Natural Computing Series

Anthony Brabazon
Michael O'Neill
Seán McGarraghy

Natural Computing Algorithms

Springer

Anthony Brabazon • Michael O'Neill
Seán McGarraghy

Natural Computing Algorithms

Springer

Anthony Brabazon
Natural Computing Research
& Applications Group
School of Business
University College Dublin
Dublin, Ireland

Michael O'Neill
Natural Computing Research
& Applications Group
School of Business
University College Dublin
Dublin, Ireland

Seán McGarraghy
Natural Computing Research
& Applications Group
School of Business
University College Dublin
Dublin, Ireland

ISSN 1619-7127
Natural Computing Series
ISBN 978-3-662-43630-1 ISBN 978-3-662-43631-8 (eBook)
DOI 10.1007/978-3-662-43631-8

Library of Congress Control Number: 2015951433

Springer Heidelberg New York Dordrecht London

Printed on acid-free paper

Springer-Verlag GmbH Berlin Heidelberg is part of Springer Science+Business Media (www.springer.com)

Contents

Part V Developmental and Grammatical Computing

1 Introduction

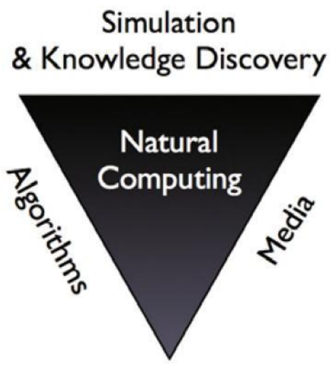


Fig. 1.1. The three facets of natural computing: 1) natural systems as computational media (e.g., DNA and molecular computing), 2) simulation of natural systems with the potential for knowledge discovery, and 3) algorithms inspired by the natural world

Although there is no unique definition of the term natural computing, most commonly the field is considered to consist of three main strands of enquiry: see Fig. 1.1. The first strand concerns the use of natural materials and phenomena for computational purposes such as DNA and molecular computing (*computing in vivo*); the second strand concerns the application of computer simulations to replicate natural phenomena in order to better understand those phenomena (e.g., artificial life, agent based modelling and computational biology); and the third strand, which forms the subject matter of this book, is concerned with the development of computational algorithms which draw their metaphorical inspiration from systems and phenomena that occur in the natural world. These algorithms can be applied to a multiplicity of real-world problems including optimisation, classification, prediction, clustering, design and model induction. The objective of this book is to provide an introduction to a broad range of natural computing algorithms.

1.1 Natural Computing Algorithms: An Overview

Natural computing is inherently multidisciplinary as it draws inspiration from a diverse range of fields of study including mathematics, statistics, computer science and the natural sciences of biology, physics and chemistry. As our understanding of natural phenomena has deepened so too has our recognition that many mechanisms in the natural world parallel computational processes and can therefore serve as an inspiration for the design of problem-solving algorithms (defined simply as a set of instructions for solving a problem of interest). In this book we introduce and discuss a range of families of natural computing algorithms. Figure 1.2 provides a high-level taxonomy of the primary methodologies discussed in this book which are grouped under the broad umbrellas of evolutionary computing, social computing, neurocomputing, immunocomputing, developmental and grammatical computing, physical computing, and chemical computing.

1.1.1 Biologically Inspired Algorithms

An interesting aspect of biological systems at multiple levels of scale (ecosystems, humans, bacteria, cells, etc.) which suggests that they may provide good sources of inspiration for solving real-world problems is that the very process of survival is itself a challenging problem! In order to survive successfully organisms need to be able to find resources such as food, water, shelter and mates, whilst simultaneously avoiding predators. It is plausible that mechanisms which have evolved in order to assist survivability of organisms in these settings, such as sensing, communication, cognition and mobility could prove particularly useful in inspiring the design of computational algorithms. Virtually all biological systems exist in high-dimensional (i.e., many factors impact on their survival), dynamic environments and hence biologically inspired algorithms may be of particular use in problem solving in these conditions.

Although many aspects of biological systems are noteworthy, a few characteristics of biological systems which provide food for thought in the design of biologically inspired algorithms include: the importance of the population; the emphasis on robustness (survival) rather than on optimality; and the existence of multilevel adaptive processes.

Populational Perspective

In many biologically inspired algorithms the search for good solutions takes place within a *population* of potential solutions. As in biological settings, indi-

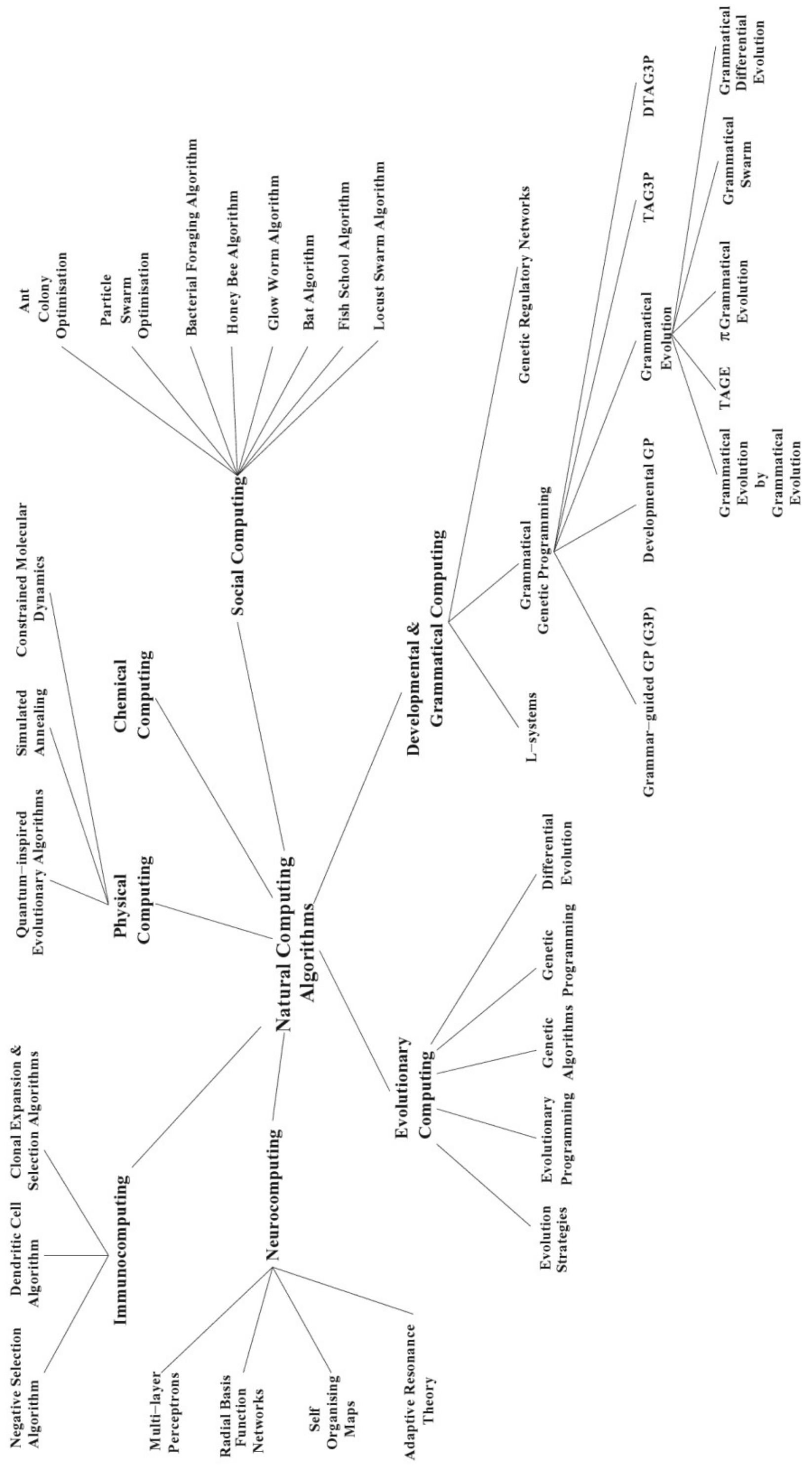


Fig. 1.2. A taxonomy of the nature inspired algorithms discussed in this book

viduals in a population can be considered an *individual hypothesis* (or learning trial) in the game of survival. From a species point of view, maintaining a dispersed population of individuals reduces the chance that environmental change will render the entire species extinct.

In contrast, in many traditional (non-natural-computing) optimisation algorithms, the paradigm is to generate a single trial solution and then iteratively improve it. Consider for example, a simple random hill-climbing optimisation algorithm (Algorithm 1.1) where an individual solution is iteratively improved using a greedy search strategy. In greedy search, any change in a solution which makes it better is accepted. This implies that a hill-climbing algorithm can find a local, but not necessarily the global, optimum (Fig. 1.3). This makes the choice of starting point a critical one. Note that this is so for all hill-climbing algorithms, including those that exploit information found so far (e.g., Newton's algorithm uses slope information, while the Nelder–Mead algorithm does not need slopes but uses several previous points' objective function values).

Algorithm 1.1: Hill Climbing Algorithm

```
Randomly generate a solution $x$;
Calculate the objective function value $f(x)$ for the solution;

repeat
    Randomly mutate solution;
    if new solution is better than the current solution then
        Replace the current solution with the new one
    end
until terminating condition;
```

Many of the biologically inspired algorithms described in this book maintain and successively update a population of potential solutions, which in the ideal case provides a good coverage (or sampling) of the environment in which we are problem-solving, resulting in a form of parallel search. This of course assumes the process that generates the first population disperses the individuals in an appropriate manner so as to maximise coverage of the environment. Ideally as search progresses it might be desirable to maintain a degree of dispersion to avoid premature convergence of the population to local optima. It is the existence of a population which allows these bioinspired algorithms the potential to achieve global search characteristics, and avoid local optima through the populational dispersion of individuals.

Dispersion and Diversity

It is important to highlight this point that we should not confuse the notions of diversity (of objective function values) and dispersion. Often we

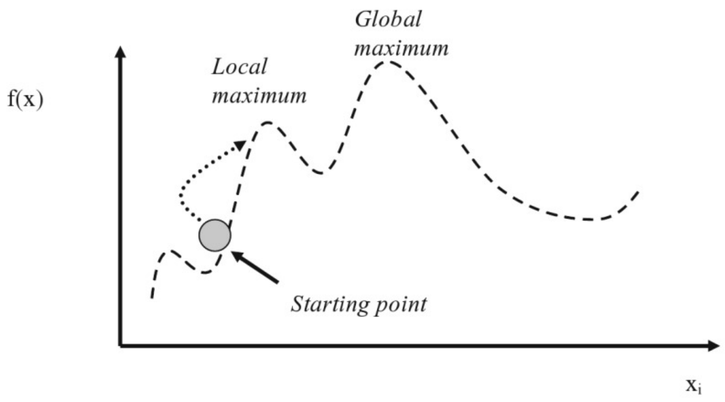


Fig. 1.3. A hill-climbing algorithm will find a local optimum. Due to its greedy search strategy it cannot then escape from this as it would require a 'downhill' move to traverse to the global optimum

(over)emphasise the value of diversity within a population. From a computational search perspective it is arguably more valuable to focus on the dispersion (or coverage) of the population. It is possible to have an abundance of diversity within a population; yet, at the same time, the population could be converged on a local optimum. However, a population which has a high value of dispersion is less likely to be converged in this manner. Figure 1.4 illustrates the difference between dispersion and diversity. The importance of dispersion is brought into sharp focus if we expose the population to a changing environment where the location of the optima might change/move over time. A population which maintains dispersion may have a better chance to adapt to an environment where the global optimum moves a relatively far distance from its current location.

Communication

Another critical aspect of most of the algorithms described in this book is that the members of the population do not search in isolation. Instead they can *communicate* information on the quality of their current (or their previous) solution to other members of the population. Communication is interpreted broadly here to mean exchange of information between members of the population, and so might take various forms: from chemical signals being left in the environment of a social computing algorithm, which can be sensed by individuals in the population; to the exchange of genes in an evolutionary algorithm. This information is then used to bias the search process towards areas of better solutions as the algorithm iterates.

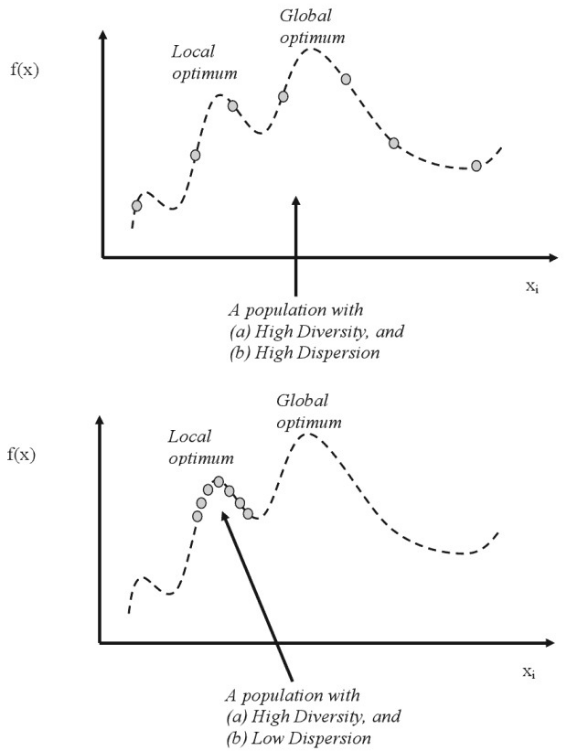


Fig. 1.4. An illustration of the difference between a dispersed (top) and a diverse (bottom) population. It is possible to have a large amount of diversity (e.g., a wide range of objective function values) but still be converged on a local optimum

Robustness

'Survival in a dynamic environment' is the primary aim in many biological systems. Therefore the implicit driver for organisms is typically to uncover and implement survival strategies which are 'good enough' for current conditions and which are *robust* to changing environmental conditions. Optimality is a fleeting concept in dynamic environments as (for example), the location of good food resources last week may not hold true next week.

Adaptiveness

Adaptiveness (new learning) occurs at multiple levels and timescales in biological systems, ranging from (relatively) slow genetic learning to (relatively) fast lifetime learning. The appropriate balance between speed of adaptation and the importance of memory (or *old learning*) depends on the nature of the dynamic environment faced by the organism. The more dynamic the environment, the greater the need for adaptive capability and the less useful is memory.

An interesting model of adaptation in biological systems is outlined in Sipper et al. [581] and is commonly referred to as the POE model. This model distinguishes between three levels of organisation in biological systems:

i. phylogeny (P),
ii. ontogeny (O), and
iii. epigenesis (E).

Phylogeny concerns the adaptation of genetic code over time. As the genome adapts and differentiates, multiple species, or phylogeny, evolve. The primary mechanisms for generating diversity in genetic codes are mutation, and, in the case of sexual reproduction, recombination. The continual generation of diversity in genetic codings facilitates the survival of species, and the emergence of new species, in the face of changing environmental conditions. Much of the research around evolutionary computation to date exists along this axis of adaptation.

Ontogeny refers to the development of a multicellular organism from a zygote. While each cell maintains a copy of the original genome, it specialises to perform specific tasks depending on its surroundings (cellular differentiation). In recent years there has been increasing interest in developmental approaches to adaptation with the adoption of models such as artificial genetic regulatory networks.

Epigenesis is the development of systems which permit the organism to integrate and process large amounts of information from its environment. The development and working of these systems is not completely specified in the genetic code of the organism and hence are referred to as 'beyond the genetic' or epigenetic. Examples include the immune, the nervous and the endocrine systems. While the basic structure of these systems is governed by the organism's genetic code, they are modified throughout the organism's lifetime as a result of its interaction with the environment. For example, a human's immune system can maintain a memory of pathogens that it has been exposed to (the acquired immune system). The regulatory mechanism which controls the expression of genes is also subject to epigenetic interference. For example, chemical modification (e.g., through methylation) of regulatory regions of the genome can have the effect of silencing (turning off or dampening) the expression of a gene(s). The chemical modification can arise due to the environmental state in which the organism lives. So the environment can effect which genes are expressed (or not) thereby indirectly modifying an organism's genetic makeup to suit the conditions in which it finds itself.

In complex biological organisms, all three levels of organisation are interlinked. However, in assisting us in thinking about the design of biologically inspired algorithms, it can be useful to consider each level of organisation (and their associated adaptive processes) separately (Fig. 1.5). Summarising

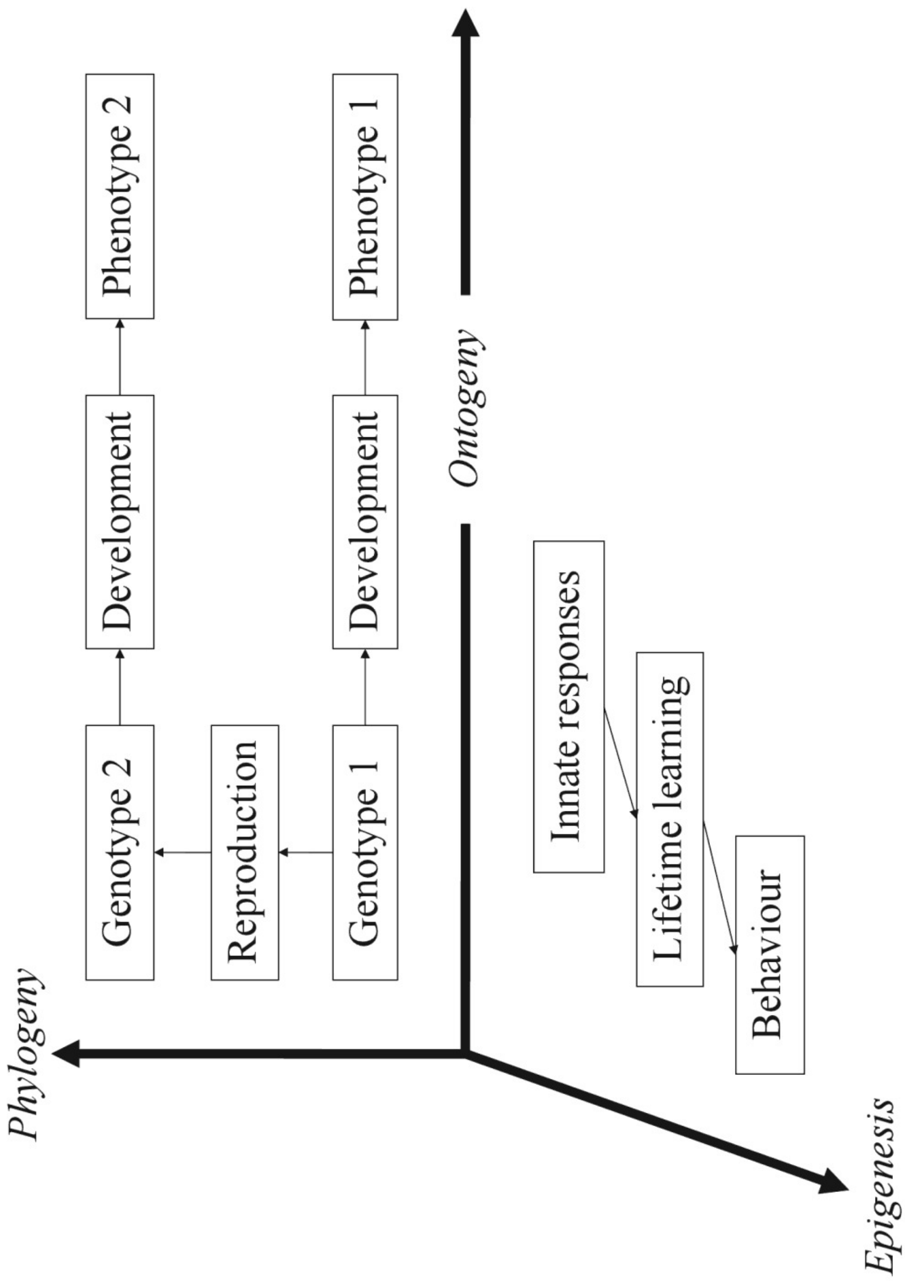


Fig. 1.5. Three levels of organisation in biological systems

the three levels, Sipper et al. [581] contends that they embed the ideas of *evolution*, structural *development* of an individual, and *learning* through environmental interactions.

Most biologically inspired algorithms draw inspiration from a single level of organisation but it is of course possible to design hybrid algorithms which draw inspiration from more than one level. For example, neuroevolution, discussed in Chap. 15, combines concepts of both evolutionary and lifetime learning, and evolutionary–development (evo–devo) approaches (e.g., see Chap. 21) hybridise phylogenetic and ontogenetic adaptation.

1.1.2 Families of Naturally Inspired Algorithms

A brief overview of some of the main families of natural computing algorithms is provided in the following paragraphs. A more detailed discussion of each of these is provided in later chapters.

Evolutionary Computing

Evolutionary computation simulates an evolutionary process on a computer in order to breed good solutions to a problem. The process draws high-level inspiration from biological evolution. Initially a population of potential solutions are generated (perhaps randomly), and these are iteratively improved over many simulated generations. In successive iterations of the algorithm, fitness based selection takes place within the population of solutions. Better solutions are preferentially selected for survival into the next generation of solutions, with diversity being introduced in the selected solutions in an attempt to uncover even better solutions over multiple generations. Algorithms that employ an evolutionary approach include genetic algorithms (GAs), evolutionary strategies (ES), evolutionary programming (EP) and genetic programming (GP). Differential evolution (DE) also draws (loose) inspiration from evolutionary processes.

Social Computing

The social models considered in this book are drawn from a *swarm* metaphor. Two popular variants of swarm models exist, those inspired by the flocking behaviour of birds and fish, and those inspired by the behaviour of social insects such as ants and honey bees. The swarm metaphor has been used to design algorithms which can solve difficult problems by creating a population of problem solvers, and allowing these to communicate their relative success in solving the problem to each other.

Neurocomputing

Artificial neural networks (NNs) comprise a modelling methodology whose inspiration arises from a simplified model of the workings of the human brain. NNs can be used to construct models for the purposes of prediction, classification and clustering.

Immunocomputing

The capabilities of the natural immune system are to recognise, destroy and remember an almost unlimited number of foreign bodies, and also to protect the organism from misbehaving cells in the body. Artificial immune systems (AIS) draw inspiration from the workings of the natural immune system to develop algorithms for optimisation and classification.

Developmental and Grammatical Computing

A significant recent addition to natural computing methodologies are those inspired by developmental biology (developmental computing) and the use of formal grammars (grammatical computing) from linguistics and computer science. In natural computing algorithms grammars tend to be used in a generative sense to construct sentences in the language specified by the grammar. This generative nature is compatible with a developmental approach, and consequently a significant number of developmental algorithms adopt some form of grammatical encoding. As will be seen there is also an overlap between these algorithms and evolutionary computation. In particular, a number of approaches to genetic programming adopt grammars to control the evolving executable structures. This serves to highlight the overlapping nature of natural systems, and that our decomposition of natural computing algorithms into families of inspiration is one of convenience.

1.1.3 Physically Inspired Algorithms

Just as biological processes can inspire the design of computational algorithms, inspiration can also be drawn from looking at physical systems and processes. We look at three algorithms which are inspired by the properties of interacting physical bodies such as atoms and molecules, namely simulated annealing, quantum annealing, and the constrained molecular dynamics algorithm. One interesting strand of research in this area is drawn from a quantum metaphor.

Quantum Inspired Algorithms

Quantum mechanics seeks to explain the behaviours of natural systems that are observed at very short time or distance scales. An example of a system is a subatomic particle such as a free electron. Two important concepts underlying quantum systems are the *superposition of states* and *quantum entanglement*. Recent years have seen the development of a series of quantum inspired hybrid algorithms including quantum inspired evolutionary algorithms, social computing, neurocomputing and immunocomputing. A claimed benefit of these algorithms is that because they use a quantum inspired representation, they can potentially maintain a good balance between exploration and exploitation. It is also suggested that they could offer computational efficiencies.

1.1.4 Plant Inspired Algorithms

Plants represent some 99% of the eukaryotic biomass of the planet and have been highly successful in colonising many habitats with differing resource potential. Just like animals or simpler organisms such as bacteria (Chap. 11), plants have evolved multiple problem-solving mechanisms including complex food foraging mechanisms, environmental-sensing mechanisms, and reproductive strategies. Although plants do not have a brain or central nervous system, they are capable of sensing environmental conditions and taking actions which are 'adaptive' in the sense of allowing them to adjust to changing environmental conditions. These features of plants offer potential to inspire the design of computational algorithms and a recent stream of work has seen the development of a family of plant algorithms. We introduce a number of these algorithms and highlight some current areas of research in this subfield.

1.1.5 Chemically Inspired Algorithms

Chemical processes play a significant role in many of the phenomena described in this book, including (for example) evolutionary processes and the workings of the natural immune system. However, so far, chemical aspects of these processes have been largely ignored in the design of computational algorithms. An emerging stream of study is beginning to remedy this gap in the literature and we describe an optimisation algorithm inspired by the processes of chemical reactions.

1.1.6 A Unified Family of Algorithms

Although it is useful to compartmentalise the field of Natural Computing into different subfields, such as Evolutionary and Social Computing, for the purposes of introducing the material, it is important to emphasise that this does not actually reflect the reality of the natural world around us. In nature all of these learning mechanisms coexist and interact forming part of a larger natural, complex and adaptive system encompassing physical, chemical, evolutionary, immunological, neural, developmental, grammatical and social processes, which, for example, are embodied in mammals. In much the same way that De Jong advocated for a unified field of Evolutionary Computation [144], we would favour a unification of all the algorithms inspired by the natural world into the paradigm of Natural Computing and Natural Computing Algorithms. Increasingly we are seeing significant overlaps between the different families of algorithms, and upon real-world application it is common to witness their hybridisation (e.g., neuroevolution, evo–devo etc.). We anticipate that the future of natural computing will see the integration of many of these seemingly different approaches into unified software systems working together in harmony.

1.1.7 How Much Natural Inspiration?

An obvious question when considering computational algorithms which are inspired by natural phenomena is how accurate does the metaphor need to be? We consider that the true measure of usefulness of a natural computing algorithm is not its degree of veracity with (what we know of) nature, but rather its effectiveness in problem solving; and that an intelligent designer of algorithms should incorporate ideas from nature — while omitting others — so long as these enhance an algorithm's problem-solving ability. For example, considering quantum inspired algorithms, unless we use quantum computers, which to date are experimental devices and not readily available to the general reader, it is not possible to efficiently simulate effects such as entanglement; hence such algorithms while drawing a degree of inspiration from quantum mechanics must of necessity omit important, even vital, features of the natural phenomenon from which we derive inspiration.

1.2 Structure of the Book

The field of natural computing has expanded greatly in recent years beyond its evolutionary and neurocomputing roots to encompass social, immune system, physical and chemical metaphors. Not all of the algorithms discussed in this book are fully explored as yet in terms of their efficiency and effectiveness. However, we have deliberately chosen to include a wide range of algorithms in order to illustrate the diversity of current research into natural computing algorithms.

The remainder of this book is divided into eight parts. Part I starts by providing an overview of evolutionary computation (Chap. 2), and then proceeds to describe the genetic algorithm (Chaps. 3 and 4), evolutionary strategies and evolutionary programming (Chap. 5), differential evolution (Chap. 6), and genetic programming (Chap. 7). Part II focusses on social computing and provides coverage of particle swarm optimisation (Chap. 8), insect algorithms (Chaps. 9 and 10), bacterial foraging algorithms (Chap. 11), and other social algorithms (Chap. 12). Part III of the book provides coverage of the main neurocomputing paradigms including supervised learning neural network models such as the multilayer perceptron, recurrent networks, radial basis function networks and support vector machines (Chap. 13), unsupervised learning models such as self-organising maps (Chap. 14), and hybrid neuroevolutionary models (Chap. 15). Part IV discusses immunocomputing (Chap. 16). Part V of the book introduces developmental and grammatical computing in Chap. 17 and provides detailed coverage of grammar-based approaches to genetic programming in Chap. 18. Two subsequent chapters expose in more detail some of grammar-based genetic programming's more popular forms, grammatical evolution and TAG3P (Chaps. 19 and 20), followed by artificial genetic regulatory network algorithms in Chap. 21. Part VI introduces physically inspired computing (Chaps. 22 to 24). Part VII introduces some other paradigms that do not fit neatly into the earlier categories, namely, plant-inspired algorithms in Chap. 25, and chemically inspired computing in Chap. 26. Finally, Part VIII (Chap. 27) outlines likely avenues of future work in natural computing algorithms.

We hope the reader will enjoy this tour of natural computing algorithms as much as we have enjoyed the discovery (and in some cases rediscovery) of these inspiring algorithms during the writing of this book.

Part I

Evolutionary Computing

2

Introduction to Evolutionary Computing

'*Owing to this struggle for life, variations, however slight and from whatever cause proceeding, if they be in any degree profitable to the individuals of a species, in their infinitely complex relations to other organic beings and to their physical conditions of life, will tend to the preservation of such individuals, and will generally be inherited by the offspring. The offspring, also, will thus have a better chance of surviving, for, of the many individuals of any species which are periodically born, but a small number can survive. I have called this principle, by which each slight variation, if useful, is preserved, by the term Natural Selection*'. (Darwin, 1859 [127], p. 115)

Biological evolution performs as a powerful problem-solver that attempts to produce solutions that are at least *good enough* to perform the job of survival in the current environmental context. Since Charles Darwin popularised the theory of Natural Selection, the driving force behind evolution, molecular biology has unravelled some of the mysteries of the components that underpinned earlier evolutionary ideas. In the twentieth century molecular biologists uncovered the existence of DNA, its importance in determining hereditary traits and later its structure, unlocking the key to the genetic code. The accumulation of knowledge about the biological process of evolution, often referred to as neo-Darwinism, has in turn given inspiration to the design of a family of computational algorithms known collectively as *evolutionary computation*. These evolutionary algorithms take their cues from the biological concepts of natural selection and the fact that the heritable traits are physically encoded on DNA, and can undergo variation through a series of genetic operators such as mutation and crossover.

A. Brabazon et al., *Natural Computing Algorithms*, Natural Computing Series,
DOI 10.1007/978-3-662-43631-8_2

2.1 Evolutionary Algorithms

Evolutionary processes represent an archetype, whose application transcends their biological root. Evolutionary processes can be distinguished by means of their four key characteristics, which are [57, 92]:

i. a population of entities,
ii. mechanisms for selection,
iii. retention of fit forms, and
iv. the generation of variety.

In biological evolution, species are positively or negatively selected depending on their relative success in surviving and reproducing in the environment. Differential survival, and variety generation during reproduction, provide the engine for evolution [127, 589] (Fig. 2.1).

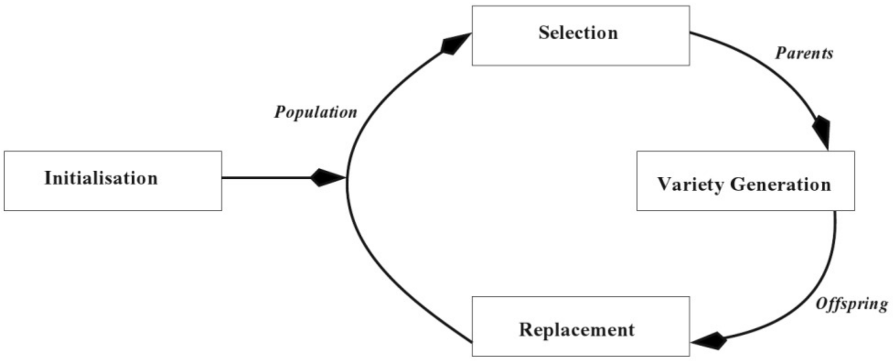


Fig. 2.1. Evolutionary cycle

These concepts have metaphorically inspired the field of evolutionary computation (EC). Algorithm 2.1 outlines the evolutionary meta-algorithm. There are many ways of operationalising each of the steps in this meta-algorithm; consequently, there are many different, but related, evolutionary algorithms. Just as in biological evolution, the selection step is a pivotal driver of the algorithm's workings. The selection step is biased in order to preferentially select better (or 'more fit') members of the current population. The generation of new individuals creates *offspring* or *children* which bear some similarity to their parents but are not identical to them. Hence, each individual represents a trial solution in the environment, with better individuals having increased chance of influencing the composition of individuals in future generations. This process can be considered as a 'search' process, where the objective is to continually improve the quality of individuals in the population.

3

Genetic Algorithm

While the development of the genetic algorithm (GA) dates from the 1960s, this family of algorithms was popularised by Holland in the 1970s [281]. The GA has been applied in two primary areas of research: optimisation, in which GAs represent a population-based optimisation algorithm, and the study of adaptation in complex systems, wherein the evolution of a population of adapting entities is simulated over time by means of a pseudonatural selection process using differential-fitness selection, and pseudogenetic operators to induce variation in the population.

In this chapter we introduce the canonical GA, focussing on its role as an optimising methodology, and discuss the design choices facing a modeller who is seeking to implement a GA.

3.1 Canonical Genetic Algorithm

In genetics, a strong distinction is drawn between the *genotype* and the *phenotype*; the former contains genetic information, whereas the latter is the physical manifestation of this information. Both play a role in evolution as the biological processes of diversity generation act on the genotype, while the 'worth' or *fitness* of this genotype in the environment depends on the survival and reproductive success of its corresponding phenotype. Similarly, in the canonical GA a distinction is made between the encoding of a solution (the 'genotype'), to which simulated genetic operators are applied, and the phenotype associated with that encoding. These phenotypes can have many diverse forms depending on the application of interest. Unlike traditional optimisation techniques the GA maintains and iteratively improves a *population* of solution encodings. Evolutionary algorithms, including the GA, can be broadly characterised as [193]:

$$x[t+1] = r(v(s(x[t]))) \tag{3.1}$$

A. Brabazon et al., *Natural Computing Algorithms*, Natural Computing Series,
DOI 10.1007/978-3-662-43631-8_3

where $x[t]$ is the population of encodings at timestep t, $v(.)$ is the random variation operator (crossover and mutation), $s(.)$ is the selection for mating operator, and $r(.)$ is the replacement selection operator. Once the initial population of encoded solutions has been obtained and evaluated, a reproductive process is applied in which the encodings corresponding to the better-quality, or *fitter*, solutions have a higher chance of being selected for propagation of their genes into the next generation. Over a series of generations, the better adapted solutions in terms of the given fitness function tend to flourish, and the poorer solutions tend to disappear. Just as biological genotypes encode the results of past evolutionary trials, the population of genotypes in the GA also encode a history (or memory) of the relative success of the resulting phenotypes for the problem of interest.

Therefore, the canonical GA can be described as an algorithm that turns one population of candidate encodings and their corresponding solutions into another using a number of stochastic operators. Selection exploits information in the current population, concentrating interest on high-fitness solutions. The selection process is biased in favour of the encodings corresponding to better/fitter solutions and better solutions may be selected multiple times. This corresponds to the idea of *survival of the fittest*. Crossover and mutation perturb these solutions in an attempt to uncover even better solutions. Mutation does this by introducing new gene values into the population, while crossover allows the recombination of fragments of existing solutions to create new ones. Algorithm 3.1 lists the key steps in the canonical genetic algorithm.

An important aspect of the algorithm is that the evolutionary process operates on the *encodings* of solutions, rather than directly on the solutions themselves. In determining the fitness of these encodings, they must first be translated into a solution to the problem of interest, the fitness of the solution determined, and finally this fitness is associated with the encoding (Fig. 3.1).

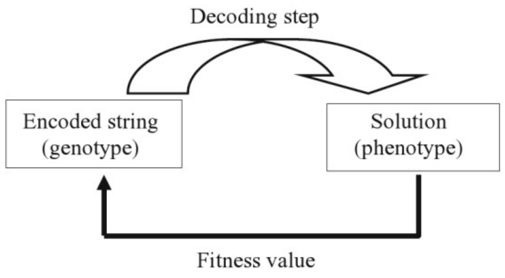


Fig. 3.1. Decoding of genotype into a solution in order to calculate fitness

Algorithm 3.1: Canonical Genetic Algorithm

```
Determine how the solution is to be encoded as a genotype and define the
fitness function;
Create an initial population of genotypes;
Decode each genotype into a solution and calculate the fitness of each of the
$n$ solution candidates in the population;

repeat
    Select $n$ members from the current population of encodings (the parents)
    in order to create a mating pool;
    repeat
        Select two parents randomly from the mating pool;
        With probability $p_{\text{cross}}$, perform a crossover process on the encodings
        of the selected parent solutions, to produce two new (child) solutions;
        Otherwise, crossover is not performed and the two children are
        simply copies of their parents;
        With probability $p_{\text{mut}}$, apply a mutation process to each element of
        the encodings of the two child solutions;
    until $n$ new child solutions have been created;
    Replace the old population with the newly created one (this constitutes a
    generation);
until terminating condition;
```

3.1.1 A Simple GA Example

To provide additional insight into the workings of the canonical GA, a simple numerical example is now provided. Assume that candidate solutions are encoded as a binary string of length 8 and the fitness function $f(x)$ is defined as the number of 1s in the bit string (this is known as the *OneMax* problem). Let $n = 4$ with $p_{\text{cross}} = 0.6$ and $p_{\text{mut}} = 0.05$. Assume also that the initial population is generated randomly as in Table 3.1.

Table 3.1. A sample initial random population

Candidate	String	Fitness
A	10000110	3
B	01101100	4
C	10100000	2
D	01000110	3

Next, a selection process is applied based on the fitness of the candidate solutions. Suppose the first selection draws candidates B and D and the second draws B and A. For each set of parents, the probability that a crossover (or recombination) operator is applied is p_{cross}. Assume that B and D are crossed over between bit position 1 and 2 to produce child candidates E and F (Table 3.2), and that crossover is not applied to B and A.

Table 3.2. Crossover applied to individuals B and D from Table 3.1, after the first element of each binary string, to produce the offspring E and F

Initial Parent	**Candidate B**	**Candidate D**
	0 1101100	0 1000110
Resulting Child	**Candidate E**	**Candidate F**
	0 1000110	0 1101100

Crossover is not applied to B and A; hence the child candidates (G and H) are clones of the two parent candidates (Table 3.3).

Table 3.3. No crossover is applied to B and D; hence the child candidates G and H are clones of their parents

Initial Parent	**Candidate B**	**Candidate A**
	01101100	10000110
Resulting Child	**Candidate G**	**Candidate H**
	01101100	10000110

Finally, the mutation operator is applied to each child candidate with probability p_{mut}. Suppose candidate E is mutated (to a 1) at the third locus, that candidate F is mutated (to a 1) at the seventh locus, and that no other mutations take place. The resulting new population is presented in Table 3.4. By biasing selection for reproduction towards more fit parents, the GA has increased the average fitness of the population in this example from 3 $(=\frac{3+4+2+3}{4})$ to 4 $(=\frac{4+5+4+3}{4})$ after the first generation and we can see that the fitness of the best solution F in the second generation is better than that of any solution in the first generation.

3.2 Design Choices in Implementing a GA

Although the basic idea of the GA is quite simple, a modeller faces a number of key decisions when looking to apply it to a specific problem:

Table 3.4. Population of solution encodings after the mutation operator has been applied

Candidate	String	Fitness
E	01100110	4
F	01101110	5
G	01101100	4
H	10000110	3

- what representation should be used?
- how should the initial population of genotypes be initialised?
- how should fitness be measured?
- how should diversity be generated in the population of genotypes?

Each of these are discussed in the following sections.

3.3 Choosing a Representation

In thinking about evolutionary processes, two distinct mapping processes can be distinguished, one between the genotype and the phenotype, and a second between the phenotype and a fitness measure (Fig. 3.2). In applying the GA, the user must select how the problem is to be *represented*, and there are two aspects to this decision. First, the user must decide how potential solutions (phenotypes) will be encoded onto the genotype. Secondly, the user must decide how individual elements of the genotype will be encoded.

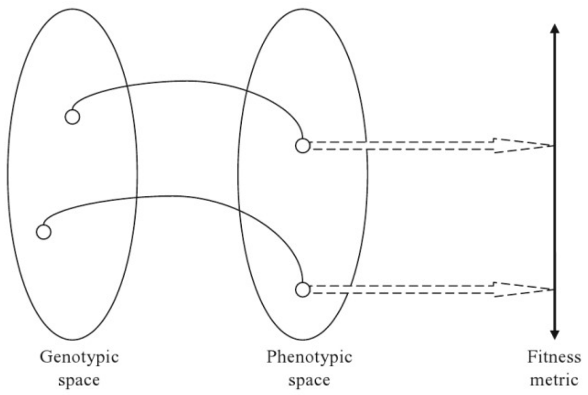


Fig. 3.2. Mapping from genotypic to phenotypic space with each phenotype in turn being mapped to a fitness measure

Table 3.5. Integer conversion for standard and Gray coding

Integer Value	**Canonical Binary Code**	**Gray Code**
0	000	000
1	001	001
2	010	011
3	011	010
4	100	110
5	101	111
6	110	101
7	111	100

For many problems, a real-valued genotype encoding is the most natural representation and most current optimisation applications of the GA use real-valued encodings (for example, $2.13, \ldots, -14.56$).

3.3.3 Representation Choice and the Generation of Diversity

The choice of representation is crucial as it determines the nature of the search space traversed by the GA. The choice also impacts on the appropriate design of diversity generation processes such as crossover and mutation.

Ideally, small (big) changes in the genotype should result in small (larger) changes in the phenotype and its associated fitness. This feature is known as *locality*. For example, if there is a good pairing of representation and diversity operators, minor mutations on the genotype will produce relatively small changes in the phenotype and its fitness, whereas a crossover between two very different parents will lead to a larger change in the phenotype. This means that the operators of mutation and crossover will perform distinctly different search processes.

Going back to the previous example of Gray coding, it can be observed from the integer to Gray mapping in Table 3.5, that a small change in the phenotype corresponds to a small change in the genotype. However, the reverse is not true. A single bit-flip in the genotype can lead to a large change in the phenotype (integer value). Hence, even a Gray encoding has poor locality properties and will not necessarily produce better results than the canonical binary coding system.

Raidl and Gottlieb [525] emphasise three key characteristics for the design of quality evolutionary algorithms (EAs):

i. locality,
ii. heritability, and
iii. heuristic bias.

Locality refers to the case where small steps in the search space result in small steps in the phenotypic space. Strong locality increases the efficiency of evolutionary search by making it easier to explore the neighbourhood of good solutions, whereas weak locality means that evolutionary search will behave more like random search.

Heritability refers to the capability of crossover operators to produce children that utilise the information contained in their parents in a meaningful way. In general, good heritability will ensure that each property of a child should be inherited from one of its parents, and that traits shared by both parents should be inherited by their child. Crossover operators with weak heritability are more akin to macro-mutation operators.

Heuristic bias occurs when certain phenotypes are more likely to be created by the EA than others when sampling genotypes without any selection pressure. In an unbiased case, each item in the phenotypic space has the same chance of occurring if a genotype is randomly generated.

Obviously, inducing heuristic bias can be good if it tends to lead to better solutions (in contrast, random search will tend to have lower heuristic bias), but inducing bias tends to reduce genotypic diversity, which can hinder the search for a global optimum. Heuristic bias arises from the choice of representation and the choice of variation operators.

3.4 Initialising the Population

If good starting points for the search process are known a priori, the efficiency of the GA can be improved by using this information to seed the initial population. More commonly, good starting points are not known and the initial population is created randomly. For binary-valued genotypes, a random number between 0 and 1 can be generated for each element of the genotype, with random numbers ≥ 0.5 resulting in the placing of a 1 in the corresponding locus of the genotype. If a real-valued representation is used, and boundary values for each locus of the genotype can be determined, each element of the genotype can be selected randomly from the bounded interval.

3.5 Measuring Fitness

The importance of the choice of fitness measure when designing a GA cannot be overstressed as this metric drives the evolutionary process. The first step in creating a suitable fitness measure is to identify an appropriate objective function for the problem of interest. This objective function often needs to be transformed into a suitable fitness measure via a transformation (for example, to ensure that the resulting fitness value is always nonnegative); hence:

$$F(x) = g(f(x)) \tag{3.2}$$

where f is the objective function, g transforms the value of the objective function into a nonnegative number, and F is the fitness measure. A simple example of a transformation is the linear rescaling of the raw objective function value:

$$g(x) = af(x) + b \tag{3.3}$$

where a is chosen in order to ensure that the maximum fitness value is a scaled multiple of the average fitness and b is chosen in order to ensure that the resulting fitness values are nonnegative. Using rescaled fitnesses rather than raw objective function values can also help control the selection pressure in the algorithm (Sect. 3.6.1).

In addition to absolute measures of fitness as just described, it is also possible to define fitness in relative rather than absolute terms, thus avoiding having to calculate explicit fitness values for each population member. For example, if our aim is to evolve a chess player we could evaluate the population by allowing individuals to play tournaments against each other where the winner of the tournament is deemed the fittest.

Estimating Fitness

Evaluating the fitness of individual members of the population is usually the most computationally expensive and time-consuming step in a GA. In some cases it is not practical to obtain an exact fitness value for every individual in each iteration of the algorithm.

A simple initial step is to avoid retesting the same individuals, so before testing the fitness of a newly created individual, a check could be made to determine if the same genotype has been tested in a prior generation. If it has been, the known fitness value can simply be assigned to the current individual.

More generally, in cases where fitness function evaluation is very expensive, we may wish to use less costly approximations of the fitness function in order to quickly locate good search regions.

One method of doing this is *problem approximation*, where we replace the original problem statement (fitness function) with a simpler one which approximates the problem of interest, the assumption being that a good solution to the simpler problem would be a good starting point in trying to solve the real problem of interest. An example of this would be the use of crash simulation systems where designs that perform well in computer simulations could then be subject to (expensive) real-world physical testing.

A second approach is to try to reduce the number of fitness function evaluations by estimating an individual's fitness based on the fitness of other 'similar' individuals. Examples of this include *fitness inheritance*, where the fitness of a child is inherited from its parent(s), or *fitness imitation*, where all the individual solutions in a cluster (those close together as defined by some distance metric) are given the same fitness (that of a representative solution of the cluster). Hence, an approximate fitness evaluation is used for much of the EA run, with the population, or a subset of the better solutions in the population, being exposed to the real (expensive) fitness function periodically during the run. This process entails a trade-off, with the gains from the reduction in the number of fitness evaluations being traded off against the risk that the search process will be biased through the use of fitness approximations.

A practical problem that can arise in applying GAs to real-world problems is that the fitness measures obtained can sometimes be noisy (for example, due to measurement errors). In this case, we may wish to resample fitness over a number of training runs, using an average fitness value in the selection and replacement process.

3.6 Generating Diversity

The process of generating new child solutions aims to exploit information from better solutions in the current population, while maintaining explorative capability in order to uncover even better regions of the search space. Too much exploitation of already-discovered good solutions runs the risk of convergence of the population of genotypes to a local optimum, while too much exploration drives the search process towards random search.

A key issue in designing a good GA is the management of the *exploration vs. exploitation* balance. The algorithm must utilise, or exploit, already-discovered fit solution encodings, while not neglecting to continue to explore new regions of the search space which may contain even better solution encodings. Choices for the selection strategy, the design of mutation and recombination operators, and the replacement strategy, determine the balance between exploration and exploitation. Selection and crossover tend to promote exploitation of already-discovered information, whereas mutation tends to promote exploration.

3.6.1 Selection Strategy

The design of the 'selection for mating' strategy determines the *selection pressure* (the degree of bias towards the selection of higher-fitness members of the population) of the algorithm. If the selection pressure is too low, information from good parents will only spread slowly through the population, leading to an inefficient search process. If the selection pressure is too high, the population is likely to get stuck in a local optimum, as a high selection pressure will tend to quickly reduce the degree of genotypic diversity in the population. Better-quality selection strategies therefore, encourage exploitation of high-fitness individuals in the population, without losing diversity in the population too quickly.

Although a wide variety of selection strategies have been designed for the GA, two common approaches are *fitness proportionate selection* and *ordinal selection*.

Fitness Proportionate Selection

The original method of selection for reproduction in the GA is fitness-proportionate selection (FPS) and under this method the probability that a specific member of the current population is selected for mating is directly related to its fitness relative to other members of the population. The selection process is therefore biased in favour of 'good' (i.e., fit) members of the current population. Given a list of each of the n individuals in the population and their associated fitnesses f_i, a simple way to implement FPS is to generate a random number $r \in \left[0, \sum_{j=1}^{n} f_j\right)$, then select the individual i such that:

$$\sum_{j=1}^{i-1} f_j \leq r < \sum_{j=1}^{i} f_j. \tag{3.4}$$

As a numerical example, suppose $n = 4$, $f_1 = f_2 = 15$, $f_3 = 10$ and $f_4 = 20$. Therefore, $\sum_{j=1}^{4} f_j = 60$. Assume a random draw from $[0, 60)$ produces 29.4. This value falls in the range $[15, 30)$ and hence results in the selection of individual 2. The FPS selection process can be thought of as spinning a roulette wheel, where the fitter individuals are allocated more space on the wheel (Fig. 3.5).

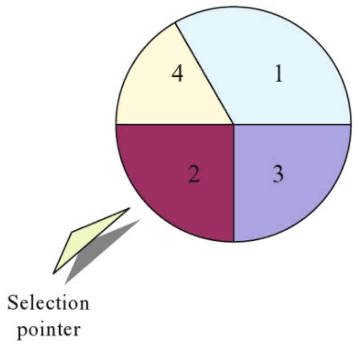


Fig. 3.5. Fitness-proportionate selection with the area on the roulette wheel corresponding to the fitness of each member of the population. Here individual 2 is selected

Although this method of selection is intuitive, it can produce poor results in practice as it embeds a high *selection pressure* in the early stage of the GA. Under FPS, the expected number of offspring for each encoding in the population is given by $\frac{P_{\text{obs}}}{P_{\text{avg}}}$, where P_{obs} is the observed performance (fitness) of the corresponding solution and P_{avg} is the average performance of all solutions in the current population.

As selection works on phenotypes (and their related fitness) it is 'representation independent'. This is not the case for the diversity generating operators of mutation and crossover.

3.6.2 Mutation and Crossover

The mutation operator plays a vital role in the GA as it ensures that the search process never stops. In each iteration of the algorithm, mutation can potentially uncover useful novelty. In contrast, crossover, if applied as a sole method of generating diversity, ceases to generate novelty once all members of the population converge to the same genotype.

The rate of mutation has important implications for the usefulness of selection and crossover. If a very high rate of mutation is applied, the selection and crossover operators can be overpowered and the GA will effectively resemble a random search process. Conversely, if a high selection pressure is used, a higher mutation rate will be required in order to prevent premature convergence of the population. In setting an appropriate rate of mutation, the aim is to select a rate which helps generate useful novelty but which does not rapidly destroy good solutions before they can be exploited through selection and crossover. In contrast to mutation, crossover allows for the inheritance of groups of 'good genes' or *building blocks* by the offspring of parents, thereby encouraging more intensive search around already discovered good solutions.

There is a close link between the choice of genotype representation and the design of effective mutation and crossover operators. Initially, mutation and crossover mechanisms for binary encodings are discussed, followed by the consideration of what modifications should be made to these for real-valued encodings.

Binary Genotypes

The original form of crossover for binary-valued genotypes was *single point crossover* (Fig. 3.7). A value p_{cross} is set at the start of the GA (say at 0.7) and for each pair of selected parents, a random number is generated from the uniform distribution $U(0, 1)$. If this value is < 0.7, crossover is applied to generate two new children; otherwise crossover is bypassed and the two children are clones of their parents. Crossover rates are typically selected from the range $p_{\text{cross}} \in (0.6, 0.9)$ but, if desired, the rate of crossover can be varied during the GA run.

One problem of single point crossover, is that related components of a solution encoding (schema) which are widely separated on the string tend to be disrupted when this form of crossover is applied. One way of reducing this problem is to implement *two point crossover* (Fig. 3.8), where the two cut positions on the parent strings are chosen randomly and the segments between the two positions are exchanged.

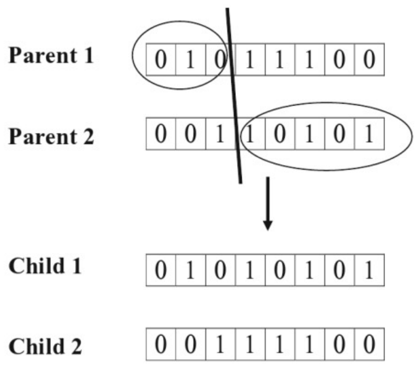


Fig. 3.7. Single point crossover where the cut-point is randomly selected after the third locus on the parent genotypes. The head and tail of the two parents are mixed to produce two child genotypes

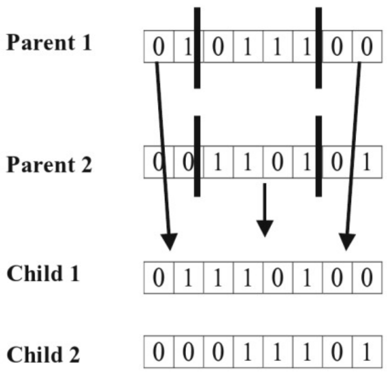


Fig. 3.8. Two-point crossover

Another popular form of crossover is *uniform crossover*. In uniform crossover, a random selection of gene value is made from each parent when filling each corresponding locus on the child's genotype. The process can be repeated a second time to create a second child, or the second child could be created using the values not selected when producing the first child (Fig. 3.9). To implement the latter approach, a random number r is drawn from the uniform distribution $U(0,1)$ for each locus. If $r < 0.5$, child 1 inherits from parent 1; else, it inherits from parent 2, with child 2 being comprised of the bit values not selected for child 1.

For binary genotypes, a mutation operation can be defined as a bit-flip, whereby a '0' can be mutated to a '1' or a '1' to a '0'. Figure 3.10 illustrates

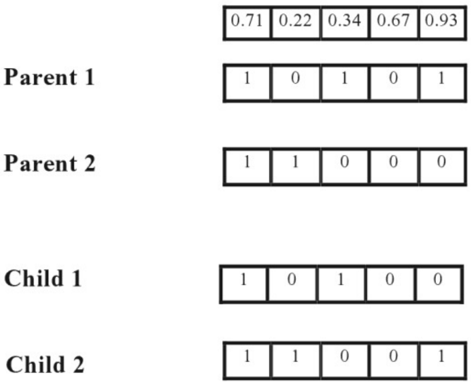


Fig. 3.9. Uniform crossover where a random choice is made as to which parent donates a bit to child 1. Child 2 is then constructed using the bits not selected for inclusion in child 1

an implementation of the mutation process, where $p_{\text{mut}} = 0.1$. Five random numbers (corresponding to a genotype length of five bits) are generated from $U(0, 1)$ and if any of these are < 0.1 then the value of that bit is 'flipped'. This mutation process is repeated for all child solutions generated by the crossover process.

Typical mutation rates for a binary-valued GA are commonly of the order $p_{\text{mut}} = \frac{1}{L}$ where L is the length of the binary string. Of course, there is no requirement that the mutation rate must remain constant during the GA run (Sect. 3.7).

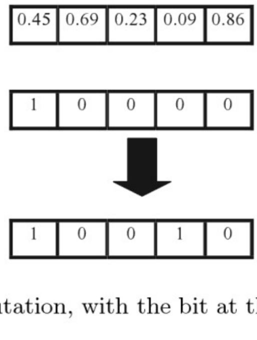


Fig. 3.10. Illustration of mutation, with the bit at the fourth locus being 'flipped'

Real-Valued Genotypes

The crossover operator can be modified for real-valued genotypes so that (for example) elements from the string of each parent are averaged in order to produce the corresponding value in their child(ren) (Fig. 3.11). Figure 3.12 illustrates this geometrically in two dimensions.

More generally, the real values in each locus of the child may be calculated as $P_1+\alpha(P_2-P_1)$, where P_1 and P_2 are the real values for that locus in each of the two parents, and α is a scaling factor randomly drawn from some interval (say $[-1.5, +1.5]$). This crossover operator defines a hypercube based on the location of the parents (Fig. 3.13).

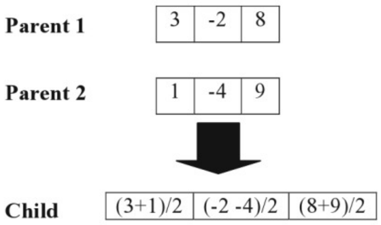


Fig. 3.11. Simple real-valued crossover with two parents producing a single child

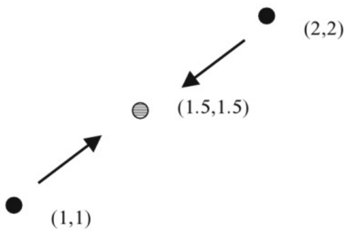


Fig. 3.12. Simple real-valued intermediate crossover with two parents (1,1) and (2,2) producing a single child at (1.5,1.5)

Many alternative mutation and crossover schemes for real-valued encodings exist. For example, a simple strategy for modifying mutation for real-valued encodings is to implement a stochastic mutation operator, where an element of a real-valued string can be mutated by adding a small (positive or negative) real value to it. Each element of the string x_i could be mutated by adding a random number drawn from the normal distribution $N(0, \alpha_i)$, where

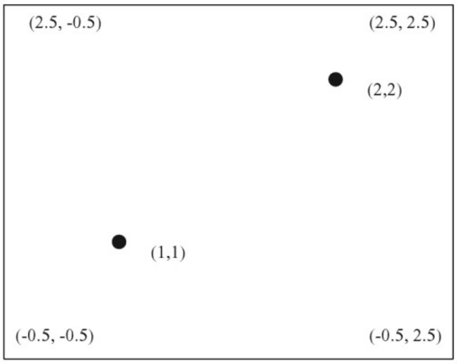


Fig. 3.13. Hypercube defined by crossover operator where parents are (1,1) and (2,2), with $\alpha \in [-1.5, 1.5]$

the standard deviation α_i is defined by the user. This mutation scheme will produce relatively small mutations most of the time, with occasional larger mutation steps.

3.6.3 Replacement Strategy

In deciding which parents and children survive into the next generation a wide variety of replacement strategies can be applied, including:

i. direct replacement (children replace their parents),
ii. random replacement (the new population is selected randomly from the existing population members and their children),
iii. replacement of the worst (all parents and children are ranked by fitness and the poorest are eliminated), and
iv. tournament replacement (the loser of the tournament is selected for replacement).

In the canonical GA, a generational replacement strategy is usually adopted. The number of children produced in each generation is the same as the current population size and during replacement the entire current population is replaced by the newly created population of child encodings.

The ratio of the number of children produced to the size of the current population is known as the *generation gap*. Hence, the generation gap is typically 1.0. It is also possible to create more offspring than members of the current population (generation gap > 1), and then select the best n (where n is the population size) of these offspring for survival into the next generation.

Many variants on the replacement process exist. As already seen, the number of children produced need not equal the current population size, and the automatic replacement of parents by children is not mandatory. A popular will be found in finite time and progress towards better solutions may be intermittent rather than gradual. Consequently, the time required to find a high-quality solution to a problem is not determinable ex ante. The GA, and indeed all evolutionary optimising methodologies, rely on feedback in the form of fitness evaluations. For some problems, measuring fitness can be difficult (perhaps fitness can only be assessed subjectively by a human) or expensive in terms of cost or computation time. In these cases, GA may not be the most suitable choice of optimising technique.

In this chapter, we have outlined the primary components and principles upon which the GA is based. The next chapter describes a number of extensions of the GA model.

4
Extending the Genetic Algorithm

The previous chapter provided an overview of the main concepts behind the GA. Since the introduction and popularisation of the GA, a substantial body of research has been undertaken in order to extend the canonical model and to increase the utility of the GA for hard, real-world problems. While it is beyond the scope of any single book to cover all of this work, in this chapter we introduce the reader to a selection of concepts drawn from this research. Many of the ideas introduced in this chapter have general application across the multiple families of natural computing algorithms and are not therefore limited to GAs. The chapter concludes with an introduction to Estimation of Distribution Algorithms (EDAs). EDAs are an alternative way of modelling the learning which is embedded in a population of genotypes in an evolutionary algorithm and have attracted notable research interest in the GA community in recent years.

4.1 Dynamic Environments

Many of the most challenging problems facing researchers and decision-makers are those with a dynamic nature. That is, the environment in which the solution exists, and consequently the optimal solution itself, changes over time. Examples of dynamic problems include trading in financial markets, time series analysis of gene expression data, and routing in telecommunication networks. Biological organisms inhabit dynamic environments and mechanisms have arisen to promote the 'survivability' of biological creatures in these environments. These mechanisms are useful sources of inspiration in helping us to design computer algorithms to attack real-world problems in dynamic environments.

A. Brabazon et al., *Natural Computing Algorithms*, Natural Computing Series,
DOI 10.1007/978-3-662-43631-8_4

4.1.1 Strategies for Dynamic Environments

In designing an evolutionary algorithm for application in a dynamic environment, the nature of the environmental changes will determine the appropriate strategy. For example, if change occurs at a slow pace, adapting the rate of mutation in a GA may be sufficient to allow the population to adjust to a slowly changing location for the global optimum. If the environment alters in a cyclic fashion, a memory of good past solutions may be useful. On the other hand, if the environment is subject to sudden discontinuous change then more aggressive adaptation strategies will be required. Hence, we can adopt a variety of strategies, including [302]:

- restart of the learning process,
- generation of more genotypic diversity if environmental change is detected,
- maintenance of genotypic diversity during the GA run,
- use of a memory mechanism to retain good past solutions (assumes cycling solutions), and
- use of multiple populations.

In an extreme case, it may be necessary to restart the learning process as past learning embedded in the population is no longer useful. More generally, if past learning still provides some guide to finding good solutions in the current environment, the focus switches to how best to adapt the current population in order to track the optimal solution as it changes. The following subsections discuss various aspects of diversity generation and maintenance. The use of multiple populations is discussed in Sect. 4.2.

4.1.2 Diversity

Maintaining diversity in the population of genotypes is important in all EC applications. Even in static environments, a population needs diversity in order to promote a good exploration of the search space. The role of diversity is even more important when faced with a dynamic environment. In the absence of any countermeasures, the canonical GA will tend to lose genotypic diversity during its run as selection and crossover will tend to push the population to a small set of genotypic states; hence the canonical algorithm needs some modification when it is applied in a dynamic environment.

Depending on the expected rate of environmental change, the modeller may decide to maintain a high degree of populational diversity at all times (useful if the environment has high rate of change), or generate it 'on demand' when a change in the environment is detected. At first glance it may appear that the better option is to maintain populational diversity at all times. However, maintaining diversity has a cost, either in terms of having a larger population, or in terms of less intensive exploitation of already discovered good regions. If the environment only changes occasionally, generation of diversity when environmental change is detected may be the better option.

Diversity Generation if Change Is Detected

Cobb's hypermutation strategy [115] was one of the earliest approaches to varying the rate of diversity generation as changes in the fitness landscape are detected. In this approach, if a change in the fitness landscape is detected, the base mutation rate of the GA is multiplied by a hypermutation factor. The size of the factor determines its effect; so if it is very large, it is equivalent to randomly reinitialising the entire population.

A common approach in the detection of environmental change is to use a *sentry strategy*. In a sentry strategy the fitness of a number of fixed genotypes (a form of memory) is monitored throughout the run. If the environment changes, the fitness of some or all of the locations of these sentries will alter and this provides feedback which is used to set the rate of mutation of the GA. If a large change in fitness occurs, indicating that the environment has changed notably, the rate of mutation is increased.

The sentry strategy can be applied in a number of ways. The sentries can remain outside the adapting population of solutions or they can be available for selection and crossover. In the latter case, while the sentries can influence the creation of new child solutions, they remain 'fixed' in location and are not mutated or replaced. Morrison [420] provides a discussion of quality strategies for sentry location, finding that random location often provides good results. In addition to providing information on whether the environment is changing, a sentry strategy can also provide information on where it is changing, thereby providing feedback on whether the changes are local or global.

Diversity Maintenance During Run

Rather than waiting for environmental change to occur and then playing 'catch-up', a strategy of maintaining continual diversity in the population can be followed. A wide variety of methods can be used for this purpose, including:

- weakening selection pressure,
- continual monitoring of populational diversity,
- restricted mating/replacement,
- fitness-sharing/crowding, and
- random immigrants.

Strong selection pressure implies that the GA will intensively sample current high fitness individuals, leading, if unchecked, to a rapid convergence of the population to similar genotypic forms. This can make it difficult for the population to adapt if environmental change occurs, particularly if the change occurs in a region of the landscape which is not currently being sampled by the population of solutions. Hence, the use of a lower selection pressure will help maintain diversity in the population of genotypes. Another related consideration when implementing a GA in a dynamic environment is what form of selection and replacement strategy to implement. The steady-state GA can offer advantages over the canonical generational GA. It allows a quicker response to a shift in the environment, as high-quality, newly created children are immediately available for mating purposes.

The degree of diversity of a population can be continually monitored in real time as the GA runs. Populational diversity can be defined on many levels, including diversity of fitness values and diversity of phenotypic or genotypic structures. Multiple measures of diversity can be defined for each of these. For example, diversity in a collection of real-valued fitnesses could be measured using the standard deviation of those values. However measured, if population diversity falls below a trigger level, action can be taken to increase diversity by raising the level of mutation or by replacing a portion of the population by newly created random individuals.
Under a *restricted mating* or restricted replacement strategy, individuals which are too similar are not allowed to mate, and in a restricted replacement strategy a newly created child is precluded from entering the population unless it is sufficiently different to existing members of the population. The object in both cases is to avoid convergence of the population to a small subset of genotypes.

A *fitness-sharing* mechanism [213] aims to reduce the chance that a multitude of similar individuals will be selected for reproduction, thereby reducing the genetic diversity of subsequent generations. An example of a fitness-sharing mechanism is:

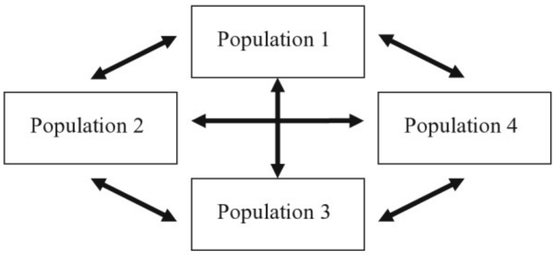
$$f'(i) = \frac{f(i)}{\sum_{j=1}^{n} s(d(i,j))} \tag{4.1}$$

where $f(i)$ represents the original raw fitness of individual i. If there are a number of individuals which are similar to i in the population, its fitness for use in the selection process is reduced. The shared (reduced) fitness of individual i is denoted as $f'(i)$, and this corresponds to i's original raw fitness, derated or reduced by an amount which is determined by a *sharing function.*

The sharing function s as in (4.2) provides a measure of the *density* of the population within a given neighbourhood of i. For any pair of individuals i, j in the population, the sharing function returns a value of '0' if i and j are more than a specified distance t apart (Fig. 4.1), and a value of '1' if they are identical.

$$s(d) = \begin{cases} 1 - \left(\frac{d}{t}\right)^{\alpha} & \text{if } d < t; \\ 0 & \text{otherwise.} \end{cases} \tag{4.2}$$

where d is a measure of the actual distance between two solutions and α is a scaling constant. To provide intuition on the sharing formula, if two individuals in the current population are virtually identical, the distance between them is close to zero. Consequently, the raw fitness of each individual is reduced by 50%, reducing each individual's chance of being selected for reproduction.



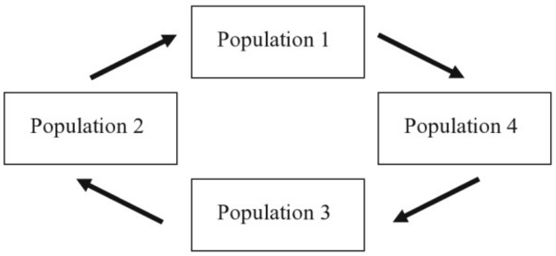


Fig. 4.2. Two examples of an island topology. The top network has unrestricted migration between all islands and the bottom network has a ring migration topology, where individuals can only migrate to one adjacent island

strategies can be used. For example, migration of individuals from one population to another may be unrestricted, or it may be confined to a predefined neighbourhood for each population (Fig. 4.2).

Illustrating one implementation of an island model, suppose there are four subpopulations with an unrestricted migration structure. A *migration pool* can be created for each subpopulation consisting of individuals selected from the other three subpopulations (perhaps their most fit individual). For each subpopulation in turn, a random selection is then made from its migration pool, with this individual replacing the worst individual in the subpopulation it enters.

Cellular EA

In cEA each individual genotype is considered as occupying a cell in a lattice (or graph) structure (Fig. 4.3). The operations of selection and recombination are constrained to take place in a small neighbourhood around each individual. When a cell is being updated, two parents are selected from its surrounding neighbourhood, and genetic operators are applied to the two parents to produce an offspring, with this offspring replacing the current genotype stored in that cell.

Each cell has its own pool of potential mates and in turn is a member of several other neighbourhoods, as the neighbourhoods of adjoining cells overlap. Therefore, in contrast to dEA, which typically has a few relatively large subpopulations, there are typically many small subpopulations in cEA.

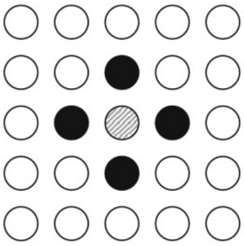


Fig. 4.3. A grid structure where a neighbourhood is defined around an individual (here the shaded cell)

In implementations of cEAs, updates of the state of each cell can be synchronous, where all cells are updated simultaneously using the cell contents in the current lattice. As the new genotypes are created, they are copied across to the next generation's lattice one at a time. Alternatively, the update process can be asynchronous, where the lattice is updated one cell at a time so that new genotypes can influence the update process as soon as they are created. One method of asynchronous update is to update all cells sequentially from left to right, and from line to line, starting from the top left corner (*fixed line sweep*). Another method of asynchronous update is to randomly select (with uniform probability and with replacement) which cell to update during each time step (*uniform choice*).

4.3 Constrained Optimisation

Many important problems consist of attempting to maximise or minimise an objective function subject to a series of constraints. The constraints serve to bound the feasible region of valid solutions, possibly to a very small subset of the entire (unbounded) search space (Fig. 4.4).

More formally, a constrained optimisation problem (assuming that the objective function is to be maximised) can be stated as follows: find the vector $x = (x_1, x_2, \ldots, x_d)^T, x \in \mathbb{R}^d$ in order to:

$$\text{Maximise } f(x) \tag{4.3}$$

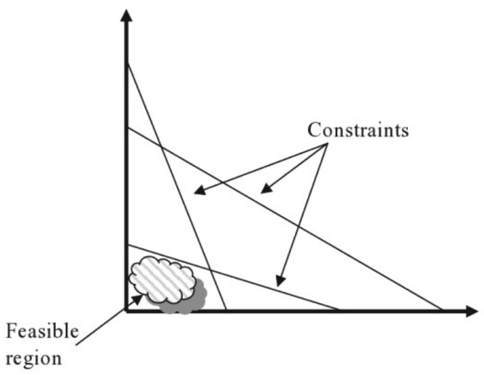


Fig. 4.4. Feasible region for a maximisation problem bounded by the x and y axes and three other constraints

subject to

$$\begin{aligned}
&\text{inequality constraints:} & g_i(x) \leq 0, &\quad i = 1, \ldots, m & (4.4)\\
&\text{equality constraints:} & h_i(x) = 0, &\quad i = 1, \ldots, r & (4.5)\\
&\text{boundary constraints:} & x_i^{\min} \leq x_i \leq x_i^{\max}, &\quad i = 1, \ldots, d. & (4.6)
\end{aligned}$$

The boundary constraints can be used to enforce physical conditions such as mass ≥ 0, etc. There are several approaches that can be taken when using a GA for constrained optimisation. A simple approach is to apply the GA as normal and assign zero fitness to any genotypes which generate an illegal solution which breaches one or more constraints. This strategy can be poetically referred to as the *death penalty* [409]. A problem with this approach is that even with low-dimensional problems it can produce a highly inefficient search process. If the problem is highly constrained, many generated genotypes may be illegal (for example, none of the initially randomly generated solutions might be feasible); hence much of the GA's effort is wasted. There is also a risk that there could be over-rapid convergence on the first feasible solution found. As the dimensionality of the problem increases, the above problems are worsened as ratio of invalid solutions outside the feasible area to valid solutions inside the feasible area will rapidly increase. Two key issues arise when applying the GA to a constrained optimisation problem:

i. it may be difficult to generate an initial population of feasible genotypes, and
ii. crossover and mutation may act to convert a legal solution into an illegal one.