs is that the user effectively becomes part of the system, acting as a guiding oracle to control the evolutionary process, where human interference changes the reproductive process. Based on functional (faster horses, higher yields) or aesthetic (nicer cats, brighter flowers) judgements, a supervisor selects the individuals that are allowed to reproduce. Over time, new types of individuals emerge that meet the human expectations better than their ancestors.

14.1.1 The effect of time

From the perspective of evolutionary time, there is a need to avoid lengthy evolution over hundreds or thousands of generations (because of fatigue and limited attention span of a human), and instead focus on making rapid gains to fit in with human needs.

14.1.2 The effect of context: What has gone before

Closely related to the issue of time, or the length of the search process, is that of context. By this we mean that human expectations, and ideas about what is a good solution, change in response to what evolution produces. If (as we hope) people are pleasantly surprised by what they see, then their expectations rise. This can mean that a solution might be judged average early in a run, but only sub-standard after more solutions have been presented. Alternatively, after a few generations of viewing similar solutions, a user may decide that they are in a 'blind alley'. Both of these imply a need to generate a diverse range of solutions - either by increasing the rate of mutation or by a restart mechanism, or by maintaining some kind of archive.

14.1.3 Advantages of IEAs

- Handling situations with no clear fitness function. If the reasons for preferring certain solutions cannot be formalised, no fitness function can be specified and implemented within the EA code. Subjective user selection circumvents this problem. It is also helpful when the objectives and preferences are changeable, as it avoids having to rewrite the fitness function
- Improved search ability. If evolution gets stuck, the user can redirect search by changing his or her guiding principle
- Increased exploration and diversity. The longer the user 'plays' with the system, the more and more diverse solutions will be encountered

14.2 Algorithmic approaches to the challenges of IEAs

14.2.1 Interactive selection and population size

In general, the user can influence selection in various ways. The influence can be very direct, for example, actually choosing the individuals that are allowed to reproduce. Alternatively, it can be indirect — by defining the fitness values or perhaps only sorting the population, and then using one of the selection mechanisms described. In all cases (even in the indirect one) the user's influence is named subjective selection, and in an evolutionary art context the term aesthetic selection is often used.
Population size can be an important issue for a number of reasons. For visual tasks, computer screens have a fixed size and humans need a certain minimum image resolution, which limits how many solutions can be

viewed at once. Another aspect is that if a screenful of solutions are being ranked, the number of pairwise comparisons and decisions to be made grows with the square of the number onscreen. For all these reasons, interactive EAs frequently work with relatively small populations.

14.2.2 Interaction in the variation process

Interactive Artificial Evolution can permit direct user intervention in the variation process. In some cases this is implicit - for example allowing the user to periodically adjust the choice and parameterisation of variation operators.
Explicit forms of control may also be used. For instance, interactive evolutionary timetabling might allow planners to inspect promising solutions, interchange events by hand and place the modified solutions back to the population.
Both types of algorithmic adaptation have in common the desire to maximise the value of each interaction with a user.

14.2.3 Methods for reducing the frequency of user interaction

The third major algorithmic adaptation attempts to reduce the number of solutions a user is asked to evaluate, while simultaneously maintaining large populations and/or using many generations. This is done by use of a surrogate fitness function which attempts to approximate the decisions a human would make. Typically the surrogate fitness models are adaptive and attempt to learn to reflect the users' decisions.

14.3 Interactive evolution as design vs. optimisation

Interactive evolution is often related to evolutionary art and design. It can even be argued that evolution is design, rather than optimisation. From this conceptual perspective the canonical task of (natural or computer) evolution is to design good solutions for specific challenges.
An alternative to parameterised representations is component-based representation. Here a set of low-level components is defined, and solutions are constructed from these components. This is a 'knowledge-lean' representation with possibly no, or only weak, assumptions of relationships between components and the ways they can be assembled. IEAs for evolutionary design commonly use this sort of representation. Also known as Generative and Developmental Systems, these quite naturally give rise to exploration, aiming at identifying novel and good solutions, where novelty can be more important than optimality (which might not even be definable). The basic template for interactive evolutionary design systems consists of five components:

1. A phenotype definition specifying the application-specific kind of objects we are to evolve
2. A genotype representation, where the genes represent (directly or indirectly) a variable number of components that make up a phenotype
3. A decoder, often called growth function or embryogeny, defining the mapping process from genotypes to phenotypes
4. A solution evaluation facility allowing the user to perform selection within the evolutionary cycle in an interactive way
5. An evolutionary algorithm to carry out the search

The basics of such evolutionary art systems are the same as those of evolutionary design in general: some evolvable genotype is specified that encodes an interpretable phenotype. The main feature that distinguishes the applica- tion of IEAs to art from other forms of design lies in the intention: the evolved objects are simply to please the user, and need not serve any practical purpose.

15 Co-evolutionary systems

Evaluating a solution may involve an element of random noise, but does not particularly depend on the context in which it is done. However, there are two obvious scenarios in which the set-up does not really hold. The first occurs when a solution represents some strategy or design that works in opposition to some competitor that is itself adapting. Example: adversarial game-playing such as chess. The second comes about when a solution being evolved does not represent a complete solution to a problem, but instead can only be evaluated as part of a greater whole, that together accomplishes some task. Example: evolution of a set of traffic-light controllers, each to be sited on a different junction, with fitness reflecting their joint performance in reducing congestion over a day's simulated traffic. Both of these are examples of co-evolution.

15.1 Co-evolution in nature

The effect of other species in determining the fitness of an organism can be positive - for example, the pollination of plants by insects feeding on their nectar - or negative - for example, the eating of rabbits by foxes. Biologists tend to use the terms mutualism and symbiosis to refer to the co-adaptation of species in a mutually beneficial way, and the terms predation or parasitism to refer to relationships in which one species has a negative effect on the survival and reproductive success of the other (antagonism).
Since evolution affects all species, the net effect is that the landscape 'seen' by each species is affected by the configuration of all the other interacting species, i.e., it will move. This process is known as co-evolution.
Despite the additional complications of co-evolutionary models, they hold some significant advantages that have been exploited within EAs to aid the generation of solutions to a range of difficult problems.

15.2 Cooperative co-evolutions

Co-evolutionary models in which a number of different species, each representing part of a problem, cooperate in order to solve a larger problem.
Advantage
It permits effective function decomposition; each sub-population is effectively solving a much smaller, more tractable problem
Disadvantage
it relies on the user to subdi- vide the problem which may not be obvious from the overall specification

15.2.1 Partnering strategies

When cooperating populations are used, a major issue is that of deciding how a solution from one population should be paired with the necessary others in order to gain a fitness evaluation.
Within the field of cooperative co-evolution it is worth mentioning the use of automatically defined functions within GP. In this extension of GP, the function set is extended to include calls to functions that are themselves being evolved in parallel, in separate populations. The great advantage of this is in permitting the evolution of modularity and code reuse.

15.3 Competitive co-evolution

In the competitive co-evolution paradigm individuals compete against each other to gain fitness at each other's expense. These individuals may belong to the same or different species, in which case it is arguably more accurate to say that the different species are competing against each other.
As with cooperative co-evolution, the fitness landscapes will change as the populations evolve, and the choice of pairing strategies can have a major effect on the observed behaviour. When the competition arises within a single population, the most common approaches are to either pair each strategy against each other, or just against a randomly chosen fixed-size sample of the others. If the competition arises between different populations, then a pairing strategy must be chosen for fitness evaluation, as it is for cooperative co-evolution.

The main advantage of the method is that it is self-scaling: early in the run relatively poor solutions may survive, for their competitors are not strong either. But as the run proceeds and the average strength of the population increases, the difficulty of the fitness function is continually scaled.

15.4 Summary of algorithmic adaptations for context-dependent evaluation

The choice of context, or equivalently the pairing strategy, can have a significant effect on how well EAs perform in this type of situation.
The second major algorithmic adaptation is the incorporation of some kind of history into the evaluation - often in the form of an archive of historically ‘good solutions against which evolving solutions are periodically tested. The reason for this is to avoid the problem of cycling: a phenomenon where evolution repeatedly moves through a series of solutions rather than making advances.

16 Theory

In this chapter we present a brief overview of some of the approaches taken to analysing and modelling the behaviour of evolutionary algorithms. The Holy Grail of these efforts is the formulation of predictive models describing the behaviour of an EA on arbitrary problems, and permitting the specification of the most efficient form of optimiser for any given problem. However, this is unlikely ever to be realised; evolutionary algorithms are hugely complex systems, involving many random factors.

16.1 Competing hyperplanes in binary spaces: The schema theorem

What is a schema?
A schema is simply a hyperplane in the search space, and the common representation of these for binary alphabets uses a third symbol - # the "don't care" symbol. Thus for a five-bit problem, the schema 11### is the hyperplane defined by having ones in its first two positions. All strings meeting this criterion are instances, or examples, of this schema. The fitness of a schema is the mean fitness of all strings that are examples of it; in practice this is often estimated from samples when there are many such strings. Global optimisation can be seen as the search for the schema with zero "don't care" symbols, which has the highest fitness.
Aggregation: all possible strings are grouped together in some way and the evolution of the aggregated variables is modelled.
Two features to describe schemata. The order is the number of positions in the schema that do not have the # sign. The defining length is the distance between the outermost defined positions (which equals the number of possible crossover points between them).

Holland's formulation for the SGA
Schema theorem; schemata of above-average fitness would increase their number of instances within the population from generation to generation.

Schema-based analysis of variation operators
Operator bias describes the interdependence of Pd(H,x) (= probability that the action of an operator x on an instance of a schema H is to destroy it) on d(H),o(H) and x, which takes two forms:

- If an operator x displays positional bias it is more likely to keep together bits that are close together in the representation. This has the effect that given two schemata H_1, H_2 with $f(H_1) = f(H_2)$ and $d(H_1)<d(H_2)$, then $Pd(H_1, x)<Pd(H_2, x)$.
- If an operator displays distributional bias then the probability that it will transmit a schema is a function of $o(H)$.

Walsh analysis and deception
Building block hypothesis: GAs begin by selecting amongst competing short low-order schemata, and then progressively combine them to create higher-order schemata, repeating this process until (hopefully) a schema of length $l - 1$ and order l, i.e., the globally optimal string, is created and selected for. Note that for two schemata to compete they must have fixed bits (1 or 0) in the same positions
Walsh functions: a set of functions that provide a natural basis for the decomposition of a binary search landscape. They can be thought of as equivalent to the way that Fourier transforms decompose a complex signal in the time do- main into a weighted sum of sinusoidal waves, which can be represented and manipulated in the frequency domain. Walsh transforms form an easily manipulable way of analysing binary search landscapes, with the added bonus that there is a natural correspondence between Walsh partitions and schemata.

16.2 Criticism and recent extensions of the schema theorem

Hitchhiking; an unfavourable allele becomes established in the population because of an early association with an instance of a high-fitness schema

16.3 Gene linkage: Identifying and recombining building blocks

Three approaches to the identification of linkage groups.
The first of these they refer to as the "direct detection of bias in probability distributions". Common to all of these approaches is the notion of first identifying a factorisation of the problem into a number of subgroups, such that a given statistical criterion is minimised, based on the current population.
The other two approaches identified by Munetomo and Goldberg use more traditional recombination and mutation stages, but bias the recombination operator to use linkage information.
Second approach: first-order statistics based on pairwise perturbation of allele values are used to identify the blocks of linked genes that algorithms manipulate.
The third approach identified does not calculate statistics on the gene interactions based on perturbations, but rather adapts linkage groups ex- plicitly or implicitly via the adaptation of recombination operators.

16.4 Dynamical systems

The dynamical systems approach:

- Start with an n-dimensional vector $\bar{p}$, where n is the size of the search space, and the component p_i^t represents the proportion of the population that is of type i at iteration t.
- Construct a mixing matrix M representing the effects of recombination and mutation, and a selection matrix F representing the effects of the selection operator on each string for a given fitness function.
- Compose a genetic operator $G = F \circ M$ as the matrix product of these two functions.
- The action of the GA to generate the next population can then be characterised as the application of this operator G to the current population: $\bar{p}^{t+1} = G\bar{p}^t$.

16.5 Markov chain analysis

Describe a system as a discrete-time Markov chain provided that the following conditions are met:

- At any given time the system is in one of a finite number (N) of states.
- The probability that the system will be in any given state X^{t+1} in the next iteration is solely determined by the state that it is in at the current iteration X^t, regardless of the previous sequence of states.

The impact of the second condition is that we can define a transition matrix Q where the entry Q_{ij} contains the probability of moving from state i to state j in a single step.
There are a finite number of ways in which we can select a finite sized population from a finite search space, so we can treat any EA working within such a representation as a Markov chain whose states represent the different possible populations.

16.6 Statistical mechanics approaches

The statistical mechanics approach focuses on modelling the behaviour of a few variables that characterise the system (macroscopic approach). The approach does not pretend to offer predictions other than of the population mean, variance and so on, so it cannot be used for all the aspects of behaviour one might desire to model.

16.7 Reductionist approaches

So far we have described a number of methods for modelling the behaviour of EAs that attempt to make predictions about the composition of the next population by considering the effect of all the genetic operators on the current population (holistic approaches).
An alternative methodology is to take a reductionist approach, and examine parts of the system separately. The advantage of taking a reductionist approach is that frequently it is possible to derive analytical results and insights when only a part of the problem is considered

16.8 Black box analysis

'Black box complexity' approach model the run-time complexity of algorithms on specific functions - that is to say on their expected time from an arbitrary starting point to reaching the global optima.

16.9 Analysing EAs in continuous search spaces

Much of the dynamics of ESs can be recovered from simpler models concerning the evolution of two macroscopic variables.
The first of the variables modelled is the progress rate, which measures the distance of the centre of mass of the population from the global optimum (in variable space) as a function of time. The second is the quality gain, which measures the expected improvement in fitness between generations.

16.10 No Free Lucnh theorem

No Free Lunch theorem (NFL) says that if we average over the space of all possible problems, then all non-revisiting black box algorithms will exhibit the same performance. By non-revisiting we mean that the algorithm does not generate and test the same point in the search space twice. By black box algorithms we mean those that do not incorporate any problem or instance-specific knowledge.

17 Evolutionary robotics

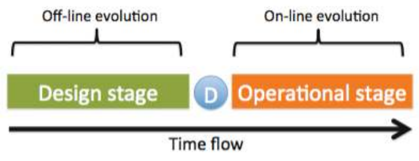
In this chapter we discuss evolutionary robotics (ER), where evolutionary algorithms are employed to design robots.

17.1 What is it all about?

Evolutionary robotics can be characterised as problems involving physical environments. Evolutionary systems in this domain have populations whose members are embedded in real time and real space. The algorithmic goal is formulated in terms of functional, rather than structural properties. In other words, it is some exhibited behaviour we are after.

17.2 Offline and online evolution of robots

Offline evolution: this means that an evolutionary algorithm to find a good controller is applied before the operational period of the robot. When the user is satisfied with the evolved controller, then it is deployed (installed on the physical robot) and the operational stage can start
Online evolution: this implies that evolutionary operators can change the robot's control software even after its deployment



17.3 Evolutionary robotics: The problems are different

Properties that make robotics a very interesting context for evolutionary computing. To begin with, let us consider the problems, in particular the fitness functions for robotics. These exhibit a number of particular features that may not be unique one by one, but together they form a very challenging combination.

- the fitness function is very noisy
  Noise is inherent in the physical world
- the fitness function is very costly
  Controller must be tested under different initial conditions - ultimately this means that many time-consuming measurements are needed to assess the fitness of a given controller
- the fitness function is very complex
  the link between actually controllable values in the genotypes and robot behaviour is complex and ill-defined without analytical models
- there may not be an explicit fitness function at all
  EC can be used in an objective-free fashion to invent robots that are well suited to some (previously unknown and/or changing) environment, in such cases robots do not have a quantifiable level of fitness
- the fitness landscape has 'no-go areas'
  testing candidate solutions that waste time must be avoided during the evolutionary search

Another special property here is that robots can be passive as well as active components of the evolutionary system. On the one hand, robots are passive from the evolutionary algorithm perspective if they just passively undergo selection and reproduction. On the other hand, robots can be active from the evolutionary algorithm perspective because they have processors on board that can perform computations and execute evolutionary operators.

17.4 Evolutionary robotics: The algorithms are different

As mentioned above, standard EAs can be used to help design robots in an offline fashion. However, new types of EAs are needed when controllers, morphologies or both are evolved on the fly. Despite a handful of promising publications, there is not much know-how on such EAs, indicating the need for further research. In the following we try to identify some of the related issues, arranged by the following aspects:

- the overall system architecture (evolutionary vs. robotic components)
  By system architecture we mean the structural and functional relations between the evolutionary and the robotics system components; for example only in a computer environment, or real robots and computer combined
  Online architectures are aimed at evolving robot controllers during their operational period. Here we can distinguish between centralised and distributed systems.
  One type of centralised system works as the online variant of the classic architecture. A master computer oversees the robots and runs the evolutionary process (fig 17.3, left). In the other type of centralised online system the computer running the evolutionary algorithm is internal to the robot, see Fig. 17.3, right.

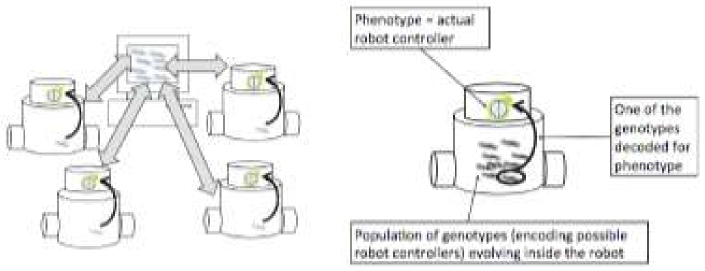

Phenotype = actual robot controller
One of the genotypes decoded for phenotype
Population of genotypes (encoding possible robot controllers) evolving inside the robot

Fig. 17.3. Centralised online evolution architectures. Left: an external master computer oversees the robots and runs the evolutionary process. Right: an internal computer (the robot's own processor) runs an encapsulated evolutionary process, where selection and reproduction do not require information from other robots

Distributed online architectures also come in two flavours. In a pure distributed system each robot carries one genome that encodes its own phenotypic features, see Fig. 17.4, left. In a hybrid system the encapsulated and the pure distributed approaches are combined, as shown in Fig. 17.4, right.

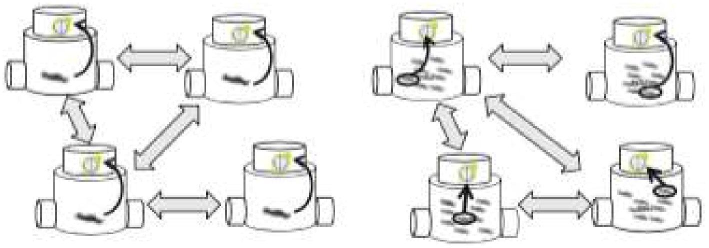


Fig. 17.4. Distributed online evolution architectures. Left: distributed system with one-robot–one-genome. Selection and reproduction require interaction between more robots. Right: a hybrid system is encapsulated and distributed. Encapsulated populations form islands that evolve independently but can crossfertilize.

- the evolutionary operators (variation and selection)
  Two prominent requirements; first and foremost, reproduction must be constrained to physically viable candidate solutions. The second requirement concerns the speed of evolution. Because of the online character of the robotic application, rapid progress is desirable.
  There are also special considerations for selection in the distributed cases. Selection operators will act on local and partial information, using fitness data of only a fraction of the whole population. In principle, this can be circumvented by using some form of epidemic or gossiping protocols that help in estimating global information.
- the fitness evaluations
  Perhaps the most important robotics- specific feature is their duration. The parameter(s) that determine the duration of fitness evaluations should have robust parameter values that work over a wide range of circumstances, or a good parameter control method should be applied that adjusts the values appropriately on the fly.
- the use of simulators
  Use simulations within robots for continuous self-modelling. In principle, this means that fitness evaluations can be performed without real-life trials during an online evolutionary process - it can save time, it can save robots, and it can help experimenters learn about the problem.