Chapter 1

An Introduction to Evolutionary Design by Computers

By Peter Bentley

1.1 Introduction

Computers can only do what we tell them to do. They are our blind, unconscious digital slaves, bound to us by the unbreakable chains of our programs. These programs instruct computers what to do, when to do it, and how it should be done.

But what happens when we loosen these chains? What happens when we tell a computer to use a process that we do not fully understand, in order to achieve something we do not fully understand? What happens when we tell a computer to evolve designs?

As this book will show, what happens is that the computer gains almost human-like qualities of autonomy, innovative flair, and even creativity. These 'skills' which evolution so mysteriously endows upon our computers open up a whole new way of using computers in design. Today our former 'glorified typewriters' or 'overcomplicated drawing boards' can do everything from generating new ideas and concepts in design, to improving the performance of designs well beyond the abilities of even the most skilled human designer. Evolving designs on computers now enables us to employ computers in every stage of the design process. This is no longer computer aided design – this is becoming computer design.

The pages of this book testify to the ability of today's evolutionary computer techniques in design. Flick through them and you will see designs of satellite booms, load cells, flywheels, computer networks, artistic images, sculptures, virtual creatures, house and hospital architectural plans, bridges, cranes, analogue circuits and even coffee tables. Out of all of the designs in the world, the collection you see in this book have a unique history: they were all evolved by computer, not designed by humans.

1.1.1 Evolutionary Tools

This may sound a little alarming to the designers and artists amongst us, but it should not be. In fact, these are the people who should feel most excited and optimistic by these advances, for it is the designer and artist who are the main beneficiaries of this field of research. Evolutionary design systems are advanced software tools which are intended to be used by people, not to replace people. They are the latest in a number of computer software advances created to improve the productivity, quality, speed and reduce the expense of designing.

EVOLUTIONARY DESIGN BY COMPUTERS
ISBN 1-55860-605-X

Today, designers recognise the usefulness of computers for data management and drawing – most art and design departments use graphics software or computer aided design (CAD) packages to draw, manipulate and store their designs. These software tools are becoming more and more advanced, with many having the ability to render designs with photorealism, produce animations, or even generate stereoscopic virtual reality worlds. Analysis tools that can simulate and measure the performance of designs are also becoming more common, with much of engineering design relying on software analysis to test designs before prototypes are built.

Evolutionary design builds on these software tools by actually taking over part of the design process. It allows designers to improve the performance of their designs automatically, judged by analysis software. It allows a designer to explore numerous creative solutions to problems (overcoming 'design fixation' or limitations of conventional wisdom) by generating these alternative solutions for the designer. It can use knowledge from designers to generate new solutions, based on many separate ideas. It can even suggest entirely new design concepts, or new ways of using existing technology. Evolutionary design can and does achieve all of this with the blinding speed and low cost of the computer.

However, although the field of evolutionary design is showing some impressive results, the computers are not fully autonomous. People are required to work out what function the design should perform, and how a computer should be applied to the problem. As this book describes, there are many complex issues involved in getting a computer to evolve anything useful at all. And although the 'design skills' of the computer are surprisingly good, they are still no match for the human brain.

1.1.2 The Unconscious Power of Evolution

In reality, evolutionary design by computers does not involve conscious design at all. How could it, for today's computers are incapable of independent conscious thought, and evolution has no consciousness of its own. Evolutionary design is simply a process capable of generating designs, it can never truly be called a designer. This can be difficult to understand – surely an intricate design must be designed? The answer is no, an intricate design can arise through slow, gradual, mindless improvement. Evolutionary biology has taught us this harsh lesson – and there are no designs more complex than those evolved in nature.

Natural evolution is, of course, the original and best evolutionary design system. Designs have been evolving in nature for hundreds of millions of years. Biological designs that far exceed any human designs in terms of complexity, performance, and efficiency are prolific throughout the living world. From the near-perfection of the streamlined shape of a shark, to the extraordinary molecular structure of a virus, every living thing is a marvel of evolved design. Moreover, as biologists uncover more information about the workings of the creatures around us, it is becoming clear that many human designs have existed in nature long before they were thought of by any human, for example: pumps, valves, heat-exchange systems, optical lenses, sonar. Indeed, many of our recent designs borrow features directly from nature, such as the cross-sectional shape of aircraft wings from birds, and velcro from certain types of 'sticky' seeds. As Ray Paton observed: 'A very good example of how biology can inspire engineering solutions is the work of Professor O. H. Schmitt who introduced the term "biomimetic" (emulating biology) into the US literature over a decade ago. It is

fascinating to see how, following his Ph.D. thesis on the simulation of nerve action, four well-known electronic devices emerged: Schmitt trigger, emitter-follower, differential amplifier and heat pipe.' (Paton, 1994, p. 51).

1.1.3 Evolutionary Design by Computers

So it is clear that evolutionary design in nature is capable of generating astonishingly innovative designs. This book demonstrates how evolutionary design by computers is also capable of such innovation. To achieve this, the highest achievers in evolutionary design have come together for the first time to contribute chapters and provide a showcase of the best and most original work in this exciting new field. The book promotes the use of the word 'Design' in its broadest sense, allowing all aspects of evolutionary design to be explored, including: evolutionary optimisation, evolutionary art, evolutionary artificial life and creative evolutionary design. Of course the number of pages available for such a volume is finite, and so not every researcher in this field can be a contributor of a chapter. As the editor of this book I have tried my hardest to ensure a coherent and definitive selection of significant developments in evolutionary design is included, but there will always be omissions, and for that I apologise.

The contributors all have considerable technical expertise in this area, but beginners to this field should take heart, for the concept of evolution is a simple one, and the simpler forms of evolutionary design do not require years of study to achieve. Indeed, to help budding evolutionary designers get started, the CD-ROM included with this book contains code from many of the contributors of the chapters, including some demonstration evolutionary design systems. Perhaps one of the primary barriers to understanding is the terminology, which often seems to be an impenetrable tangle of words such as *meiosis*, *allele*, *epistasis* and *embryogeny*. Never fear: even the most experienced of us sometimes forget what the latest term to be stolen from biology means, so do not be afraid to consult the glossary included in the book!

And finally: before we open up evolutionary design by computers and explore its gory innards, a warning. This has been an area of computer science which has fascinated and thrilled me for some years. Like any researcher with a 'pet subject', I cannot pretend to hold unbiased views in this area. But I still find the excitement of my computer evolving an innovative design is undiminished, despite the hundreds I have already been privileged enough to see evolving before my eyes. I hope I can transfer some of my enthusiasm to you, my perceptive reader, so sit back and enjoy the ride!

1.1.4 What's to Come

This chapter gives an introduction to evolutionary design by computers. It is structured into three major sections: first, a general summary of evolutionary computation and the dominant evolutionary algorithms is given. Second, definitions and reviews of the significant aspects of evolutionary design are provided, to place the contents and structure of the rest of the book into context. Finally, some important technical issues in evolutionary design are explored.

However, before we explore these more detailed aspects of evolutionary design by computers, there is a question which must be tackled:

1.2 Why *Evolve* Designs?

This is an important and fundamental question, asked by many people. There are a number of reasons why we choose to use evolution, most which boil down to: 'because it seems to work rather well'. In more detail, there are perhaps four main reasons why the choice of evolutionary algorithms (EAs) is appropriate for design problems:

REASON 1: *Evolution is a good, general-purpose problem solver.*

Evolutionary algorithms are just one of many types of method known in computer science. It is currently not possible to define exactly which of these methods is best for which problem or even class of problems (Fogel, 1997), except in a very broad sense. However, it is possible to identify methods that consistently produce improved results (compared to results produced by other techniques) for a wide range of different problems. Indeed, as will be explained later, the evolutionary algorithms fall into this category, having been demonstrated successfully with hundreds of different types of problem. Table 1.1 lists some of these types of application. (It should be noted that there are literally hundreds of researchers working in each of the areas listed, all developing their own evolutionary systems.)

Researchers and software developers apply computers to a wide variety of design problems. Rather than spending time, effort and money developing new specialised computational techniques for every new problem, most developers prefer to use an algorithm proven through extensive trials to be reusable and robust – such as an evolutionary algorithm.

REASON 2: *Uniquely, evolutionary algorithms have been used successfully in every type of evolutionary design.*

Although there are contenders to the throne of computational design, evolutionary algorithms are, without doubt, the leading techniques at present. Hill-climbing, simulated annealing, Tabu search and other techniques have all been applied successfully in certain areas, but only evolutionary algorithms such as the genetic algorithm have been used successfully in all types of automated design system. Indeed, the popularity of genetic algorithms in engineering design

Table 1.1 Examples of types of applications tackled successfully by evolutionary computation.

Control systems	(Husbands et al., 1996).
Data mining	(Radcliffe and Surrey, 1994b).
Fault-tolerant systems	(Thompson, 1995).
Game playing	(Axelrod, 1987).
Machine learning	(Goldberg, 1989).
Ordering problems	(Schaffer and Eshelman, 1995).
Scheduling	(Yamada and Nakano, 1995).
Set covering and partitioning	(Levine, 1994).
Signal timing	(Foy et al., 1992).
Strategy acquisition	(Greffenstette, 1991).

has led to workshops, conferences, and books devoted entirely to this subject (Fleming et al., 1995; Gen and Cheng, 1997; Bentley, 1998b).

REASON 3: *Evolution and the human design process share many similar characteristics.*

Some researchers claim that natural evolution and the human design process are directly comparable (Fogel et al., 1966; Goldberg 1991; French 1994). It is clear that our designs have evolved, as flint hand-axes became arrowheads, as the first primitive computers have become the powerful supercomputers of today. The 'arms race' which is known to dramatically increase the complexity of our designs is thought by biologists to be responsible for the development of the complexity in living creatures (Dawkins, 1982). Comparative studies of our own designs also reveals the development of 'species' of designs which fit within clearly defined 'niches' (French and Ramirez, 1996).

Indeed, Goldberg actually attempts to formally define human design in terms of evolution by the genetic algorithm (Goldberg, 1991). He compares the recombination of genetic material from parent solutions when forming a new child solution, with a human designer combining ideas from two solutions to form a new solution. (These ideas and others are explored further in the first section of the book.)

REASON 4: *The most successful and remarkable designs known to mankind were created by natural evolution, the inspiration for evolutionary algorithms.*

Natural evolution has been creating designs successfully for an unimaginable number of years. Even a cursory study of the myriad of extraordinary designs in nature should be sufficient to inspire awe in the power of evolution. Indeed, conceivably the most complex and remarkable miracle of design ever created – the human brain – was generated by evolution in nature. Not only is it an astonishing design in its finished form, but equally astonishingly, its huge complexity grew from a single cell using instructions contained in one molecule of DNA. This is perhaps the most conclusive demonstration of all that the evolution-based techniques of evolutionary computation are highly suitable for design problems.

1.3 Evolutionary Computation

Evolutionary computation is all about *search*. In computer science, search algorithms define a computational problem in terms of search, where the *search-space* is a space filled with all possible solutions to the problem, and a point in that space defines a solution (Kanal and Cumar, 1988). The problem of improving parameter values for an application is then transformed into the problem of searching for better solutions elsewhere in the solution space, see fig 1.1. There are many types of search algorithm in existence, of which evolutionary search is a recent and rapidly growing subset.

Evolutionary search algorithms are inspired by and based upon evolution in nature. These algorithms typically use an analogy with natural evolution to perform search by *evolving* solutions to problems.[1] Hence, instead of working with one solution at a time in the search-space, these algorithms consider a large collection or *population* of solutions at once.

Although evolutionary algorithms (EAs) do make computers evolve solutions, this evolution is not explicitly specified in an EA, it is an *emergent property* of the algorithm. In fact, the

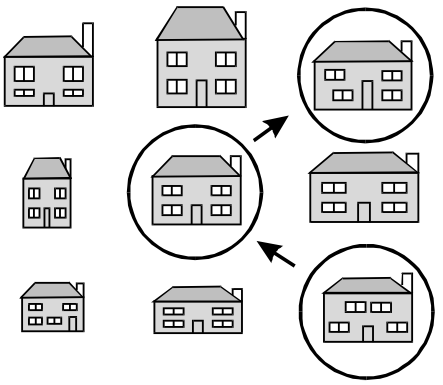


Figure 1.1 Searching for a solution in an example search space of house designs.

computers are not instructed to evolve anything, and it is currently not possible for us to explicitly 'program-in' evolution – for we do not fully understand how evolution works. Instead, the computers are instructed to maintain populations of solutions, allow better solutions to 'have children', and allow worse solutions to 'die'. The 'child solutions' inherit their parents' characteristics with some small random variation, and then the better of these solutions are allowed to 'have children' themselves, while the worse ones 'die', and so on. This simple procedure causes evolution to occur, and after a number of generations the computer will have evolved solutions which are substantially better compared to their long-dead ancestors at the start, see fig. 1.2.

By considering the search space, it is possible to get an idea of how evolution finds good solutions. Figure 1.3 shows the search space for the example shown in fig. 1.2. It should be clear that evolution searches the space in parallel (in the example, it considers four house

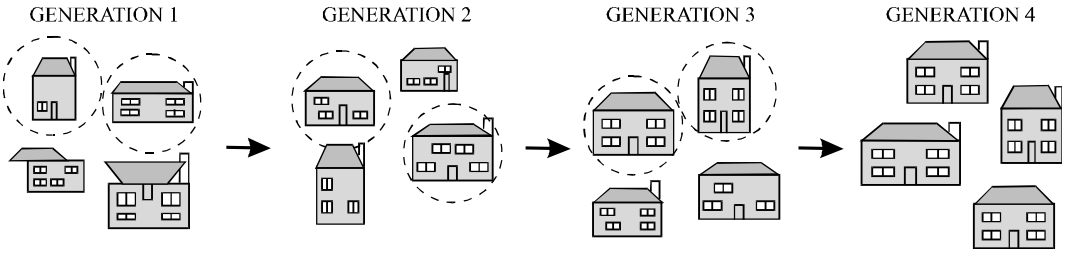


Figure 1.2 Four generations of evolving house designs using a population size of four. Parents of the next generation are circled.

[1] It must be stressed that evolution is not simulated in these algorithms, *it actually happens*. While EAs may simulate natural evolution, to call this process simply 'simulated evolution' is incorrect – an EA no more simulates evolution than a pocket calculator simulates addition, or a typewriter simulates text. (Indeed, it could be argued that compared to our pocket calculators, we are the ones who simulate addition, for we often rely on memory to provide us with answers, but the calculator must always calculate the sum.) Evolutionary search generates evolution in a different medium compared to evolution in nature, but both are equally valid forms of evolution.

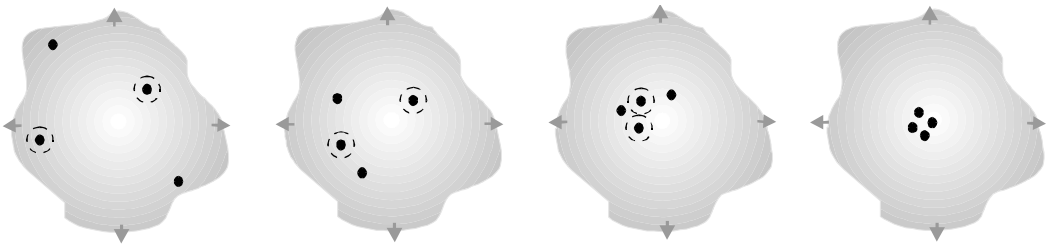


Figure 1.3 The location of the evolving houses in the space of house designs, each generation. Better solutions are found in the centre of this example space.

designs at a time). It should also be clear that evolution quickly 'homes in' on the best area of the search space, resulting in some good designs after only four generations.

All EAs require guidance to direct evolution towards better areas of the search space. They receive this guidance by *evaluating* every solution in the population, to determine its *fitness*. The fitness of a solution is a score based on how well the solution fulfils the problem objective, calculated by a *fitness function*. Typically, fitness values are positive real numbers, where a fitness of zero is a perfect score. EAs are then used to minimise the fitness scores of solutions, by allowing the fitter solutions to have more children than less fit solutions. In the 'house' example, the problem objective might be to find a house design which has four evenly placed windows, a door in the centre, a chimney, and so on. The fitness function would take a solution as input and return a fitness value based on how well the solution satisfies these objectives, e.g. when evaluating the number of windows, the fitness score could simply be incremented by:

$$| 4 - \textit{no. of windows in solution} |.$$

Fitness values are often plotted in search spaces, giving mountainous *fitness landscapes*, where a high peak corresponds to solutions in that part of the search space which have optimal fitnesses (i.e., low fitness scores). If the problem has many separate optima (i.e., if the fitness function is *multimodal*), finding a globally optimal solution (the top of the highest mountain) in the landscape can be difficult, even for an EA.

There are four main types of evolutionary algorithm in use today, three of which were independently developed more than thirty years ago. These algorithms are: the **genetic algorithm** (GA) created by John Holland (1973, 1975) and made famous by David Goldberg (1989), **evolutionary programming** (EP) created by Lawrence Fogel (1963) and developed further by his son David Fogel (1992), and **evolution strategies** (ES) created by Ingo Rechenberg (1973) and today strongly promoted by Thomas Bäck (1996). The fourth major evolutionary algorithm is a more recent and very popular development of John Koza (1992), known as **genetic programming** (GP). The field of evolutionary computation has grown up around these techniques, with its roots still firmly in evolutionary biology and computer science, see fig. 1.4. Today researchers examine every conceivable aspect of EAs, often using knowledge of evolution from evolutionary biology in their algorithms, and more recently, using EAs to help biologists learn about evolution (Dawkins, 1986).

Evolution-based algorithms have been found to be some of the most flexible, efficient and robust of all search algorithms known to computer science (Goldberg, 1989). Because of these

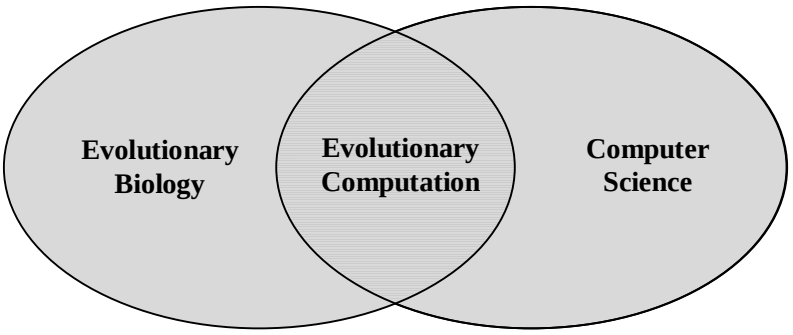


Figure 1.4 Evolutionary computation has its roots in computer science and evolutionary biology.

properties, these methods are now becoming widely used to solve a broad range of different problems (Holland, 1992).

The following sections briefly summarise the four dominant types of EA, and then a general architecture for EAs is introduced, to show how these separate techniques follow a common evolutionary paradigm.

1.3.1 Genetic Algorithms

A Summary

The genetic algorithm is perhaps the most well known of all evolution-based search algorithms. GAs were developed by John Holland in an attempt to explain the adaptive processes of natural systems and to design artificial systems based upon these natural systems (Holland, 1973, 1975). (Precursors of GAs were developed by Alex Fraser in 1957 and Hans Bremermann in 1962.[2]) Whilst not being the first algorithm to use principles of natural selection and genetics within the search process, the genetic algorithm is today the most widely used. More experimental and theoretical analyses have been made on the workings of the GA than any other EA. Moreover, the genetic algorithm (and enhanced versions of it) resembles natural evolution more closely than most other methods.

Having become widely used for a broad range of optimisation problems in the last fifteen years (Holland, 1992), the GA has been described as being a 'search algorithm with some of the innovative flair of human search' (Goldberg, 1989). GAs are also very forgiving algorithms – even if they are badly implemented, or poorly applied, they will often still produce acceptable results (Davis, 1991). GAs are today renowned for their ability to tackle a huge variety of optimisation problems and for their consistent ability to provide excellent results, i.e. they are *robust* (Holland, 1975; Goldberg 1989; Davis 1991; Fogel 1994).

Genetic algorithms use two separate spaces: the search space and the *solution space*. The search space is now a space of *coded* solutions to the problem, and the solution space is the space of actual solutions. Coded solutions, or *genotypes* must be mapped onto actual solutions, or *phenotypes*, before the quality or *fitness* of each solution can be evaluated, see fig. 1.5.

[2] This information was kindly provided by David Fogel, private communication.

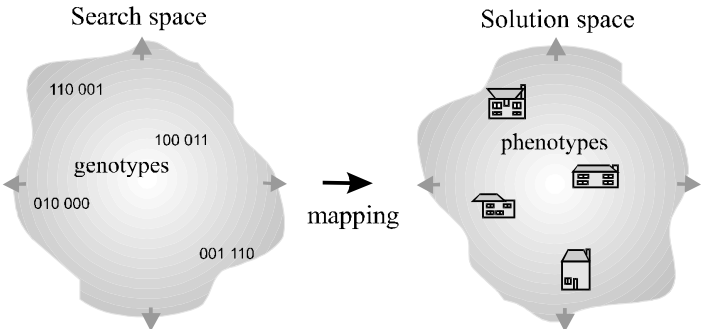


Figure 1.5 Mapping genotypes in the search space to phenotypes in the solution space.

GAs maintain a population of *individuals* where each individual consists of a genotype and a corresponding phenotype. Phenotypes usually consist of collections of parameters (in our 'house' example, such parameters might define the number and position of windows, the position of the roof, the width and height of the house, and so on). Genotypes consist of coded versions of these parameters. A coded parameter is normally referred to as a *gene*, with the values a gene can take being known as *alleles*. A collection of genes in one genotype is often held internally as a string, and is known as a *chromosome*.

The simplest form of GA, the *canonical* or *simple* GA, is summarised in fig. 1.6. This algorithm works as follows: The genotype of every individual in the population is initialised with random alleles. The main loop of the algorithm then begins, with the corresponding phenotype of every individual in the population being evaluated and given a fitness value according to how well it fulfils the problem objective or fitness function. These scores are then used to determine how many copies of each individual are placed into a temporary area often termed the 'mating pool' (i.e. the higher the fitness, the more copies that are made of an individual).

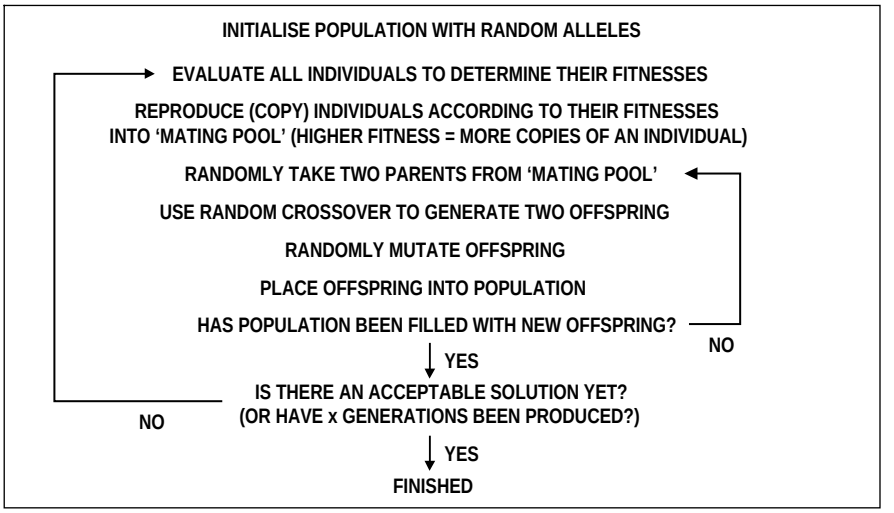


Figure. 1.6 The simple genetic algorithm.

Two parents are then randomly picked from this area. Offspring are generated by the use of the crossover operator, which randomly allocates genes from each parent's genotype to each offspring's genotype. For example, given two parents: 'ABCDEF' and 'abcdef', and a random crossover point of, say, 2, the two offspring generated by the simple GA would be: 'ABcdef' and 'abCDEF', see fig. 1.7. (Crossover is used about 70% of the time to generate offspring, for the remaining 30% offspring are simply clones of their parents.) Mutation is then occasionally applied (with a low probability) to offspring. When it is used to mutate an individual, typically a single allele is changed randomly. For example, an individual '111111' might be mutated into '110111', see fig. 1.8.

Using crossover and mutation, offspring are generated until they fill the population (all parents are discarded). This entire process of evaluation and reproduction then continues until either a satisfactory solution emerges or the GA has run for a specified number of generations (Holland, 1975; Goldberg, 1989; Davis, 1991).

The randomness of the genetic operators can give the illusion that the GA and other EAs are nothing more than parallel random search algorithms, but this is not so. Evolutionary search has a random element to its exploration of the search space, but the search is unquestionably *directed* by selection towards areas in the search space that contain better solutions. Unless the genetic operators are very badly designed, an EA will always 'home-in' on these areas, and because the search is performed in parallel, these algorithms are rarely fooled by local optima, unlike many other search algorithms (Goldberg, 1989).

However, the simple GA is just that – very simple and a little naïve. This GA is favoured by those that try to theoretically analyse and predict the behaviour of genetic algorithms, but in reality, typical GAs are usually more advanced. Common features include: more realistic

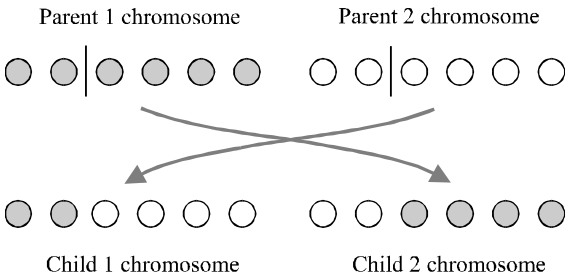


Figure 1.7 The behaviour of the crossover operator. The vertical line shows the position of the random crossover point.

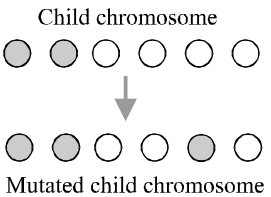


Figure 1.8 The behaviour of the mutation operator.

natural selection, more genetic operators, ability to detect when evolution ceases, and overlapping populations or elitism (where some fit individuals can survive for more than one generation) (Davis, 1991). Because of this improved analogy with nature, the term *reproduction* is normally used as it is in biology to refer to the entire process of generating new offspring, encompassing the crossover and mutation operators. (This is in contrast to the somewhat confusing use of the word 'reproduction' to mean an explicit copying stage within the simple GA.)

GA Theory

Whilst there is no formal proof that the GA will always converge to an acceptable solution to any given problem, a variety of theories exist (Holland, 1975; Kargupta, 1993; Harris, 1994), the most accepted of these being Holland's Schema Theorem (Holland, 1975) and the Building Block Hypothesis (Goldberg, 1989).

Briefly, a *schema* is a similarity template describing a set of strings (or chromosomes) which match each other at certain positions. For example, the schema *10101 matches the two strings {110101, 010101} (using a binary alphabet and a metasymbol or *don't care* symbol *). The schema *101* describes four strings {01010, 11010, 01011, 11011}. As Goldberg (1989) elucidates, in general, for alphabets of cardinality (number of alphabet characters) k, and string lengths of l characters, there are $(k + 1)^l$ schemata.

The *order* of a schema is the number of fixed characters in the template, e.g. the order of schema *1*110 is 4, and the order of schema *****0 is 1. The *defining length* of a schema is the distance between the first and last fixed character in the template, e.g. the defining length of 1****0 is 5, the defining length of 1*1*0* is 4, and the defining length of 0***** is 0.

Holland's Schema Theorem states that the action of reproduction (copying, crossover and mutation) within a genetic algorithm ensures that schemata of short defining length, low order and high fitness increase within a population (Holland, 1975). Such schemata are known as building blocks.

The building block hypothesis suggests that genetic algorithms are able to evolve good solutions by combining these fit, low order schemata with short defining lengths to form better strings (Goldberg, 1989). However, this still remains an unproven (though widely accepted) hypothesis.

GA Analyses

Experimental results show that for most GAs (initialised with random values), evolution makes extremely rapid progress at first, as the diverse elements in the initial population are combined and tested. Over time, the population begins to converge, with the separate individuals resembling each other more and more (Davis, 1991). Effectively this results in the GA narrowing its search in the solution-space and reducing the size of any changes made by evolution until eventually the population converges to a single solution (Goldberg, 1989). When plotting the best fitness value in each new population against the number of generations, a typical curve emerges, fig 1.9 (Parmee and Denham, 1994).

Theoretical research to investigate the behaviour of the various varieties of GAs for different problems is growing rapidly, with careful analyses of the transmission of schemata being made (De Jong, 1975; Kargupta, 1993). The use of Walsh function analysis (Deb et al., 1993)

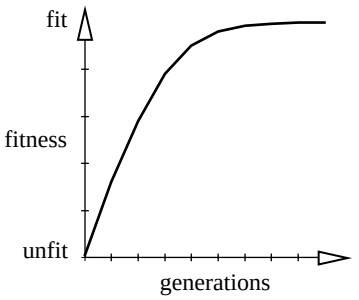


Figure 1.9 Typical curve of evolving fitness values over time.

and Markov chain analysis (Horn, 1993) has led to the identification of some 'deceptive' and 'hard' problems for GAs (Deb and Goldberg, 1993). Chapter 4: *The Race, the Hurdle, and the Sweet Spot* by David Goldberg summarises some of the significant advances in the understanding of GAs made to date.

Advanced Genetic Algorithms

When applying GAs to highly complex applications, some problems do occasionally arise. The most common is *premature convergence* where the population converges early onto non-optimal local minima (Davis, 1991). Problems are also caused by deceptive functions, which are, by definition, 'hard' for most GAs to solve. In addition, noisy functions (Goldberg et al., 1992) and the optimisation of multiple criteria within GAs can cause difficulties (Fonseca and Fleming, 1995a). In an attempt to overcome such problems, new, more advanced types of GA are being developed (Goldberg, 1994). These include:

- **Steady-state GAs**, where offspring are generated one at a time, and replace existing individuals in the population according to fitness or similarity. Convergence is slower, but very fit solutions are not lost (Syswerda, 1989).
- **Parallel GAs**, where multiple processors are used in parallel to run the GA (Adeli and Cheng, 1994; Levine, 1994).
- **Distributed GAs**, where multiple populations are separately evolved with few interactions between them (Whitley and Starkweather, 1990)
- **GAs with niching and speciation**, where the population within the GA is segregated into separate 'species' (Horn, 1993; Horn and Nafpliotis, 1993; Horn et al., 1994).
- **Messy GAs (mGA)**, which use a number of 'exotic' techniques such as variable-length chromosomes and a two-stage evolution process (Deb, 1991; Deb and Goldberg, 1991).
- **Multiobjective GAs (MOGAs)**, which allow multiple objectives to be optimised with GAs (Schaffer, 1985; Srinivas and Deb, 1995; Bentley and Wakefield, 1997c).
- **Hybrid GAs (hGAs)**, e.g. memetic algorithms, where GAs are combined with local search algorithms (George, 1994; Radcliffe and Surrey, 1994a).
- **Structured GAs (sGAs)**, which allow parts of chromosomes to be switched on and off using evolveable 'control genes' (Dasgupta and McGregor, 1992; Parmee and Denham, 1994).

- **GAs with diploidy and dominance**, which can improve variation and diversity in addition to performance (Smith and Goldberg, 1992).
- **Mutation-driven GAs**, such as Harvey's SAGA (Harvey and Thompson, 1997), which uses converged populations modified primarily by mutation to allow the constant 'incremental evolution' of new solutions to varying fitness functions.
- **GAs with 'genetic engineering'**, which identify beneficial genetic material during evolution and prevent its disruption by the genetic operators (Gero and Kazakov, 1996).
- **Injection Island GAs (IIGAs)**, which evolve a number of separate populations ('islands') with representations and fitness functions of different accuracy, and occasionally 'inject' good solutions from one island into another (Eby et al., 1997).

Most of these advanced types of GA are described further in the chapters of this book.

Recommended Books for GAs:

Adaptation in Natural and Artificial Systems.
by John Holland (1975).

Genetic Algorithms in Search, Optimization & Machine Learning.
by David Goldberg (1989).

The Handbook of Genetic Algorithms.
edited by Lawrence Davis (1991).

Genetic Algorithms + Data Structures = Evolution Programs.
by Zbigniew Michalewicz (1996).

Practical Handbook of Genetic Algorithms.
edited by Lance Chambers (1995).

An Introduction to Genetic Algorithms.
by Melanie Mitchell (1996).

Evolutionary Computation: Toward a New Philosophy of Machine Intelligence.
by David B. Fogel (1995).

The Design of Innovation: Lessons from Genetic Algorithms
by David Goldberg (1998).

1.3.2 Genetic Programming

A Summary

Genetic programming is not, strictly speaking, a separate evolutionary algorithm in its own right. It is a specialised form of genetic algorithm, which manipulates a very specific type of solution using modified genetic operators.

GP was developed by Koza (1992) in an attempt to make computers program themselves (i.e., perform *automatic programming*) by evolving computer programs. Perhaps because of this application, or perhaps because of the higher conceptual level at which the algorithm

operates, GP has become immensely popular amongst computer scientists, and it seems likely that the number of publications on this new evolutionary technique will soon approach the wealth of publications in its parent field, genetic algorithms. Practitioners of GP are beginning to move away from the original application of evolving computer programs, with GP now being applied in alternative areas, evolutionary design amongst them. John Koza describes one such application in his chapter *The Design of Analog Circuits by Means of Genetic Programming*, in the last section of the book.

GP follows essentially the same procedure as described previously for GAs. Populations of individuals are maintained. These individuals are initialised randomly, evaluated and parents are selected for reproduction based on fitness. Offspring are generated using crossover and mutation operators, and these offspring replace some or all of their parents in the population. The individuals are then evaluated, parents are selected for reproduction, and so on.

However, unlike GAs, GP does not make a distinction between the search space and the solution space. In GP, genotypes are the same as phenotypes, i.e. GP does not manipulate coded versions of the solutions, it manipulates the solutions themselves. This means that for GP, the search space and solution space are identical. In addition, unlike GAs (which use almost any conceivable representation), GP represents solutions in a very specific, hierarchical manner. Figure 1.10 shows an example solution for GP.

A strong motivation in the use of such hierarchical representations was the problem of applying crossover to variable-length chromosomes. Computer programs are obviously of variable sizes – they can be anything from a few characters to thousands of lines long. The standard crossover operator for GAs simply cannot cope with performing recombination with two chromosomes that are of different lengths. To illustrate this, consider two chromosomes: 'A+B/C' and '~A/B+C'. Using the simple crossover of GAs, if the random crossover point happened to be 1, the two child chromosomes would be: '~+B/C' and 'AA/B+C'. These are clearly meaningless and invalid expressions. The solution suggested by Koza for GP is to arrange its solutions in hierarchical tree-structures, and then crossover can be used to interchange randomly chosen branches of the parents' trees without the syntax of the programs being disrupted, as shown by fig. 1.11. (In fact, this is just one solution to the problem of variable-length crossover. There are many types of GA which use a number of alternatives (Bentley and Wakefield, 1996c).)

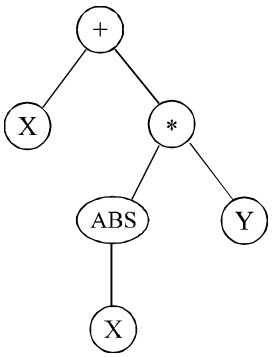


Figure 1.10 A simple computer program defined by GP's hierarchical representation.

GP also has a modified mutation operator. This operator picks a random point in a tree and deletes everything below it, replacing it with a randomly generated subtree, see fig. 1.12. However, because the crossover operator plays a similar role to mutation in GP, mutation is often considered unnecessary (Koza,1992).

The other major distinction between GAs and GP is in the evaluation of solutions. As described previously, GAs require the genotypes of individuals to be mapped onto the phenotypes before evaluation. Phenotypes are then analysed by fitness functions. In standard GP, there is no mapping process – because GP evolves phenotypes directly – and the evaluation process is very different. To calculate the fitness of solutions, these evolved programs must be *run* to find out what they do. Normally a series of input values and desired output values are provided, and the fitness of the program is based on how closely actual output values match the desired output values, for each set of input values. GP terminates evolution when a solution has been evolved which has a satisfactory fitness value (or after a predefined number of generations).

GP normally evolves symbolic expressions (S-expressions) in languages such as LISP – a computer programming language which uses combinations of functions written as lists. For

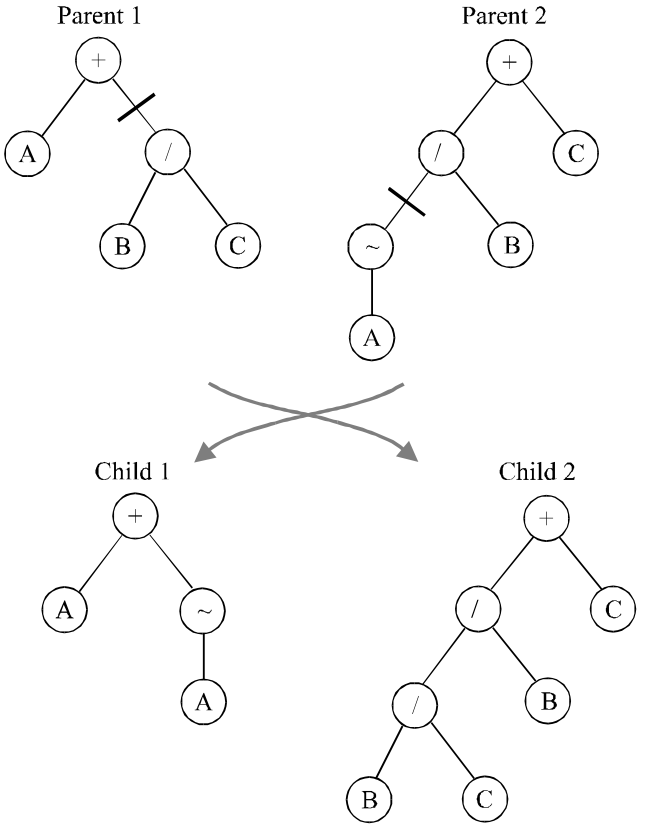


Figure 1.11 The behaviour of the crossover operator in GP. The thick lines show the positions of the random crossover points.

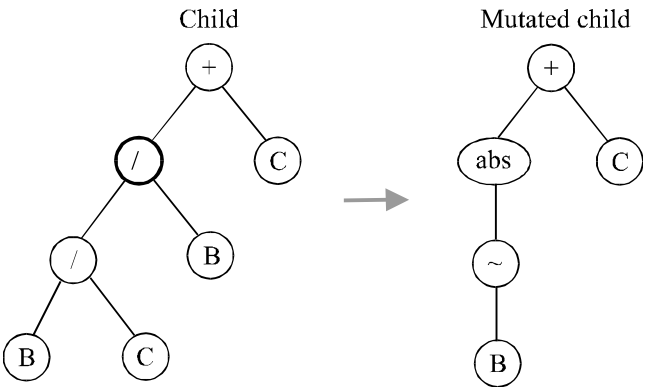


Figure 1.12 The behaviour of the GP mutation operator.
The random mutation point is shown in bold.

example, the two parent solutions used to illustrate the crossover operator in fig. 1.11 would be written in LISP as:

(+ A (/ B C)) and
(+ (/ (~ A) B) C).

LISP allows the usual high-level programming operators, including conditional operators, to be applied in the same way. For example, the following LISP S-expression tells the computer to add 1, 2 and 3 (if *time* is greater than 10) or 4 (if *time* is less than 10):

(+ 1 2 (IF (> time 10) 3 4))

Because solutions can contain such conditional statements, it is possible for evolving solutions to contain redundant code which is never executed. For example, the following S-expression will always return the result of A+B. The sub expression (− A (/ B C)) will never be executed by the computer:

(IF (5 > 0) (+ A B) (− A (/ B C)))

Solutions with redundant code in them are said to contain *junk* or *introns*. Such solutions are common in GP, with most solutions steadily increasing in size as evolution progresses. This tendency is known as *bloat* (Langdon and Poli, 1997). The effects of bloat can be reduced by penalising the fitness of any solutions that become oversized.

Conditional statements in GP allow a simple form of implicit *dominance* to occur in evolving S-expressions. Normally implementations of dominant and recessive alleles in EAs require diploid chromosomes (pairs of chromosomes) where the value of a gene is the combined meaning of both alleles from the twin chromosome. Certain alleles are defined to be dominant and others recessive, ensuring that the phenotypic effect of a gene is caused by dominant alleles in preference to recessive ones. GP allows a simpler version of this with conditional

statements. For example, if in the following S-expression, *X* can only take the values 0, 1, or 2, then the result A could be regarded as being dominant to the result *B*:

(IF (> X 0) A B)

Conditional statements in GP also resemble the operons and regulons found in our own DNA, used to switch on and off other genes during the development of the organism (Paton, 1994).

GP Theory

Because there are significant differences between the representation and operators of GP and GAs, it is not clear whether the Schema Theorem and Building Block Hypothesis described previously can be applied to GP. Koza (1992) attempts to achieve this by defining a schema to be the set of all individual trees from the population that contain, as subtrees, one or more specified subtrees. So for GP, disruption is smallest and the deviation from the optimum number of trials is smallest when the schema is defined in terms of a single compact subtree (Koza, 1992). As long as crossover is not too disruptive, the fittest of such compact subtrees should exponentially grow within the population, and be used as building blocks for constructing fit new individuals.

Unfortunately, it seems that crossover in GP is often too disruptive for such theories to be applicable. Definitions of schema and the Schema Theorem are still the subject of much research in the GP community (O'Reilly and Oppacher, 1995; Poli and Langdon, 1997b).

Advanced GP

Just as GAs have many other advanced genetic operators, GP has a number of specialised operators. These include: *permutation*, which swaps two characters in a tree; *editing*, which allows the optimisation and reduction of long S-expressions; and *encapsulation* which allows a subtree to be converted into a single node, preserving it from disruption by crossover or mutation.

One commonly used technique, which is a more advanced version of encapsulation, is the evolution of *automatically defined functions* (ADFs). Typically, the GP system is set up to evolve a predetermined number of functions in addition to the main program. Each function can then be called in the program, allowing the multiple use of code without the need to re-evolve it each time, and also minimising the danger of disruption by crossover or mutation. The use of ADFs has been shown to enhance the performance of GP (Koza, 1992).

Figure 1.13 shows an example solution with two ADFs. The first ADF (ADF0) takes a single argument and returns its cube, the second (ADF1) takes two arguments and returns 1/(ARG0*ARG1). The main program (at the right of the tree) adds the result of calling both ADFs with parameters A and B. When expanded into a single LISP S-expression, the solution becomes:

(+ (* (* A A) A) (/ (1 (* A B))))

Should it be necessary, hierarchical ADFs can be employed to allow one ADF to call another (Koza, 1992). Researchers are now exploring other styles of function creation, e.g. Yu

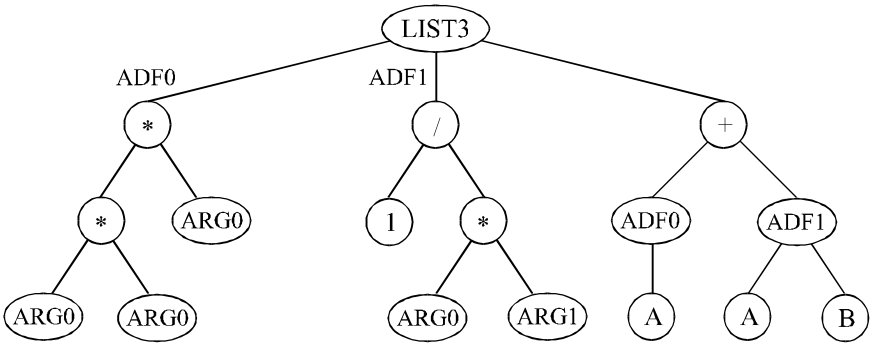


Figure 1.13 A solution with two ADFs (the two left branches of the tree). Note the function calls within the program in the right branch of the tree.

evolves anonymous functions known as λ-abstractions, which are reused through recursion (Yu and Clack, 1998a,b). Current research also investigates the automatic creation of iterations (ADIs), loops (ADLs), recursions (ADRs), and various types of memory stores (ADS), see (Koza et al., 1999).

The hierarchical tree representation used by GP allows crossover and mutation to manipulate solutions whilst preserving the syntax of LISP S-expressions. However, there are problems of excessive disruption with these genetic operators (O'Reilly and Oppacher, 1995). In GAs, the operators tend to be constructive: many offspring generated using crossover or mutation from fit parent solutions will be at least as fit as their parents. In GP, the reverse is true: the operators tend to be destructive, with many offspring less fit than their parents. This sorry state of affairs has led to the development of new crossover operators, designed to minimise the disruption of good solutions. Most of these new operators limit the use of crossover to the recombination of similarly structured parent solutions (Poli and Langdon, 1997a).

Recommended Books for GP:

Genetic Programming: On the Programming of Computers by Means of Natural Selection
by John Koza (1992).

Genetic Programming II: Automatic Discovery of Reusable Programs
by John Koza (1994).

Genetic Programming III
by Koza, Andre, Bennett and Keane (1999).

Advances In Genetic Programming
Edited by Kenneth E. Kinnear Jr. (1994).

Advances In Genetic Programming 2
by Peter Angeline and Kenneth E. Kinnear Jr. (Eds) (1996).

Genetic Programming – an Introduction
by Wolfgang Banzhaf, Peter Nordin, Robert E. Keller and Frank D. Francone (1998).

Genetic Programming and Data Structures
by Bill Langdon (1998)

1.3.3 Evolution Strategies

A Summary

Evolution strategies (or *evolutionstrategie*) were developed in Germany in the 1960s by Bienert, Rechenberg and Schwefel (Bäck, 1996). Evolutionary design was one of the very first applications of this technique, involving shape optimisation of a bent pipe, drag minimisation of a joint plate and structure optimisation of nozzles. However, these early experiments were not performed using computers. Instead, actual physical designs were built, tested and mutated by changing the joint positions or adding and removing segments.

The first ES computer algorithm was demonstrated initially by Schwefel (1965), and then developed further by Rechenberg (1973). This simple form of ES, known as the *two membered* ES, used only two individuals: a parent and child. Like GP, it made no distinction between genotype and phenotype, each individual being represented as a real-valued vector. It has a simple operation: the child solution is generated by randomly mutating the problem parameter values of the parent. Mutation is performed independently on each vector element by aggregating a normal-distributed random variable with zero mean and a pre-selected standard deviation value. The child is then evaluated, and if its fitness is better than the fitness of its parent, the child survives and becomes the parent solution. Otherwise, the child is discarded and the original parent is mutated once again to produce another child solution. This selection scheme is known as (1+1)-selection.

Unfortunately, there were a couple of drawbacks to the (1+1)-ES: the point-to-point search made the procedure susceptible to stagnation at local optima, and the constant standard deviation for each vector element made the procedure slow to converge on optimal solutions.

Advanced ES

Other ES selection schemes followed, incorporating the idea of populations of solutions. By the early 1980s, the current state-of-the-art evolutionary strategies had been developed (Bäck, 1996), known as the $(\mu+\lambda)$-ES and (μ,λ)-ES (where μ is the number of parents and λ is the number of offspring). These new types of ES now strongly resemble the genetic algorithm, by maintaining populations of individuals, selecting the fittest individuals, and using recombination and mutation operators to generate new individuals from these fit solutions.

There are some significant differences between ES and GAs, however. For example, although ES maintains populations of solutions, it separates the parent individuals from the child individuals. In addition, as mentioned above, ES does not manipulate coded solutions like GAs. Instead, like GP, the decision variables of the problem are manipulated directly by the operators. Also unlike the probabilistic selection of GAs and GP, ES selects its parent solutions deterministically.

The $(\mu+\lambda)$-ES picks the best μ individuals from both child and parent populations. The (μ,λ)-ES picks the best μ individuals from just the child population. In order to ensure that a selection pressure is generated, the number of parents, μ must always be smaller than the number of offspring, λ. Bäck (1996) recommends the use of the (μ,λ)-ES with a parent:offspring ratio of 1:7.

Figure 1.14 shows the operation of the population-based evolutionary strategy. The ES is initialised with a population of random solutions, or from a population of solutions mutated from a single solution provided by the user. Parent solutions are chosen randomly from the 'parent population', and a number of random recombination operators are used to generate child solutions, which are placed in the 'child population'. These operators may recombine the values within two parents like the crossover operator of GAs, or they may use a parent for every decision variable in the solution. New solutions are then mutated using *strategy parameters* within each solution.

Mutation plays an important role in ES, and is regarded as the primary search operator. Unlike the entirely random mutation of simple GAs and GP, the mutation of ES follows a *normal distribution*. The distribution of possible mutated values is guided by two types of strategy parameter: standard deviation σ and rotation angle α for each decision variable in the solution. (In fact, ES does not require these two parameters for every variable in the solution – it permits the use of the same strategy parameters for multiple variables.) The strategy parameters influence the direction of search in the search space taken by mutation, and by modifying the values of σ and α. for each variable, the ES is able to use mutation to follow the contours of the search space and quickly find the optimal solutions, see fig. 1.15.

But the ES has another trick up its sleeve. Not only does it have directed mutation, it actually *evolves* this direction. ES achieves this by placing the strategy parameters for each

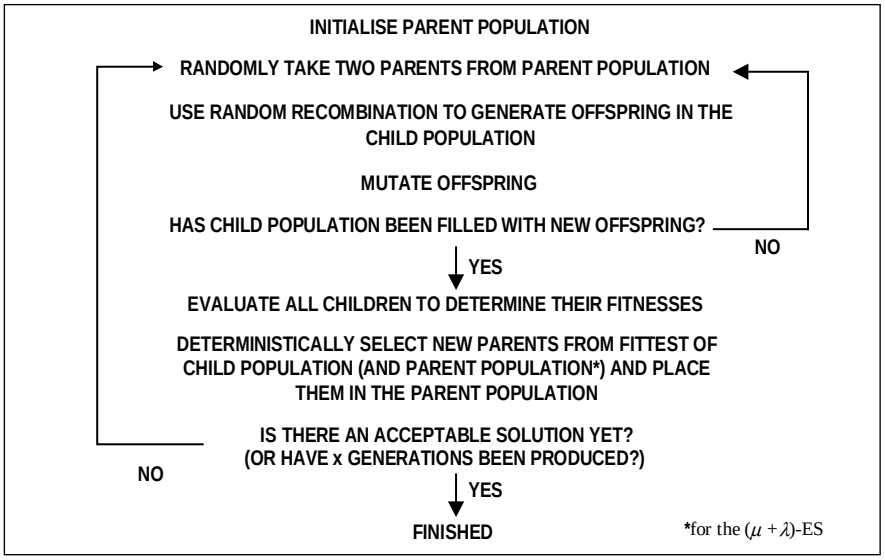


Figure 1.14 The evolutionary strategy.

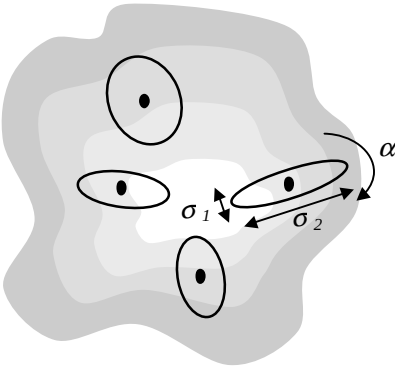


Figure 1.15 Mutation hyperellipsoids defining equal probability density around each solution. In this example, each solution has two σ parameters and one α parameter to guide mutation. Note how mutations towards the better area of the search space are encouraged.

variable within the individual solution. The operators of ES are then able to modify the strategy values in addition to the values of the variables within individuals, and hence optimise the direction of mutation in parallel to the optimisation of the variables. This important feature (which has only been introduced into advanced GAs in the last 5 years or so) is known as *self-adaptation*.

Once mutation and recombination has generated enough new individuals to fill the child population, these individuals are evaluated to obtain fitness measures, as described earlier for GAs. The fittest offspring are then deterministically selected to become parents and these are placed into the parent population.

With the parent population filled with (mostly) new individuals, the ES randomly picks parents from this population to generate new child solutions, which are then evaluated, and so on. Evolution terminates after a predefined number of generations, or when a solution of sufficient quality has been generated.

ES Theory

Like GA theory, the theory of ES was developed in the 1970s, and applies to the simplest form of the algorithm, in this case the (1+1)-ES with no self-adaptation. Rechenberg analysed the convergence rates of this two-membered ES for two objective functions, and calculated the optimal standard deviation and probability values for a successful mutation. From this he formulated the *1/5 success rule*:

The ratio of successful mutations to all mutations should be 1/5. If it is greater than 1/5, increase the standard deviation, if it is smaller, decrease the standard deviation. (Rechenberg, 1973.)

In order to apply the 1/5-success rule, the ES keeps track of the observed ratio of successful mutations to the total number of mutations, measured over intervals of $10 \times n$ trails, where n is the number of variables in the individual. According to this ratio, the standard deviation is increased, decreased, or remains constant. Born (1978) subsequently formally proved that this

simple form of (1+1)-ES will result in global convergence with a probability of one under the condition of a positive standard deviation. Other theoretical analyses have shown that the introduction of populations in ES causes a logarithmic speed-up in evolution, compared to the (1+1)-ES.

As Bäck (1996) describes, Schwefel derived an approximation theory for the convergence rate of the simplified (μ,λ)-ES and ($\mu+\lambda$)-ES (one standard deviation for all object variables; no crossover nor self-adaptation). More recently, Rudolph (1996, 1997, 1998) has used Markov chains to investigate the convergence theory in EAs.

Recommended Books for ES:

Evolutionstrategie: Optimierung Technischer Systeme nach Prinzipien der Biologischen Evolution
by Ingo Rechenberg (1973).

Numerical Optimization of Computer Models
by H.-P. Schwefel (1981).

Evolution and Optimum Seeking
by H.-P. Schwefel (1995).

Evolutionstrategie'94 (volume 1 of ***Werkstatt Bionik und Evolutionstechnik)***
by Ingo Rechenberg (1994).

Evolutionary Algorithms in Theory and Practice
by Thomas Bäck (1996).

Evolutionary Computation: Toward a New Philosophy of Machine Intelligence
by David B. Fogel (1995).

1.3.4 Evolutionary Programming

A Summary

Evolutionary programming resembles evolutionary strategies closely, although EP was developed independently (and earlier) by Lawrence Fogel in the 1960s. The early versions of EP were applied to the evolution of transition tables of finite state machines[3] (FSMs), and the fitness of individuals was based on how closely the output sequence of letters generated by each individual matched a target sequence. A single population of solutions was maintained, and reproduction used mutation alone (Fogel et al., 1966).

[3] In fact, EP was proposed as a procedure to generate machine intelligence. Intelligent behaviours were viewed as the ability to predict one's environment and to provide a suitable response in order to achieve a given goal.

For the sake of generality, the behaviours were represented in finite state machines (FSMs). For each state of an FSM, each possible input symbol has an associated output symbol and next-state transition. A sequence of input symbols such as 010001 is given to an FSM as an observed environment. A behaviour is then considered to be 'intelligent' if it predicts what the next symbol will be and satisfies a pay-off function.

Unfortunately, despite many theses written on EP at New Mexico State University during the 1970s, this algorithm was either misunderstood or overlooked for many years. It was only when Lawrence Fogel's son, David Fogel, redeveloped EP in the late 1980s, that this technique was rediscovered by the research community.

Advanced EP

David Fogel extended the original EP, which could only evolve discrete parameterisations, to allow it to be used for continuous parameter optimisation (fixed-length real-valued vectors). Another important addition to EP was self-adaptation – the use of evolveable strategy parameters to guide mutation in a very similar way to ES.

There are three main types of EP in use: *standard EP*, *meta-EP*,[4] and *Rmeta-EP*. These three types differ by the level of self-adaptation employed. Standard EP uses no self-adaptation, meta-EP incorporates mutation variance parameters in individuals to allow self-adaptation, and Rmeta-EP incorporates mutation variance and covariance parameters into individuals to permit more precise self-adaptation (Fogel, 1995). Figure 1.16 shows the working of a general evolutionary program.

Like ES, EP operates on the decision variables of the problem directly (i.e. the search space is the same as the solution space). During initialisation, these variables are given random values, often with a uniform sampling of values between predefined ranges. These solutions are then evaluated to obtain the fitness values (which in EP may involve some form of scaling or the addition of a random perturbation). Next, parents are picked using *tournament selection*. This type of selection is commonly used in GAs, and involves a series of tournaments between

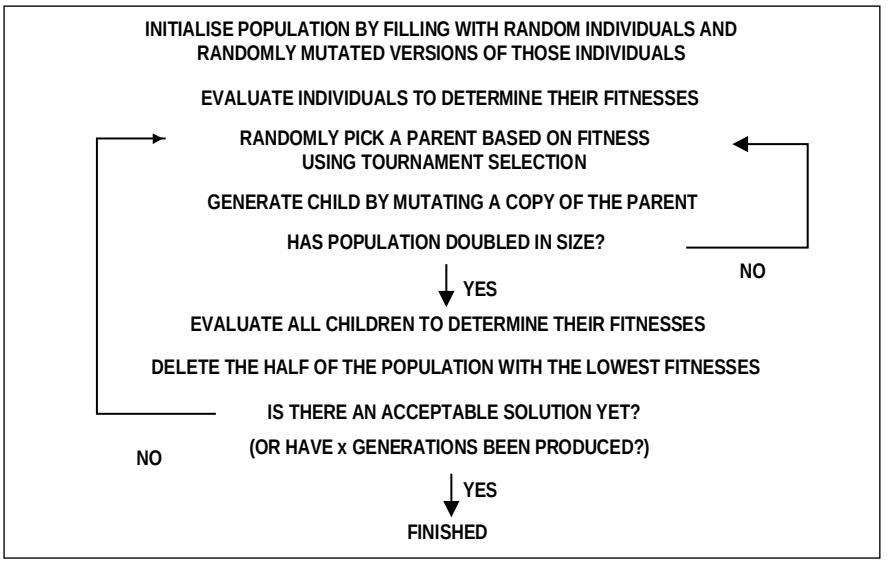


Figure 1.16 The evolutionary programming algorithm.

[4] Meta-EP is now the standard form of EP in use today, and is usually called simply EP.

each individual and a group of randomly chosen individuals. The probability of being selected for reproduction is then based on how many other individuals the prospective parent has managed to beat during the tournament (i.e., a solution with a better fitness than most of the others it 'played' in the tournament has a higher probability of being selected than a solution with a worse fitness compared to its competitors). The number of individuals in each tournament is a global parameter of the algorithm.

Children are generated asexually, by simply creating a copy of the parent and mutating it. In a similar way to ES, meta-EP and Rmeta-EP ensure that mutation will favour the search towards the areas of the search space containing better solutions, by the use of variance and covariance strategy parameters held within individuals for each variable. EP allows mutation to modify these parameters, thus permitting self-adaptation to occur. Unlike any of the evolutionary algorithms mentioned so far, EP does not use any form of recombination operator. All search is performed by mutation, following the assumption that mutation is capable of simulating the effects of operators such as crossover on solutions (Fogel, 1995).

Child solutions are generated and placed into the population until the population has doubled in size. All new solutions are evaluated, and then the half of the population with the lowest fitnesses are simply deleted. Parents are then picked, offspring generated, and so on. Evolution is typically terminated after a specific number of generations have passed.

All three types of EP can also be *continuous* (Fogel and Fogel, 1995), where instead of generating offspring until the population size has doubled and then deleting the unfit half, a single individual is generated, evaluated, and inserted into the population, replacing the least fit individual. This *replacement* method strongly resembles that used by steady-state GAs (Syswerda, 1989).

New advances in EP continue, as this EA is applied to new types of problem. Chellapilla (1997) has recently expanded EP to allow it to evolve program parse trees. Various subtree mutation operators were designed in solving four benchmark problems. He reported that the experiment results showed the technique compares well with EAs which use the crossover operator.

EP Theory

Although ES theory can be applied with little modification to EP (Bäck, 1996), Fogel has performed independent analyses of various forms of EP, including the case where the population size $= 1$ (in which case EP strongly resembles the $(1+1)$-ES (Bäck, 1996)). He calculated that mutations should increment or decrement the value of a decision variable by no more than the square root of the fitness score of the solution.

Fogel proved that the simple EP will converge with a probability of one, however convergence rates (time taken to converge) and other significant features of EP remain unsolved. Complete details of the proofs are beyond the scope of this chapter; interested readers should consult (Fogel, 1992b).

Yao and Liu (1996) proposed a Cauchy instead of Gaussian mutation operator as the primary search operator in EP. In (Yao, Lin and Liu, 1997), an analysis based on the study of neighbourhood and step size of the search space is performed to compare these two mutation operators. Their work provides a theoretical explanation, supported with empirical evidence, of when and why Cauchy mutation is better than Gaussian mutation in EP search. The long jumps provided

by Cauchy mutation increase the probability of finding a near-optimum when the distance between the current search point and the optimum is large. The Gaussian mutation, on the other hand, is a better search strategy when the distance is small.

A study by Gehlhaar and Fogel (1996) also indicates that the order of the modifications of object variables and strategy parameters has a strong impact on the effectiveness of self-adaptation. It is important to mutate the strategy parameters first and then use them to modify the object variables. In this way, the potential of generating good object vectors that are associated with poor strategy parameters can be reduced. Thus the likelihood of stagnation can be reduced.

Experimental comparisons between the different types of EP are ongoing. For example, EP with self-adaptation has now been reapplied to the original task of evolving FSMs (Fogel et al., 1995; Angeline and Kinnear, 1996). Each state in an FSM was associated with mutation parameters to guide which component of the FSM was to be mutated and how to perform the mutation. Angeline and Kinnear (1996) reported that EP with self-adaptation performs better than a standard EP when solving a predication problem.

Recommended Books for EP:

Artificial Intelligence through Simulated Evolution
by L. J. Fogel, A. J. Owens and M. J. Walsh (1966)

System Identification through Simulated Evolution: A Machine Learning Approach to Modeling
by David B. Fogel (1991).

Evolutionary Computation : Toward a New Philosophy of Machine Intelligence
by David B. Fogel (1995).

Evolutionary Algorithms in Theory and Practice
by Thomas Bäck (1996).

1.3.5 A General Architecture for Evolutionary Algorithms (GAEA)

Sadly, the four major types of evolutionary algorithm are rarely considered in unison. Most researchers are genetic algorithmists, genetic programmers, evolutionary strategists, or evolutionary programmers – there are very few evolutionary computationists. Researchers in the field of evolutionary computation spend considerable time and energy exploring their favourite EA, and usually no time at all on considering the general concepts behind EAs.[5] This is unfortunate, because it should be clear that these algorithms are hardly different at all. Consequently, rather than dwell on the differences between the four major types of EA summarised previously, it is perhaps more appropriate to stress the similarities of these techniques.

[5] As the editor of this book, I cannot claim to be any different. My first venture into this field was when, as a teenager, I wrote a program called Evolve, which I discovered years later was essentially both a real-coded steady-state genetic algorithm, and a continuous standard evolutionary program, except that all evolutionary pressure was exerted using negative selection. Despite being yet another ‘independent discoverer’ of an evolutionary algorithm, it will be apparent to any knowledgeable reader that today I am a genetic algorithmist at heart.

In general, as suggested by the theory of Universal Darwinism (Dawkins, 1983), for evolution to occur, the following criteria must be met (where 'transmission' has been expanded to 'reproduction' and 'inheritance' for clarity):

reproduction **inheritance** **variation** **selection**

In other words, as long as some individuals generate copies of themselves which inherit their parents' characteristics with some small variation, and as long as some form of selection preferentially chooses some of the individuals to live and reproduce, evolution will occur. As Chapter 3: *The Memetics of Design*, describes in this book, this is true regardless of what the individual is, be it an ant, an artificial creature, or an idea.

Evolutionary search algorithms are no exception. All EAs perform the **reproduction** of individuals, either directly cloning parents or by using recombination and mutation operators to allow **inheritance** with **variation**. These operators may perform many different tasks, from a simple random bit inversion to a complete local search algorithm. All EAs also use some form of **selection** to determine which solutions will have the opportunity to reproduce, and which will not. The key thing to remember about selection is that it exerts *selection pressure*, or *evolutionary pressure* to guide the evolutionary process towards specific areas of the search space. To do this, certain individuals must be allocated a greater probability of having offspring compared to other individuals. Selection is often misunderstood by developers of EAs, who often regard it to mean simply 'selection of parents'. As will be shown, however, selection does not have to mean parent selection – it can also be performed using fertility, replacement, or even 'death' operators. It is also quite common for multiple evolutionary pressures to be exerted towards more than one objective in a single EA.

Unlike natural evolution, evolutionary algorithms also require three other important features:

initialisation **evaluation** **termination**

Because we are not prepared to wait for the computer to evolve for several million generations, EAs are typically given a head start by **initialising** (or *seeding*) them with solutions that have fixed structures and meanings, but random values. In our earlier 'house' example, each solution might consist of 'number of windows', 'position of roof', 'height' and 'width' parameters.

Evaluation in EAs is responsible for guiding evolution towards better solutions. Unlike natural evolution, evolutionary algorithms do not have a real environment in which the 'survivability' or 'goodness' of its solutions can be tested, they must instead rely on simulation, analysis and calculation to evaluate solutions.

Extinction is the only guaranteed way to **terminate** natural evolution. This is obviously a highly unsuitable way to halt EAs, for all the evolved solutions will be lost. Instead, explicit *termination criteria* are employed to halt evolution, typically when a good solution has evolved or when a predefined number of generations have passed.

There are two other important processes, which although not necessary to trigger or control evolution, will improve the capabilities of evolution enormously. These processes are:

mapping **moving**

Currently only the genetic algorithm separates the search space (containing genotypes) from the solution space (containing phenotypes), and has an explicit **mapping** stage between the two. This is unfortunate, for embryogeny and ontogeny are known by biologists to be highly significant, allowing highly complex solutions to be specified using a compact set of instructions, incorporating constraint handling and error checking.

Using EAs to evolve more than one population or *species* concurrently and separately is becoming an important tool in the optimisation of difficult or multimodal functions. **Moving** (or *migrating* or *injecting*) individuals from one population to another, or from one species to another, allows separately evolving individuals to occasionally share useful genetic information, reducing premature convergence within species, reducing the number of evaluations needed, and improving the quality of solutions evolved.

Figure 1.17 shows the general architecture of evolutionary algorithms (GAEA). This architecture should be regarded as a general framework for evolutionary algorithms, not an algorithm itself. Indeed, most EAs use only a subset of the stages listed. For example, EP uses: *initialisation*, *evaluation*, *selection*, *reproduction*, *replacement* and *termination*. Alternatively, a simple GA uses: *initialisation*, *mapping*, *evaluation*, *selection*, *fertility*, *reproduction* and *termination*. (Each optional stage in the architecture is marked as such.)

To help give the reader a more general view of using computers to perform evolution, this introductory section of the chapter will now briefly examine each of the stages in GAEA.

Initialisation

Evolutionary algorithms typically seed the initial population with entirely random values (i.e., starting from scratch). If, like the genetic algorithm, a distinction is made between the search space and the solution space, then the *genotype* of every individual will be filled with random alleles. Evolution is then used to discover which of the randomly sampled areas of the search space contain better solutions, and then to converge upon that area. Sometimes the entire population is constructed from random mutants of a single user-supplied solution. Often random values are generated between specified ranges (a form of constraint handling). It is not uncommon for explicit constraint handling to be performed during initialisation, by deleting any solutions which do not satisfy the constraints and creating new ones.

More complex problems often demand alternative methods of initialisation. Some researchers provide the EA with ‘embryos’ – simplified non-random solutions which are then used as starting points for the evolution of more complex solutions. John Koza describes such an approach in Chapter 16. Some algorithms actually attempt to evolve representations or low-level building blocks first, then use the results to initialise another EA which will evolve complex designs using these representations or building blocks. John Gero and Michael Rosenman describe this approach in Chapter 15.

Although most algorithms do use solutions with fixed structures (i.e. a fixed number of decision variables), some, like GP, allow the evolution of the number and organisation of parameters in addition to parameter values. In other words, some evolve *structure* as well as *detail*. For such algorithms, initialisation will typically involve the seeding of solutions with both random values and random structures.

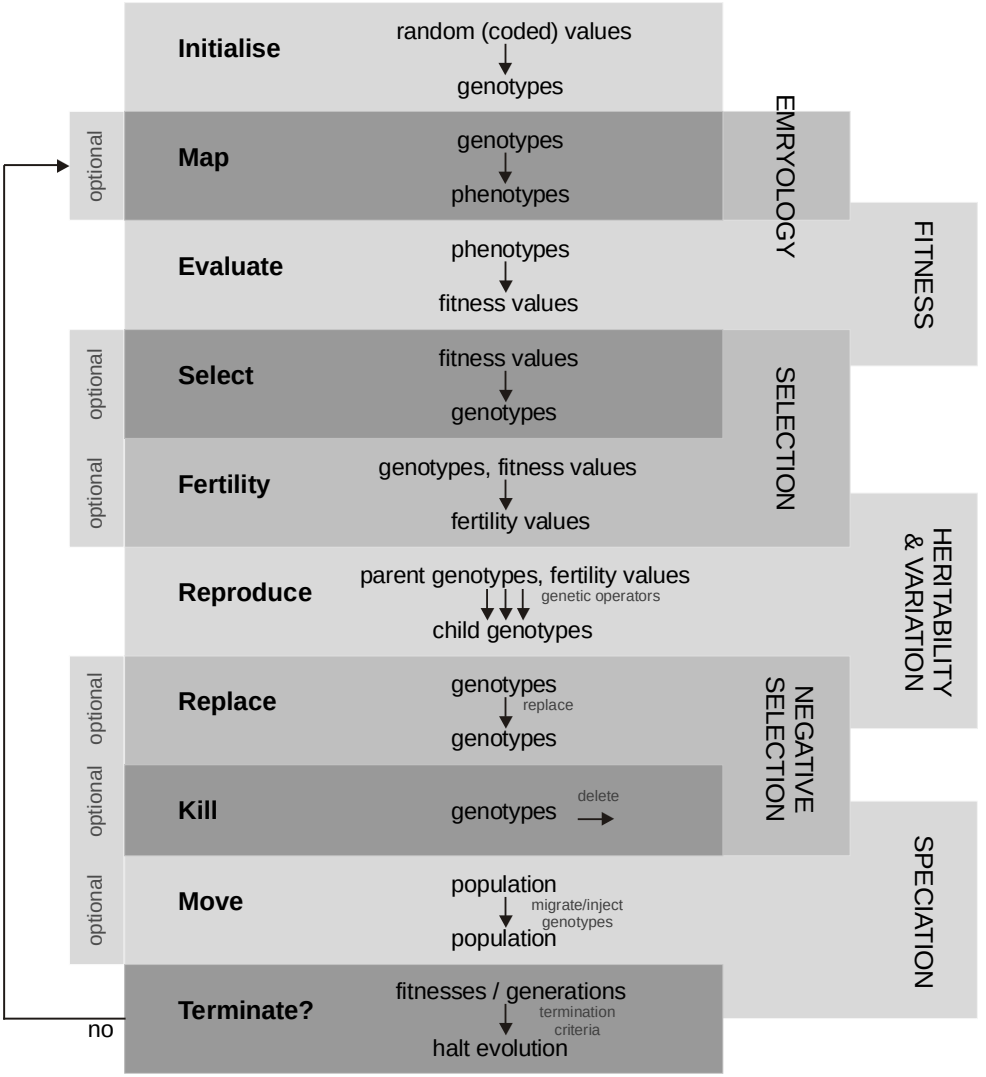


Figure 1.17 The general architecture of evolutionary algorithms (GAEA).

Map

Only algorithms which make a distinction between the search space and the solution space require a mapping stage to convert genotypes into phenotypes. Consequently, the genetic algorithm is the only major EA which uses mapping. This mapping stage is often trivially simple, e.g. converting a binary allele in the genotype to a decimal parameter value in the phenotype. However, as some chapters in this book describe, this mapping stage can also be very complex.

So why bother with it? This is a difficult question which neatly divides evolutionary computation into two camps: those who believe the effects of genetic operators can be simulated on

phenotypes without needing to resort to storing, modifying, and mapping genotypes, and those who prefer to 'do it properly' and actually perform search at the genotype-level. There can be no doubt that evolution will occur whether genotypes are maintained or not, but most genetic algorithmists would argue that evolutionary search is improved if genotypes are employed. Certainly the latest advances in the understanding of natural evolution are from the level of the gene, not of the organism (Dawkins, 1976, 1986, 1996).

But the mapping process should not be viewed as a time-consuming side-effect of maintaining genotypes. Quite the opposite – this process, known by biologists as *embryogeny*, is highly important in its own right. Indeed, the advantages of embryogeny have caused some researchers to introduce genotypes into traditionally 'phenotype-only' algorithms such as GP (Banzhaf, 1994).

There are many good reasons to use a mapping stage in an EA. These include:

- **Reduction of search space**. Embryogeny permits highly compact genotypes to define phenotypes. This reduction (often recursive, hierarchical and multifunctional) results in genotypes with fewer parameters than their corresponding phenotypes, causing a reduction in the dimensionality of the search space, and hence a smaller search space for the EA.
- **Better enumeration of search space**. Mapping permits two very differently organised spaces to coexist, i.e. a search space designed to be easily searched can allow the EA to locate corresponding solutions within a hard-to-search solution space.
- **More complex solutions in solution space**. By using 'growing instructions' within genotypes to define how phenotypes should be generated, a genotype can define highly complex phenotypes. Chapter 14 gives an excellent example of this, using a Lindenmayer system as the embryogeny process.
- **Improved constraint handling**. Mapping can ensure that all phenotypes always satisfy all constraints, without reducing the effectiveness of the search process in any way, by mapping every genotype onto a legal phenotype.

The use of simple mapping stages is increasing, but research in artificial embryogenies is still in its infancy, with most researchers designing their own. Many of the chapters in the last two sections of this book describe approaches for evolutionary design. The design of artificial embryogenies is not trivial, so it seems likely that researchers will attempt to evolve them in the future (just as our own embryogeny evolved in nature). Section 1.5.2 in the final part of this chapter describes embryogenies in more detail.

Evaluation

Every new phenotype must be evaluated to provide a level of 'goodness' for each solution. Often a single run of an EA will involve thousands of evaluations, which means that almost all computation time is spent performing the evaluation process (most EAs use negligible processing time themselves). In evolutionary design, evaluation is often performed by dedicated analysis software which can take minutes or even hours to evaluate a single solution, so there is often a strong emphasis towards reducing the number of evaluations during evolution. Chapters 5, 6 and 7 describe various advanced techniques in EAs to reduce evaluation times.

Evaluation involves the use of fitness functions to assign fitness scores to solutions. These fitness functions can have single or multiple objectives, they can be unimodal or multimodal, continuous or discontinuous, smooth or noisy, static or continuously changing. EAs are known to be proficient at finding good solutions for all these types of fitness function, but specialised techniques are often required for multimodal, multiobjective, noisy and continuously changing functions. Some of these are summarised at the end of this chapter.

Evaluation is not always performed by explicit fitness functions. Some EAs employ human evaluators to view and judge their solutions – the chapters on evolutionary art describe this approach. Fitness can also be determined by competition between solutions (for example, each solution may represent a game playing strategy, and the fitness of each strategy depends on how many other solutions in the population the current strategy can beat (Axelrod, 1987)).

Once fitness values have been calculated, some form of scaling is common. For example, if multiple species of individuals are being evolved, *fitness sharing* reduces the fitness of any individuals in large groups to encourage the development of more groups containing smaller numbers of individuals (Goldberg, 1989). Fitness scores are also commonly scaled as part of multiobjective optimisation (Bentley and Wakefield, 1997c) or to prevent unwanted biases during parent selection (Goldberg, 1989).

Parent Selection

Parent solutions are always required in an EA, or no child solutions can be generated. However, the *preferential selection* of some parents instead of others is not essential to evolution. (If parent selection is not present in the EA, then all solutions are permitted to generate offspring with equal probability.) Every one of the major EAs does perform parent selection, but evolution will still occur without it, as long as evolutionary pressure is exerted by one of the three other selection methods: *fertility*, *replacement* and *death*.

Choosing the fitter solutions to be parents of the next generation is the most common and direct way of inducing a selective pressure towards the evolution of fitter solutions. Typically, one of three selection methods are utilised: fitness ranking, tournament selection, or fitness proportionate selection. Fitness ranking sorts the population into order of fitness and bases the probability of a solution being selected for parenthood on its position in the ranking. Tournament selection bases the probability of a solution being selected on how many other randomly picked individuals it can beat (see the section on EP). Fitness proportionate selection (or roulette wheel selection) bases the probability of a parent being selected on the relative fitness scores of each individual, e.g. a solution ten times as fit as another is ten times more likely to be picked as a parent (Goldberg, 1989). This method also incorporates the *fertility* selection method, see below.

Although fitter parents are normally selected, this does not have to be the case. It is possible to select parents based on how many constraints they satisfy, or how well they fulfil other criteria, as long as a fitness-based selection pressure is introduced elsewhere in the algorithm. The selection of pairs of parents is usually limited by EAs with speciation or multiple populations (i.e. two parents from different species or different populations/islands will not be permitted to generate offspring together). In algorithms that record the age of individuals, parent selection may be limited to individuals that are ‘mature’ or individuals which are below their maximum lifespans. Any individual that is sterile (see below) should not be selected for parenthood.

Fertility

The *fertility* of a parent solution is the number of offspring that parent can have, e.g. a parent with a high fertility will have more offspring than a parent with low fertility. The fertility of parent solutions is often confused with the selection of parent solutions. For example, fitness proportionate selection not only selects parents based on their relative fitnesses, it also increases the fertility of fitter parents based on fitnesses. This confusion is unfortunate, because in reality, changing the fertility of parents is an independent and separate way to exert selection pressure in EAs.

The separation of parent selection and fertility is perhaps clearest in natural evolution. A peahen *selects* the most attractive peacock it can find (perhaps based on the size and pattenation of the tail) to be the parent of its offspring (Dawkins, 1986). Conversely, the *fertility* of the two parents is an innate characteristic of the parents, based on their fitness, their age, and, for some birds, the number of other birds sharing the same neighbourhood (Dawkins, 1976). Because unhealthy birds tend to have lower fertilities than healthy birds,[6] an evolutionary pressure is exerted towards healthy birds, in addition to the selective pressure towards birds with ornate tails.

Currently the use of fertility to induce selection pressures towards explicit goals in EAs has been limited to constraint handling. Experimental results have shown that selecting parents based on fitness, and setting fertility values based on how many constraints each solution satisfies, can be a very effective method of performing twin-objective optimisation using EAs (Yu and Bentley, 1998).

Individuals in the population may have reduced fertilities, or may even be sterile (i.e., have a fertility of zero) if immature or too old. Two parents from different species are typically regarded as having very low fertilities (e.g., a one in twenty chance of generating any offspring). EAs that do not employ fertility ensure that every parent has a fixed, unchanging number of offspring – usually with each parent producing one child.

Reproduction

Reproduction is the cornerstone of every evolutionary algorithm – it is the stage responsible for the generation of child solutions from parent solutions. Crucially, child solutions must *inherit* characteristics from their parents, and there must be some *variability* between the child and parent. This is achieved by the use of the genetic operators: recombination and mutation.

Recombination operators require two or more parent solutions. The solutions (or the genotypes, if the algorithm distinguishes between search and solution spaces) are ‘shuffled together’ to generate child solutions. EAs normally use recombination to generate most or all offspring. For example, ES often uses recombination to generate offspring, GAs typically use recombination with a probability of 0.7, with the remaining offspring being clones of their parents. Only EP uses no recombination at all.

[6] This is an oversimplification: in nature most creatures have an optimal fertility rate – if it is too low, none may survive predation, if too high, the excessive cost of rearing all the offspring may cause all to die of malnutrition. If the creature is unfit, it may have too many or too few offspring, with the net result that fewer survive (Dawkins, 1976).

Recombination is normally performed by crossover operators in EAs. Examples of the working of various types of crossover were provided in the previous sections on specific EAs.

Mutation operators modify a single solution at a time. Some EAs mutate a copy of a parent solution in order to generate the child, some mutate the solution during the application of the recombination operators, others use recombination to generate children, and then mutate these children. In addition, the probability of mutation varies depending on the EA. For example, GAs use low probabilities, often between 0.01 and 0.001 per bit in the genotype, whereas EP always uses mutation.

There are huge numbers of different mutation operators in use today. Examples include: bit-mutation, translocation, segregation, inversion (GAs), structure mutation, permutation, editing, encapsulation (GP), mutation directed by strategy parameters (ES and EP), and even mutation using local search algorithms (memetic algorithms) (Goldberg, 1989; Koza, 1992; Radcliffe and Surrey, 1994a; Bäck, 1996). The previous sections on EAs described some of these further.

An important feature of both recombination and mutation is *non-disruption*. Although variation between parent and child is essential, this variation should not produce excessive changes to phenotypes. In other words, child solutions should always be near to their parent solutions in the solution space. If this is not the case, i.e. if huge changes are permitted, then the semblance of inheritance from parent to child solutions will be reduced, and their position in the solution space will become excessively randomised. Evolution relies on inheritance to ensure the preservation of useful characteristics of parent solutions in child solutions. When disruption is too high, evolution becomes no more than a random search algorithm.

This is the reason why researchers are trying to improve the crossover and mutation operators of GP – the existing operators disrupt so many of the offspring that most are less fit than their parents. This is also the reason why ES and EP use a normal distribution to guide their mutation operators – the distribution encourages smaller mutations.

Replacement

Once offspring have been created, they must be inserted into the population. EAs usually maintain populations of fixed sizes, so for every new individual that is inserted into the population, an existing individual must be deleted. The simpler EAs just delete every individual and replace them with new offspring. However, some EAs (such as the steady-state GA and continuous EP) use an explicit replacement operator to determine which solution a new child should replace. Replacement is often fitness-based, i.e. children always replace solutions less fit than themselves, or the weakest in the population are replaced by fitter offspring. Indeed, the use of fitness-based replacement exemplifies the famous Darwinian phrase ‘survival of the fittest’, for by replacing all the less-fit solutions, the fittest literally have an increased chance of survival.

Replacement is clearly a third method of introducing evolutionary pressure to EAs, but instead of being a selection method, it is a *negative selection* method. In other words, instead of choosing which individuals should reproduce or how many offspring they should have, replacement chooses which individuals will die. (Negative selection also takes place within immune systems; recent work using computers to evolve artificial immune systems uses negative selection as the sole purveyor of evolutionary pressure (Forrest et al., 1995)).

Replacement need not be fitness-based, it can be based on constraint satisfaction, the similarity of genotypes, the age of solutions, or any other criterion, as long as a fitness-based evolutionary pressure is exerted elsewhere in the EA. Replacement is also limited by speciation within EAs: a child from two parents of one species/population/island should not replace an individual in a different species/population/island.

Kill

The fourth and final way to induce evolutionary pressure in an EA is very similar to the replacement method. It involves 'killing' individuals based on some criterion, and consequently is also a negative selection method. 'Kill' is related to replacement, but is subtly different. Replacement involves the comparison of a child with the solution it may replace. 'Kill' operates on a single solution – if the solution does not fulfil the criterion, it is removed from the population. Also unlike replacement, the 'kill' operator is usually used for constraint satisfaction rather than to help generate fit solutions. (Non-continuous EP is the exception to this – 'kill' is used to remove the weakest half of the population after the generation of offspring.)

Typically, the child solution is 'killed' before it has a chance to reproduce. This may happen during initialisation or during reproduction, but in either case, once a solution has been deleted, another attempt will be made to generate a solution (possibly using the same parents). If every child that does not fulfil the criterion is deleted without exception, the maximum possible level of negative selection is exerted, i.e. every solution will always satisfy the criterion. Simply deleting solutions in an EA is very much a brute-force method, often used to enforce hard constraints. As will be described later in this chapter, this approach suffers from significant drawbacks such as reduced diversity and increased difficulty of search for the EA.

'Kill' can also be used as a 'die of old age' operator. This is normally achieved by recording the number of generations each individual has been in the population in the EA and 'killing' solutions when they reach a maximum lifespan (although most will have been replaced by fitter offspring long before then). Deleting older individuals can be a useful method of preventing very fit individuals from becoming 'immortal' and corrupting evolution by filling the entire population with their numerous progeny – particularly if the immortal individual only received a good fitness score because of random noise in the fitness function (Bentley, 1997).

Move

All evolutionary algorithms search populations of solutions in parallel, but most converge onto a single point in the search space after a number of generations. If the fitness function has a single optimal solution, this behaviour is acceptable, but if the fitness function is multimodal (i.e., it has multiple optima), then it is often desirable to use an EA to converge on as many of the separate optima as possible. This is achieved by evolving a number of separate, non-interbreeding groups of individuals (sometimes referred to as separate populations, species, islands, or *demes*), and allowing each of these groups to converge onto potentially different optima. Such EAs are often termed *parallel* or *distributed*. Other motivations for the use of these EAs include the reduction of evaluation times and improvement of solution quality (Eby et al., 1997).

If the separate groups of individuals evolving in the EA never interact in any way, this becomes equivalent to running multiple EAs at the same time, or running one EA many times. However, by permitting the occasional migration or injection of an individual from one group to another, the behaviour becomes distinct.

'Move' encompasses such operators in EAs. 'Move' operators allow characteristics evolving in one group of individuals to be passed to another group of individuals, to help propagate the best independently evolving features. Typically only a single individual is moved (or migrated, or injected) at a time, and usually this occurs infrequently. Individuals may be randomly chosen, or, more commonly, selected according to fitness. The 'movement' may also involve a translation from one representation to another. Chapter 7: *The Optimization of Flywheels using an Injection Island Genetic Algorithm* describes this process in more detail.

Termination

Evolution by an EA is halted by termination criteria, which are normally based on solution quality and time. Most EAs use quality-driven termination as the primary halting mechanism: they simply continue evolving until an individual which is considered sufficiently fit (or for GP, until a program with hits for every exemplar) has been evolved. Some EAs will also reinitialise and restart evolution if no solutions have attained a specific level of fitness after a certain number of generations.

For algorithms which use computationally heavy fitness functions, or for algorithms which must generate solutions quickly, the primary termination criterion is based on time. Normally evolution is terminated after a specific number of generations, evaluations, or seconds. In order to reduce the number of unnecessary generations, some algorithms measure the convergence rates during evolution, and terminate when convergence has occurred (i.e. when the genotypes, phenotypes or fitnesses of all individuals are static for a number of generations). Many EAs also permit the user to halt evolution – an option often misused by more impatient users.

Some EAs do not use explicit termination criteria. These rare beasts are used to continuously adapt to changing fitness functions, and so must evolve new solutions unceasingly. Experiments have shown that EAs with self-adaptation (such as ES and EP) seem to provide the best results when trying to evolve solutions towards a 'moving target' (Bäck, 1996). Harvey and Thompson (1997) describe a GA created for this purpose. Nevertheless, even these EAs are subject to the most fundamental of termination criteria: a power failure.

1.3.6 From Evolutionary Algorithms to Evolutionary Design

This section of the chapter has introduced the concept of using computers to *search* for good solutions in a *search space*. The parallel searching mechanism used by evolutionary algorithms was described, and the four major EAs were summarised: genetic algorithms, genetic programming, evolution strategies, and evolutionary programming. The section concluded by describing the general architecture of evolutionary algorithms, showing that all EAs are fundamentally the same.

Having now explained how computers are used to perform evolution, the following section describes how computers perform evolutionary design.

1.4 Evolutionary Design

Evolutionary design has its roots in computer science, design, and evolutionary biology. It is a branch of evolutionary computation, it extends and combines CAD and analysis software, and it borrows ideas from natural evolution, see fig. 1.18.

The use of evolutionary computation to generate designs has taken place in many different guises over the last 10 or 15 years. Designers have optimised selected parts of their designs using evolution, artists have used evolution to generate aesthetically pleasing forms, architects have evolved new building plans from scratch, computer scientists have evolved morphologies and control systems of artificial life.

In general, these varied types of evolutionary design can be divided into four main categories: *evolutionary design optimisation*, *creative evolutionary design*, *evolutionary art*, and *evolutionary artificial life forms*, see fig. 1.19. As is usually the case with any kind of classification system, the work of a few researchers does not fall neatly within one category, but may be included in two or more categories. Such work comprises four 'overlapping' types of evolutionary design: *integral evolutionary design*, *aesthetic evolutionary design*, *artificial life-based evolutionary design*, and *aesthetic evolutionary AL* (Bentley, 1998a). Figure 1.19 shows all of these areas of research and how they relate to each other.

This middle section of the chapter summarises the scope of research in each area of evolutionary design. The aims and objectives of researchers in each area are described, and some key contributions to the fields of research is examined. For each of the four major aspects of evolutionary design, examples are provided of how designs are represented, which evolutionary algorithms are used and what designs have been evolved.

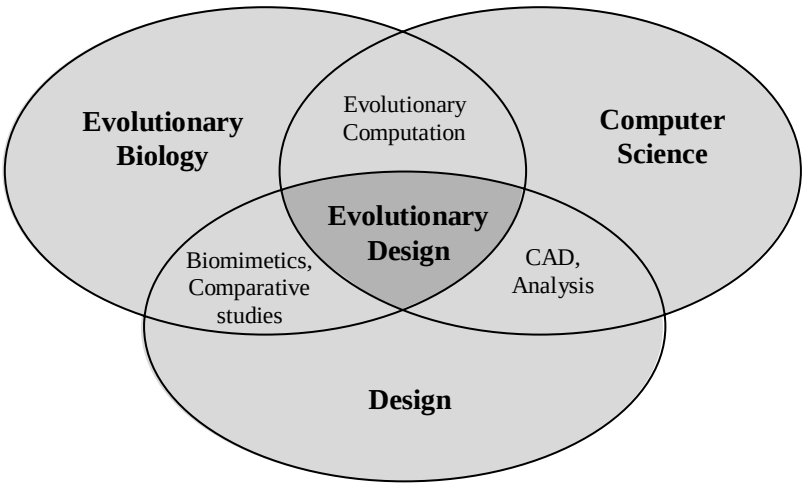


Figure 1.18 Evolutionary design has its roots in computer science, design, and evolutionary biology.

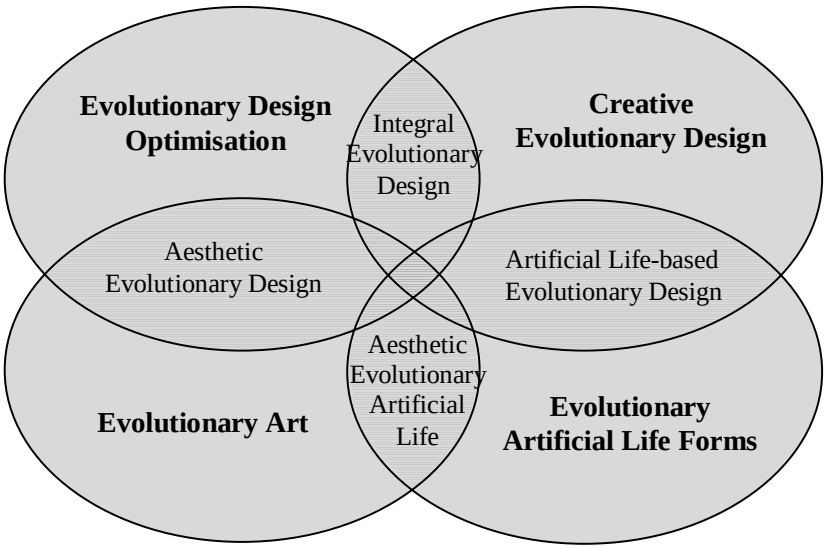


Figure 1.19 Aspects of evolutionary design by computers.

1.4.1 Four Aspects

Evolutionary Design Optimisation

The use of evolutionary computation to optimise existing designs (i.e., perform *detailed design* or *parametric design*) was the first type of evolutionary design to be tackled. Over the last fifteen years, a huge variety of different engineering designs have been successfully optimised (Holland, 1992; Gen and Cheng, 1997), from flywheels (Eby et al., 1997) to aircraft geometries (Husbands et al., 1996). Other more unusual types of evolutionary design optimisation include reliability optimisation (see Chapter 8) and techniques for solving the protein folding problem (Canal et al., 1998).

Although the exact approach used by developers of such systems varies, typically this type of evolutionary design cannot be classed as *generative* or *creative* (see next section). Practitioners of evolutionary optimization usually begin the process with an existing design, and parameterise those parts of the design they feel need improvement. The parameters are then encoded as genes, the alleles (values) of which are then evolved by an evolutionary search algorithm. The designs are often judged by interfacing the system to analysis software, which is used to derive a fitness measure for each design.

Phenotype representations for these design optimisation problems are application-specific, consisting of existing designs for that application, with the evolved parameter values simply inserted into the corresponding parameterised elements. Artificial embryogenies (mapping or 'growth' stages from genotypes to phenotypes (Dawkins, 1989)) are often rudimentary or non-existent, simply because they are not necessary for such evolutionary design. Because of this, genotype representations may match phenotype representations closely, often with a one-to-one mapping between genes and parameters. Consequently, the addition or deletion of

genes in genotypes, and parameters in phenotypes, is usually not performed by the evolutionary algorithm for evolutionary design optimisation.

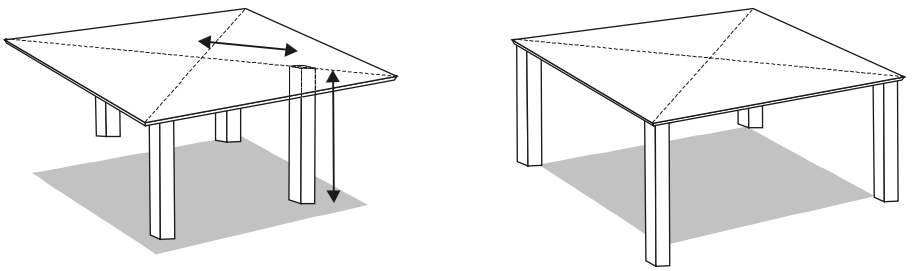
To illustrate this form of evolutionary design, consider the design of a four-legged table. A typical approach to evolutionary design optimisation would be to parameterise part of the table design – for example, the position and length of the legs – and use an EA to optimise the values of those parameters for some criteria – for example, maximise the stability of the table (Bentley and Wakefield, 1996b). As fig. 1.20 shows, phenotypes could consist of eight parameters, with genotypes being 64 bits.

In this trivial example, the optimal solution is clearly a table design with all four legs the same length, and with the legs placed at the four corners of the table top, fig. 1.20 (right).

As the example illustrates, evolutionary optimisation finds functionally optimal (or at least functionally good) permutations of the form of existing designs. However, it is incapable of changing the design concept (i.e. a table top resting on a single pedestal with a wide base will never be 'invented').

Evolutionary optimisation places great emphasis upon finding a solution as close to the global optimal as possible – perhaps more so than for any other type of evolutionary design. Often designs that are already of good quality are to be improved, and it can be a challenge to improve them at all. Because of this motivation towards global optimality, researchers tend to concentrate on methods for evolutionary search which reduce any tendencies towards convergence upon local optima. In addition, because the analysis software used to provide fitness functions for solutions can have heavy computational demands, there is often a strong emphasis towards reducing the number of evaluations required before a final solution is found.

Numerous techniques have been tried to achieve these goals. To improve performance, sometimes multiple genetic representations are used in parallel (see Chapter 7). Many complex types of genetic algorithm are used (Gen and Cheng, 1997). For example, Husbands et al. (1996)



Phenotype:
Table consisting of fixed top and four legs defined by:
Length of leg 1, *Distance of leg 1 from centre*
Length of leg 2, *Distance of leg 2 from centre*
Length of leg 3, *Distance of leg 3 from centre*
Length of leg 4, *Distance of leg 4 from centre*

Genotype:

11010110	10101101	10101110	10011010	01101010	10001010	11110010	00101110
Length 1	*Distance 1*	*Length 2*	*Distance 2*	*Length 3*	*Distance 3*	*Length 4*	*Distance 4*

Figure 1.20 Evolutionary optimisation of a table.

describes the use of a distributed GA and a distributed GA hybridized with gradient descent techniques to evolve the cross-section of optimal aircraft wingboxes. The research found that hybrid GAs outperformed many other search algorithms for this problem (Husbands et al., 1996).

The *Evolutionary Optimization* section in this book provides two chapters on 'classic' evolutionary optimisation. In Chapter 6, Andy Keane descibes the minimisation of the structural vibration of designs such as satellite booms, using a GA to minimise fitness values returned by a statistical energy analysis package. Gordon Robinson also describes his work to optimise strain in load cells. In Chapter 7, Erik Goodman describes the optimisation of flywheels using GAs. Cross-sections of flywheels are parameterised (into a collection of height parameters) and evolved, see fig. 1.21. Evaluation of structure is performed using multiple finite element models (Eby et al., 1997). In Chapter 8, Mitsuo Gen and Jong Ryul Kim describe a more unusual application for evolutionary optimisation: reliabity design. For more examples of evolutionary optimisation of designs, see Gen and Cheng's recent book: *Genetic Algorithms and Engineering Design* (Gen and Cheng, 1997).

Creative Evolutionary Design

Calling anything generated by computer 'creative' is fraught with ambiguity and controversy, so this section will begin by attempting to define, for the purposes of evolutionary design, what 'creative' actually means.

Writing about this very subject in his paper 'Computers and Creative Design', Gero (1996) makes the distinction between cognitive and social views, i.e. an individual can display creativity when designing, and a design can have characteristics which may be regarded as being creative. Gero concentrates on the former definition, and concludes that a computer is designing creatively when it explores the set of possible design state spaces in addition to exploring parameters within individual design spaces. In other words, Gero indicates that by evolving the

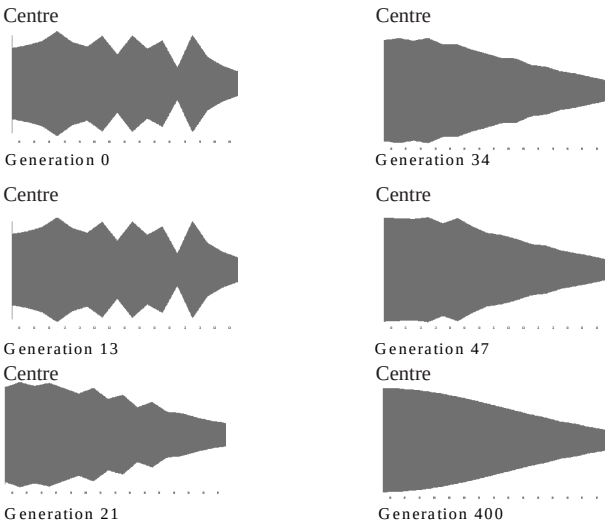


Figure 1.21 Goodman's evolutionary optimisation of flywheels.

number of decision variables in addition to evolving the *values* of those variables, a computer is being creative (Gero, 1996). In a similar vein, Boden (1992) suggests in her book *The Creative Mind*, that creativity is only possible by going beyond the bounds of a representation, and by finding a novel solution which simply could not have been defined by that representation. Boden, however, does not feel that computers are capable of such creativity (Boden, 1992).

Other definitions for creative design include: the transfer of knowledge from other domains (see Chapter 4), having the ability to generate 'surprising and innovative solutions', or the creation of 'novel solutions that are qualitatively better than previous solutions' (Gero and Kazakov, 1996). However, for the purposes of this book, Rosenman's description seems most apt:

The lesser the knowledge about existing relationships between the requirements and the form to satisfy those requirements, the more a design problem tends towards creative design.
(Rosenman, 1997).

Consequently, the main feature that all creative evolutionary design systems have in common, is the ability to generate entirely new designs starting from little or nothing (i.e. random initial populations), and be guided purely by functional performance criteria. In achieving this, such systems often do vary the number of decision variables during evolution (Bentley and Wakefield, 1997b; Rosenman, 1997). They can often generate surprising and innovative solutions, or novel solutions qualitatively better than others (Bentley and Wakefield, 1997a; Harvey and Thompson, 1997). Whether this means that these systems are really 'designing creatively', or whether they simply generate 'creative designs', will be left for the reader to decide.

Research in the field of creative evolutionary design is concerned with the preliminary stages of the design process. There are two main approaches. Both involve the use of evolutionary computation to generate entirely new designs from scratch, however the level at which these designs are represented is different:

Conceptual evolutionary design
– *the production of high-level conceptual frameworks for designs.*
In this type of evolutionary design, the relationships and arrangements of high-level design concepts are evolved in an attempt to generate novel preliminary designs. A good example of this is the work of Pham, who describes his preliminary design system known as TRADES (TRAnsmission DESigner) (Pham and Yang, 1993). TRADES uses a genetic algorithm to evolve the organisation of a set of conceptual building blocks (such as rack and pinion, worm gear, belt drive). When given the type of input (e.g. rotary motion) and the desired output (e.g. perpendicular linear motion), the system generates a suitable conceptual transmission system to convert the input into the output.

In these systems, evolution is used to search through the possible networks of interconnected conceptual building blocks. The genotype and phenotype representations of these systems are often simple, with rudimentary embryogenies, if any. Returning to our 'table' example, the typical approach to conceptual evolutionary design would be to devise a number of conceptual building blocks, each having a specific behaviour, and use evolution to find a suitable organisation of the blocks to ensure that the whole design behaves as a table, see fig. 1.22.

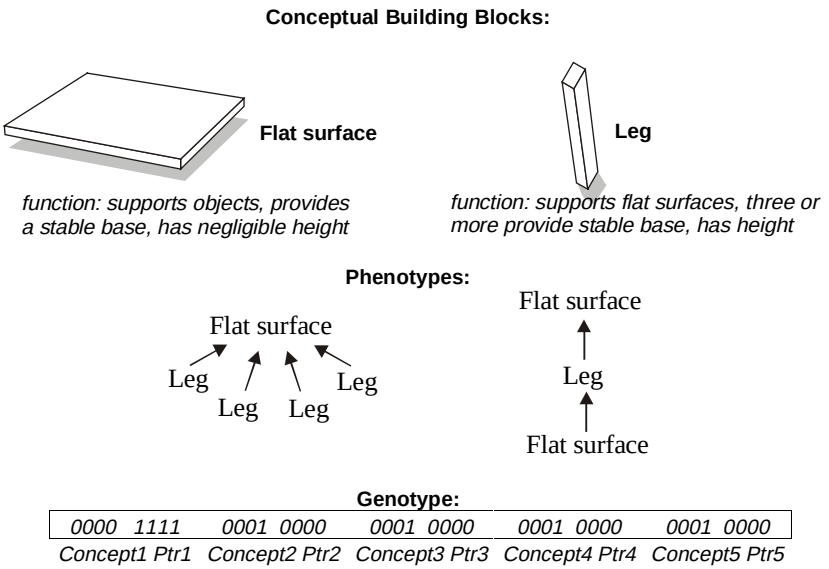


Figure 1.22 Conceptual evolutionary design of a table.

The example uses only two conceptual building blocks: *flat surface* and *leg*, each of which have their behaviours predefined. Phenotypes consist of networks of these concepts. Figure 1.22 shows the phenotype representation of a four-legged table (middle left) and a table with its table top resting on a single pedestal with a wide base (middle right). If GP is used to evolve these designs, the phenotypes could be directly modified, and the number of concepts would be variable. If a GA is used to evolve the designs, a genotype representation similar to the one shown in fig. 1.22 (bottom) would be required. It should be clear that, unlike evolutionary optimisation, conceptual evolutionary design is capable of generating new design concepts. However, such systems are inevitably limited to the building blocks and their functions which have been provided by the designer.

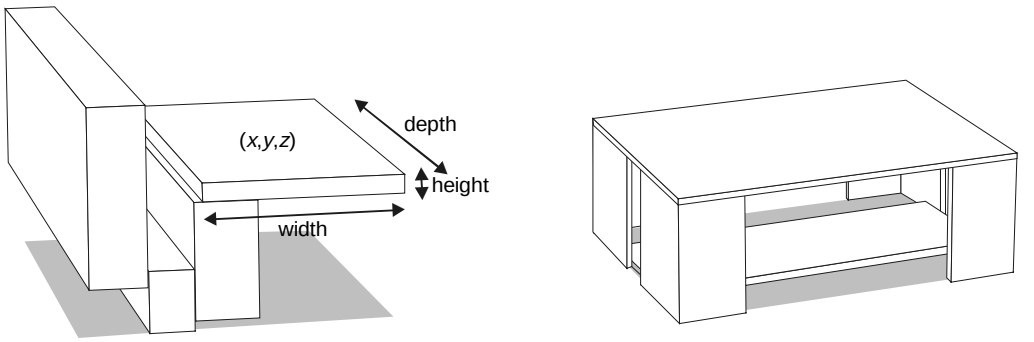
Basic evolutionary algorithms are usually sufficient for conceptual evolutionary design. There are always exceptions to every rule, however, and the work of Parmee provides such an exception. Parmee (1996) describes the use of structured GA to evolve a large-scale hydropower system (this is intended to take place at the feasibility/bid stages of the design process, after the conceptual design stage). His advanced GA manipulates a design hierarchy of ‘sites’, ‘dam types’, ‘tunnel lengths’, ‘modes of operation’, etc., and allows appropriate elements to be switched on or off by control genes during evolution (Parmee, 1996a). This work is mentioned by Ian Parmee in Chapter 5.

Generative evolutionary design (or genetic design)
– *the generation of the form of designs directly.*
Using computers to generate the form of designs rather than a collection of pre-defined high-level concepts has the advantage of giving greater freedom to the computer. Typically such systems are free to evolve any form capable of being represented, and the evolution of such forms may well result in the emergence of implicit design concepts (Bentley and Wakefield,

1997a; Harvey and Thompson, 1997). However, the difficulty of this type of creative evolutionary design is severe, since it often involves the creation of dynamically specified representations and complex evaluation routines (see below).

Often involving just the preliminary stages of design, the emphasis for this type of evolutionary design is on the generation of novelty and originality, and not the production of globally optimal solutions. Representations of form vary tremendously, but they do all share certain features. Because the emphasis is on the generation of new forms, phenotype representations are typically quite general, capable of representing vast numbers of alternative morphologies (this is in contrast to representations for optimisation, which can only define variations of a single form). Representations range from direct spatial partitioning (e.g. voxels), which have one-to-one mappings between genes in genotypes and elements of phenotypes (Baron et. al., 1997), to highly indirect representations which use shape grammars or cellular automata (CA) with some advanced embryogenies to map genotypes to phenotypes (Frazer, 1995; Coates, 1997; Rosenman, 1997).

Figure 1.23 shows how a generative evolutionary design system approaches the task of evolving a table. The initial population begins with randomly shaped 'blobs' (fig. 1.23, left), and evolution is used to gradually fine-tune these shapes until they function as tables (fig. 1.23, right). The representation is crucial for such systems – every part of the design must be alterable. In this example, designs (phenotypes) are represented by a number of blocks, defined by their 3D position and size. Genotypes define *desired* 3D position and size genes. As alleles are mapped to parameter values, the values may change slightly (i.e. two overlapping blocks are 'squashed' until they touch rather than overlap). Genotypes may define partial designs, which are subsequently reflected to form symmetrical phenotypes (fig. 1.23, right). These simple mapping processes mean that there is no longer a direct one-to-one mapping between genes



Phenotype:

x, y, z, width, height, depth of block 1 *x, y, z, width, height, depth of block 2*
x, y, z, width, height, depth of block 3 *x, y, z, width, height, depth of block 4*
x, y, z, width, height, depth of block 5 *…x, y, z, width, height, depth of block 16*

Genotype:

11010110	10101101	10101110	10011010	01101010	01101010	…	10001010	10001010	10001010
desired xpos 1	*desired ypos 1*	*desired zpos 1*	*desired width 1*	*desired height 1*	*desired height 1*	…	*desired width 4*	*desired height 4*	*desired depth 4*

Figure 1.23 Generative evolutionary design of a table.

and parameters – the form of designs is affected by interactions between different genes. More details of this type of representation and how it can be used in a generic evolutionary design system are provided in Chapter 18.

Because designs are generated 'from the bottom up', generative evolutionary design has yet to be used for the evolution of complex designs with moving parts. Nevertheless, this type of creative evolutionary design is capable of greater creativity than conceptual evolutionary design, as it uses only the fitness functions to provide guidance. It also overcomes potential limitations of 'conventional wisdom' and 'design fixation' by evolving forms without the use of knowledge of existing designs or design components.

The final section of this book explores this exciting type of creative evolutionary design. For example, Mike Rosenman describes the use of a GA to evolve house plans, using a design grammar of rules to define how polygons should be constructed out of edge vectors (Chapter 15). John Koza uses genetic programming (GP) to evolve novel analogue circuits (Chapter 16). John Gero attempts to evolve new higher-level representations of form, suitable for subsequent evolution of house plans in a specific architectural style (Chapter 15).

Simple evolutionary algorithms are often sufficient for the design systems that employ simpler representations (Baron et al., 1997). However, for creative evolutionary design systems with advanced representations and corresponding embryogenies (e.g. to 'grow' phenotypes from a set of shape-grammar rules defined in the genotypes), more advanced EAs are essential. Typically, these EAs are used to evolve the larger structure of the representation (e.g. number and organisation of shape-rules) in addition to the detail (e.g. type and content of the individual rules). In other words, these EAs are capable of evolving designs which have representations of variable length – they explore new and different search spaces in addition to the parameter values within each space (Bentley and Wakefield, 1996a). Typically, GAs and GP are used to evolve these highly variable forms (Frazer, 1995; Bentley and Wakefield 1997b; Coates, 1997), usually beginning from random, simple forms, and gradually improving the structure and detail of these designs until some functional criteria are met. The objective of this type of research is not normally to use computers to generate a single global optimal solution, but rather to generate a number of 'creative' alternatives. The evaluation of designs can be more difficult, since most off-the-shelf analysis packages are limited to judging specific types of designs – when presented with some of the initially random forms generated by these systems, they simply generate errors. Consequently, many of these systems rely on simplified custom-written evaluation routines, which can analyse everything presented to them, but perhaps not always with the desired accuracy (Bentley and Wakefield, 1997b).

However, this is one of the most recent and exciting types of evolutionary design, and is already showing great potential in a number of application areas. Chapter 18 describes the evolution, from scratch, of a number of unusual and inventive designs for numerous applications, such as tables, heatsinks, optical prisms, aerodynamic and hydrodynamic forms, etc., see fig. 1.24. In Chapter 17, Jordan Pollack demonstrates the use of a GA to evolve novel bridge and crane structures, which are subsequently built as LEGO™ models. An application area currently receiving much media attention is 'evolveable hardware', where new logic circuits are evolved and evaluated in real silicon using FPGAs. Evolution is also now being used to generate analogue circuits (described by John Koza in Chapter 16). Already some surprising and novel electronic solutions have been found using these techniques (Harvey and Thompson, 1997).

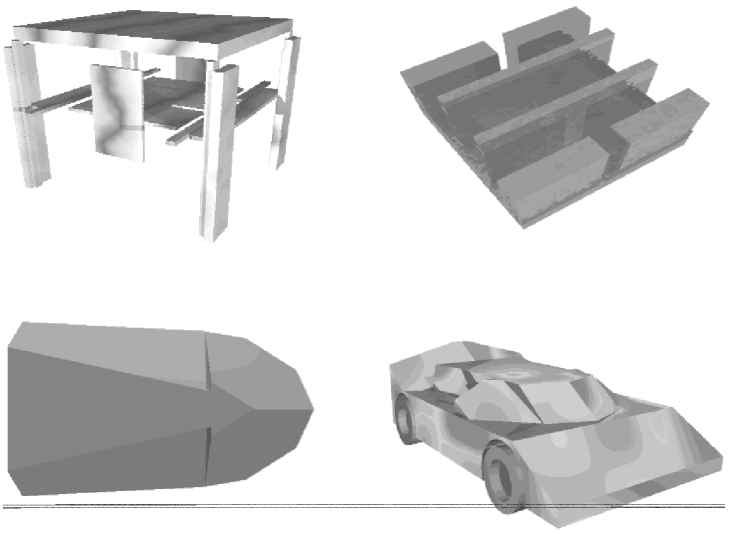


Figure 1.24 Examples of creative evolutionary design.

Evolutionary Art

Evolutionary art is perhaps the most commercially successful type of evolutionary design. Although academic research in this area is less common than in the other fields, there are more evolutionary art products available today than any other type of evolutionary design system (see Chapter 11 for a review).

Most evolutionary art systems tend to resemble each other closely. They all generate new forms or images from scratch (random initial populations). They rely completely upon a human evaluator to set fitnesses for each member of the population – normally based on aesthetic appeal. Population sizes are usually very small (often less than ten individuals), to allow them all to be quickly judged every generation. User-interfaces are often similar, with members of the current population shown on the screen in the form of a grid, allowing the user to rank them, or assign fitness scores by clicking on them with a mouse.

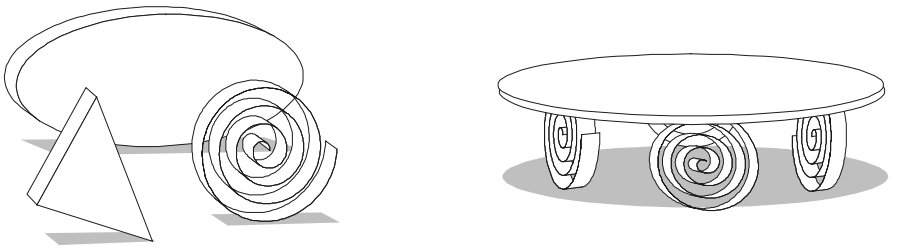
The main differences between these systems lie in their phenotype representations. A large variety of alternative representations have been employed, from fractal equations (such as John Mount's 'Interactive Genetic Art', which is shown on-line at http://www.geneticart.org/), to recursive grammar-rules using constructive solid geometry (Todd and Latham, 1992).

These representations are created with different intentions. For example, Dawkins' recursive tree-like structures were intended to resemble the recursive embryogenies found in nature, in the hope that natural looking forms would emerge (Dawkins, 1986, 1989). Todd and Latham's representation was based upon repeated elements such as spheres and tori, used to form 'horns' and 'ribs' out of which images are constructed (Todd and Latham, 1992). Colour and texture can also be incorporated into these representations (Dawkins, 1989; Sims, 1991; Todd and Latham, 1992).

Perhaps surprisingly, many of the representations are evolved with fixed structures (e.g. Todd and Latham (1992) hand-designed the structures of forms, then evolved the detail within these structures). Allowing evolution to vary structures (e.g. change the number of rules or primitive shapes), as is done in creative evolutionary design, could possibly increase the creativity of such systems.

Returning once again to the 'table' example, fig. 1.25 (left) shows the type of primitive shapes an evolutionary art system might use to represent forms. Figure 1.25 (right) shows an 'artistic table' generated from such shapes. As this example shows, it is quite common for the artist to employ a shape description language to specify the fixed structure of the designs to be evolved. In the example, each 'artistic table' must be constructed from an ellipse and a variable number of differently positioned and rotated swirls. This structure then defines how many genes will be evolved by the system, and how the values of the genes will be used to generate the phenotypes, i.e. the shape description language defines the genome and embryogeny for evolutionary art systems. By evolving the values of the genes, a number of unusual and hopefully aesthetically pleasing designs (some of which may behave as tables) will emerge.

Evolutionary art is an effective way of creating highly original and unusual pieces of art, but it is rarely used to generate anything as practical as a table. Since forms are not analysed



Phenotype:
Ellipse, *width 80, height 50, depth 4, rotated by 90 degrees*
Spiral, *radius 40, curliness 4, depth 6, shifted horizontally by 50, vertically by -20*
Spiral, *radius 40, curliness 4, depth 6, shifted horizontally by 50, vertically by -20, rotated 90 degrees*
Spiral, *radius 40, curliness 4, depth 6, shifted horizontally by 50, vertically by -20, rotated 180 degrees*
Spiral, *radius 40, curliness 4, depth 6, shifted horizontally by 50, vertically by -20, rotated 270 degrees*

Fixed structure (embryogeny)
using Shape Description Language:

```
Table =   { Ellipse ( width, height, depth )
            YZ_Rotate ( angle ) }
          { Leg
            X_Shift ( distance )
            Y_Shift ( distance )
            Rotate & Duplicate ( angle, #duplicates ) }
Leg =     Spiral ( radius, curliness, depth )
```

Genotype:

80	50	4	90	40	4	6	50	-20	90	4
width	*height*	*depth*	*angle*	*radius*	*curliness*	*depth*	*distance*	*distance*	*angle*	*#duplicates*

Figure 1.25 Evolving 'artistic tables'.

for their functionality (although users may be able to choose forms which appear more functionally valid than others), the output from evolutionary art systems is usually attractive, but non-functional.

One undesired side-effect of many of these representations is that they generate pieces of art which have very distinct styles. Often the style of form generated using a particular representation is more identifiable than the style of the artist used to guide the evolution. This can cause problems if the artist wishes to take the credit for the piece. The cause of this 'style problem' is perhaps due to the initial preconceptions and assumptions of the designer of the representation. By limiting the computer to a specific type of structure, or a specific set of primitive shapes and constructive rules, it will inevitably always generate forms with many common and identifiable elements.

Because evolution is guided by a human selector (i.e. the 'fitness function' is an artist), the evolutionary algorithm does not have to be complex. Evolution is used more as a continuous novelty generator, not as an optimiser. The artist is likely to score designs highly inconsistently as he/she changes his/her mind about desirable features during evolution, so the *continuous* generation of new forms based on the fittest from the previous generation is essential. Consequently, an important element of the EAs used is *non-convergence*. If the populations of forms were ever to loose diversity and converge onto a single shape, the artist would be unable to explore any further forms. Because of this, most evolutionary art systems do not employ crossover within their EAs. Typically only mutation is used, with all offspring being mutated copies of their parents (and often only a single parent is used per generation). This mutation-driven evolution is similar to the approach used in EP and ES, which are known to be excellent for finding solutions to problems with continuously changing fitness functions (Bäck, 1996).

Examples of such systems include Dawkins' biomorphs program (included on the CD-ROM) (Dawkins 1986, 1989), Todd and Latham's evolutionary art (described in Chapter 9), Rowbottom's Evolutionary Art (described in Chapter 11, and shown in fig. 1.26) and Sims' evolved computer graphics (Sims, 1991). Today, numerous evolutionary art systems are available on-line (see Chapter 11).

Evolutionary Artificial Life-forms

Evolutionary computation plays a significant role in many aspects of the new field of computer science known as artificial life (AL). Artificial 'brains', behaviour strategies, methods of communication, distributed problem solving and many other topics are commonly explored using genetic algorithms and other evolutionary search techniques (Cliff et al., 1994).

Although all types of evolved AL could be described as aspects of evolutionary design, it is clear that certain topics within AL fall into the 'evolutionary design' category more comfortably than others. For the purposes of this book, AL research that can be readily categorised as an aspect of evolutionary design will be defined as research which aims to evolve 'artificial life-forms'. Examples of evolutionary AL-forms include: Lohn's cellular automata (CA) evolved to be capable of self-replication (Lohn and Reggia, 1995), Harvey's evolved layout and structure of neurons (Harvey, 1997), and the evolved plant-like and animal-like morphologies of Dawkins (1986, 1989) and Sims (1994a,b)

Motivations for the creation of evolutionary AL-forms are usually theoretical. The goals of such research are often to discover more about the mechanisms of natural evolution, to find

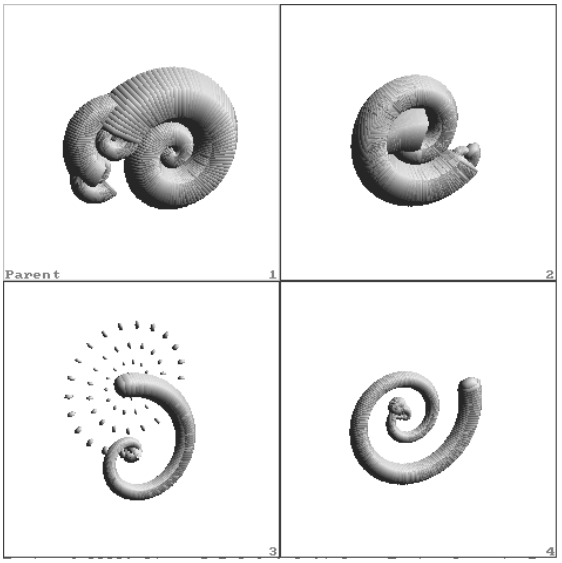
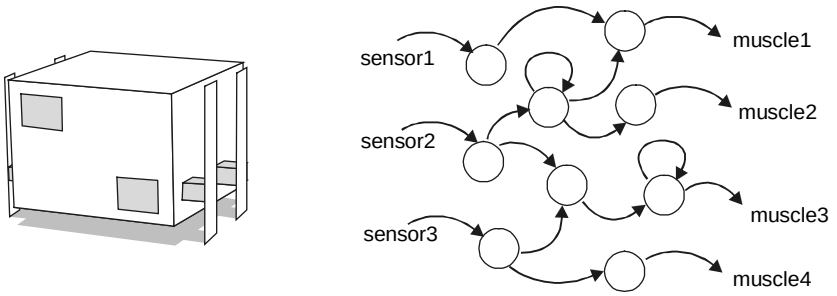


Figure 1.26 Rowbottom's evolutionary art.

explanations of forms observed in nature, or to exploit the solutions proven in nature by attempting to duplicate them. Often the evolved AL-forms show the enormous potentials of this type of evolutionary design, but as yet, practical applications are still scarce.

To illustrate this type of evolutionary design, we return to the 'table' example one final time. However, instead of a static table, we now require a robot/virtual creature/animat capable of carrying objects around – a robot waiter, perhaps. Figure 1.27 shows the dual nature of these designs: evolutionary AL-forms typically involve the evolution of the form (or some aspect of the form) and the brain. In this example, the form, or 'body' is defined by a collection of variable-sized blocks (which may be 'sensors', 'body' or 'muscles'). The 'brain' is defined by a network of neurons which receive input from 'sensory blocks' and produce output to the 'muscle blocks'. Each part of the phenotype is encoded as a variable-length chromosome in the genotype. The fitness function judges phenotypes on their ability to move whilst keeping the flat upper surface level. Over time, evolution will co-evolve both chromosomes in individuals to generate a virtual creature capable of supporting objects and movement in a virtual world.

Figure 1.27 shows just one example of how such animats can be represented. In reality, representations are typically specific to each system. For example, Lohn and Reggia (1995) use CA 'rule tables' within the chromosomes of their GA. Sims (1994a,b) uses a hierarchical chromosome structure to define both 'brain' and 'body'. Ventrella (1994) combines ordered morphology and control parameters of his animats in a flat chromosome structure, as does Harvey et al. (1993) for his evolved robots. Many of these representations are inspired by the genotype structure of natural organisms, and some researchers have attempted to evolve AL-forms with complex embryogenies. Other researchers invent their own intricate coding schemes (Cliff et al., 1994). Most such representations are highly flexible and of variable length, requiring complex genetic operators with the EAs.



Phenotype:

body

type, x, y, z, width, height, depth of block 1 *type, x, y, z, width, height, depth of block 2*
type, x, y, z, width, height, depth of block 3 *… type, x, y, z, width, height, depth of block 9*

brain

sensor1 excites neuron 1, weight 5 *neuron2 inhibits neuron 5, weight 0*
neuron1 excites neuron 6, weight 6 *sensor3 excites neuron 3, weight 8*
sensor2 excites neuron 2, weight 4 *… neuron 3 excites neuron 9, weight 4,*
neuron2 excites neuron 4, weight 3 *output of neuron 9 to muscle4*

Genotype:

Chromosome 1

11	11010110	10101101	10101110	10011010	01101010	01101010	…	10001010	10001010	10001010
type1	*xpos 1*	*ypos 1*	*zpos 1*	*width 1*	*height 1*	*depth 1*	…	*width 9*	*height 9*	*depth 9*

Chromosome 2

0000	10	1001	1	0110	1	…	1110
neuron1 marker	*neuron1 type (in/intl/out)*	*neuron1 link1*	*neuron1 link1 type (ex/inhib)*	*neuron1 link2*	*neuron1 link1 type (ex/inhib)*	…	*neuron9 link2*

Figure 1.27 Evolutionary artificial life form.

Because research in this field is still very much at the ‘blue-sky’ stage, evolutionary techniques are often used as exploration tools, in a similar way to evolutionary art. These algorithms can be used to generate multiple solutions, incorporating niching, speciation, parasitism, competition, co-operation and other advanced methods (Cliff et al., 1994). Evaluation usually consists of analysing behaviour in simulated virtual worlds (although some researchers do test solutions using real robots (Harvey, 1997)), and can be very time-consuming. To try to shorten evolution run-times, advanced methods are commonly used, e.g. steady-state GAs, parallel GAs, hybrid GAs (Cliff et al., 1994). Many systems that evolve AL-forms also use changing fitness functions, which necessitate the use of other specialised genetic search techniques (Harvey, 1997). Most systems evolve the forms from scratch (the initial population is random), however some occasionally seed initial populations with the fittest individuals from previous runs (Sims, 1994b). Figure 1.28 shows perhaps the most notable work in this field: Sims’ evolved virtual creatures (see Chapter 13 for full details).

1.4.2 Combining Good Ideas by Merging the Boundaries

Many researchers confine themselves to one of the four aspects of evolutionary design mentioned above, and seem loath to consider alternative approaches. However, more recently, some have begun to combine ideas from one or more of these areas in their work. This is

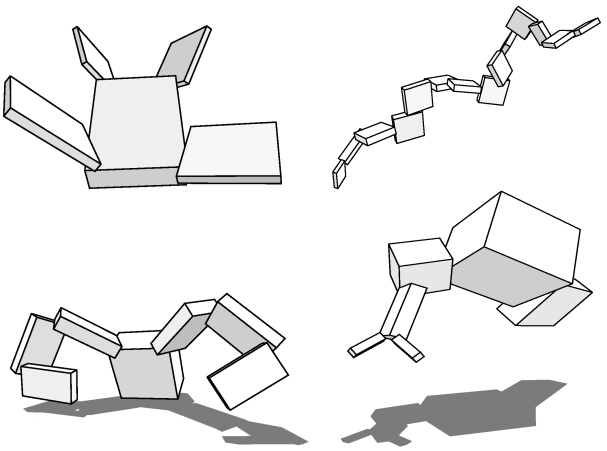


Figure 1.28 Sims' evolved artificial life forms.

leading to four more (still very new and relatively unexplored) 'overlapping' areas of research in evolutionary design (see fig. 1.19).

Integral Evolutionary Design

The evolution of engineering designs is becoming widespread today, with numerous academic engineering design centres exploring these ideas. Although most research seems to fall into either the evolutionary optimisation category, or the creative evolutionary design category, some work does attempt to combine the two into unified, or *integral* evolutionary design systems.

For example, Parmee suggests that computers can be used within the entire design process, both the early conceptual design stages and the later, detailed design stages, by using a number of adaptive techniques. He discusses how the use of several different systems, each dedicated to a specific stage of design could be used in combination, thus 'integrating adaptive search at every stage of the design process' (Parmee, 1996a). Ian Parmee gives more details of these ideas in Chapter 5.

Alternatively, Chapter 18 describes the investigation of the use of a single generic evolutionary design system, to perform the complete design process without making any distinction between the stages of design. (It is 'generic' because it is capable of evolving designs for multiple different design tasks.) This work has shown that it is possible to use a computer to evolve new designs from scratch, and optimise them, such that they fulfil specific functional criteria (Bentley and Wakefield, 1996b, 1997a,b).

With many optimisation systems beginning to be applied to more and more detailed parameterisations of designs, and many creative design systems beginning to be used to optimise the designs they generate, the field of integral evolutionary design looks set to grow rapidly.

Artificial Life-based Evolutionary Design

Work in artificial life has generated forms of astonishing diversity and creativity, so some researchers are now using some of the techniques from AL in their creative evolutionary design

systems, in an attempt to improve the quality and originality of evolved engineering designs. For example, Parmee (1996b) borrows distributed agent methods from AL to increase performance of search, in his 'ant colony' method, which he combines with evolutionary search. Coates (1997) employs L-systems with GP to evolve new architectural forms (described in Chapter 14).

Many other unexplored possibilities still exist in this area. For example, Bonabeau et al. (1994) describes an AL computer simulation of a swarm of artificial wasps, which build intricate three-dimensional nest architectures. One future avenue of research may be to evolve artificial wasps capable of building new engineering designs between them.

Aesthetic Evolutionary Artificial Life

The evolution of aesthetically pleasing AL was perhaps first performed by Dawkins (1986), who hand-selected his artificial 'biomorphs' for reproduction in exactly the way artists select their forms using evolutionary art systems. Ventrella (1994) has taken this one step further, and has evolved aesthetically pleasing animats which resemble animated stick-men. These are evolved for their ability to walk naturally in a virtual world, and evolution is also guided by the aesthetic judgement of the user. Alternatively, Lund et al. (1995) and Tabuada et al. (1998) describe neural networks which judge the aesthetics of evolving images.

Although to date there have been few applications to benefit from this area of research, the use of computers to evolve amusing or attractive animated characters may well be lucrative in the computer games industry, or for television advertisements.

Aesthetic Evolutionary Design

The evolution of aesthetic designs is an area of research with obvious importance, which should perhaps receive more attention than it does. Few designs are purely functional, most are chosen partly because of their aesthetics, and some functionally outstanding designs are discarded purely because of their ugly appearance. Furuta et al. (1995) describes an approach in which bridge designs can be optimised using GAs, based on their appearance, using 'psychovectors' to quantify the aesthetic factors of the structures. Frazer (1995) describes his substantial and pioneering research on the evolution of architectural forms, using a combination of formal analysis and aesthetic guidance from designers. Husbands et al. (1996) describes the evolution of 3D solid objects resembling propellers, using a superquadric shape-description language and guided by 'the eye of the beholder'.

1.4.3 Recommended Reading for Evolutionary Design

Advances in Design Optimization
by Hojjat Adeli (Ed) (1994).

Genetic Algorithms and Engineering Design
by Mitsuo Gen and Runwei Cheng (1997).

Artificial Intelligence in Design '94, '96 & '98
by John Gero and Fay Sudweeks (Eds) (1994, 1996, 1998).

Modeling Creativity and Knowledge-Based Creative Design
by John Gero and Mary Lou Maher (Eds) (1993).

An Evolutionary Architecture
by John Frazer (1995).

The Creative Mind: Myths & Mechanisms
by Margaret Boden (1992).

Evolutionary Art and Computers
by Stephen Todd and William Latham (1992).

The Blind Watchmaker
by Richard Dawkins (1986).

Climbing Mount Improbable
by Richard Dawkins (1996).

Artificial Life: an Overview
by Chris Langton (Ed) (1995).

Artificial Life: Grammatical Models
by Gheorghe Paun (Ed) (1995).

Modern Heuristic Search Methods
by Victor Rayward-Smith et al. (Eds) (1996).

Soft Computing in Engineering Design and Manufacturing
by P. K. Chawdhry, R. Roy and R. K. Pant (Eds) (1997).

Advances in Soft Computing – Engineering Design and Manufacturing
by R. Roy, T. Furuhashi and P. K. Chawdhry (Eds) (1998).

1.4.4 From Perusals to Problem Solving

In summary, this middle section of the chapter has described how computers are used to perform evolutionary design. The four main aspects: *evolutionary design optimisation*, *creative evolutionary design*, *evolutionary art*, and *evolutionary artificial life forms*, were surveyed in detail. In addition, the four 'overlapping' types of evolutionary design: *integral evolutionary design*, *aesthetic evolutionary design*, *artificial life-based evolutionary design*, and *aesthetic evolutionary AL* were introduced.

Having now explained both evolutionary computation and evolutionary design by computers, the final major section of this chapter discusses some of the significant technical issues faced by developers of evolutionary design systems.

1.5 Enumerations, Embryogenies and other Problems

There are a number of common problems encountered when attempting to perform evolutionary design using computers. As is usual when applying the techniques of evolutionary computation to anything, there are issues related to the fitness functions, such as noisy functions,

discontinuous functions, and multimodal functions (Bentley and Wakefield, 1996c; Parmee, 1996a). Ian Parmee discusses some of these in Chapter 5. There are, however, some more specific problems that typically arise more often with evolutionary design (Roston, 1997).

1.5.1 Enumerating the Search Space

Before an evolutionary algorithm can be applied to a problem, a suitable genotype and phenotype representation must be created. The genotype representation enumerates the search space of the problem, i.e. it defines which genotypes should be next to each other in the search space. The phenotype representation enumerates the solution space, i.e. it defines which phenotypes should be next to each other in the solution space. Both spaces must be carefully designed to ensure that the task of finding good solutions is not made any harder than it needs to be. The key thing to remember when developing these representations is that two genotypes which are *close* to each other in the search space should map onto two solutions which are *similar* to each other in the solution space. In other words, *a small change in the genotype should produce a small change in the phenotype*. (If the EA makes no distinction between genotypes and phenotypes, this equates to: *a small change in the value of any decision variable should produce a small change in the design*.)

To illustrate this concept, consider the problem of evolving a two-dimensional rectangle of specific size. Figures 1.29 and 1.30 show two ways in which the genotypes and corresponding phenotypes can be represented for this problem. In fig. 1.29, the genotype representation defines the path of a turtle using the three instructions: 'forwards' (F), turn right 'R', and 'turn left' (L). The rectangle phenotype is defined by the starting and ending positions of the turtle. It should be clear from the example that this is a very poor genotype representation – a small change in the genotype will often cause a big change in the phenotype. Alternatively, fig. 1.30 shows a genotype representation which defines the corners of a rectangle using angle and length parameters. This representation is far more conducive to search – a small change in the genotype will usually produce a small change in the phenotype.

The first genotype representation is worse than the second because its enumeration of the search space is very discontinuous: genotypes that map onto very dissimilar phenotypes are

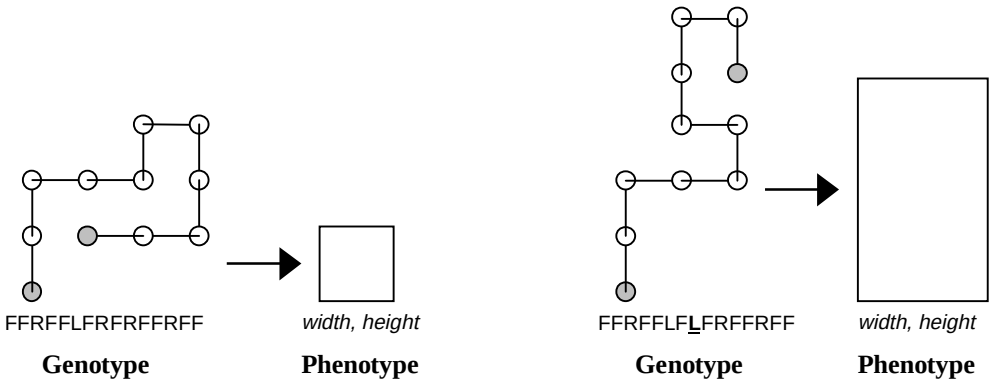


Figure 1.29 Representing rectangles using the start and end points of a turtle trail. Note how the smallest possible change in the genotype causes a very large change to the phenotype.

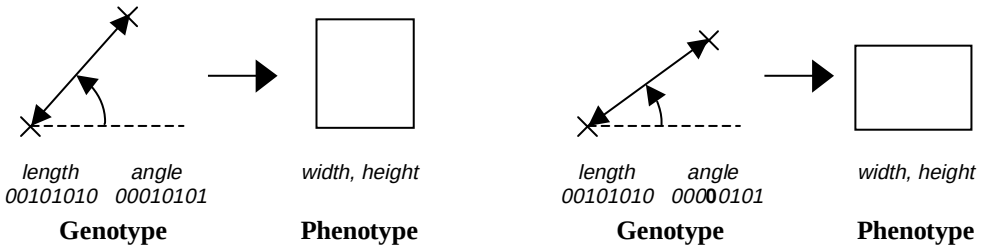


Figure 1.30 Representing rectangles using length and angle parameters. Note how a small change in the genotype causes a small change to the phenotype.

placed next to each other in the search space. Evolution relies on inheritance with a small degree of variation to ensure that most offspring resemble their parents and thus have similar fitnesses to their parents. If the search space is too discontinuous, then every application of crossover or mutation will generate offspring which hardly resemble their parents at all. This reduces the effectiveness of traversing from parent solution to child in the search space, and the search deteriorates into random exploration. As mentioned earlier in the chapter, this effect is caused by the genetic operators disrupting the parent solutions too much. It should now be clear that the level of disruption is often determined by the representations employed in the EA (a poor representation will be disrupted by all standard operators – and may require the creation of specially designed 'non-disruptive' operators).

Poorly designed representations are most problematical towards the end of an evolutionary run. As the population converges onto a small area of the search space, fine-tuning these solutions becomes nearly impossible if every minor change in the genotype causes a major change in the phenotype. Consequently, to ensure the successful evolution of good designs, all representations should be created with care.

Often, the hardest type of representation to design is a genotype representation which incorporates some form of *embryogeny* to 'grow' phenotypes from the genotypes. Such representations often use chains of rules which are epistatically linked – removing or altering one rule can radically alter the action of many others, resulting in major phenotypic changes.

1.5.2 Designing Embryogenies

As mentioned earlier, an embryogeny is an advanced form of mapping, from genotypes to phenotypes. Embryogenies have a number of advanced features:

- **Compression**. Because of their ability to allow simple genotypes to define complex phenotypes, many of these mappings resemble compression techniques, with genes performing more than one function during the development of the phenotype.
- **Repetition**. Properly designed embryogenies can improve the ability of evolution to generate solutions with repeating structures such as symmetry, segmentation, and subroutines.
- **Adaptation**. This is one of the most significant features of the use of embryogenies. It is possible to 'grow' phenotypes from genotypes adaptively, allowing constraints to be satisfied (Yu and Bentley, 1998), improvement to variable conditions, and correction of

malfunctions in designs (Sipper, 1997). Such adaptation seems likely to play significant roles in future applications of evolutionary design. For example, if the problem was to evolve a building which had good access to fire exits, various simulations modelling 'virtual people' trying to escape fires could be performed during the 'growth' of each building, thus ensuring that the final design was developed to maximise access (and satisfy the constraint).

Unfortunately, embryogenies can suffer from some drawbacks:

- **Hard to design**. All types of embryogeny require careful design, and to date, only those few researchers capable of performing this difficult art have demonstrated successful results.
- **Hard to evolve**. Many embryogenies introduce problems for evolutionary algorithms. Bloat, epistasis and excessive disruption of child solutions is common, resulting in the need for carefully designed genetic operators.

In nature, embryogenies are defined by the interactions between genes, their phenotypic effects and the environment in which the embryo develops. In evolutionary design by computers, we can define embryogenies in three main ways: *externally*, *explicitly*, and *implicitly*.

External (non-evolved) Embryogenies

Embryogenies are, in a very real sense, complex designs in their own right. Most embryogenies are hand-designed and are defined globally and externally to genotypes. For example, evolutionary optimisation systems usually use very simple, fixed, non-evolveable mapping procedures to specify how the genes in the genotype are mapped to the parameters in the phenotype. Evolutionary art systems often use more complex embryogenies defined by fixed, non-evolveable structures which specify how phenotypes should be constructed using the genes in the genotypes (see section 1.4.1, and Chapter 9). The advantage with such external embryogenies is that the user retains more control of the final evolved forms, and can potentially improve the quality of evolved designs by careful embryogeny design. In addition, this type of embryogeny produces the fewest harmful effects for evolution, and requires no specialised genetic operators. The disadvantage of this approach is that these embryogenies are not evolved, so they remain static and unchanging during the evolution of genotypes. This does not necessarily imply that the evolved designs will be any less fit, but it does mean that the designer of the embryogeny must take care to ensure that this complex mapping process will always perform the desired function. Figure 1.31 provides a simple example of an external embryogeny.

Explicit (evolved) Embryogenies

If each step of an embryogeny is explicitly specified in a data structure, the embryogeny resembles a computer program. Designs are 'grown' by following the instructions in this program, and these instructions may contain conditional statements, iteration, and even subroutines.

Although it is possible to hand-design such 'programs', genetic programming allows us to evolve them. Typically, the genotype and embryogeny are combined, allowing the evolution of both simultaneously. Clearly, this approach avoids the need to hand-design embryogenies, and

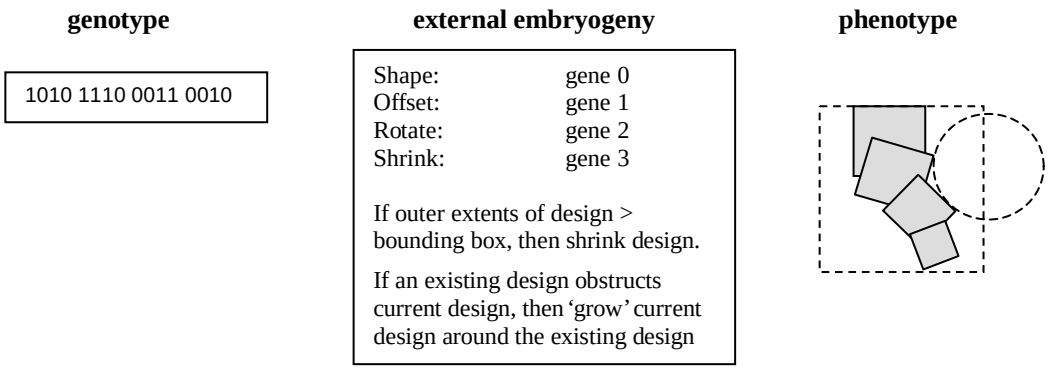


Figure 1.31 An example of an external embryogeny. Note how the embryogeny is adaptive, ensuring that the phenotype fits within a bounding box, and forcing the phenotype to 'grow' around a circular obstacle.

allows the emergence of adaptive mapping from genotype to phenotype (i.e., different initial conditions acting on conditional statements could trigger the growth of different phenotypes). There are some disadvantages, however. The creation of suitable representations can be difficult. Successfully evolving such representations can also be difficult (often specialised genetic operators are required to ensure disruption is minimised (Koza et al., 1998)). In addition, because the complete embryogeny process must be defined explicitly, advanced features such as iteration, subroutines and recursion must be manually added to the GP system – they cannot emerge spontaneously (see below).

Figure 1.32 gives a simple example of an explicit embryogeny. For a more detailed example of the use of explicit embryogenies, readers should consult Chapter 16, which summarises the use of David Andre's cellular encoding embryogeny to evolve analogue circuits using GP.

Implicit (evolved) Embryogenies

Natural evolution does not use externally defined embryogenies, nor does it explicitly represent embryogenies in our genes. Instead, natural evolution uses highly indirect chains of interacting 'rules' to generate complex embryogenies, which result in the development of living creatures. The flow of activation is not completely predetermined and preprogrammed, it is dynamic, parallel and adaptive.

To summarise in very simple terms, natural embryogenies use chemicals surrounding each cell to activate or suppress genes within the chromosomes of the cell, triggering patterns of cellular growth. Cellular death, differentiation, and the production of chemicals is also triggered by genes. Living creatures are grown in wombs or eggs with chemicals carefully placed to guide the early development of the embryo. As embryos develop, complex chains of gene activation occur, cells grow and die to form the appropriate shapes, and cells are differentiated to perform specialised functions. Even the movement of the developing muscles of the embryo affects the development and placement of cells.

Few researchers have explored the use of implicit embryogenies for evolutionary design, and yet the potential advantages of this approach are significant. Because of the way in which

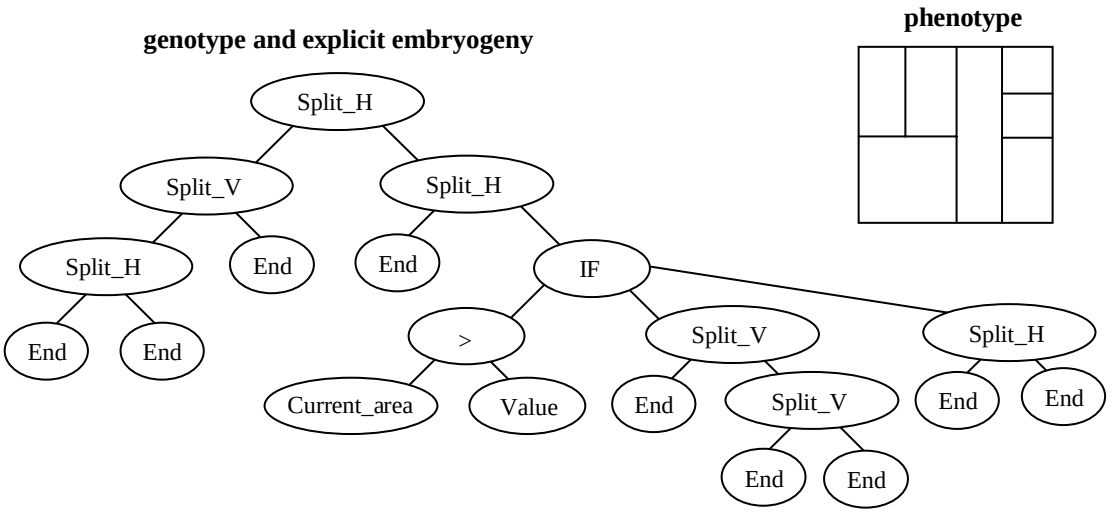


Figure 1.32 An example of an explicit embryogeny, incorporated into the genotype. Note how the embryogeny is adaptive, varying the phenotype depending on 'Value'. Also note that the same evolved embryogeny will generate different phenotypes if provided with a different initial starting shape.

genes can be activated and suppressed many times during the development of phenotypes, because the same genes can be used to specify multiple functions, and because of the inherent parallelism of gene activation, such implicit embryogenies go far beyond today's genetic programming. Through emergence during evolution, these implicit embryogenies incorporate all concepts of conditional iteration, subroutines, and parallel processing which must be manually introduced into explicit GP embryogenies. There is a serious disadvantage with the use of implicit embryogenies, however. Currently, the design of suitable genetic representations is proving prohibitively difficult, with very few useful designs having been evolved using this approach. Figure 1.33 shows a simple example of an implicit embryogeny. For some more detailed examples, readers should consult Chapter 12 by Hugo de Garis.

From Embryogeny to Ontogeny

Embryogeny defines the growth of a phenotype from zygote to new-born baby. Ontogeny defines the growth (as specified by its genes) throughout the life of the phenotype. In nature, our genes continue to affect our bodies and behaviour throughout our lives, defining when we become sexually mature, how we grow, which diseases we will be immune to, and even affecting aspects of our personalities and behaviour tendencies. To date there has been little research performed on evolving and growing adult designs from child designs (basing fitness on the life-time functionality of the design). Yet it is clear that our designs do often go through such growth and change. For example a mature 'adult' building can be very different from the 'child' building designed by an architect, as walls and facades are added and removed, roofs replaced, extensions added. If approximations of such phenotype growth were also incorporated into the genetic description (as an ontogeny) it might be possible to evolve designs

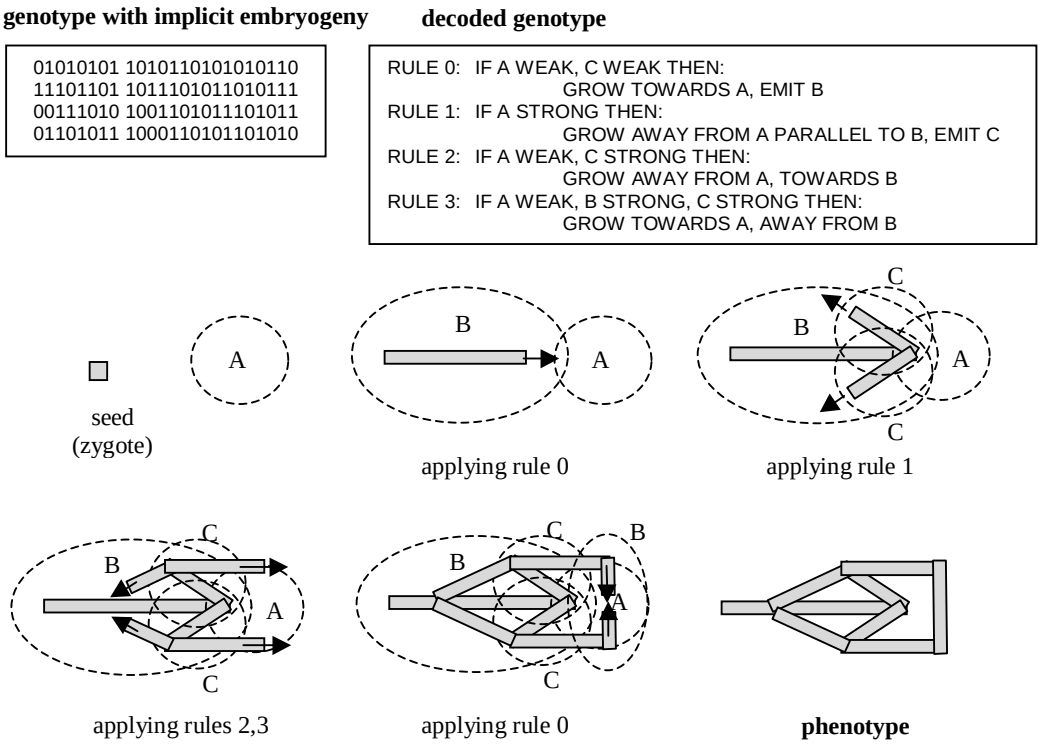


Figure 1.33 An example of an implicit embryogeny in a genotype. The decoded genotype provides an English description of the rules defined by the genes. Various components of the phenotype are grown towards and away from the chemicals A, B and C, by switching on and off rules. Note the reuse of rule 0.

which remain useful for many years, instead of simply evolving designs which satisfy certain criteria at 'birth'.

Our genes also control aspects of our bodies' repair mechanisms: continuously defining where cells should grow, and what type of cells they should be, to replace dying cells. By evolving designs which have self-repair mechanisms defined in their genes, the possibility of designs that can automatically correct faults within themselves becomes conceivable. Moshe Sipper (1997) is leading research in this area, by attempting to evolve electronic hardware capable of surviving 'injury'.

1.5.3 Epistasis

Epistasis means the 'degree of dependency' between multiple genes in a chromosome. Significantly, epistasis is all about *genes* acting in combination to produce *solutions*. Consequently, epistasis is defined by the genotype (and embryogeny) representation, and not by the fitness function.[7] A genetic representation with high epistasis may have many genes whose

[7] Epistasis causes confusion in the GP community, as practitioners of GP make no distinction between genotypes and phenotypes, so the epistatic properties of conditional statements in the evolved programs only

phenotypic effect relies to a large degree on the alleles of other genes. For example, a single shape-rule in a rule-based representation may have very different phenotypic effects, depending on which other shape-rules precede and succeed it. Conversely, a representation with low epistasis has few or no genes whose phenotypic effect relies on the alleles of other genes. For example, a simple voxel representation in which every gene switches on or off a single voxel in a grid has zero epistasis.

Experiments investigating whether epistasis should be high or low have so far been inconclusive (Schoenauer, 1996). However, a simple thought experiment can help explain this dilemma. Consider a (fictional) representation which uses, say, ten genes to represent the entire form of designs, and the phenotypic effect of every gene is *completely dependent* on all of the other genes through some embryogeny process. With this (maximum) amount of epistasis, these ten genes effectively become elements of a single overall gene. So our ten-gene representation with complete epistasis could be considered as a single-gene representation. Consider what effect varying any part of that gene would have on the phenotype. With every part of the design epistatically linked to every other part, any attempt to improve just one small area would result in changes to all of the rest of the design (*pleiotropy*) – making evolution to acceptable designs very difficult, if not impossible.

Alternatively, consider a representation with zero epistasis (e.g. a voxel representation using a 3D array). This requires no embryogeny, since every gene maps directly onto a specific area of the phenotype, and only that area of the phenotype. It should be clear that such a representation is well suited for evolution of small-scale detail, but evolution of large-scale characteristics becomes immensely difficult, e.g. the scaling of the entire form in one dimension, or the duplication or mirroring of an existing feature in the design. (To duplicate or mirror an existing feature in the way segmentation or symmetry does, each new duplicate part would have to be re-evolved in entirety – highly unlikely to occur (Bentley and Wakefield, 1996b).

Having examined the two extreme cases, it should be clear that both too much epistasis and too little epistasis in a representation is undesirable. Perhaps the ideal representation for evolutionary design should have a ‘middling’ amount of epistasis. However, it seems likely that the best tutor on this subject will be natural evolution, which seems to use varying degrees of epistasis in a single living creature, with recombination of DNA carefully controlled to avoid disruption, and the harmful effects of pleiotropy minimised (Altenberg, 1995).

1.5.4 Incorporating Knowledge in Evolutionary Design

Experience, insight and judgement make a good designer. Evolutionary design often relies only on the last of these attributes, for evolutionary algorithms are usually guided solely by the fitness function. In other words, design knowledge is provided in terms of an objective which must be met. As described in the previous section, this can be a significant strength of evolutionary design, allowing the generation of creative designs, art and artificial life by the

---

seem to appear during the evaluation of those programs. The confusion can be avoided if the hierarchical program structure is considered to be the genotype, and the *result of the action of the program as it runs* is considered to be the phenotype. This corresponds nicely with nature: we are the result of the action of our DNA – a continuously running program which builds and repairs us throughout our lives.

computer. However, the use of a single fitness function to guide the evolution of designs does prevent the computer from benefiting from the substantial knowledge of designers.

It is possible to add further knowledge to EAs by using more than one fitness function. In this way, multiple design objectives can be specified, including partially or fully contradictory objectives. These fitness functions need not define desired functionality – they can be used to compare evolving designs with a database of different good designs, allowing evolution to be guided by the case-based knowledge contained within each example design. Additional knowledge can also be used to constrain evolution, and prevent it from generating designs which are known to be unsatisfactory. (Handling multiple objectives and constraints in EAs is discussed later in this section.) There are also two other ways to provide additional knowledge to an evolutionary design system:

Knowledge-rich Representations

As described in the previous section, evolutionary optimisation uses EAs to optimise parameterised portions of existing designs. In other words, within the phenotype representation, the EA is provided with knowledge of the general form or structure of a rough design, which it then fine-tunes using the judgement provided by the fitness function. This is one way to incorporate knowledge into the representation of an EA. Another method is to provide a series of building blocks from which designs can be constructed. Conceptual evolutionary design uses this approach, allowing designers to use EAs to 'juggle' with their knowledge and find new ways of using it in combination. Alternatively, as John Gero illustrates in the first part of Chapter 15, it is possible to use evolution to *learn* representations which incorporate knowledge in the form of architectural styles, and then evolve new designs using such knowledge-rich representations.

Knowledge Seeding

Evolutionary algorithms are often initialised with random values, but this is not a prerequisite to evolutionary design. A common practice by some researchers is to seed the initial population with non-random values, i.e. give the EA some examples of good designs to work from. The advantage of this case-based design approach is the simplicity of introducing knowledge into the system. Unfortunately, the disadvantages are two-fold: firstly a pair of very fit, but very different parent designs provided by a designer may well generate nothing but unfit malformations because of the large differences in their structures. Secondly, if one example design is substantially fitter than the others provided by the designer, evolution will quickly seize upon it, and base almost all future generations on that single design – disregarding the knowledge contained within the other, less fit designs. Nevertheless, when performed with care, seeding populations with designs (whether they are designed by human or previously evolved) can provide a boost to the quality of designs evolved by computers (Bentley and Wakefield, 1997a).

1.5.5 Multiple Objectives

A substantial proportion of evolutionary design problems involve the evolution of solutions to problems with more than one criterion. More specifically, such problems consist of several separate objectives, with the required solution being one where some or all of these objectives are satisfied to a greater or lesser degree. Perhaps surprisingly then, despite the large numbers

of these multiobjective applications being tackled using EAs, only a small proportion of the literature explores exactly how they should be treated with EAs.

With single objective problems, the evolutionary algorithm stores a single fitness value for every solution in the current population of solutions. This value denotes how well its corresponding solution satisfies the objective of the problem. By allocating the fitter members of the population a higher chance of producing more offspring than the less fit members, the EA can create the next generation of (hopefully better) solutions. However, with multiobjective problems, every solution has a number of fitness values, one for each objective. This presents a problem in judging the overall fitness of the solutions. For example, one solution could have excellent fitness values for some objectives and poor values for other objectives, whilst another solution could have average fitness values for all of the objectives. The question arises: which of the two solutions is the fittest? This is a major problem, for if there is no clear way to compare the quality of different solutions, then there can be no clear way for the EA to allocate more offspring to the fitter solutions.

The approach most users of EAs favour to the problem of ranking such populations, is to weight and sum the separate fitness values in order to produce just a single fitness value for every solution, thus allowing the EA to determine which solutions are fittest as usual. However, as noted by Goldberg: '. . . there are times when several criteria are present simultaneously and it is not possible (or wise) to combine these into a single number'. (Goldberg, 1989). For example, the separate objectives may be difficult or impossible to manually weight because of unknowns in the problem. Additionally, weighting and summing could have a detrimental effect upon the evolution of acceptable solutions by the EA (just a single incorrect weight can cause convergence to an unacceptable solution). Moreover, some argue that to combine separate fitnesses in this way is akin to comparing completely different criteria; the question of whether a good apple is better than a good orange is meaningless.

The concept of Pareto optimality helps to overcome this problem of comparing solutions with multiple fitness values. A solution is Pareto optimal (i.e., Pareto minimal, in the Pareto optimal range, or on the Pareto front) if it is *not dominated* by any other solutions. As stated by Goldberg (1989):

> A vector $\mathbf{x}$ is partially less than $\mathbf{y}$, or $\mathbf{x} <_p \mathbf{y}$ when:
> $$(\mathbf{x} <_p \mathbf{y}) \Leftrightarrow (\forall_i)(\mathbf{x}_i <= \mathbf{y}_i) \wedge (\exists_i)(\mathbf{x}_i < \mathbf{y}_i)$$
> $\mathbf{x}$ *dominates* $\mathbf{y}$ iff $\mathbf{x} <_p \mathbf{y}$.

However, it is quite common for a large number of solutions to a problem to be Pareto optimal (and thus be given equal fitness scores). This may be beneficial should multiple solutions be required, but it can cause problems if a smaller number of solutions (or even just one) is desired.

For example, consider the multiobjective function (to be minimised):

$$\begin{aligned} f_1 &= (x + 50)^2 \\ f_2 &= (x - 50)^2 \qquad \text{where } -64 \leq x \leq 64. \end{aligned}$$

For this twin-objective function, the Pareto minimal solutions range from -50 to 50 – so almost every allowable value of x is Pareto optimal, see fig. 1.34. Although this is an extreme

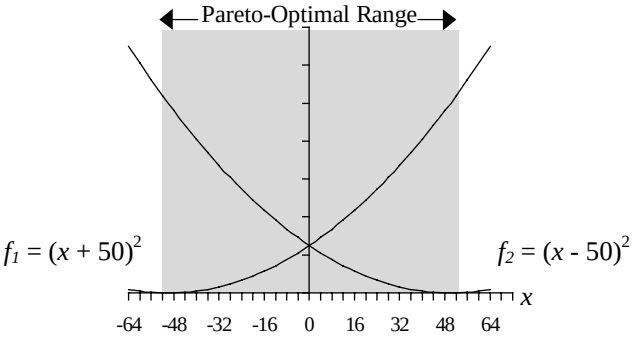


Figure 1.34 The Pareto optimal range of solutions for some multiobjective functions can include almost all allowable solutions to the problem.

case, it does illustrate a fundamental flaw with the concept of Pareto optimality: the Pareto front can be so large that it becomes infeasible to use the non-dominance of solutions as the sole fitness measure for solutions in an EA.

Hence, for many problems, the set of solutions deemed acceptable by a user will be a small subset of the set of Pareto optimal solutions to the problems (Fonseca and Fleming, 1995b). Manually choosing an acceptable solution can be a laborious task, which would be avoided if the EA could be directed by a ranking method to converge only on acceptable solutions (Bentley and Wakefield, 1997c).

So why do multiobjective problems cause such difficulties for EAs? Fundamentally, successful multiobjective optimisation is all about *range-independence*.

Range-Independence

Throughout the evolution by the EA, every separate objective (fitness) function in a multiobjective problem will return values within a particular range. Although this range may be infinite in theory, in practice the range of values will be finite. This 'effective range' of every objective function is determined not only by the function itself, but also by the domain of input values that are produced by the EA during evolution. These values are the parameters to be evolved by the EA and their exact values are normally determined initially by random, and subsequently by evolution. The values are usually limited still further by the coding used, for example 16 bit sign-magnitude binary notation per gene only permits values from −32768 to 32768.

Although occasionally the effective range of all of the objective functions will be the same, in most more complex multiobjective tasks, every separate objective function will have a different effective range (i.e., the function ranges are non-commensurable (Schaffer, 1985)). This means that a bad value for one could be a reasonable or even good value for another, see fig. 1.35. If the results from these two objective functions were simply added to produce a single fitness value for the EA, the function with the largest range would dominate evolution (a poor input value for the objective with the larger range makes the overall value much worse than a poor value for the objective with the smaller range).

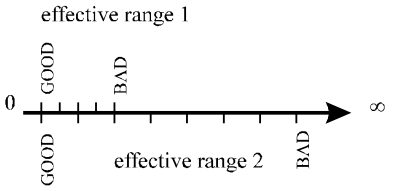


Figure 1.35 Different effective ranges for different objective functions (to be minimised).

For example, consider the two objective functions:

$f_{11} = x^2$

$f_{12} = (x - 2)^2 / 1000$

(both to be minimised).

Given a non-optimal input value, the output value from f_{11} will normally be three orders of magnitude worse than that from f_{12} (i.e., the second function will be approximately one thousand times closer to the minimum of zero). As can be seen in the simplest of tests, if the outputs from both were simply summed, the first function would completely dominate the second, resulting in the effective evolution of a good solution only to the first function.

Thus, the only way to ensure that all objectives in a multiobjective problem are treated equally by the EA is to ensure that all the effective ranges of the objective functions are the same (i.e., to make all the objective functions commensurable), or alternatively, to ensure that no objective is directly compared to another. In other words, either the effective ranges must be converted to make them equal, and a range-dependent ranking method used, or a range-independent ranking method must be used (Bentley and Wakefield, 1997c). Typically, range-dependent methods (e.g., 'sum of weighted objectives', 'distance functions', and 'min-max formulation') require knowledge of the problem being searched to allow the searching algorithm to find useful solutions (Srinivas and Deb, 1995). Range-independent methods require no such knowledge, for being independent of the effective range of each objective function makes them independent of the nature of the objectives and overall problem itself. Hence, a ranking method should not just be independent of individual applications (i.e., problem independent), as stated by Srinivas and Deb (1995), it should be independent of the effective ranges of the objectives in individual applications (i.e., range-independent).

For example, the standard 'sum of weighted objectives' method favoured by so many, uses the weights to make the effective domains of each objective equal, then provides a single fitness value by summing the resulting values. This is a range-dependent method, for it relies completely on the weights being set precisely for every problem. Should any of the objectives be changed, or the allowable domain of input values be changed (perhaps by a change in coding, or seeding the initial population with anything other than random values), then these weights may have to be changed.

Alternatively, the non-dominated sorting method, and variants of it, is a range-independent method. It requires no weighting of the objective values, for the fitness values from each objective function are never directly compared with each other. Only values from the same objective are ever compared in the process of determining the non-dominance of solutions (Goldberg, 1989). For complex multiobjective problems, this range-independence is extremely

advantageous: good results do not depend on the ability of the user to fine-tune weights correctly. However, a disadvantage of non-dominated sorting is that all Pareto optimal solutions are considered equally good, regardless of what the user actually regards as being acceptable.

For a detailed review, analysis and investigation of six different multiobjective handling methods for GAs, see Bentley and Wakefield (1997c).

1.5.6 Constraints

Constraints form an integral part of every problem, and yet they are often overlooked in evolutionary algorithms (Michalewicz, 1995, 1996). It is vital to perform constraint handling with care, for if evolutionary search is restricted inappropriately, the evolution of good solutions may be prevented.

A problem with constraints has both an objective, and a set of restrictions. For example, when designing a VLSI circuit, the objective may be to maximize speed and the constraint may be to use no more than 50 logic gates. When writing a computer program, the objective is to generate a program which performs a specific task and a constraint is not to violate the syntax of the language. A good problem solution must both fulfil the objective and satisfy these restrictions.

In the same way that phenotypes are evaluated for fitness, not genotypes, it is the phenotypes which must satisfy the problem constraints, not the genotypes (although their enforcement may result in the restriction of some genotypes[8]). However, unlike the fitness evaluation, constraints can be enforced at any point in the algorithm to attain legal phenotypes. As is described by Yu and Bentley (1998), they may be incorporated into the genotype or phenotype representations, during the seeding of the population, during reproduction, or handled at other stages.

There are two main types of constraint: the *soft constraint* and the *hard constraint*. Soft constraints are restrictions on phenotypes that should be satisfied, but will not always be. Such constraints are often enforced by using penalty values to lower fitnesses. Illegal phenotypes (which conflict the constraints) are permitted to exist as second-class, in the hope that some portions of their genotypes will aid the search for fit phenotypes (Michalewicz, 1995). Hard constraints, on the other hand, must always be satisfied. Illegal phenotypes are not permitted to exist (although their corresponding genotypes may be, if the constraints are enforced in the mapping stage).

Just as evolution requires selection pressure to generate phenotypes that satisfy the objective function, evolution can have a second selection pressure placed upon it in order to generate phenotypes that do not conflict the constraints. However, using *pressure* in evolutionary search to evolve legal solutions is no guarantee that all of the solutions will always be legal (i.e., they are soft constraints).

Constraints can also be handled in two other ways: solutions that do not satisfy the constraints can be *prevented* from being created, or they can be *corrected*. Such methods can have significant drawbacks such as loss of diversity and premature convergence. Nevertheless, these

[8] There are other types of constraints, e.g. minimisation of the number of evaluations, which may be applied to the evolutionary algorithm as a whole. This section focuses on evolving solutions which satisfy constraints, not on handling constrained evolutionary algorithms.

Table 1.2 Classification of constraint handling.

Prevention	HARD
Correction	HARD
Pressure	SOFT

two types of constraint handling ensure that all solutions are always legal (i.e., they are hard constraints). Table 1.2 shows the three conceptual categories for constraint handling.

As is typical in evolutionary computation, researchers typically investigate constraint handling from the perspective of a single evolutionary algorithm. For a general analysis and investigation of constraint handling in evolutionary algorithms, see Yu and Bentley (1998).

1.5.7 Evolving Designs with Interdependent Elements

Opponents of Darwin's theory of natural selection often give the example of the eye as a structure 'impossible to occur by evolution'. They state that the eye consists of many interdependent parts: the iris, the retina, the cornea, the lens, with each element relying on the correct functioning of all the other elements for the eye to work as a whole. Dawkins (1986) convincingly argues that there does exist a series of gradual evolutionary steps from *no eye* to *eye*, and that not only has the eye evolved, but it has evolved many times independently in different species. However, although it is clear that such intricate designs have been evolved in nature, it is also clear that using evolutionary computation to generate designs with interdependent elements is a very difficult task.

For example, Bentley and Wakefield (1997a) describe (amongst other things) the evolution of a Penta prism. This design must bounce light twice using total internal reflection within its solid glass structure in order to reflect an image through 90 degrees whilst keeping the output image the right way up, see fig. 1.36. Although it is a single component, it does have two interdependent elements: the two reflective parts. If either of these internally reflective sides are imprecisely oriented, or omitted, the design will not function correctly. Not only that, but the first reflection must direct the light in a direction almost opposite to the final, desired direction: so a design without the second reflective part is actually worse than a design with no reflective parts at all (Bentley and Wakefield, 1997a).

Evolution 'prefers' to begin with a single functional element that performs the function to some extent, however small, then slowly and incrementally build up the complexity of the

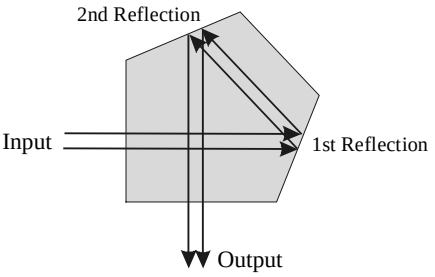


Figure 1.36 A Penta prism uses two interdependent internal reflections.

design, adding new elements *if they improve the fitness of the design* (Dawkins, 1989). However, if care is not taken in the design of the representation and operators, evolution may commit itself too early to simple approximations of the desired design. In the example of the Penta prism, evolution normally began by evolving a simple right-angle prism (using a single reflection), which directed the light in the correct direction, but oriented the wrong way up, fig. 1.37. Having committed itself to this simple, but unsatisfactory type of solution, evolution was then unable to proceed to the more complex Penta-prism design. Perhaps because of limitations of the representation, the only way that evolution could be forced to abandon this unsatisfactory local optimum was by penalising all such designs with a fitness constraint (Bentley and Wakefield, 1997a).

Such examples of evolutionary design illustrate that design problems requiring solutions with many interdependent elements can have large numbers of local optima – some of which do not resemble the functionally correct designs at all. Penalising all such unsatisfactory designs is not always a practical solution, so for design problems of this type, very careful consideration should be given to the creation of a representation and genetic operators that will always permit evolutionary paths from local optima to global optima.

1.5.8 Evolving Structure and Detail

As described in previous sections, some types of evolutionary design allow evolution to explore the *structure* (e.g. number of parameters/rules/primitive shapes) of designs in addition to the *detail* (e.g. parameter values) of designs. In most representations, varying the structure of designs has considerably more impact on the fitness of these designs, compared to varying the detail. Because of this, evolution typically converges on suitable design structures long before converging on design details. (This effect is also evident in the convergence of most significant bits before least significant bits in a binary coded genotype.) Although this is not always a disadvantage (e.g. the skeletal structure of all mammals seems to permit sufficient diversity, as does the cellular structure of plants), in some instances evolution may prematurely converge onto an inappropriate structure. If this happens, the evolution of detail around this structure may be insufficient to allow the production of an acceptable design.

There are two potential solutions to this problem. First, an improved representation capable of allowing changes in structure without disastrous fitness changes would allow structure and detail to converge simultaneously. This could be accomplished if the evolution of detail

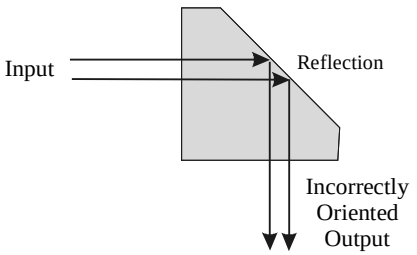


Figure 1.37 A right-angle prism is an easily found solution which does a similar job to the Penta prism, but there is no easy evolutionary path from a right-angle prism to a Penta prism.

could directly affect the degree to which changes in structure affected fitnesses. For example, in nature, a superfluous bone will be gradually reduced in size by evolution (a change in design detail), whilst at the same time improving other aspects of the organism because of that change. This reduction continues until at some point the bone becomes sufficiently small and redundant that evolution can remove the entire bone (structure mutation), without decreasing the fitness of the organism.

The second way to prevent premature convergence of structure is to reduce selection pressure once in a while, and permit less-fit designs to propagate. Dawkins (1989) suggests that certain landmark mutations in structure such as the development of segmentation may have initially resulted in solutions with worse fitnesses. It seems likely that some of the more gross mutations are more likely to survive during 'good times' where food is plentiful, predators are scarce, and hence selection pressure low (Dawkins, 1989). By reducing selection pressure in our evolutionary design systems in a similar way, we may well permit changes in structure to occur at later stages of evolution.

1.5.9 Summary

The final major section of this chapter has explained eight significant issues which prospective evolutionary designers should consider: enumerating the search space, designing embryogenies, epistasis, incorporating knowledge, handling multiple objectives, handling constraints, interdependent elements in designs, and the evolution of structure and detail. Not every evolutionary design system is affected by all of these issues, but most will be affected by some. More solutions to these and other problems are discussed in the other chapters of this book.

1.6 Summary of Chapter

This introductory chapter has attempted to explain the fundamental issues surrounding evolutionary design by computers. The technical jargon and equations have deliberately been minimised in the chapter in an attempt to provide a gentle introduction and an intuitive *feel* of evolutionary design. Beginning with a justification of why evolution is used to generate designs, the chapter then explored the techniques and ideas of evolutionary computation, reviewed the different types of evolutionary design, and discussed the problems faced by creators of evolutionary design systems.

The stress throughout has been the promotion of *understanding*. This chapter does not provide a set of instructions for, or results of, performing evolutionary design. It provides instead a series of explanations of how evolutionary design can be performed, in the hope that you, the reader, will create your own original evolutionary design system. And remember: this is still a new and rapidly growing field. There are no hard and fast rules which must always be followed. As the following chapters of this book show: the results of evolutionary design are limited only by our imagination – and the unlimited imagination of evolution.

1.7 Organisation of the Book

The book has five major sections, intended to cover the major aspects of evolutionary design by computers.

The first section, **Evolution and Design**, describes and explores the relationships between the design process and evolution. The four chapters discuss whether our own method of design bears any resemblance to evolution, whether insight and creativity can be achieved by evolution, and how evolution should be used in the design process.

The second section, **Evolutionary Optimisation of Designs**, provides examples of design optimisation using genetic algorithms. The three chapters describe detailed investigations of how satellite booms, load cells, flywheels and reliable networks can be optimised using evolution.

The artistic side of evolutionary design is shown by the third section, **Evolutionary Art**. The three chapters in this section give very graphic illustrations of the artistic capabilities and creativity of evolutionary design.

The fourth section of the book, **Evolutionary Artificial Life Forms**, explores the exciting use of evolutionary design to generate astonishing virtual creatures, and investigates the use of biologically inspired embryogenies and other techniques to evolve forms.

The fifth and final section, **Creative Evolutionary Design**, examines the creative potential of evolution to generate novel and useful designs. The chapters in this section show the remarkable diversity of original designs that are now being evolved, and give clues to the future of evolutionary design.

The books ends with a glossary of commonly used terms.

Acknowledgements

Thanks to Tina Yu, who provided help, time, and some of the text for the GP, ES and EP summaries. Thanks to Suran Goonatilake and Phil Treleaven for their advice, and to David Fogel, Laura Decker and Tina Yu for proof-reading sections of this monstrosity of a chapter. Thanks to Jonathan Wakefield, Sanjeev Kumar and all the members of UCL's Design Group for providing stimulating discussions and helpful criticism. Thanks also to Ying Li for the idea of including 'recommended reading' sections, and to Jim Viner for the title of section 1.4.4. Portions of the middle section of this chapter have appeared as the chapter 'Aspects of Evolutionary Design by Computers' in *Advances in Soft Computing – Engineering Design and Manufacturing*, Springer-Verlag, London, 1998, reprinted with permission.

References

Adeli, H. (ed.) (1994). *Advances in Design Optimization*, Chapman and Hall, London.

Adeli, H. and Cheng, N. (1994). Concurrent Genetic Algorithms for Optimization of Large Structures. *ASCE Journal of Aerospace Engineering* **7:3**, 276–296.

Alternberg, L. (1995). Genome Growth and the Evolution of the Genotype-Phenotype Map. In *Evolution and Biocomputation: Computational Models of Evolution*. Springer-Verlag, pp. 205–259.

Angeline, P. and Kinnear Jr., K. E. (eds) (1996). *Advances in Genetic Programming 2*. MIT Press, Cambridge, MA.

Axelrod, R. (1987). The Evolution of Strategies in the Iterated Prisoner's Dilemma. In Davis, L. (ed.), *Genetic Algorithms and Simulated Annealing*, Pitman, London, pp. 32–41.

Bäck, T. (1996). *Evolutionary Algorithms in Theory and Practice*. Oxford University Press, New York.

Banzhaf, W. (1994). Genotype-Phenotype-Mapping and Neutral Variation – A Case Study in Genetic Programming. In Davidor, Y., Schwefel, H.-P. and Mnner, R. (eds), *Parallel Problem Solving From Nature*, 3. Springer-Verlag, pp. 322–332.

Banzhaf, W., Nordin, P., Keller, R. E. and Francone, F. D. (1998). *Genetic Programming – an Introduction*. Morgan Kaufmann Publishers, San Francisco.

Baron, P., Fisher, R., Mill, F., Sherlock, A. and Tuson, A. (1997). A Voxel-based Representation for the Evolutionary Shape Optimisation of a Simplified Beam: A Case-Study of a Problem-Centred Approach to Genetic Operator Design. *2nd On-line World Conference on Soft Computing in Engineering Design and Manufacturing (WSC2)*.

Bentley, P. J. (1997). The Revolution of Evolution for Real-World Applications. *Emerging Technologies '97: Theory and Application of Evolutionary Computation*, 15th December, University College London.

Bentley, P. J. (1998a). Aspects of Evolutionary Design by Computers. In *Proceedings of the 3rd On-line World Conference on Soft Computing in Engineering Design and Manufacturing (WSC3)*.

Bentley, P. J. (ed.) (1998b). *Proc. of the Workshop on Evolutionary Design, 5th International Conference on Artificial Intelligence in Design '98*, Instituto Superior Técnico, Lisbon, Portugal, 20–23 July 1998.

Bentley, P. J. and Wakefield, J. P. (1996a). Generic Representation of Solid Geometry for Genetic Search. *Microcomputers in Civil Engineering* 11:3, Blackwell Publishers, 153–161.

Bentley, P. J. and Wakefield, J. P. (1996b). The Evolution of Solid Object Designs using Genetic Algorithms. In Rayward-Smith, V. (ed.), *Modern Heuristic Search Methods*, Ch. 12, John Wiley and Sons Inc., pp. 199–215.

Bentley, P. J. and Wakefield, J. P. (1996c). Hierarchical Crossover in Genetic Algorithms. In *Proceedings of the 1st On-line Workshop on Soft Computing (WSC1)*, Nagoya University, Japan, pp. 37–42.

Bentley, P. J. and Wakefield, J. P. (1997a). Conceptual Evolutionary Design by Genetic Algorithms. *Engineering Design and Automation Journal* 3:2, John Wiley and Sons, Inc, 119–131.

Bentley, P. J. and Wakefield, J. P. (1997b). Generic Evolutionary Design. In Chawdhry, P. K., Roy, R. and Pant, R. K. (eds), *Soft Computing in Engineering Design and Manufacturing*, Springer Verlag, London, Part 6, pp. 289–298.

Bentley, P. J. and Wakefield, J. P. (1997c). Finding Acceptable Solutions in the Pareto-Optimal Range using Multiobjective Genetic Algorithms. In Chawdhry, P. K., Roy, R. and Pant, R. K. (eds), *Soft Computing in Engineering Design and Manufacturing*. Springer Verlag, London, Part 5, pp. 231–240.

Boden, M. A. (1992). *The Creative Mind: Myths and Mechanisms*. Basic Books.

Bonabeau, E., Theraulaz, G., Arpin, E. and Sardet, E. (1994). The Building Behaviour of Lattice Swarms. In Brooks, R. and Maes, P. (eds), *Artificial Live IV*, Proc. of the 4th Int. Workshop on the Synthesis and Simulation of Living Systems, MIT Press, Cambridge, MA, pp. 307–312.

Born, J. (1978). *Evolutionstrategien zur Numerischen Lösung von Adaptationsafgaben*. Dissertation A, Humboldt-Universität, Berlin.

Canal, E., Krasnogor, N., Marcos, D. H., Pelta, D. and Risi, W. A. (1998). Encoding and Crossover Mismatch in a Molecular Design Problem. In Bentley, P. J. (ed.), *Proceedings of the AID '98 Workshop on Evolutionary Design*, July 19, 1998, Lisbon, Portugal.

Chambers, L (1995). *Practical Handbook of Genetic Algorithms*, CRC Press, Boca Raton.

Chawdhry, P. K., Roy, R. and Pant, R. K. (eds) (1997). *Soft Computing in Engineering Design and Manufacturing*. Springer-Verlag, London.

Chellapilla, K. (1997). Evolutionary Programming with Tree Mutations: Evolving Computer Programs without Crossover, In Koza, J., Kalyanmoy, D., Marco, D., Fogel, D., Garzon, M., Hitoshi, I. and Rick, R. (eds), *Genetic Programming 1997: Proceedings of the Second Annual Conference*. Morgan Kaufmann, San Francisco, CA.

Cliff, C., Husbands, P., Meyer, J. and Wilson, S. W. (eds) (1994). *From Animals to Animats 3. Proceedings of the Third International Conference on Simulation of Adaptive Behaviour*, MIT Press, Cambridge, MA.

Coates, P. (1997). Using Genetic Programming and L-Systems to explore 3D design worlds. In Junge, R. (ed.), *CAADFutures '97*, Kluwer Academic Publishers, Munich.

Dasgupta, D. and McGregor, D. R. (1992). Nonstationary Function Optimization using the Structured Genetic Algorithm. In *Parallel Problem Solving from Nature 2*, Elsevier Science Pub., Brussels, pp. 145–154.

Davis, L. (1991). *The Handbook of Genetic Algorithms*. Van Nostrand Reinhold, New York.

Dawkins, R. (1976). *The Selfish Gene*. Oxford University Press.

Dawkins, R. (1982). *The Extended Phenotype*. Oxford University Press.

Dawkins, R. (1983). Universal Darwinism. In Bendall, D. (ed.), *Evolution from Molecules to Men*. Cambridge University Press.

Dawkins, R. (1986). *The Blind Watchmaker*. Longman Scientific and Technical, Harlow.

Dawkins, R. (1989). The Evolution of Evolvability. In Langton, C. G. (ed.), *Artificial Life. The Proceedings of an Interdisciplinary Workshop on the Synthesis and Simulation of Living Systems*, vol. VI, September, 1987, Los Alamos, New Mexico. Addison-Wesley Pub. Corp, pp. 201–220.

Dawkins, R. (1996). *Climbing Mount Improbable*. Penguin Books, Harmondsworth.

De Jong, K. A. (1975). *An Analysis of the Behaviour of a Class of Genetic Adaptive Systems*. Doctoral dissertation, University of Michigan, Dissertation Abstracts International.

Deb, K. (1991). *Binary and Floating Point Function Optimization using Messy Genetic Algorithms*. Illinois Genetic Algorithms Laboratory (IlliGAL), report no. 91004.

Deb, K. and Goldberg, D. E. (1991). *mGA in C: A Messy Genetic Algorithm in C*. Illinois Genetic Algorithms Laboratory (IlliGAL), report no. 91008.

Deb, K. and Goldberg, D. E. (1993). Analyzing Deception in Trap Functions. In *Foundations of Genetic Algorithms 2*, Morgan Kaufmann Pub.

Deb, K., Horn J. and Goldberg, D. E. (1993). Multimodal Deceptive Functions. *Complex Systems* 7:2, 131–153.

Eby, D., Averill, R., Gelfand, B., Punch, W., Mathews, O. and Goodman, E. (1997). An Injection Island GA for Flywheel Design Optimization. In *5th European Congress on Intelligent Techniques and Soft Computing EUFIT '97*, vol. 1, Verlag Mainz, Aachen, pp. 687–691.

Fleming, P., Zalzala, A., Bull, D., Fonseca, C. and Patton, R. (1995). *Proceedings of the Genetic Algorithms in Engineering Systems: Innovations and Applications (GALESIA '95)*, Sept. 1995, Sheffield. IEE, London.

Fogel, D. B. (1991). *System Identification through Simulated Evolution: A Machine Learning Approach to Modeling*. Ginn Press, Needham Heights.

Fogel, D. (1992a). Evolving Artificial Intelligence. PhD thesis, University of California, San Diego, CA.

Fogel, D. (1992b). An Analysis of Evolutionary Programming. *Proc. of the 1st Annual Conf on Evolutionary Programming*, San Diego. Evolutionary Programming Society, San Diego, CA, pp. 43–51.

Fogel, D. B. (1994). Asymptotic Convergence Properties of Genetic Algorithms and Evolutionary Programming: Analysis and Experiments. *J. of Cybernetics and Systems* **25**, Taylor and Francis Pub., 389–407.

Fogel, D. B. (1995). *Evolutionary Computation: Towards a New Philosophy of Machine Intelligence*. IEEE Press.

Fogel, D. B. (1997). The Advantages of Evolutionary Computation. *Biocomputing and Emergent Computation (BCEC97)*.

Fogel, G. B. and Fogel, D. B. (1995). Continuous Evolutionary Programming: Analysis and Experiments. *J. of Cybernetics and Systems* **26**, Taylor and Francis Pub., 79–90.

Fogel, L. J. (1963). *Biotechnology: Concepts and Applications*. Prentice Hall, Englewood Cliffs, NJ.

Fogel, L. J., Owens, A. J. and Walsh, M. J. (1966). *Artificial Intelligence through Simulated Evolution*. Wiley, New York.

Fogel, L., Angeline, P. and Fogel, D. (1995) An Evolutionary Programming Approach to Self-adaptation on Finite State Machines.In J.R. McDonnell, R.G. Reynolds and D.B. Fogel (eds), *Proceedings of the Fourth International Conference on Evolutionary Programming*, MIT Press, Cambridge, MA.

Fonseca, C. M. and Fleming, P. J. (1995a). An Overview of Evolutionary Algorithms in Multiobjective Optimization. *Evolutionary Computation* 3:1, 1–16.

Fonseca, C. M. and Fleming, P. J. (1995b). Multiobjective Genetic Algorithms Made Easy: Selection, Sharing and Mating Restriction. *Genetic Algorithms in Engineering Systems: Innovations and Applications*, Sheffield. IEE, London, 45–52.

Forrest, S., Perelson, A. S., Allen, L. and Cherukuri, R. (1995). A Change-Detection Algorithm Inspired by the Immune System. Submitted to *IEEE Transactions on Software Engineering*.

Foy, M. D. et al. (1992). Signal Timing Determination using Genetic Algorithms. *Transportation Research Record #1365*, National Academy Press, Washington, DC, 108–113.

Frazer, J. (1995). *An Evolutionary Architecture*. Architectural Association, London.

French, M. J. (1994). *Invention and Evolution: Design in Nature and Engineering*, 2nd Edition. Cambridge University Press.

French, M. and Ramirez, A. C. (1996). Towards a Comparative Study of Quarter-turn Pneumatic Valve Actuators. *Journal of Engineering Manufacture*, part B, 543–552.

Funes, P. and Pollack, J. (1997). *Computer Evolution of Buildable Objects*. Brandeis University Computer Science Technical Report CS-97-191.

Furuta, H., Maeda, K. and Watanabe, W. (1995). Application of Genetic Algorithm to Aesthetic Design of Bridge Structures. In *Microcomputers in Civil Engineering* 10:6, Blackwell Publishers, MA, 415–421.

Gehlhaar, D. K. and Fogel, D. B. (1996). Tuning Evolutionary Programming for Conformationally Flexible Molecular Docking. In L. Fogel, P. Angeline and T. Back (eds), *Proceedings of the Fifth International Conference on Evolutionary Programming*, MIT Press, Cambridge, MA.

Gen, M. and Cheng, R. (1997). *Genetic Algorithms and Engineering Design*, John Wiley and Sons.

Gero, J. S. (1996). Computers and Creative Design, In Tan, M. and Teh, R. (eds), *The Global Design Studio*, National University of Singapore, pp. 11–19.

Gero, J. S. and Kazakov, V. (1996). An Exploration-based Evolutionary Model of Generative Design Process. *Microcomputers In Civil Engineering* 11, 209–216.

Gero, J. S. and Maher, M. L. (eds) (1993). *Modeling Creativity and Knowledge-Based Creative Design*, Lawrence Erlbaum, Hillsdale, NJ.

Gero, J. S. and Sudweeks, F. (eds) (1994). *Artificial Intelligence in Design '94*, Kluwer, Dordrecht.

Gero, J. S. and Sudweeks, F. (eds) (1996). *Artificial Intelligence in Design '96*, Kluwer, Dordrecht.

Gero, J. S. and Sudweeks, F. (eds) (1998). *Artificial Intelligence in Design '98*, Kluwer, Dordrecht.

Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley.

Goldberg, D. E. (1991). Genetic Algorithms as a Computational Theory of Conceptual Design. In *Proc. of Applications of Artificial Intelligence in Engineering* **6**, pp. 3–16.

Goldberg, D. E. et al. (1992). Accounting for Noise in the Sizing of Populations. In *Foundations of Genetic Algorithms* **2**, Morgan Kaufmann Pub., pp. 127–140.

Goldberg, D. E. (1994). Genetic and Evolutionary Algorithms Come of Age. *Communication of the ACM*, 37:3, 113–119.

Goldberg, D. (1998). *The Design of Innovation: Lessons from Genetic Algorithms*. (in press)

Greffenstette, J. J. (1991). Strategy Acquisition with Genetic Algorithms. Ch. 12 in *The Handbook of Genetic Algorithms*. Van Nostrand Reinhold, New York, pp. 186–201.

Harris, R. A. (1994). An Alternative Description to the Action of Crossover. In *Proceedings of Adaptive Computing in Engineering Design and Control – '94*. University of Plymouth, Plymouth. pp. 151–156.

Harvey, I. (1997). Cognition is Not Computation: Evolution is Not Optimisation. In Gerstner, W., Germond, A., Hasler, M. and Nicoud, J.-D. (eds), *Artificial Neural Networks – ICANN97*, Proc. of 7th International Conference on Artificial Neural Networks, 7–10 October 1997, Lausanne, Switzerland, Springer-Verlag, LNCS 1327, pp. 685–690.

Harvey, I., Husbands, P. and Cliff, D. (1993). Issues in Evolutionary Robotics, In J.-A. Meyer, H. Roitblat and S. Wilson (eds), *From Animals to Animats 2: Proc. of the Second Intl. Conf. on Simulation of Adaptive Behavior, (SAB92)*, MIT Press/Bradford Books, Cambridge, MA, pp. 364–373.

Harvey, I. and Thompson, A. (1997). Through the Labyrinth Evolution Finds a Way: A Slicon Ridge. In Higuchi, T. and Iwata, M. (eds), *Proceedings of the 1st Int. Conf. on Evolveable Systems: From Biology to Hardware (ICES96)*. Springer Verlag, LNCS 1259, pp. 406–422.

Holland, J. H. (1973). Genetic Algorithms and the Optimal Allocations of Trials. *SIAM Journal of Computing* 2:2, 88–105.

Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor.

Holland, J. H. (1992). Genetic Algorithms. *Scientific American*, 66–72.

Horn, J. (1993). Finite Markov Chain Analysis of Genetic Algorithms with Niching. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, Morgan Kaufmann Pub., pp. 110–17.

Horn, J. and Nafpliotis, N. (1993). *Multiobjective Optimisation Using the Niched Pareto Genetic Algorithm*. Illinois Genetic Algorithms Laboratory (IlliGAL), report no. 93005.

Horn, J., Goldberg, D. E. and Deb, K. (1994). *Implicit Niching in a Learning Classifier System: Nature's Way*. Illinois Genetic Algorithms Laboratory (IlliGAL), report no. 94001.

Husbands, P., Jermy, G., McIlhagga, M. and Ives, R. (1996). Two Applications of Genetic Algorithms to Component Design. In Fogarty, T. (ed.), *Selected Papers from AISB Workshop on Evolutionary Computing*. Springer-Verlag, Lecture Notes in Computer Science, pp. 50–61.

Kanal, L. and Cumar, V. (eds) (1988). *Search in Artificial Intelligence*. Springer-Verlag.

Kargupta, H. (1993). *Information Transmission in Genetic Algorithm and Shannon's Second Theorem*. Illinois Genetic Algorithms Laboratory (IlliGAL), report no. 93003.

Keane, A. (1994) Experiences with Optimizers in Structural Design. In I. C. Parmee, *Proceedings of the Conference on Adaptive Computing in Engineering Design and Control '94*. University of Plymouth, Plymouth, pp. 14–27.

Kinnear, Jr., K. E. (ed.) (1994). *Advances In Genetic Programming*. MIT Press, Cambridge, MA.

Koza, J. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA.

Koza, J. (1994) *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge, MA.

Koza, J., Bennett, III, F. H., Andre, D. and Keane, M. A. (1999). *Genetic Programming III*. Morgan Kaufmann, San Francisco.

Langdon, B. (1998). Genetic Programming and Data Structures: *Genetic Programming + Data Structures = Automatic Programming!* Kluwer, Boston.

Langdon, B. and Poli, R. (1997). Fitness Causes Bloat. *2nd On-line World Conference on Soft Computing in Engineering Design and Manufacturing (WSC2).*

Langton, C. (ed.) (1995). *Artificial Life: an Overview*. MIT Press, Cambridge, MA.

Levine, D. (1994). A Parallel Genetic Algorithm for the Set Partitioning Problem. D. Phil. dissertation, Argonne National Laboratory, Illinois, USA.

Lohn, J. and Reggia, J. (1995). Discovery of Self-Replicating Structures Using a Genetic Algorithm. *1995 IEEE Int. Conf. on Evolutionary Computation (ICEC '95)*, vol. 1, Perth, Western Australia, pp. 678–683.

Lund, H., Pagliarini, L. and Miglino, O. (1995). Artistic Design with GA and NN. *Proc. of the 1st Nordic Workshop on Genetic Algorithms and Their Applications (1NWGA)*, University of Vaasa, Finland, pp. 97–105.

Michalewicz, Z. (1995). A Survey of Constraint Handling Techniques in Evolutionary Computation Methods. *Proc. of the 4th Annual Conf. on Evolutionary Programming*, MIT Press, Cambridge, MA, pp. 135–155.

Michalewicz, Z. (1996). *Genetic Algorithms + Data Structures = Evolution Programs*. 3rd extended edn, Springer, Berlin.

Michalewicz, Z., Dasgupta, D., Le Riche, R. G. and Schoenauer, M. (1996). Evolutionary Algorithms for Constrained Engineering Problems, *Computers and Industrial Engineering Journal*, 30:2, September, 851–870.

Mitchell, M. (1996). *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA.

O'Reilly, U.-M. and Oppacher, F. (1995). The Troubling Aspects of a Building Block Hypothesis for Genetic Programming. In Witley, L. D. and Vose, M. D. (eds), *Foundations of Genetic Algorithms*. Morgan Kaufman, San Fransisco, CA, pp. 72–88.

Parmee, I. (1996a) Towards an Optimal Engineering Design Process using Appropriate Adaptive Search Strategies. *Journal of Engineering Design*, 7:4, Carfax Pub.

Parmee, I. (1996b) The Development of a Dual-Agent Strategy For Efficient Search Across Whole System Engineering Design Hierarchies. *4th Int. Conf. on Parallel Problem Solving From Nature*, Berlin, Germany, September 22–27.

Parmee, I. C. and Denham, M. J. (1994). The Integration of Adaptive Search Techniques with Current Engineering Design Practice. In *Proc. of Adaptive Computing in Engineering Design and Control '94*, University of Plymouth, Plymouth, pp. 1–13.

Paton, R. (1994). Enhancing Evolutionary Computation using Analogues of Biological Mechanisms. In *Evolutionary Computing, AISB Workshop*. Springer-Verlag, pp. 51–64.

Paun, G. (ed.) (1995). *Artificial Life: Grammatical Models*. Black Sea University Press, Romania.

Pham, D. T. and Yang, Y. (1993). A Genetic Algorithm Based Preliminary Design System. *Journal of Automobile Engineers*, 207:D2, 127–133.

Poli, R. and Langdon, B. (1997a). Genetic Programming with One-point Crossover. In Chawdhry, P. K., Roy, R. and Pant, R. K. (eds), *Second On-line World Conference on Soft Computing in Engineering Design and Manufacturing*. Springer-Verlag, London.

Poli, R. and Langdon, B. (1997b). A New Schema Theorem for Genetic Programming with One-point Crossover and Point Mutation. In Koza, J., Goldberg, D., Fogel, D. and Riolo, R. L. (eds), *Genetic Programming 1997: Proceedings of the Second Annual Conference on Genetic Programming*. Morgan Kaufmann, San Francisco, CA, pp. 278–285.

Radcliffe, N. J. and Surry, P. D. (1994a). Formal Memetic Algorithms. *Edinburgh Parallel Computing Centre*.

Radcliffe, N. J. and Surry, P. D. (1994b). Co-operation through Hierarchical Competition in Genetic Data Mining. (Submitted to) *Parallel Problem Solving From Nature*.

Rayward-Smith, V., Osman, I. H., Reeves, C. R., Smith, G. D. (1996). *Modern Heuristic Search Methods*. John Wiley and Sons, London.

Rechenberg, I. (1973). *Evolutionstrategie: Optimierung Technisher Systeme nach Prinzipien der Biologischen Evolution*. Frommann-Holzboog Verlag, Stuttgart.

Rechenberg, I. (1994). *Evolutionstrategie '94*, volume 1 of *Werkstatt Bionik und Evolutionstechnik*. Frommann-Holzboog, Stuttgart.

Rosenman, M. (1997). The Generation of Form Using an Evolutionary Approach. In Dasgupta, D. and Michalewicz, Z. (eds), *Evolutionary Algorithms in Engineering Applications*, Springer-Verlag, pp. 69–86.

Roston, G. (1997). Hazards in Genetic Design Methodologies. In Dasgupta, D. and Michalewicz, Z. (eds), *Evolutionary Algorithms in Engineering Applications*, Springer-Verlag, pp. 135–154.

Roy, R., Furuhashi, T. and Chawdry, P. K. (eds) (1999). *Advances in Soft Computing – Engineering Design and Manufacturing*. Springer-Verlag, London.

Rudolph, G. (1996). Convergence of Evolutionary Algorithms in General Search Spaces. *Proceedings of the Third IEEE Conference on Evolutionary Computation*, Piscataway, NJ. IEEE Press, pp. 50–54.

Rudolph, G. (1997). Convergence Rates of Evolutionary Algorithms for a Class of Convex Objective Functions, *Control and Cybernetics* 26(3), 375–390.

Rudolph, G. (1998). Local Convergence Rates of Simple Evolutionary Algorithms with Cauchy Mutations, *IEEE Transactions on Evolutionary Computation* 1(4).

Schaffer, J. D. (1985). Multiple Objective Optimization with Vector Evaluated Genetic Algorithms. *Genetic Algorithms and Their Applications: Proceedings of the First International Conference on Genetic Algorithms*, pp. 93–100.

Schaffer, J. D. and Eshelman, L. (1995). Combinatorial Optimization by Genetic Algorithms: The Value of Genotype/Phenotype Distinction. In *Proc. of Applied Decision Technologies (ADT '95)*, April 1995, London, pp. 29–40.

Schnier, T. and Gero, J. S. (1996). Learning Genetic Representatrions as an Alternative to Hand-coded Shape Grammars. In Gero, J. and Sudweeks, F. (eds), *Artificial Intelligence in Design '96*, Kluwer, Dordrecht, pp. 39–57.

Schoenauer, M. (1996). Shape Representations and Evolution Schemes. *Proc. of the 5th Annual Conf. on Evolutionary Programming*, MIT Press, Cambridge, MA, pp. 121–129.

Schwefel, H.-P. (1965). *Kybernetische Evolution als Strategie der experimentellen Forschung in der Strömungstechnik*. Diplomarbeit, Technische Universität, Berlin.

Schwefel, H.-P. (1981). *Numerical Optimization of Computer Models*. Wiley, Chichester.

Schwefel, H.-P. (1995). *Evolution and Optimum Seeking*, Wiley, New York.

Sims, K. (1991). Artificial Evolution for Computer Graphics. *Computer Graphics*, 25, 4, 319–328.

Sims, K. (1994a). Evolving Virtual Creatures. In *Computer Graphics*, Annual Conference Series (SIGGRAPH '94 Proceedings), July 1994, 15–22.

Sims, K. (1994b). Evolving 3D Morphology and Behaviour by Competition. In Brooks, R. and Maes, P. (eds), *Artificial Life IV Proceedings*, MIT Press, Cambridge, MA, pp. 28–39.

Sipper, M. (1997). A Phylogenetic, Ontogenetic, and Epigenetic View of Bio-Inspired Hardware Systems. *IEEE Transactions On Evolutionary Computation*, 1:1.

Smith, R. E. and Goldberg, D. E. (1992). Diploidy and Dominance in Artificial Genetic Search. *Complex Systems* 6, 251–285.

Srinivas, N. and Deb, K. (1995). Multiobjective Optimization Using Nondominated Sorting in Genetic Algorithms. *Evolutionary Computation*, 2:3, 221–248.

Syswerda, G. (1989). Uniform Crossover in Genetic Algorithms. In Schaffer, D. (ed.), *Proc. of the Third Int. Conf. on Genetic Algorithms*. Morgan Kaufmann Pub,

Tabuada, P., Alves, P., Gomes, J. and Rosa, E. A. (1998). 3D Artificial Art by Genetic Algorithms. In Bentley, P. J. (ed.), *Proc. of the Workshop on Evolutionary Design*, 5th International Conference on Artificial Intelligence in Design '98, Instituto Superior Técnico, Lisbon, Portugal, 20–23 July 1998.

Todd, S. and Latham, W. (1992). *Evolutionary Art and Computers*. Academic Press,

Thompson, A. (1995). Evolving Fault Tolerant Systems. In *Genetic Algorithms in Engineering Systems: Innovations and Applications*, IEE Conf. Pub. No. 414, pp. 524–529.

Ventrella, J. (1994). Explorations in the Emergence of Morphology and Locomotion Behaviour in Animated Characters. In Brooks, R. and Maes, P. (eds), *Artificial Life IV*, Proc. of the 4th Int. Workshop on the Synthesis and Simulation of Living Systems, MIT Press, Cambridge, MA, pp. 436–441.

Whitley, D. and Starkweather, T. (1990). GENITOR II: A Distributed Genetic Algorithm. *Journal of Experimental and Theoretic Artificial Intelligence* 2:3, 189–214.

Yamada, T. and Nakano, R. (1995). A Genetic Algorithm with Multi-step Crossover for Job-shop Scheduling Problems. In *Genetic Algorithms in Engineering Systems: Innovations and Applications*, IEE Conf. Pub. No. 414, pp. 146–151.

Yao, X. and Liu, Y. (1996). Fast Evolutionary Programming, In Fogel, L., Angeline, P. and Back, T. (eds) *Proceedings of the Fifth International Conference on Evolutionary Programming*, MIT Press, Cambridge, MA.

Yao, X. Lin, G. and Liu, Y. (1997). An Analysis of Evolutionary Algorithms Based on Neighbourhood and Step Sizes. In Angeline, P., Reynolds, R., McDonnell, J. and Eberhart, R. (eds), *Proceedings of the Sixth International Conference on Evolutionary Programming*. Springer, pp. 298–307.

Yu, T. and Bentley, P. (1998). Methods to Evolve Legal Phenotypes. *Fifth Int. Conf. on Parallel Problem Solving From Nature*. Amsterdam, Sept 27–30. Springer.

Yu, T. and Clack, C. (1998a). Recursion, Lambda Abstractions and Genetic Programming, *Genetic Programming 1998: Proceedings of the Third Annual Conference*. Morgan Kaufmann, San Francisco.

Yu, T. and Clack, C. (1998b). PolyGP: A Polymorphic Genetic Programming System in Haskell (ed.), *Genetic Programming 1998: Proceedings of the Third Annual Conference*. Morgan Kaufmann, San Francisco.