Applied Evolutionary Algorithms in Java

Springer Science+Business Media, LLC

Robert Ghanea-Hercock

Applied Evolutionary Algorithms in Java

With 57 Illustrations

Springer

Robert Ghanea-Hercock
BTexact Technologies
Intelligent Systems Laboratory
Main Lab Block, First Floor, pp12
Adastral Park
Martlesham Heath, Ipswich IP5 3RE
UK
robert.ghanea-hercock@bt.com

Library of Congress Cataloging-in-Publication Data
Ghanea-Hercock, Robert.
Applied evolutionary algorithms in Java / Robert Ghanea-Hercock.
p. cm.
Includes bibliographical references and index.
ISBN 978-1-4684-9526-3
1. Evolutionary programming (Computer science) 2. Genetic algorithms. 3. Java (Computer program language) I. Title.
QA76.618 .G453 2003
005.1—dc21 2002042740

Printed on acid-free paper.

Additional material to this book can be downloaded from http://extra.springer.com.

ISBN 978-1-4684-9526-3 ISBN 978-0-387-21615-7 (eBook)
DOI 10.1007/978-0-387-21615-7

9 8 7 6 5 4 3 2 1 SPIN 10891089

www.springer-ny.com

Springer-Verlag New York Berlin Heidelberg
A member of BertelsmannSpringer Science+Business Media GmbH

Preface

> The Quantum Weather Butterfly (Papilio tempestae) is an undistinguished yellow colour. Its outstanding feature is its ability to create weather. This presumably began as a survival trait, since even an extremely hungry bird would find itself inconvenienced by a nasty localized tornado.
>
> Terry Pratchett, *Interesting Times*

The principle of adaptive evolution is strongly intuitive, as we all perceive its effects in the natural world and the Darwinian perspective has permeated common culture. Computing researchers working in the fields of Artificial Intelligence and Machine Learning realised over 30 years ago that such a principle could be simulated in software, and hence exploited as a tool for automating problem solving. The breakthrough in understanding how to apply evolution is normally credited to the group led by John Holland and their seminal work on the Genetic Algorithm, in 1975 at the University of Michigan.

However, it has taken several decades for evolution-based methods to become an accepted element of computing techniques. Part of the reason for this is the relative complexity of the theories underlying evolution-based algorithms, which are still poorly understood.

This text is aimed at providing the reader with a practical guide to applying evolutionary algorithms to engineering and scientific problems. The concepts are illustrated through clear examples, ranging from simple to more complex problem domains.

A major aspect of this text is the use of a Java toolkit for exploring Genetic Algorithms, with examples in digital image processing and mobile robot control. A hands-on approach is encouraged throughout the text, in order to develop an intuitive understanding of evolution-driven algorithms. The toolkit provides an easy-to-use visual interface, with integrated graphing and analysis tools, which aid in the task of visualising the underlying processes. Full code for the toolkit is available from the author's web site, where regular updates can be found (http://www.cybernetics.org.uk), and on the included CD-ROM.

Chapter 1 provides a basic introduction to the history and background of evolution-based algorithms. Chapter 2 covers some basic but essential background material and principles from biological evolution. Chapters 3 and 4 cover the theory and operation of Genetic Algorithms and Genetic

Programming, respectively, in greater detail, as these represent the main techniques in common use. Chapter 5 provides a detailed pair of examples from the engineering domain in image processing and mobile robot control. Chapters 6 and 7 finish with a review of future directions in the field and some conclusions on the benefits of evolution-based processing. Appendix A provides a listing of some useful web-based references and software. Appendix B describes the specific genetic algorithm software used in the text and the Eos evolution toolkit from the Intelligent Systems laboratory at BTexact Technologies. Appendix C provides some essential background knowledge in Fuzzy Logic, which is required for the mobile robot control example. Appendix D is a detailed user guide for the excellent mobile robot simulator, which has been generously contributed by Gary Lucas.

This book is intended for anyone interested in evolution-based software at the graduate and postgraduate levels. No mathematics beyond basic algebra and Cartesian graph methods is used, as the aim is to encourage use of the Java toolkit in order to develop an appreciation of the power of these techniques. Several further texts are referenced that provide extensive coverage of the theory and mathematical basis of genetic operators. The author welcomes suggestions and feedback on the text and software.

Description and Use of CD-ROM

The accompanying CD-ROM in the back cover of the book contains the Java code for the example applications from Chapter 5. It also contains a selection of associated open-source Java code libraries for Evolutionary Applications and robot simulators from other sources. In addition the papers directory contains a selection of useful papers on Evolutionary Computing. Installation instructions are contained in the CD file ReadMe.txt.

Acknowledgements

I would like to acknowledge the following people who for various reasons were involved in contributing to the creation of this book. I must particularly thank Gary Lucas, the author of the Rossum mobile robot simulator, which forms the basis for the second application in Chapter 5, as it was ideally suited to the purpose of this text to promote Java and evolutionary algorithms in graduate studies. Thanks also go to members of the Future Technologies group at BTexact Technologies – in particular to Richard Tateson, Rob Shipman and Paul Marrow for contributions, Erwin Bonsma for the Eos toolkit and appendix material, and Mark Shackleton for allowing me time to work on this project.

Special thanks also go to my parents, Joyce and Derek Hercock. Finally, thanks to my wife for tolerating my obsessive interest in artificial evolution.

Contents

1
Introduction to Evolutionary Computing

> Four billion years ago, the Earth was a molecular Garden of Eden. There were as yet no predators. Some molecules reproduced themselves inefficiently, competed for building blocks and left crude copies of themselves. With reproduction, mutation and the selective elimination of the least efficient varieties, evolution was well under way, even at the molecular level. As time went on, they got better at reproducing. Molecules with specialized functions eventually joined together, making a kind of molecular collective - the first cell.
>
> Carl Sagan, *Cosmos*, 1980

1.1 Evolutionary Computation

Evolution is a ubiquitous natural force that has shaped all life on Earth for approximately 3.2 billion years. For several thousand years humanity has also utlised artificial selection to shape domesticated plant and animal species. In the past few decades, however, science has learned that the general principles at work in natural evolution can also be applied to completely artificial environments. In particular, within Computer Science the field of automated machine learning has adopted algorithms based on the mechanisms exploited by natural evolution.

One driving motivation for many researchers in Evolutionary Algorithms (EA) and other machine learning fields is to create a set of automatic processes that can translate high-level task descriptions into well-coded solutions. The problem is that almost all commercially available software relies on handcrafted and custom-written lines of code. This is always a time-consuming, expensive, and error-prone process. For example, a typical modern operating system contains several million lines of code and thousands of bugs, most of which are tolerated and slowly weeded out by consumer effort (i.e., you). This may work at present, but the cost and success rate of large and complex pieces of software are a major concern to software developers. The result is a widening gap between the capabilities of computing hardware and the quality and efficiency of the software it runs. One solution being pursued is the open source movement

R. Ghanea-Hercock, *Applied Evolutionary Algorithms in Java*

in which code is opened to a large community of developers who contribute in an ongoing process to its development (the principal example being development of the Linux operating system). Good solutions are then incorporated into the final product and poor solutions culled, which is not entirely dissimilar to the approch of evolution-based algorithms.

There are many flavours of machine learning algorithms, from Case Based Reasoning to Artificial Neural Networks, each of which has particular strengths and weaknesses. However, there is a particular attraction of EA to many computer scientists. This is difficult to convey until you have experimented with such algorithms and been fascinated by their ability to produce a functioning solution to a problem out of initially random bit sequences. It is the aim of this text to enable readers to quickly acquire confidence in building and developing their own EA programs and hence to gain some understanding of the power of EA as a machine learning method.

1.2 History of Evolutionary Computing

It is particularly useful to consider the history of evolution within computing as it covers much of the timeframe of computing itself. Some of the earliest work can be traced back to Friedberg (1958), who introduced the idea of an evolutionary algorithm approach for automatic programming. Later significant developments included the creation of evolutionary programming by Fogel (Fogel et al., 1966). John Holland (1967) founded the initial work on genetic algorithms at the University of Michigan. Parallel work was also initiated by Bienert, P. Rechenberg, and Schwefel in evolutionary strategies (1966).

However, the first evolutionary algorithm in the form most commonly utilised today was the genetic algorithm (GA) created by John Holland and his students at the University of Michigan. The methods defined by the GA are the basis for this text as they represent the most commonly used EA methods and have proved to be highly flexible in real-world applications. Endless variations of the standard GA operators have also been developed, such as variable-length chromosomes, multipoint crossover, and many other flavours. Chapter 3 provides an introduction to the core GA concepts and some of the common variations in present use.

The next major development in EA systems was Genetic Programming, introduced by John Koza (1992). Koza formulated a general-purpose system for program induction based on a tree-based representation scheme, which has been applied to a very wide range of machine learning problems. Genetic programming is allocated to Chapter 4 precisely because of its broad applicability and to present an alternative approach to evolving computer algorithms. In addition, it allows a more complex genome data representation, which can be mapped to the object-based approach presented in the applications in Chapter 5.

The remainder of this chapter will first review the challenges encountered in EA and the software and hardware issues that impact the exploitation of artificial evolution as a problem-solving method. Second, a brief overview of the history and methods of machine learning is provided, as this introduces some core concepts required later in the text.

1.3 Obstacles to Evolutionary Computation

A major barrier to the early adoption of EA in the computing domain came from opposition within the computer science community itself. This was often based on the mistaken belief that such algorithms, with probabilistic processes as a core mechanism, would not be amenable to producing functional code. Particularly as normal computer code is generally brittle, and very susceptible to minor variations in structure. In Chapters 2 and 3 we develop the necessary background to EA methods to understand why probabilistic selection methods can generate functional code. As Goldberg states:

> In our haste to discount strictly random search techniques, we must be careful to separate them from randomised techniques. The genetic algorithm is an example of a search procedure that uses random choice as a tool to guide a highly exploitative search through a coding of a parameter space. (Goldberg 1989, p. 5)

The second barrier to EA development was the problem that contemporary computing technology in software, and particularly hardware, in the early 1970s was barely capable of generating useful results in acceptable time scales (i.e., less than a few weeks). This problem added to the belief that such methods, while theoretically interesting, would never be capable of useful applications. Fortunately the exponential development in computing systems has generated both software and hardware that offer the necessary processing power and also software sophistication in terms of efficient compilers and object-based design.

A modern high-specification desktop can generate useful results from an EA within a few hours, in a typical application such as those described in Chapter 5. In Chapter 6 some speculative ideas are discussed on the possibilities future hardware will enable, such as using EA within real-time adaptive applications. In particular, the possibility of using reconfigurable hardware based on Field Programmable Gate Array devices, opens entirely new applications to EA methods. This area is explored in Chapter 6.

1.4 Machine Learning

Machine learning explores the mechanisms by which knowledge or skills can be acquired through computing processes. It includes models for learning based on statistics, logic, mathematics, neural structures, information theory, and heuristic

search algorithms. It also involves the development and analysis of algorithms that identify patterns in observed data in order to make predictions about unseen data. Mitchell gives a useful definition:

> "..machine learning is the study of computer algorithms that improve automatically through experience." (Mitchell,1996).

In contrast, the early history of computer intelligence was dominated by the field of 'Expert Systems' in which the emphasis was in precisely defining human knowledge into machine processable tasks. Some success was achieved with the expert systems approach, but it became increasingly obvious that such systems were highly task-specific with minimal ability to generalise the knowledge supplied. Hence increased interest developed during the 1980s in giving machines the ability to *learn* new knowledge, leading to a revival of machine learning processes.

Longer-term goals that the pioneers of the field considered achievable, such as general learning and cognition, have been shelved until far more work has been completed in understanding the basics of specific machine skills. Expert and knowledge-based systems will therefore continue to develop in parallel with adaptive and soft computing processes, such as EA and Fuzzy Logic systems.

1.5 Problem Domains

There is an increasing range of complex problems across commerce and science, which require automated and adaptive computational techniques. For example, relational databases with millions of records are now common in business, astronomy, engineering, and the sciences. The problem of extracting useful information from such data sets is an important (and lucrative) commercial problem. Evolution-based algorithms have been applied to a wide range of technical problems, including handwriting recognition, scheduling in manufacturing, speech and image processing. Other examples of EA applications include

- Gas pipeline optimisation (Goldberg, 1983)
- Robot kinematics (Davidor, 1991)
- Image Processing (Harvey & Marshall, 1996)
- Evolved art (Todd & Latham, 1992)
- Network design (Choi, 1996)

1.5.1 Search Spaces

A central aspect of machine learning is the requirement of an algorithm to perform an automated search of a complex and often multidimensional problem space, (as shown in figure 1.1). The problem space is effectively a mapping of the problem parameters into an abstract n-dimensional state space. In state space

search, a programmer first assigns a computer an initial state. A set of operators is automatically used to move from one state to the next. In addition, some halting condition is required, which specifies when a sufficiently close solution has been achieved. Common search algorithms applied to AI search strategies include breadth-first and depth-first search.

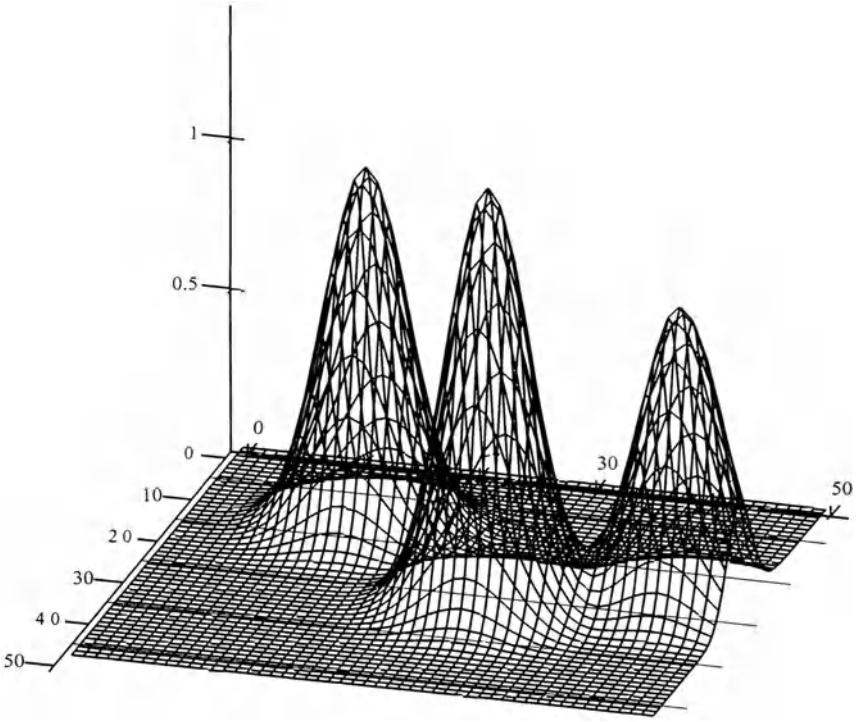

FIGURE 1.1 A typical multipeak search space, in which a machine learning algorithm can become trapped on a suboptimal solution.

A useful starting point for developing search algorithms is the text *Fundamentals of Data Structures in C++* by Horowitz et al, (Horowitz et al, 1995). Simple hill-climbing or related calculus-based methods may have great difficulty in solving such multioptima spaces, and are of even less use in discontinuous and irregular spaces. Unfortunately, many if not all real-world problems are of this form. All too often researchers test new algorithms and strategies on relatively simple test cases and then fail to achieve any qualitative improvement when running the algorithm on a real data set. It is therefore

instructive to consider the idea that in all machine search methods the *no free lunch theorem* applies (Wolpert and Macready, 1995) that is:

> all algorithms that search for an extremum of a cost function perform exactly the same, when averaged over all possible cost functions. In particular, if algorithm A outperforms algorithm B on some cost functions, then loosely speaking there must exist exactly as many other functions where B outperforms A. (Wolpert and Macready, 1995)

1.5.2 Optimisation versus Robustness

It is of some importance when considering the value of any machine learning algorithm to determine the correct measure of effectiveness (or MOE as defined in military strategy studies). Real-world search spaces as discussed are invariably complex and ill defined, hence spending large resources on locating a precise optimum may be futile. A better approach may be to employ a spread of algorithms, which generate a robust and usable set of good solutions. The lessons learned from the underachievement of expert systems were exactly that they frequently generated optimised but fragile and nonadaptive solutions. EA can also experience the same problems if poorly designed.

Another way of reading this is to apply quick and cheap searches if they approach a working solution. Fuzzy logic and EA are examples of broad and robust search algorithms, and their robustness attributes will be considered in some detail later in the text.

1.5.3 Expert Systems and AI

This section provides a very cursory review of rule-based AI methods, as a comparison with soft computing and EA. Expert systems utilise rule-based programming rules to represent heuristics, or "rules of thumb," which specify a set of actions to be performed for a given situation. A rule is composed of an *if* portion and a *then* portion. The *if* portion of a rule is a series of patterns that specify the facts (or data) that cause the rule to be applicable (the precondition). The process of matching facts to patterns is called pattern matching. The expert system tool provides a mechanism, called the inference engine, which automatically matches facts against patterns and determines which rules are applicable. The *if* portion of a rule can actually be thought of as the *whenever* portion of a rule since pattern matching always occurs whenever changes are made to facts. The *then* portion of a rule is the set of actions to be executed when the rule is applicable (the postcondition). Hence the actions of applicable rules are executed when the inference engine is instructed to begin execution. The inference engine selects a rule and then the actions of the selected rule are executed (which may also affect the list of applicable rules by adding or removing facts).

A useful publicly available rule-based expert system is CLIPS (the C Language Integrated Production System), which was originally developed at NASA. CLIPS is available from www.ghg.net/clips/CLIPS.html. (A useful Java-based rule system is the IBM AbleBeans code library, available from http://alphaworks.ibm.com.) For further material a good introductory text to modern AI is that by Russell and Norvig (1995).

Current work in expert systems has now shifted to Case Based Reasoning and agent-based systems, each of which extends and adapts the use of heuristic inferencing. An interesting (and profitable) application domain example involves online e-commerce systems (e.g., Maes et al., 1999, and Ardissono et al., 1999), in which agents and CBR are combined to support intelligent user services. However, the majority of working business and e-commerce systems still rely heavily on simple rule-based inferencing.

1.5.4 Fuzzy Logic

One highly successful computational intelligence method that has emerged is Fuzzy Logic. Fuzzy logic (Zadeh, 1965) has enjoyed wide popularity in engineering as an advanced control and AI technique; particularly within the Japanese and Asian markets. However, it is only in the past two decades that it has been recognised in Europe and America as a useful AI system. Driankov et al., (Driankov, Hellendoorn, and Reinfrank, 1996) provides an excellent introduction to applied fuzzy control methods. It provides a particularly clear insight into the theory and application of fuzzy techniques.

Since Zadeh's original work on Fuzzy Logic (Zadeh, 1965), intense debate has surrounded the subject, with vigorous defenders and opponents on both sides. In terms of the underlying theory the argument has revolved around whether fuzzy logic is just another form of probabilistic reasoning or whether probability is a subset of fuzziness (Kosko, 1992).

A fuzzy rule system effectively embeds expert knowledge into a set of flexible overlapping rules in order to create an adaptive data processing system. Its prime application area is in advanced control systems, such as robotics, industrial plant control, transport speed controllers, and washing machines! Its strength lies in its ability to control a complex system, even if no precise mathematical model of the underlying processes is available. The key issue revolves around designing the required input and output rule sets.

Since a fuzzy system normally contains no learning ability in itself, it is often combined with evolutionary algorithms or neural networks, which act to learn the necessary rule sets from some training data. Appendix C provides an introductory tutorial to fuzzy systems, as they form an important part of the mobile robot control application presented in Chapter 5. Figure 1.2 provides an illustration of a fuzzy clustering algorithm applied to an e-commerce application.

As with all of the machine learning methods fuzzy logic is not a panacea, but it is highly effective in the control of complex engineering systems and can provide a useful interface between other adaptive algorithms and physical systems. An excellent introduction to fuzzy systems, which includes material on neural networks and how the two systems can be combined, is Kosko's text (Kosko, 1992). The advantages of fuzzy control systems can be divided into theoretical and practical issues.

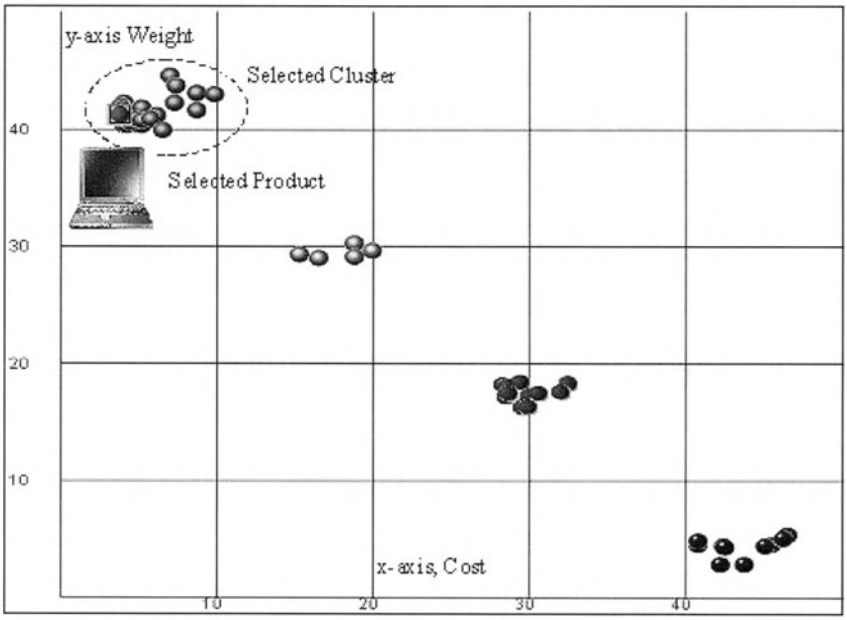

FIGURE 1.2 Example fuzzy clustering technique applied to data points for an e-commerce application. Using 4 input attributes to generate 4 clusters, from a range of 64 laptop computers. In this example a fuzzy rule system can help identify a product based on a customers combined set of specified preferences. (Copyright BT ISR Laboratories, 1999.)

First, the arguments for fuzzy control include

- They permit the incorporation of linguistic knowledge into a control architecture.
- It is a model-free system, hence no underlying mathematical model of the control process or plant is required.
- It provides an efficient nonlinear control mechanism, which is justified by the universal-approximation theorem.
- Ease of implementation.

- Simple to design, compared to an equivalent conventional controller for the same system (particularly true for nonlinear systems).
- Cheap to develop, with well-established software packages available, and fast to prototype new code if required.
- Possible to implement as a VLSI chip, which are now widely available (in contrast to the inherent complexity of hardware neural network systems).

1.5.5 Bayesian networks

A Bayesian network is basically a model representation based on reasoning with uncertainty, using a statistical approach. A problem domain is divided into a number of entities or events, which are represented as separate variables. One variable could represent the event that a server in a communications network has failed. The variables representing different events are then connected via directed edges to describe relations between events. An edge between two variables X and Y then simply represents a possible dependence relation between the events represented by X and Y, as illustrated in Figure 1.3.

An edge could, for instance, describe a dependence relation between a disease and a symptom (a common application domain for Bayesian net software). Hence, edges can be used to represent cause-effect relations. These dependence relations between entities of the problem domain may be organised as a graphical structure. This graphical structure describes the possible dependence relations between the entities of the problem domain. The advantage of such Bayesian networks is their compactness and efficiency.

The uncertainty of the specific problem domain is then represented through conditional probabilities. Conditional probability distributions specify our belief about the strengths of the cause-effect relations; for example, a server crash may not be due to network traffic but could also be due to a power outage, each of which is assigned some probability value.

Thus, a Bayesian network consists of a qualitative part, which describes the dependence relations of the problem domain, and a quantitative part, which describes our belief about the strengths of the relations. The framework of Bayesian networks offers a compact, intuitive, and efficient graphical representation of dependence relations between the entities of a problem domain.

The graphical structure reflects properties of the problem domain in an intuitive way, which makes it easy for nonexperts to understand and build this kind of knowledge representation. It is possible to utilise both background knowledge such as expert knowledge and knowledge stored in databases when constructing Bayesian networks.

(A freeware tool for experimenting with Bayesian networks is available from the Microsoft research web site at http://research.microsoft.com/msbn/.)

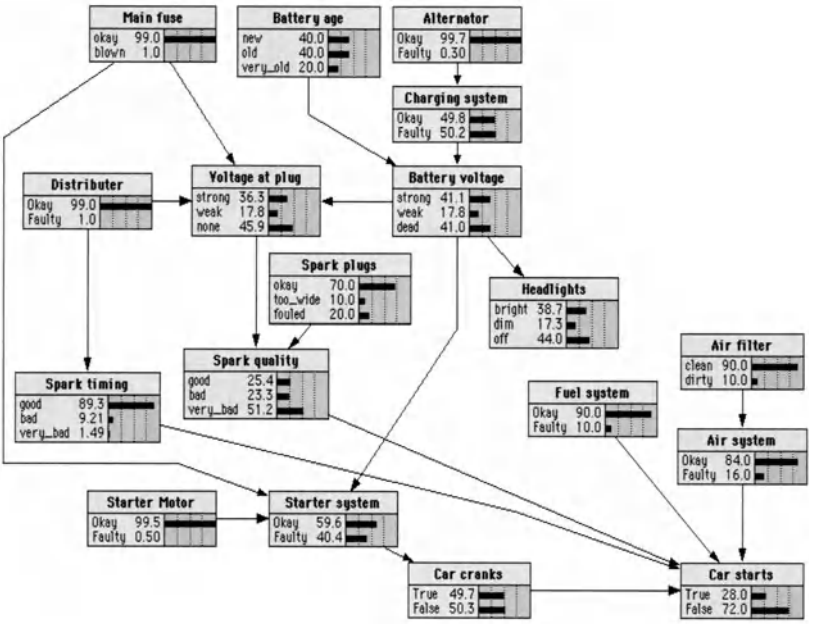

FIGURE 1.3 Example graphical representation of a Bayesian network showing a simple example belief network for diagnosing why a car won't start, based on spark plugs, headlights, main fuse, etc. (Copyright Norsys Software Corp., 2001.)

1.5.6 Artificial Neural Networks

A dominant field in machine learning is that of artificial neural networks (ANN). Original work in this field can be traced back to Mc Culloch and Pits (1943) and their development of artificial models of neuron behaviour. In this model the neuron receives one or more inputs and produces one or more identical outputs. Each output is then a simple nonlinear function of the sum of the inputs to the neuron.

Given the obvious fact that the most intelligent computer on the planet weighs about 2.5 pounds and is composed of a jellylike mass of biological neurons, there has been significant interest in the computer science community in learning how to replicate the capabilities of the biological neuron. Strong interest in the AI community continued until 1969, when a text by Marvin Minsky and Seymour Papert emphasised the limitations of the Perceptron model

of ANN, which was dominant at the time; this virtually halted further research in the subject.

In 1982, however, a seminal paper by Hopfield (Hopfield, 1982) reignited interest in the field with a new model of a bidirectional ANN. Since then a large research community has grown around the investigation of ANN models and their application. One useful definition is

> a neural network is a system composed of many simple processing elements operating in parallel whose function is determined by network structure, connection strengths, and the processing performed at computing elements or nodes. (DARPA Neural Network Study, 1988, AFCEA Int. Press.)

1.5.7 Feedforward Networks

The commonest ANN is the three-layer feedforward network, which has an input layer, one hidden layer, and an output layer. The network operates as a transfer function acting on a vector applied to the input layer, as illustrated in Figure 1.4.

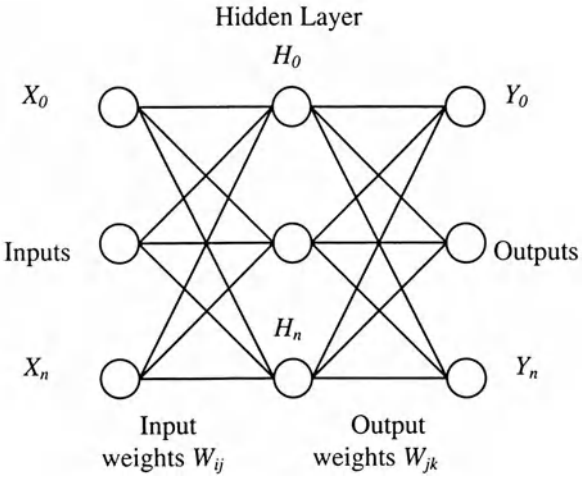

FIGURE 1.4 Example of a three-layer feedforward network (Welstead, 1994).

The network's behaviour is determined by the number of layers and nodes within each layer and the value of each weight linking the nodes. Through a training process involving the presentation of large sets of input-output pairs the network is able to learn the required transfer function. The popularity of these systems is based on the mathematical result that such networks are *universal approximators*, hence they can model any reasonable function (i.e., one having a small number of discontinuities and defined on a closed, bounded subset of $\mathbf{R}^N$)

based on a deterministic process (see Kosko, 1992). The most frequently used transfer function for a neural network is a sigmoidal function, as shown in Figure 1.5.

$$f(x) = \frac{1}{1 + e^{(-a.x)}} \tag{1.1}$$

The use of weighted threshold connections is a basic approximation of the way in which biological synapses operate. Nonlinear transfer functions increase the computational range of a parallel network and help in noise suppression. However, they increase the complexity of analysing the trained network. Learning in natural organisms utilises adjustments in the strength of the synaptic connections, and similarly ANNs are also able to encode learned responses in the set of weight vectors.

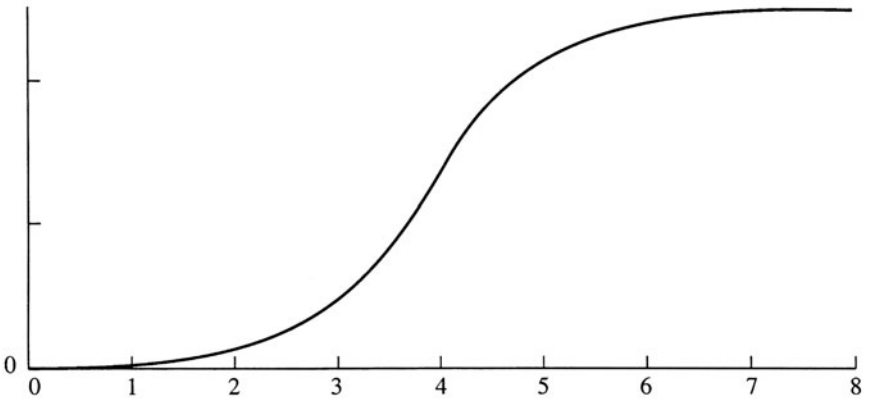

FIGURE 1.5 Example sigmoidal threshold function, used in ANN nodes, based on the logistic sigmoid function in (1.1) a = gain factor, i.e., response time of the network.

1.6 Applications

The prime benefit of ANN is the ability to act as pattern recognisers and noisy data classifiers. They can generalise decisions from poor-quality input data and hence are ideal for a range of classification problems, such as

- Handwriting recognition
- Speech processing
- Image processing
- Digital and analogue signal processing

- Robot control

ANN may be categorised into two general types; feedforward and recurrent (utilising feedback). They can also be divided into whether they use training data to learn the required response (supervised) or use a self-organising method (unsupervised) to learn. An example of the later is Kohenen networks (Kohenen, 1984). Unsupervised networks operate by clustering data into related groups based on measuring the features acting as inputs to the network.

1.6.1 Problems

While there is an intuitive and appealing relationship between ANN and biological neural systems, the artificial variety suffers from several disadvantages. First, the universal approximation property may be theoretically useful, but it fails to inform us how many hidden layers and nodes the network may actually require for a given function. Second, the number of pairs of training data pairs required to train even a simple network is often several thousand. This generally requires a significant amount of computation. In addition, the algorithms used to compute the weights are nontrivial and require extensive processing. Finally, the most significant problem relates to interpretation of the final trained network and its set of activation weights. It is extremely difficult to extract the logical inference, which relates a given input vector to an output response, namely, to understand how the network is performing its specific task. This has led to the formation of the new subfield of Rule-Extraction from ANN. In effect, the trained network is a black-box mechanism. In contrast, fuzzy rule-based systems are intrinsically transparent to inspection and analysis of which rules relate to specific behaviours of the system.

It is also very important to note that contrary to popular opinion, biological neurons are *not* simple function transducers (Koch, 1997); rather they should be modeled as individual agents in their own right; or at least as having adaptive properties. Cortical neurons contain a nucleus and cellular body and are specialised but fully functional cells. Constructing simple three-layer feedforward networks based on sigmoid functions and expecting them to model natural neural computational structures is simply unrealistic. Significantly more complex neural systems have, however, been designed. These may be based on a cellular model, which can act in a co-operative manner to process complex signals. Recent work on using cellular neural systems for controlling autonomous software agents with a more realistic biological model has been demonstrated by Cyberlife Technology Inc., in their "Creatures" product (Cyberlife, Grand, 1996).

However, ANN systems do possess intrinsic learning and adaptive capabilities, and these have been exploited in a range of advanced self-organising controllers. An interesting example is mobile robot control systems.

Robotic systems are a useful test platform for a wide range of machine learning methods and frequent use of ANN has been made in this domain, as they offer a way to investigate sensory integration and processing. They may also be used to directly implement the sensor-actuator link or to control the interaction of lower-level reactive behaviours.

1.7 Evolution-Based Search

The central concept of EA is that repeated mixing, mutation, and a selective filtering action on a population of individual bit string solutions (i.e., chromosomes) is an effective method for locating a solution point within a complex search space. Chapters 2 and 3 cover the field of EA in greater detail.

1.7.1 Languages for Evolutionary Computing

In order to implement an evolutionary algorithm, we need two components, a sufficiently powerful computer and a suitable programming language. Fortunately both of these components have matured since Holland's early work, and modern computers and high-level languages make the task of building evolutionary algorithms far easier. However, the software aspect of implementing EA is frequently a source of frustration for new researchers and students in the field. It is a nontrivial exercise to code even a minimal EA. This is particularly the case with languages that are not object-oriented, as an EA is best represented as a collection of interacting data objects, each possessing multiple parameters. In addition, the classic programming problem of memory management is of importance, as individual chromosomes are being added and deleted from the pool of evolving code at some rate, leading to the need for careful deallocation of objects from memory. Since the advent of new object-based languages such as Java, C++, and more advanced compilers, these have greatly simplified the programming effort required to develop EA systems.

The relative merits of some suitable languages will now be considered. All of the examples and exercises in this book are based on the Java programming language from Sun MicroSystems, in its current version JDK1.3. There are several strong reasons why this language was selected for this text, and these will be discussed with comparisons to alternative languages.

1.7.2 C and C++

The principal computing language used to date for implementing EA has been C or its derivative, C++. It offers several advantages well suited to EA work;

- High speed: from being a compiled medium-level language.
- Compact size .
- Wide range of efficient compilers and support tools available.
- C++ is an early object-based language that builds on the strengths of C.

As a result there are now a significant number of code libraries in both C and C++ available to perform a vast range of EA operations. In Appendix A, a few of the principal libraries are listed, particularly those available for open source distribution.

However, there are also some disadvantages to these languages. First, C is a relatively low-level language with few operators for complex data manipulation. It can also be difficult for new students to learn and it can be difficult to implement large-scale programs. C++ was an attempt to add object-based functionality to C, which has been partially successful. However, both languages offer little in the way of automatic memory management, which places a burden on the developer to manually code explicit deallocation of used memory resources. C++ has also been criticised for its complexity and lack of standardisation. It can be a major task to port code developed on one specific hardware platform to another.

1.7.3 Pascal and Fortran

Pascal and more frequently Fortran have also been used to develop EA systems. However, there are far fewer code libraries available than C or C++. Both languages suffer to some extent from the disadvantages of C and usually lack the object capabilities of C++ (Delphi, a strong object-based language derived from Pascal, is an exception).

1.7.4 Visual Basic

A modern object oriented language that could be used for EA work is Visual Basic; however, it is relatively slow, compared to fully compiled languages, and quite platform-specific. Some source code for EA in VB is available on the web.

1.7.5 Java

Java has several major advantages for developing EA applications:

- Automatic memory management.
- Pure object-oriented design.
- High-level data constructs: for example, the Vector and ArrayList objects offer dynamically resizable arrays.
- Platform independent code is easily ported among Unix, Linux, Windows, and Mac systems.
- Several complete EA libraries are available for EA systems (see Appendix A).

Since life is rarely perfect, however, there is a price to pay! Java is an interpreted language, meaning it is converted from its bytecode format into a platform-specific executable at run-time. This significantly slows down Java applications, relative to equivalent C or C++ code. However, recent work by Sun and third-party companies has resulted in "Just in Time" compilers for Java

that preload and compile the Java bytecode. The end result is performance, which is within 30-50% of an equivalent C application. (See the Bibliography for references in this area.)

1.7.6 Object-Oriented Design

One of the fundamental advances in software development was the introduction of object-oriented (OO) design. The following quotes attempt to define the basic idea behind OO design:

> *The first principle of object oriented programming might be called intelligence encapsulation: view objects from outside to provide a natural metaphor of intrinsic behavior.* (Rentsch et al., 1982)

> *The basic support a programmer needs to write object-oriented programs consists of a class mechanism with inheritance and a mechanism that allows calls of member functions to depend on the actual type of an object (in cases where the actual type is unknown at compile time).* (Stroustrup, 1991)

> *The object-oriented approach combines three properties: encapsulation, inheritance, and organization.* (Nguyen, 1986)

In basic terms rather than use a procedure or structured programming format, OO design works by encapsulating related methods and variables within a single code entity, i.e., an object. Hostetter (Hostetter, 2002) gives a useful description of the evolution of OO languages. One of the first such languages was SmallTalk, but the first widespread use of OO came in 1983 when the first version of C++ was released. More features have been continually added, until an ISO standard version of C++ was agreed, in 1998.

1.7.7 Java and OO Design

Java was originally designed for programming embedded systems (e.g., smart toasters). Because of this, the ideas of platform independence and run-time safety are key aspects of its design. However, it is also completely object-oriented in every respect. The Java online tutorial provides an excellent definition of the OO approach:

> Real-world objects share two characteristics: They all have state and behaviour. For example, dogs have state (name, colour, breed, hungry) and behaviour (barking, fetching, and wagging tail). Bicycles have state (current gear, current pedal cadence, two wheels, number of gears) and behaviour (braking, accelerating, slowing down, changing gears). Software objects are modelled after real-world objects in that they too have state and behaviour. A software object maintains its state in one or more variables. A variable is an

item of data named by an identifier. A software object implements its behaviour with methods. A method is a function (subroutine) associated with an object.

Definition: An object is a software bundle of variables and related methods.

http://java.sun.com/docs/books/tutorial/java/concepts/object.html

From the object approach we get two key benefits:

Modularity: The source code for an object can be written and maintained independently of the source code for other objects. Also, an object can be easily passed around in the system.

Information hiding: An object has a public interface that other objects can use to communicate with it. The object can maintain private information and methods that can be changed at any time without affecting the other objects that depend on it.

On balance the advantages of Java as a research and development language for EA systems are therefore very strong. Once a useful solution has been created via an EA method, it can be translated into a fully compiled language if maximum speed is an issue in its deployment. The applications in Chapter 5 will also make reference to particular features available in Java, which make it an ideal language for EA development. Future developments in computer languages may lead to a better alternative to Java, such as improved speed of operation. One example is the recent release of C# from Microsoft, which shares many of the more advanced features of Java, such as a clean object-based design.

1.8 Summary

The availability of new high-level languages and high-speed desktop machines now makes the process of learning and implementing evolutionary algorithms considerably easier for the student. This chapter introduced the essential ideas behind some common machine learning algorithms and some of the generic problems in this domain. As with any tool it is important to know when EAs are the best approach to a particular computational problem.

Chapter 2 presents some useful background material taken from natural evolution and biology and attempts to illustrate how these principles have inspired the process of artificial evolution.

Chapter 3 provides a detailed overview of the basic genetic algorithm. It also covers features of the principal operators and methods for selecting which operator and representation scheme to choose for a specific application. Chapter

4 provides an introduction to Genetic Programming with some analysis of its relative merits in comparison to GA techniques. Chapter 5 then describes a detailed example using GA methods to solve two classes of engineering problem. The first is based on image processing and the second on controller design for a mobile robot. Chapter 6 reviews some possible future directions for EA techniques in general, covering both software and hardware issues. Chapter 7 offers a summary of the application of EA and GA methods in particular.

However, the central aim of this text is to encourage readers to experiment with the included source code and applications in order to appreciate the practical potential of genetic algorithms. EAs are an exciting domain of machine learning and require a broad, multidisciplinary approach in application and comprehension. The referenced texts provide greater theoretical coverage and applications, but hopefully this volume will inspire students to seek a deeper appreciation of the field.

Further Reading

Goldberg D., *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley, ISBN: 0201157675, 1989.

Haupt R.L. & Haupt S.E., *Practical Genetic Algorithms*, John Wiley & Sons; ISBN: 0471188735, 1998.

Holland J., *Adaptation in Natural and Artificial Systems: An Introductory Analysis With Applications to Biology, Control, and Artificial Intelligence*, Bradford Books; ISBN: 0262581116, reprint 1992.

Kosko B., *Neural Networks and Fuzzy Systems, A Dynamical Systems Approach to Machine Intelligence*, Prentice Hall, 1992.

Koza J.R., *Genetic Programming*, Cambridge, MIT Press, 1992.

Mitchell M., *An Introduction to Genetic Algorithms*, Complex Adaptive Systems Series, MIT Press; ISBN: 0262631857, reprint 1998.

Mitchell T., *Machine Learning*, New York, McGraw Hill, , 1996.

Russell S.J. & Norvig P., *Artificial Intelligence: A Modern Approach*, Prentice Hall, ISBN: 0131038052, 1994.

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

2
Principles of Natural Evolution

> It is raining DNA outside. On the bank of the Oxford canal at the bottom of my garden is a large willow tree, and it is pumping downy seeds into the air. The whole performance, cotton wool, catkins, tree and all, is in aid of one thing and one thing only, the spreading of DNA around the countryside. Not just any DNA, but DNA whose coded characters spell out specific instructions for building willow trees that will shed a new generation of downy seeds. Those fluffy specks are, literally, spreading instructions for making themselves. It is raining instructions out there; it's raining programs; it's raining tree-growing, fluff-spreading, algorithms. That is not a metaphor, it is the plain truth. It couldn't be any plainer if it were raining floppy discs.
>
> Richard Dawkins, *The Blind Watchmaker*, 1986

2.1 Natural Selection

This chapter first provides a brief overview of naturally evolving systems, as these form the source of inspiration for evolution within artificial systems. We then consider the abstracted processes from nature, which form the basis of most evolutionary algorithms. It is, however, important to note that software-based EAs are not directly modeled on actual mechanisms in biological evolution, which are vastly more complex than current software or hardware could accommodate. The goal in computer-based evolution has rather been to use the broad principles within natural evolution of fitness-based selection and recombination, in order to automatically generate a phenotype (i.e., program) suited to a specific problem.

2.1.1 Genes and Chromosomes

Charles Darwin (Darwin, *On the Origin of Species*, 1859) first proposed that there are four essential requirements for the process of evolution to occur:

1. Reproduction of some individuals within a population.
2. A degree of variation that affects probability of survival.

R. Ghanea-Hercock, *Applied Evolutionary Algorithms in Java*

3. Heritable characteristics, that is, similar individuals arise from similar parents.
4. Finite resources, which drive competition and fitness selection.

The consequence of these processes is the gradual adaptation of the individuals in a population to the specific ecological niche they occupy. This can therefore be viewed as a form of long-term learning by a population, of the characteristics suited to their particular environment.

The second major discovery, which helped form the modern understanding of genetics, was the discovery by Watson and Crick in 1953 of the double helix nature of DNA and the mechanisms of DNA replication. The success of Darwinian theory inspired a number of early computer scientists to propose that the same principles could be applied to automated machine learning (Friedberg, 1958).

In order to understand the particular mechanisms of interest, we first need to review the theoretical basis of natural evolution. (The bibliography provides references with far greater coverage of this subject and should be consulted for a serious review of the themes in this chapter.)

2.1.2 Biological Genes

The primary unit on which natural evolution operates is defined as the gene. However, within biology the concept of a gene is surprisingly difficult to precisely define. A working definition is, "unit of hereditary information that occupies a fixed position (locus) on a chromosome. Genes achieve their effects by directing the synthesis of proteins" (www.Britannica.com).

In general, a gene is a sequence of DNA segments that acting together via transcription code for a single protein or polypeptide (subcomponents of proteins). However, the sections of DNA active in the transcription process into proteins are also separated by long sequences of "junk" DNA which are not known to play any active role in the coding of proteins.

Each possible alternative expression of a specific gene sequence is known as an allele; an example is the gene variation that codes for blue or brown eyes.

2.2 DNA Structure

The building blocks of a strand of DNA are the four nucleic acid bases adenine, guanine, cytosine, and thymine, that is, A,G,C,T. A single strand of DNA is composed of the four bases in a tightly linked chain. Each base is chemically bonded to another under the fixed rule A > T and C > G. These base pairs then bind via weak links to a second chain in a complex three-dimensional strand, forming the famous double helix of DNA.

The sequence of bases in a DNA strand can be decoded into a sequence of amino acids. The DNA is read three bases at a time. Each triplet of bases is

known as a codon. Each codon then codes for the production of a specific amino acid; for example, AGA codes for Arginine. However, some codons code for the same amino acid, such that only 20 are actually used by DNA. This results in a significant amount of redundancy in the possible coding process, since four bases in a three-letter grammar allows for up to 64 possible expressions.

Three of the possible sequences are used to code for special stop signals, which are used during the transcription process to indicate new sequences in the DNA. The process of decoding DNA into a protein requires two intermediate stages – transcription and translation – which are described next.

2.2.1 Transcription – from DNA to RNA

Transcription is the process of converting a nucleotide sequence of DNA into RNA, which then forms part of the protein synthesis mechanism. Transcription involves the synthesis of an RNA molecule, which is complementary in its base sequence to the coding strand of the DNA duplex. An RNA polymerase (an unzipping enzyme) binds to the DNA duplex, causing a local unwinding of the helix. The RNA molecule grows by polymerisation of nucleotides, whose sequence is set by the base pairing to the coding strand of the DNA template. This proceeds until a stop signal is encountered and the new RNA molecule is then released.

2.2.2 Translation – from RNA to Protein

The second phase in the creation of the polypeptide subcomponents of the required proteins is translation. This is a complex process involving the growth of a polypeptide chain, which is specified by the order of nucleotides in the RNA strand. The process is mediated by the ribosomes, which serve as the site of protein synthesis during translation.

The ribosomes can be thought of as precision-engineered nanoscale machines, which assemble new proteins. They are highly complex structures whose precise nature is still being deduced by molecular biologists.

The processes of transcription and translation may appear to be quite abstract with respect to developing computer-based evolutionary algorithms; however, they are highly efficient and robust mechanisms that could aid in the design of EA. However, they have rarely been considered within the EA research community.

2.2.3 Genotype

The total genetic description of an organism is defined as the genotype or genome. Each parent in a sexually reproducing organism donates half of each genotype. It is the genome that is acted on by the genetic operations of mutation and recombination. The phenotype of an organism is the set of observable properties or traits it possesses and is determined by expression of its genotype.

The process of natural selection is the preferential selection of specific phenotypes resulting in adaptations to an environment.

Since those organisms that survive are those responsible for the next generation, well-adapted phenotypic traits will be spread through a population. (A fascinating text on why evolution is actually a universally applicable process is described in Richard Dawkins key work, *The Selfish Gene* (Dawkins 1989, 2nd ed.)

2.2.4 No Lamarckianism!

A further aspect of all biological evolution is that all phenotypic traits are transmitted via expression of a specific gene or set of genes within the genotype. It is not the case (except in a few rare exceptions which prove the rule) that changes in an individual's phenotype directly affect the genotype it transmits to its offspring (a theory known as Lamarckianism; Lamarck, 1815).

However, it is certainly possible to utilise Lamarckian selection within a computer-based EA, (Sasaki & Tokoro, 1998). A number of researchers have tried this method, and it is an option that can be used in addition to the normal Darwinian model of selection.

2.2.5 Evolution and Variation

There is a fundamental tension within natural evolution centered on the conflict between the need to maintain the transmission of useful traits from parent to offspring and the need to allow for the emergence of new, better-adapted traits.

Recent evidence on the frequency and scale of extinction events in the Earth's history point to the consequences of failures in evolution to allow sufficient adaptation (Erwin, 2000). We first need to consider the mechanisms by which natural selection maintains the accurate copying of one generation's genotypic traits to the next.

2.2.6 Redundancy

As previously mentioned, the base pair coding in the DNA codons leads to redundancy in the formation of amino acids by transcription. For example, there are two codons for histidine and six for leucine. The key benefit of this encoding redundancy is to alleviate the consequences of single point random mutations, when one base pair is incorrectly replaced by another. Due to the redundancy there is a high probability that the new codon will express the same amino acid, resulting in no modification to the organism's phenotype.

2.2.7 Self-Maintenance

Within the DNA molecule itself there are also processes that allow for the correction of copying errors. DNA therefore has a very high fidelity rate, that is, a low frequency of copying errors.

2.2.8 *Evolvability*

In order for an organism to evolve, there must be a means for modifications to the genotype to occur. Recent work has considered this issue using the theme of the "evolution of evolvability" (Wagner and Altenberg, 1996, and Altenberg, 1994). They nicely summarise the relevance of this process to EA:

> In evolutionary computer science it was found that the Darwinian process of mutation, recombination and selection is not universally effective in improving complex systems like computer programs or chip designs. For adaptation to occur, these systems must possess "evolvability," i.e. the ability of random variations to sometimes produce improvement. It was found that evolvability critically depends on the way genetic variation maps onto phenotypic variation, an issue known as the representation problem. (Wagner and Altenberg, 1996.)

The important issue of representation in EA is therefore addressed in Chapter 3. Several basic specific mechanisms for variability will first need to be presented, as follows.

2.2.9 *Mutation*

An obvious way in which a genotype may be modified is if some process modifies part of the actual DNA sequence of base pairs such that different codons are expressed. This may be due to several factors, such as the presence of chemical mutagens, radiation, or pathogen activity. There are three major types of mutation that can occur, having significantly different consequences for the resulting organism.

Point or base-pair switch mutation: A single base pair (e.g., A-T) is replaced with C-G. These occur with some small probability due to copying errors, typically every several million replications. Presuming that codon redundancy fails to cover the error, then some small phenotypic effect may result. Such mutations are therefore very useful in allowing the incremental development of an organism's phenotype to find useful adaptations.

Frame-shift mutations: A more serious class of mutation is known as a frame-shift. This may be a + or – frame-shift mutation, that is, a new base pair may be inserted or deleted from the DNA sequence resulting in a scrambling of the coded sequence from that point on in the DNA. Clearly, if a significant length of the DNA has been modified, then major changes in the expressed phenotype will occur, usually with strongly negative consequences for the organism.

Large-scale sequence mutation: It is also possible for large sections of a genome's DNA to be rearranged, which would normally be fatal to the organism. This may be due to serious radiation damage or a failure of the cell's normal replication systems.

Mutation rates are normally very low, typically one every several million to maybe one per billion replications of the DNA base-pair material. Also, most mutations are detrimental to the operation of the host cell or are neutral in effect. In contrast, most EAs typically use a mutation rate of less than 1%, (0.001-0.1), which are orders of magnitude higher. However, it is debatable whether any direct comparison of artificial and natural mutation rates can be made given the vastly different replication and recombination mechanisms involved.

2.2.10 Sexual Recombination

An obvious source of variability in natural organisms is the process of offspring creation from sexual reproduction. This is also a commonly imitated method within many EA systems.

During the sexual reproduction of two parent individuals DNA from each parent is combined to generate a new child organism. This is referred to as homologous genetic exchange. In homologous genetic exchange sequences of DNA are exchanged such that the function of DNA transcription segments is preserved and the length of both DNA molecules. Unlike a mutation operation, this recombination is a precisely controlled process. It is therefore dependent on an interchange between organisms of the same species, which share virtually identical DNA.

Other forms of genetic exchange are also possible such as transposon recombination that occurs in some bacteria. In this case intact gene sequences are inserted into the receiving bacteria.

The text by Banzhaf (Banzhaf et al., 1998) provides a more detailed overview of how EAs relate to natural evolutionary mechanisms, with an emphasis on Genetic Programming.

2.2.11 Nonselectionist Issues

There are also a number of important evolution-related processes, that fall outside commonly described genetic mechanisms. Two of these will be considered in order to present a balanced perspective on modern biology with respect to the evolution of organisms.

2.2.12 Epigenesis

This is generally defined as

> The concept that an organism develops by the new appearance of structures and functions, as opposed to the hypothesis that an organism develops by the unfolding and growth of entities already present in the egg at the beginning of development (preformation). (Development in which differentiation of an individual's cells into organs and systems arises primarily through their interaction with the environment and each other.) (King, 1990)

This concept expresses the idea that some of an organism's phenotypic features may be more a function of environmental or physical processes than genetic mechanisms. It may therefore be of some use in EA but has been overshadowed by the dominance of Darwinian genetics.

2.2.13 Dynamics and Morphogenesis

An alternative perspective on the role of genetics in the evolution of organisms also exists within biology, which advocates the role of development and morphogenesis in parallel with genetic operators. The prime advocate of this position is Brian Goodwin (Goodwin, 1997), who clearly explains the idea:

> in an extended view of the living process, the focus shifts from inheritance and natural selection to creative emergence as the central quality of the evolutionary process. And, since organisms are primary loci of this distinctive quality of life, they become again the fundamental units of life, as they were for Darwin. Inheritance and natural selection continue to play significant roles in this expanded biology, but they become parts of a more comprehensive dynamical theory of life, which is focused on the dynamics of emergent processes. (Goodwin, 1997, p. xiii)

Such a position is also advocated in work by Kauffman (Kaufffman, 1993) and forms a potentially useful alternative to the ultra-Darwinian position adopted by many biologists. Such ideas have inspired a number of researchers in complex adaptive systems, in particular those seeking to design complete autonomous agents, such as mobile robots (Cliff & Miller, 1994), in which the role of a physical or simulated phenotype is of importance. (In contrast, most EA research focuses on the role and behaviour of the simulated genotype.)

2.3 Summary

The sheer range and scale of natural evolution, have inspired computer scientists to abstract evolutionary mechanisms into the computational domain. While some success has been achieved from applying the broad principles of mutation, recombination, and selection, a great deal more remains to be learned from natural genetic mechanisms. The power of transcription and translation in converting base-pair sequences into complex three-dimensional proteins is one

example that could be fruitfully explored. The exact role of mutational processes and their relationship to the rates of evolvability also need investigation in the context of EA.

However, as pointed out by Fogel (Fogel, 1995), a simplistic mapping of low-level DNA operations into computational systems may actually lead to a reduction in algorithm performance. Clearly, any further developments in our understanding of biological evolution may be of assistance in designing EA, but it is important to consider that computational systems are still an artificial environment with a distinct physics and geometry. However, in Chapter 6 the emergence of hardware-based synthetic evolution with embodied phenotypes raises an entirely new set of questions with respect to imitating natural evolution. Chapter 3 considers exactly how the principal evolutionary mechanisms can be applied within the genetic algorithm model.

Further Reading

Darwin C., *On the origin of Species by Means of Natural Selection or the Preservation of Favoured Races in the Struggle for Life*, London, Murray, 1859. (Current ed., Wordsworth Editions Ltd., 1998.)

Dawkins R., *The Blind Watchmaker*, Penguin Books, new ed., 2000.

Dawkins R., *The Selfish Gene*, 2nd ed., Oxford Paperbacks; ISBN: 0192860925, 1989.

Goodwin B., *How the Leopard changed its spots, the evolution of Complexity*, Phoenix Edition, 1997.

Maynard-Smith J., *Evolutionary Genetics*, Oxford, Oxford University Press, 1994.

3
Genetic Algorithms

> In the beginning the Universe was created. This has made a lot of people very angry and been widely regarded as a bad move.
>
> Douglas Adams, *Hitchhiker's Guide to the Galaxy*

3.1 Genetic Algorithms

There is a single dominant evolutionary algorithm that encapsulates many of the biological evolutionary mechanisms outlined in Chapter 2. This is the standard genetic algorithm first described by John Holland (Holland, 1975). As we will describe later there are numerous variations of the basic GA, but few (with the possible exception of genetic programming) represent a significantly new methodology in EA. In the literature however, GA is frequently used as a generic term, which incorporates many evolutionary algorithms. In the general sense a genetic algorithm is any population-based model that includes selection and recombination operators. This chapter outlines the common operators used by GA, a brief introduction to GA theory, and a consideration of which operators and representation scheme to select for a given problem.

3.2 GA Basics

The defining characteristics of a Holland-type GA are a bitstring representation, a proportional selection method, and crossover variation. As pointed out by Eshelman (Eshelman, p. 64, Back et al., 2000) it is the emphasis on crossover, which has been adopted by most following EA methods. The rationale is related to Holland's schema hypothesis, which is discussed later.

3.2.1 Fitness and Evaluation Functions

In most EAs once a population has been initialised each member must be evaluated and assigned a fitness value f_i. The evaluation function provides a

R. Ghanea-Hercock, *Applied Evolutionary Algorithms in Java*

measure of the individual chromosome's performance with respect to some set of parameters. In the robot example in Chapter 5 this translates into a performance metric for an autonomous robot, based on navigation and obstacle avoidance. A fitness function may then translate the assigned performance value into the reproductive opportunities for the individual. For a real-world problem, defining the evaluation and fitness functions can be a major component in resolving a problem, that is, creating a GA capable of solving it. A detailed knowledge of the problem domain is therefore required in order to select the best parameters for optimisation via a GA (Grefenstette, 1986).

The following sequence is a common starting point for most GA-class algorithms:

1. Create a randomly generated population of N chromosomes, each of some length m bits.

2. Test each chromosome (i.e., a possible task solution) within the problem space and assign a measure of fitness $f(x)$.

3. Selection phase: Select a pair of chromosomes from the population with probability based on their fitness.

4. Apply a set of genetic operators to the two parent chromosomes: With some crossover probability p_c, apply crossover at some randomly selected point along each chromosome.

5. Apply mutation to each new chromosome with a probability p_m.

6. Place the new chromosomes in the new population.

7. Replace the old population with the new population, or we may use a steady-state method, in which an overlapping population model is maintained (a useful comparison is in Rogers & Prügel-Bennett, 1999).

8. Test if target termination criteria is met, such as a specified best fitness value; else repeat from step 2.

Each loop of the sequence, illustrated in Figure 3.1, is termed a generation. The central concept of the GA is the *chromosome,* which is the encoding of information in a string of symbols (usually binary digits). These strings can be manipulated by a set of genetic operators. Using the process of fitness proportional selection, the chromosome strings, which encode a potential solution to the specified task or function, evolve toward an improved solution.

In general, Holland defines a GA as being composed of a problem E, the population of encoded problem solutions:

A (i.e., chromosomes) and the set of genetic operators Ω that modify A according to a plan τ of information.

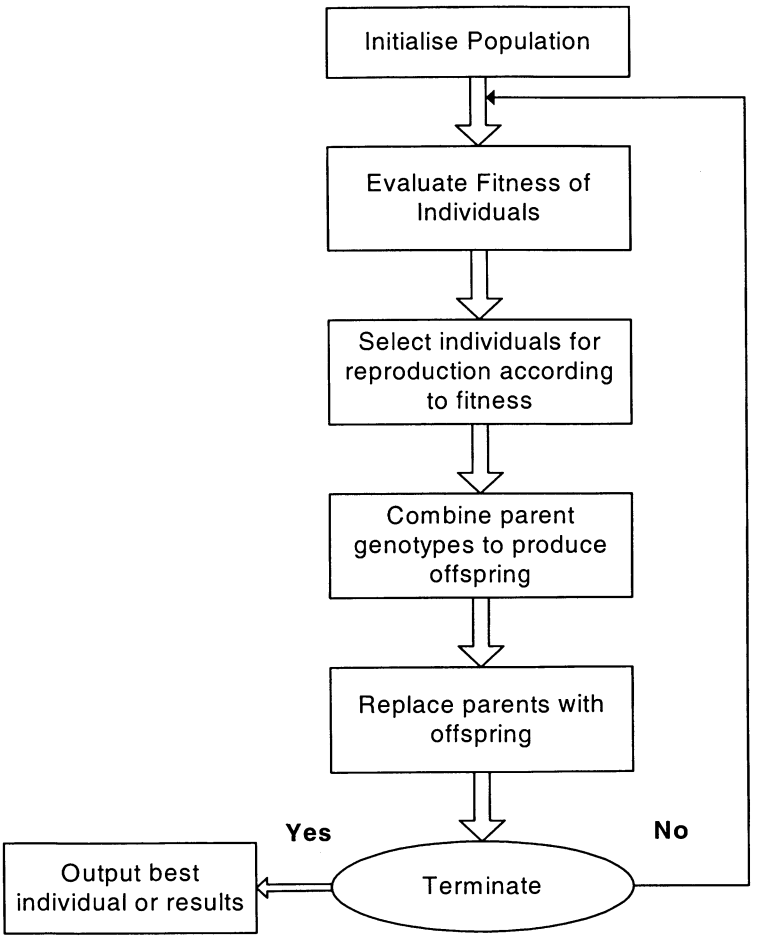

FIGURE 3.1 GA sequence of operations.

The GA works by creating a set or *population* of chromosomes where each member string is an *individual.* An individual is assigned a real number, which represents its fitness value, defined by the particular problem function. The objective of the GA is then to maximise this fitness function for each individual.

The process begins by assigning random fitness values to every individual in the initial population and then creating a new population using a selection mechanism that is proportional to the fitness of the individual [Eq.(3.1)].

$$P_{select}(n) = \frac{f(n)}{\sum_{k=1}^{pop} f(k)} \tag{3.1}$$

Where n is the nth string, p = total number of strings in the population, and $f(n)$ is the fitness of the nth string. The selected individuals are then copied into the next generation using the set of genetic operators, normally composed of mutation and crossover. However, a large number of variations exist for both the basic selection process and the operators.

3.3 GA Theory

Most GA analysis assumes the representation scheme is binary in order to simplify the theoretical model. In effect when we use a GA we are seeking a particular bit sequence or group of closely related sequences from the possible search space formed by bit strings of the specified length L. The size of the search space is 2^L and forms a hypercube of L dimensions. Since a useful bit string chromosome is typically of length >30, then the associated search space is very large: ~ 10^9. However, with strings of length >100, the space grows to 10^{30} which is an immense number. Hence the difficulty of searching the problem space grows as a power of the length of the bit string (i.e., the GA genome).

A large number of papers have covered the theoretical aspects of GA Holland, 1975; Goldberg, 1989; and Whitley, 1993). The common view of how a GA works is that the processes of recombination, mutation, and selection act to group useful substrings within the chromosome into "building blocks", which are then gradually combined to create the final solution. Holland (Holland, 1975) first introduced the "building block hypothesis," which he termed schemas or schemata. A schema is defined as a set of bit strings that comprise a template of binary 1's 0's and wild cards *.

A useful notation was introduced by Goldberg (Goldberg, 1989) to represent a hyperplane H, an example being H = 1***01. Example strings that match this schema would be 100001, or 111101. Since this schema has two defining bits, it has an *order* of 2. The power of the GA lies in the fact that an *implicit parallelism* (Holland, 1975) is contained in the schema sampling process, that is, the GA operates by implicitly sampling hyperplane sections of the search space, while explicitly evaluating the fitness of n strings in the population.

The schema description also highlights a difference in the variation effects of mutation and crossover, that is, in mutation new variations are introduced uniformly across the search space, while under crossover the variations are constrained to be in the vicinity of the hyperplanes occupied by schemata from the parent individuals.

Hence in order for the building block hypothesis to operate, the crossover process needs to both preserve and propagate good schemata and recombine them with other fit schemata. However, to achieve this constructive linking of schemata may require a high level of disruptive recombination which destroys the underlying useful schemata that have already been discovered. Holland's original theory on schema assumed that important building blocks could be discovered of relatively short length. However, it has been pointed out (Eshelman, p. 64, 2000, ed. Back et al.) that there is no a priori reason why real-world problems can be represented such that useful building blocks will be of short length.

3.3.1 Deception

A related issue to the preservation of useful schema is the concept of *deception* (Goldberg, 1997). This is defined such that a problem is deceptive if given two incompatible schema A and B, the average fitness of A is greater than B, even if B includes a string with greater fitness than any in A. The result is that the lower-order building blocks tend to lead the GA away from any global optimum.

3.3.2 Messy Genetic Algorithm

A significant alternative to the standard GA, which attempts to resolve some of the problems with the building block hypothesis, is the Messy GA (MGA) (Goldberg et al., 1989). In MGA variable-length strings represent individual chromosomes. The description of the GA comes from the fact that some bit positions may be under or overspecified if the string is too short or too long. The MGA therefore uses "moving-locus" representations for genomes, such that as genes are moved during recombination useful schema remain intact. MGA uses a two-phase commitment process in which the first phase aims to use genomes of limited commitment, followed by a second stage of full commitment.

A useful analysis of the MGA, which identifies the value of adopting variable-length genomes, is recent work by Watson and Pollack (Watson & Pollack, 1999). They extended the MGA concept of using a two-phase commitment operation and specify a more generalised "Incremental Commitment" GA. In this ICGA model partial commitment and recombination operations are applied throughout the operation of the algorithm.

Significantly more detailed descriptions of the normal GA mechanisms can be found in Mitchell's text (Mitchell, 1996), which offers a useful introduction to the processing of schema within a GA. Whitley (Whitley, 1994) also offers a clear description of the nature of hyperplane sampling in the simple GA.

However, the schema theorem is not without controversy and debates have focused on the degree to which crossover operators disrupt or preserve useful schemata (Grefenstette, 1991, 1993).

3.4 GA Operators

3.4.1 Mutation

The simplest GA operator to understand is mutation. Mutation is a mechanism where a randomly selected gene within the chromosome is replaced with an alternative allele. In the case of a binary representation scheme this amounts to flipping n bits. In the examples in Chapter 5 an integer-valued representation is used and the mutation operator then selects a suitable value from a prespecified viable range of values.

Within the research community there are ongoing debates regarding the value of mutation as a genetic operator and the relative rate of mutation. Most researchers agree that mutation rates should be low, with probabilities ranging from 0.001 to 1%, typically at the lower end of this range, less than 0.05%. However, some argue that mutation can be used as the sole source of variation with no recombination (Mathias and Whitely, 1994).

A common perspective is that mutation is primarily a secondary operator and acts to replace or regenerate bits (or genes) lost during the crossover process. At best mutation can help move a chromosome away from local optima by injecting new genes into the population of chromosomes. Recent work has considered the relative importance of mutation versus crossover (Hinterding et al., 1995). It is interesting to compare that in the field of natural evolution mutation is considered an essential process in allowing organisms to evolve by enabling new variants to appear in a population.

It would be useful to investigate whether more accurate models of biological mutation processes would be of benefit within EA, for example via a frame-shift mutation mechanism, in which subsections of a chromosome have new genetic sequences inserted or deleted. Of course, such a process would have a large-scale impact on the final phenotype, but increased variability may be worth the effect in some task domains. As Hinterding et al., point out, if we utilise a binary coding, then bit flipping mutation operators are acting at a subgene level: "We argue this effect (mutation) is more like that of a nucleotide within a gene, than a gene within a chromosome" (Hinterding et al., 1995.)

Hence the representation scheme used within a particular GA can also have a critical effect on the power of the mutation operator. Finally, a particular aspect of many GAs is when the algorithm has operated for a number of generations without converging on a satisfactory solution. The population is then said to be prematurely converged. Mutation can therefore operate, as a means of reintroducing novel genes into the converged population which crossover alone

would be unable to achieve. It is also the case that mutation alone can act as a hill-climbing search method (Whitley, 1994).

Finally, the selection point chosen for a mutation operator to act on can have a specified probability distribution, rather than a purely random distribution. One common method uses a Gaussian distribution function to select the element of the current chromosome for mutation. One reason for this nonuniform selection may be that different sections of the chromosome have a biased weighting effect on the fitness evaluation of the GA. This can be particularly true of a binary representation (Michalewicz, 1999).

3.4.2 Crossover

In a simple GA the crossover operator works by selecting two parent individuals from the population with a fitness-dependent probability, and swapping sections of each individuals chromosome.

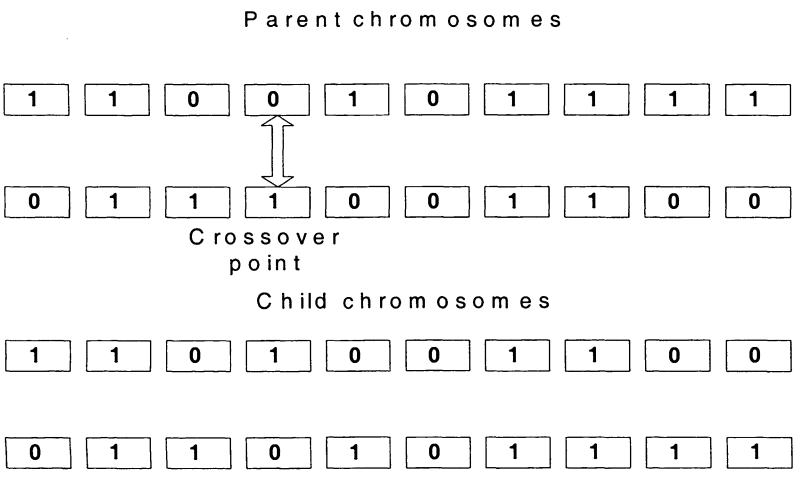

FIGURE 3.2 Example schematic of single-point crossover.

A common method is to randomly select a single crossover site on each individual and swap over the remaining chromosome sections from each parent, (i.e., single-point crossover). Figure 3.2 provides a basic illustration of single-point crossover.

Single-point crossover suffers from one particular disadvantage, which relates to the schema theorem discussed in Section 1.2, namely the maintenance of useful schema can be degraded as genes or bits that are not co-located along the schema may be disrupted.

As with most GA mechanisms the optimal crossover method is dependent on the application domain and the representation scheme. It is therefore a matter for further research, and the relationship between schema optimisation and preservation is still not understood.

3.4.3 Multipoint Crossover

One obvious solution to the problem is to use a multipoint crossover mechanism. In this case multiple crossover points are randomly selected, as in Figure 3.3.

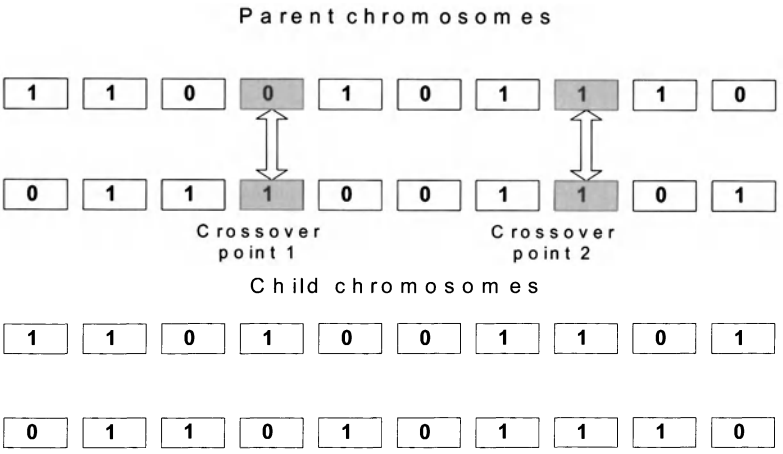

FIGURE 3.3 Multipoint crossover.

A third alternative is to apply a uniform crossover method, in which offspring individuals are created from a randomly generated uniform bit mask. This has been studied by Ackley (Ackley, 1987) and has the advantage of being unbiased with respect to the length of a schema but is clearly going to be more disruptive than a single-point crossover (Whitley, 1994). However, with small populations it may be the case that a greater disruptive effect is essential to overcome convergence of the population (Spears & De Jong, 1991).

3.4.4 Selection

Once we have a population of individuals with assigned fitness values, the next step is how to preferentially select a subset of individuals that should survive into the next generation.

The purpose of selection is therefore to increase the frequency of fitter individuals within the population over repeated generations. However there is always a pressure between exploiting the population through selection and exploring the search space via crossover and mutation (see Figure 3.4).

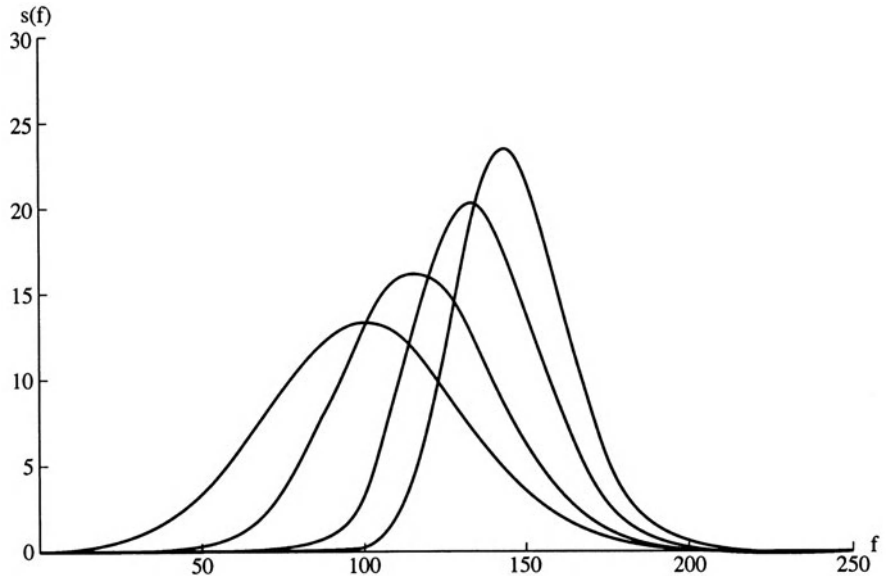

FIGURE 3.4 Illustrates the effect of applying a selection function on the distribution of fitness in a population. This also indicates the effect of loss of genetic diversity as the range of fitness is reduced, (Reproduced from Blickle & Thiele, 1995).

It is an intuitive process summed up by the Darwinian maxim:

> . . . [C]an we doubt . . . that individuals having any advantage, however slight, over others, would have the better chance of surviving and of procreating their kind? This preservation of favourable individual differences and variations, and the destruction of those which are injurious, I have called Natural Selection, or the Survival of the Fittest.

(Darwin, *On The Origin of Species*, 1859)

Excessive selection will lead to fit but suboptimal individuals taking over the population before a target solution is found. It is then difficult for the population to recover sufficient diversity to explore the remaining search space. If the selection pressure is too weak, however, the rate of evolution will fail to

converge on a useful solution (real-world problems always have finite resources available).

As with other GA operators a wide range of selection processes has been researched (Whitley, 1994; Goldberg & Deb, 1991). Work by Blickle and Thiele (Blickle & Thiele, 1995) provides a detailed comparison of some common selection methods. A useful definition of the fitness distribution within a population is as follows:

Definition: (Fitness distribution) The function s: $\mathbf{R} \longrightarrow \mathbf{Z}^+$ assigns to each fitness value $f \in \mathbf{R}$ the number of individuals in a population $P \in J^N$ carrying this fitness value. S is called the fitness distribution of a population P (Blickle & Thiele, 1995).

3.4.5 Fitness-Proportionate Selection

In this method the number of times an individual is allowed to reproduce is equal to the individual's fitness divided by the average fitness from the population, namely fitter individuals have more chance to generate offspring. A very popular method for implementing this is termed "Roulette Wheel" selection (see Figure 3.5) The steps of the roulette wheel method are as follows:

1. Sum the total fitness values of the individuals in the population, where sum = T.
2. Repeat N times, for population of size N.
3. Generate a random integer r between 0 > T.
4. Step through the population and sum the fitness values until the current sum is >= r. The individual whose value shifted the sum over the limit is the one selected.

A pseudo-code example is given below:

```
/* Select an individual for the next generation in proportion to its
contribution to the total fitness of the population */
/* Assumes fitness values in global array fitness */
/* Returns a single selected individual from global array*/

int select (real sum_of_fitness_values) int index;
index = 0; sum = 0.0;
/* Create a random number between 0 and fitness total */
r = rand() * sum_of_fitness_values;
do index++;
sum = sum + fitness[index];
while (index < SIZE-1) and (sum < r);
return index;
```

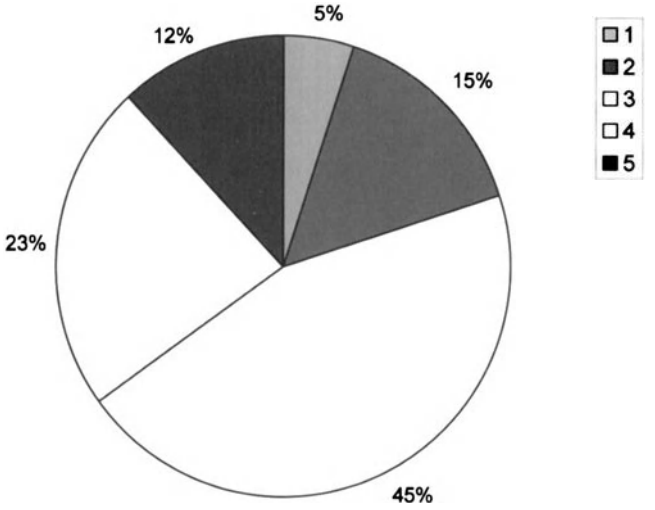

FIGURE 3.5 Graphical representation of roulette wheel selection, where the probability of selection is proportional to an individual's fitness, against the total fitness of the population (Goldberg, 1989).

3.4.6 Disadvantages of Fitness-Proportionate Selection

A common problem with this method is that the initial fitness variance in the population is high and the fittest individual will rapidly take over the population, at the expense of genetic diversity, hence leading to strong premature convergence.

3.4.7 Rank Selection

Rank based selection (Baker, 1985 and Grefenstette & Baker, 1989) was designed to overcome the disadvantages of fitness-proportionate methods. In this scheme individuals are sorted according to their fitness value and a rank N-1 is assigned to the best individual and a rank of 1 to the least-fit individual. A selection probability is then linearly assigned to each individual according to the rank value. The principal effect is to avoid assigning too large a proportion of each generation's offspring to a few fittest individuals. However, it maintains sufficient selection pressure when the fitness variance in the population is low, hence removing the need for fitness scaling. Rank selection is therefore a simple and efficient selection method.

Linear Ranking

In this form, a selection probability is assigned to an individual that is directly proportional to the individual's rank. In this case the rank of the least-fit member is zero and the most fit is assigned a rank of n -1, for a population of n individuals.

Nonlinear Ranking

Under nonlinear ranking selection probabilities are assigned based on an individual's rank, biased by some nonlinear function. A simple example would be to take probabilities proportional to the square of the rank. However, as increasingly nonlinear functions are applied, the fittest individuals will once again dominate the population in early generations, leading to premature convergence and the associated loss of genetic diversity.

Disadvantages of Rank Selection

Clearly, by reducing the strength of the selection pressure, even if adaptively, the convergence of the GA to its solution may be slowed.

3.4.8 Tournament Selection

Another popular selection method is tournament selection (see Goldberg & Deb, 1991). The name conveys how this process works, that is, a tournament is repeatedly held in which N individuals are selected from the current population and the fittest individual is copied into the intermediate population (this may be with or without replacement). The process is repeated until a new population is created.

Typically the size of the tournament group is a low value, between 1-5. It is relatively obvious that as the tournament size increases an increasing percentage of genetic diversity will be discarded, and we return again to the issue of premature convergence.

This effect is illustrated in Figure 3.6, where the loss of diversity is given by Eq.(3.2), for tournament size t. (Blickle & Thiele, 1995). As shown with a tournament size of only 5, we have already lost 50% of the population diversity.

$$f(t) := t^{\frac{-1}{t-1}} - t^{\frac{-t}{t-1}} \tag{3.2}$$

One major advantage over other selection methods, however, is, its computational efficiency as fewer operations on the population are required; for example, no fitness scaling is required. It is therefore particularly popular as a selection method in genetic programming.

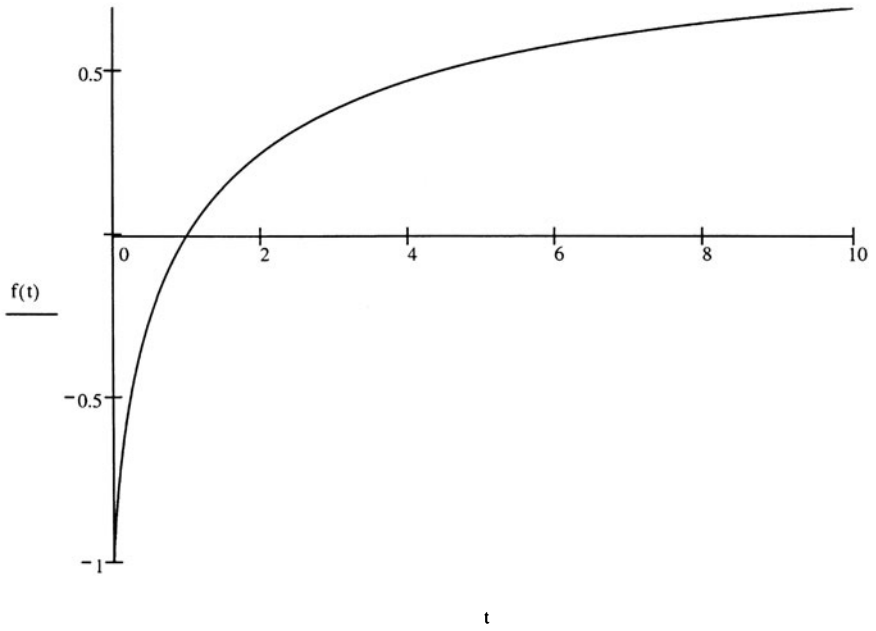

FIGURE 3.6 Graph showing the loss of diversity $p(t)$ under tournament selection (Blickle & Thiele, 1995).

3.4.9 Scaling Methods

With most selection methods as discussed the fitness variance of the population rapidly reduces as the algorithm converges. The resulting narrow range of fitness values makes differentiating between individuals increasingly difficult for the selection operators. One method to resolve this is to apply a scaling mechanism to the population, that is, the fitness values of each individual are reassigned according to some magnification factor. For example, in linear scaling if the fitness values have converged to a range of 1.91 to 1.95 with a target value of 2, then scaling would shift the lower value to some nonzero minimum, for instance, 0.05. This method is used in the algorithm for the application examples in Chapter 5. Of course some nonlinear scaling factor could be applied such as a power or exponential function. (See Forest, 1985, for a more advanced example termed "Sigma Scaling.")

3.5 Pros and Cons of Genetic Algorithms

There are numerous advantages in using a GA, such as not depending on analytical knowledge of the function to be optimised, robustness, intuitive

operation, and the ability to optimise over multiple parameter sets. The GA has therefore been an obvious candidate in search and optimisation problems. However, serious limitations with the standard GA have led many researchers to search for a more powerful evolutionary system. The disadvantages include

1. It is very expensive in computational resources.
2. It is a probabilistic algorithm.
3. It is frequently prone to premature convergence (i.e., sticking on local maxima in the fitness landscape).
4. It is often difficult to encode a problem in the form of a suitable chromosome (representation issue).

This last point (i.e., representation) has been the source of active research to discover alternative schemes and evolutionary strategies. Two significant alternatives are Messy GAs (Goldberg, 1989) and Genetic Programming (Koza, 1992). Other alternatives have also been developed, such as work by Leitch (Leitch, 1995), who uses a "context-dependent coded GA" that contains several features aimed at mimicking the operation of real genomes, and which attempts to alleviate the perennial problem of premature convergence. Chapter 6 reviews some of the more advanced methods currently being exploited to enhance the capabilities of the standard GA.

3.6 Selecting GA Methods

From this chapter it will hopefully be apparent that designing a suitable GA for a real-world task is a nontrivial exercise, almost an art. The interactions between representation, recombination, mutation, and selection are a complex balance between exploitation and exploration. In Chapter 6 an overview of adaptive techniques in GA methods is presented that aims to show how this balance can be maintained.

In general, if the search space of a particular application is multimodel and course grained, then a GA approach should outperform a gradient ascent algorithm (e.g., hill-climbing).

3.6.1 Encoding Choice

A key issue with most EA techniques is the choice of a suitable encoding scheme, namely whether to use binary, floating-point, or some grammar-based representation. Holland (1975) used the argument that a genome with a small number of alleles but long strings has a higher degree of parallelism than a numeric scheme with a larger number of alleles but short (floating-point) strings. However as Mitchell (1996) points out, for real-world applications it is frequently more natural to use a decimal or symbolic representation scheme, as

this is an easier mapping to the actual representation of the problem space; for example, the weights in a neural network.

The text by Michalewicz also offers a useful analysis of the relative merits of binary versus floating-point representations. The conclusion is that a floating-point scheme is faster, is more consistent between runs, and can provide a higher precision for large domain applications.

> The binary alphabet offers the maximum number of schemata per bit of information of any coding and consequently the bit string representation has dominated genetic algorithm research. This coding also facilitates theoretical analysis and allows elegant genetic operators. But the implicit parallelism does not depend on using bit strings and it may be worth-while to experiment with large alphabets. In particular for parameter optimisation problems with variables over continuous domains, we may experiment with real-coded genes together with special genetic operators developed for them. (Michalewicz, 1996)

A good example is the traveling salesman problem (discussed in Appendix B), where a binary chromosome representation suffers from several disadvantages such as bit sequences that do not correlate with any city. Alternatively, an integer representation is a more direct mapping to the problem domain.

A specific example provided by Michalewicz is to compare the functional and computational performance of a binary and floating-point chromosome at solving a dynamic control problem. The results for the CPU performance are given in Table 3.1. As the table shows, the performance of the floating-point representation is superior and scales better than the binary scheme.

However, as with many aspects of EA development the matching of operators and representation to the problem domain is critical and can impact the performance of the solution enormously.

The applications in Chapter 5 make use of a floating-point scheme, which is truncated, to an integer representation, as these map well onto the problem representations in fuzzy rules and image operator lookup.

No. of Elements	5	15	25	35	45
Binary	1080	3123	5137	7177	9221
Float	184	398	611	823	1072

TABLE 3.1 CPU time (sec) to solution, as a function of the number of elements for an evolved dynamic control problem (Michalewicz, 1999).

Michalewicz also looks at the use of an EA in designing a mobile robot path-planning algorithm, which provides useful background material to the second application in Chapter 5.

3.6.2 Operator Choice

Regarding which operators to use is a difficult question; however, some general guidelines can be offered. First, many applications benefit from a two-point crossover operator, which reduces the disruption of schemas with a long defining length. The choice of what mutation rate to use is also application-dependent and a practical alternative is to use an adaptive mutation rate which is parameterised within the genome of the EA. Alternatively, a meta-parameter system that allows self-adaptation of the mutation rate can be utilised (Bäck, p. 143 2000). The actual merit of using mutation at all remains a debated research issue, (see Grefenstette, 1986, and Hinterding, 1995).

3.6.3 Elitism

Another very common technique used in many GA systems is to retain the best performing individual (or several of them) in each generation and to add this to the next generation (De Jong, 1975). This can give a significant performance increase to the GA, although with the risk of reducing diversity.

In most cases, however, the performance gain over the whole GA run is substantial and it provides the benefit of ensuring that the final best solution is from the entire run and not just the final generation.

3.7 Example GA application

This section outlines the design and implementation of a simple GA applied to a numerical problem, based on optimisation of a function. The target function is the equation

$$f(x) = \frac{a.\sin(x).\cos(x)}{x} \tag{3.3}$$

This is the classic sinc function as shown in Figure 3.7. The objective of the GA is to find the value of x that maximises the function $f(x)$. Typically this would be approached using a binary representation chromosome (e.g., Man et al., 1999). In this case we use a floating-point representation scheme.

Each chromosome is an array of float values, which is evaluated using the following algorithm, taken from the example code. (The complete GA code in Java for this problem is available from the author's web site.)

Appendix B also provides a generic listing of the basic GA classes required to evaluate such functions. The target function is evaluated at a number of sample points, equal to the length of the chromosome.

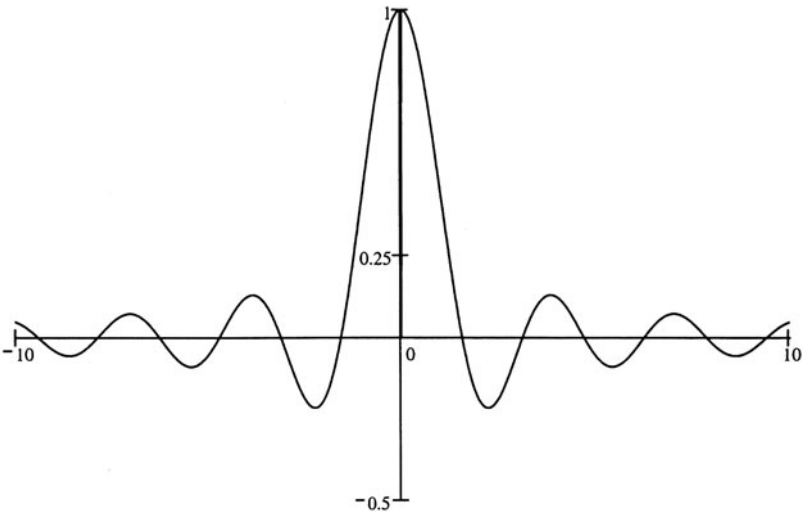

FIGURE 3.7 Graph of the function $f(x) = a.sin(x).cos(x)/x$ for $a = 2.0$.

The result is then compared against the corresponding element of the chromosome. By sorting the output array, one can obtain the maximum (or minima if required) of the function.

```
double evaluate(double[] genome){

double answer=0.0; double err = 0.00001;

int shiftX = genome.length/2;

for(int i=0;i<genome.length;i++) {

testarray[i] = alleleSize * Math.abs((Math.sin((i+err)-
shiftX))/((i+err)-shiftX));

}

for(int i=0;i<genome.length;i++){

answer += Math.abs(testarray[i] - genome[i]);

}
```

```
max_value = sort(genome);

//select whether fitness is across the whole function or
//just for the maxima.

If(select_function_max) //externally set boolean switch

answer = max_value;

else answer = 1.0 - (answer/genome.length);

return answer;

}
```

Listing 3.1 Evaluation method for optimisation of the function $f(x) = a.sin(x)/x$. The error value `err` is used to prevent division by zero and to ensure the function is evaluated in the region of $x = 0$.

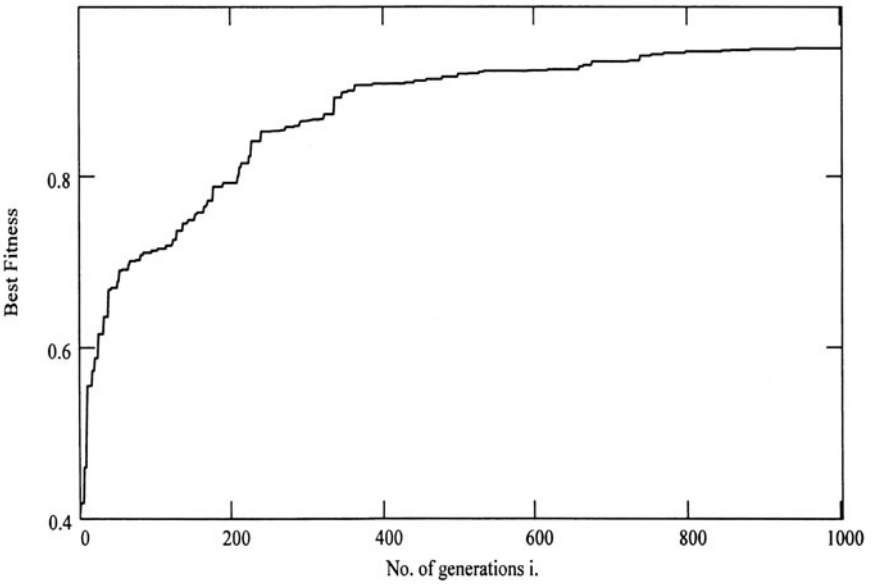

FIGURE 3.8 Graph of best fitness data from optimisation of the function in Eq. (3.3). (Evaluating the function over a set of values $f(x_{0\text{-}n})$, where n = length of the chromosome.)

The full code includes a graphical output for the problem to allow a user to see the result of the GA for each generation. Figure 3.8 shows the fitness response of the GA for 1000 generations with a relatively small population of 60 individuals and a mutation rate of 0.05 (using single-point random crossover and roulette wheel selection).

As indicated in Figure 3.8, this representation is very effective at solving this class of problem, even using a very basic set of genetic operators.

3.8 Summary

The impact of Holland's work on the genetic algorithm has been far broader than its original goal of creating and modeling adaptive systems. An entire industry of academic and commercial research programs has been generated which exploits the central ideas of the GA and extends them. An extensive range of applications has emerged, from evolved artwork to evolved designs for supersonic jet turbines.

Also, a number of standardised numeric optimisation test functions have been developed since the GA was first developed to allow some measure of comparative performance to be applied. Commonly used functions include DeJong's five test functions, Schwefel's function, and Schafer's test function. In Chapter 5 example applications demonstrate how a GA system can be applied to the solution of a complex control design problem in autonomous mobile robots and for an image processing problem.

The classical GA has proved to be a flexible and productive machine learning method, which is well suited to a range of complex technological problem domains. Further research is required, however, to expand its theoretical foundations and the exact interactions between the principal operators of recombination, mutation, and selection.

In order to gain a perspective of the possible space of evolutionary algorithms, Chapter 4 introduces the field of genetic programming (Koza, 1992). This incorporates the application of GA operators, but within a radically different representational medium, namely an entire functional program.

Further Reading

Back T., Fogel D.B. & Michalewicz T., (eds.) *Evolutionary Computation 1, Basic Algorithms and Operators*, Bristol and Philadelphia, Inst. of Physics, 2000.

Goldberg D.E., *Genetic Algorithms, in Search Optimization & Machine Learning*, Reading, Massachusetts, Addison-Wesley, 1989.

Michalewicz Z., *Genetic Algorithms + Data Structures = Evolution Programs*, New York, 3rd ed. Springer, 1999.

Mitchell M., *An Introduction to Genetic Algorithms*, Complex Adaptive Systems Series, MIT Press; ISBN: 0262631857, reprint 1998.

Whitley D., *A Genetic Algorithm Tutorial*, J. of Statistics and Computing, Vol. 4: 65-85, 1994.

4
Genetic Programming

> As the poet said, "Only God can make a tree" – probably because it's so hard to figure out how to get the bark on.
>
> Woody Allen

4.1 Genetic Programming

A large number of alternative evolutionary algorithms to the GA have been developed over the past two decades. Most share the same set of evolution-style operators of crossover, selection, and mutation. One important variation was developed by John Koza (Koza, 1992), termed Genetic Programming (GP). GP is distinctive in seeking to automate the generation of complete computer programs and adopts a tree-based representation scheme to allow the encoding of sufficiently complex logic structures within a chromosome.

GP has been widely adopted within the AI community and now has dedicated conferences and a large volume of published papers in the field. It is an appealing machine learning method as it offers the possibility of automated program discovery. As an evolutionary computing method it inherits many of the mechanisms used by other EAs but it also adopts a high-level approach to the representation of problems, which is a distinctive contribution to the whole field of machine learning. For this reason this chapter provides a review of GP and its applications.

4.2 Introduction to Genetic Programming

Banzhaf (Banzhaf et al., 1998) gives a useful starting definition of the GP class of systems:

> genetic programming, shall include systems that constitute or contain explicit references to programs (executable code) or to programming language expressions. So, for example, evolving LISP lists are clearly GP because

R. Ghanea-Hercock, *Applied Evolutionary Algorithms in Java*

> LISP lists constitute programming language structures and elements of those lists constitute programming language expressions. Similarly, the common practice among GP researchers of evolving C data structures that contain information explicitly referring to programs or program language tokens would also be GP, (Banzhaf et al., 1998, p. 5)

Such a definition hints at the very broad range of possible evolutionary mechanism, which might be defined as instances of GP. (The text by Banzhaf et al., provides an excellent overview and introduction to GP.) Koza (Koza, 1992) aimed to address the general problem of how to evolve complete programs with the capacity to solve any specified problem. The search space for GP is therefore the space of all possible computer programs derived from a set of *functions* (data processing elements) and *terminals* (data value elements) relevant to the problem.

There are five steps in using GP to solve a problem, which are to determine

1. The set of terminals
2. The set of primitive functions
3. The fitness measure
4. The parameters for controlling the run
5. The method for designating a result and the criterion for terminating a run

The terminal set is the set of inputs for the GP such as the set of real numbers. The function set is the collection of necessary logical or arithmetic functions required by the GP to solve the problem. Clearly, some prior knowledge of the possible functions to solve a problem is required in determining this set. The GP process works by combining a terminal and function set in a population of programs (tree structures), which are then individually evaluated against some fitness criteria, and the EA operators of recombination, mutation, and selection are then applied to produce the next generation.

Some distinctive claims have been made for GP as a machine learning system:

> Genetic programming is fundamentally different from other approaches to artificial intelligence, machine learning, adaptive systems, automated logic, expert systems, and neural networks in terms of (i) its representation, (ii) the role of knowledge (none), (iii) the role of logic (none), and (iv) its mechanisms (gleaned from nature) for getting to a solution within the space of possible solutions. (Koza, 1998)

The fundamental difference, which Koza emphasises, is the use of a program representation scheme in GP.

4.2.1 Variable-Length and Tree-Based Representations

We first need to refer to a potential limitation of the standard GA, namely its use of a fixed-length genome, which can restrict the algorithm to a nonoptimal region of the problem search space. Within the GA community this led to the creation of variable-length genome architectures, such as the Messy GA (Goldberg et al., 1989). The basic premise is that in designing a suitable representation and chromosome for a problem domain we may have no prior knowledge of how long a solution chromosome is required to be. In addition, from schema theory we may expect that if we allow the length of chromosomes to increase, then improved sequences of building blocks may be strung together.

By adopting a tree-based representation scheme, as in GP, we automatically have the potential for chromosomes of any length to occur (potential problems in this approach are discussed later).

4.2.2 GP Terminal Set

The nodes that terminate the endpoints of a GP tree are naturally named terminals (or leaves). When invoked either during evaluation or after completion of the GP run, a terminal returns some numeric value.

> The terminal set is comprised of the inputs to the GP program, the constants supplied to the GP program, and the zero-argument functions with side effects executed by the GP program, (Banzhaf et al 1998)

A terminal set also includes constants, typically a set of real-numbered constants, defined as R. GP is also able to generate new constants through the combination of the available constants in a given run. (Combined with the ability to synthesise new functions from the available set, this represents the inherent power of the GP technique.)

4.2.3 GP Function Set

"The function set is composed of the statements, operators, and functions available to the GP system" (Banzhaf et al.,1998). The function set is the application specific content of a GP program. In contrast to the limited range of chromosome types used in most GA systems, GP can utilise any available functional constructs relevant to the application. For example, any logic operator may be included: IFTHEN, AND, OR, NOT, any arithmetic operator: +, -, /, *, and any trigonometric entity: sin, cos, tan. In addition, any user-defined function can be added, allowing for the inclusion of subroutines.

Immediately from such flexibility we can see that GP allows for the construction of any level of program complexity. However, in selecting the function set we must be careful to include sufficient functions to address the problem space but avoid burdening the search process with an excessive number of functions. Remember each additional function vastly increases the

dimensionality of the search space. A wise approach is therefore to start with a small function set and only add functions if the system is failing to converge on a solution.

4.2.4 Function Closure

An obvious problem may be apparent in this approach to anyone with experience of creating function-based code, that is, how to ensure that a function returns a valid value and can handle exceptions correctly. A prime example is the division operator, which may crash the current GP run if input zero is received. A separate user-created protected division function must therefore be written to catch any zero input and return some safe value. Any unsafe function will rapidly become visible during the thousands of operations performed in a typical GP run, so careful function design is essential.

In addition, as many researchers in the field of GP have found, very application-specific functions are often not required, as GP can build its own required functions from subsets of simpler generic functions. Hence it is smarter to take advantage of this mechanism and focus on other issues, such as which selection or recombination operators to use.

4.2.5 Tree Structure Processing

Several possible representation schemes can be used in a GP system. The most frequently used is a tree structure, (see Figure 4.1), but linear and graph structures have also been used, which will also be considered. Koza's original format used LISP 's' expressions, as these are symbol strings, ideally suited to creating a GP tree representation (Koza, 1992).

There is a standard convention for the processing of such structures, namely, repeatedly evaluate the leftmost node for which inputs are currently available. This method is termed postfix order processing, as operators occur after the operands. The reverse form is known as prefix ordering, in which nodes are executed in order from the root node. This may save on processing operations if conditional branches are tested prior to being traversed such that unevaluated branches can be ignored.

4.2.6 Linear Structure Encoding

An alternative GP structure is to use a linear chain of instructions, which uses a separate memory structure to link functions and terminals together (Nordin, 1994, is one example).

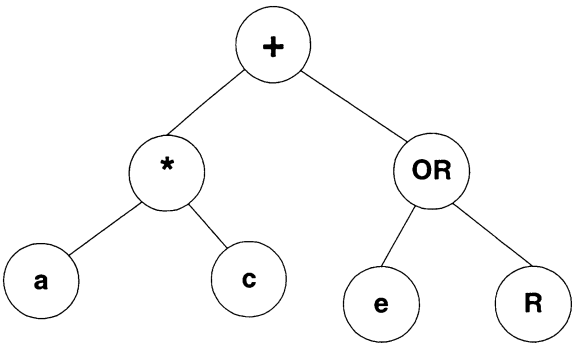

FIGURE 4.1 Example of a GP tree structure genome, where R = Koza's random constant (Koza, 1992).

4.2.7 Graph Structure Encoding

A recent alternative to tree-based structures is to use a more generic graph format structure (e.g., Teller & Veloso, 1995). In this case a program phenotype is represented by a graph structure with a set of vertices (nodes) and connecting edges. Such a structure facilitates the use of recursive and loop-style operations. The method uses a special start and end node to control the execution of an individual graph program, and the flow of the program is determined by the edges of the graph. Each node can be any user-specified function from the original function set.

4.2.8 GP Initialisation

The process of creating the initial population in a GP system is somewhat more complex than in a GA. Koza (1992) defined two methods termed *full* and *grow*. The grow method randomly selects nodes from the function and terminal set which are added to a new individual, until a terminal node is reached, which terminates that branch. This method frequently leads to trees with an irregular shape.

An important measure of the tree's structure is its depth, which is the minimal number of nodes that must be traversed to reach from the root node to the selected node.

The full method works by selecting nodes from only the function set until a node is at a specified maximum depth when it then switches to selecting only terminals. Hence every branch reaches the maximum depth, creating more regular-shaped trees.

An improved initialisation method, termed ramped half-and-half, attempts to combine both full and grow methods to ensure good diversity exists in the population. In this case half of the required trees are initialised with the full method and the other half with the grow method. However, the population is also first divided equally into groups of varying depth ranging from 2 up to the maximum depth.

4.3 GP Operators

GP shares many similarities with the broader domain of evolutionary algorithms, in particular in using crossover, mutation, and selection operators in order to generate novel evolved phenotypes.

4.3.1 GP Crossover

Crossover is performed through exchanging branches of two trees such that *syntactic correctness* is maintained, that is, the descendent programs are always valid programs that can be evaluated. Figure 4.2 illustrates this process.

As in the standard GA two parent individuals are first selected. We then select a random subtree from each parent, which are then exchanged at the corresponding node location in each offspring individual. (Similar but more complicated processes are used in linear and graph structure representations.)

4.3.2 Mutation

As in the GA, new individuals generated by crossover are selected with some low probability to have a mutation operator applied to them. The normal method is to randomly select one node in the individual and replace the current subtree at that point with a new randomly generated subtree (ensuring that the original requirements on maximum depth are observed).

However, as should be apparent from the mechanism just described, this is quite different to the simple point mutation effect occurring in GA-related methods. This operation is more like a frame-shift biological mutation as the resulting phenotype may be radically changed. (Check the Bibliography for GP references relating to mutational effects.)

4.3.3 Selection Operators in GP

GP may use either generational or steady-state mechanisms, according to the preferences of the researcher and the characteristics of the application. However, a popular selection method is tournament selection (i.e., a steady-state process), as it does not require a centralised fitness comparison between all individuals (Kinnear, 1993). A continuous interaction of individuals takes place with the

normal mechanisms of crossover and mutation, with new offspring replacing losers in the tournament process.

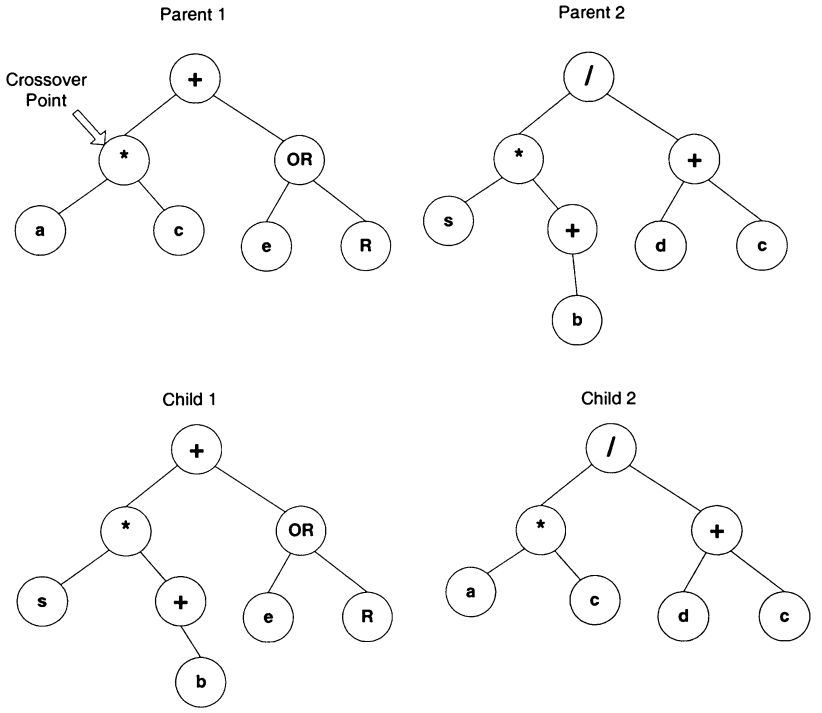

FIGURE 4.2 An example of the crossover process in genetic programming.

4.3.4 Controlling Genome Growth

The smarter student of this text will have already noted a potential problem with the GP approach. If the crossover operator can concatenate subtrees onto offspring individuals, then the new trees will tend to grow to reach the maximum allowed depth. Such a process leads to what is termed GP *bloat*. This was noted by Tackett (1994), who suggested that this is due to blocks of code in proximity to high fitness blocks being positively selected, known as free-riding.

A related issue is that many evolved GP individuals are highly complex in structure with many redundant subsections of code with no apparent purpose. These have been compared to the biological structures termed introns, namely, apparently redundant sequences of basepairs that don't actively code for RNA-forming proteins. This appears to be an inherent process in GP runs and can lead

to large proportions of the GP individuals being comprised of introns (Nordin et al., 1995). A common hypothesis is that the emergence of introns at the start of a GP run helps protect useful building blocks of code from destructive crossover effects, (Banzhaf et al., 1998, provides a detailed consideration of intron effects.) However, at the end of a run the growing mass of introns leads to a stagnant population.

The solution to negative intron effects in GP is to apply some form of selective pressure on finding shorter individuals. Hence longer programs incur an additional cost penalty in their fitness assignment.

4.4 Genetic Programming Implementation

GP has been used to solve a wide variety of problems, including how to evolve a control system for navigating an autonomous robot (Koza, 1994; Ghanea-Hercock & Fraser, 1994). The wider application of GP, however, has been hindered by the lack of suitable tools for implementing GP on standard software platforms, as most early research was performed in LISP.

Some compilers for C++ now exist (ftp://ftp.salford.ac.uk, GP in C++, by Adam Fraser), and Java versions can be found online at
http://www.cs.ucl.ac.uk/staff/A.Qureshi or
http://www.cs.umd.edu/projects/plus/ec/ecj/.

The Gpsys library of code is particularly suitable for initial experimentation with GP as it provides a well-structured set of Java classes and a GUI example for the lawnmower problem. A brief description of the system is presented in Appendix B.

A very illustrative example applet demonstrating GP in action is also available from http://alphard.ethz.ch/gerber/approx/default.html. The applet displays the output of a GP run while solving the symbolic regression problem, a classic example from Koza's original text.

4.4.1 Advances in GP – Automatically Defined Functions

Koza (1994) also introduced a significant development in GP, that is, a modular subroutine approach termed Automatically Defined Functions (ADFs). Using ADFs, we can divide an individual GP tree into two parts:

1. A result-forming branch evaluated during the fitness calculation;
2. A function defining branch, containing the definition of one or more ADFs.

The root node is now termed *Program* and acts as a placeholder for the subsections of the individual. Figure 4.3 shows an example ADF containing individual. A *defun* node is the root node of an ADF definition branch. The value node defines the result-producing branch of the individual. There may be multiple instances of the ADF and value returning branches.

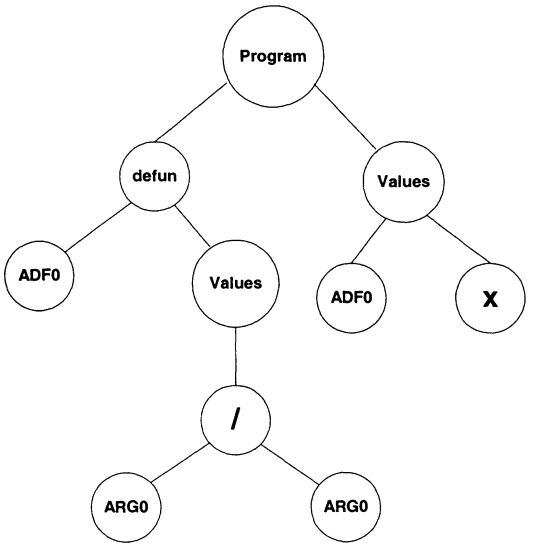

FIGURE 4.3 Example of an ADF GP tree.

One additional requirement with ADFs is the need for a new crossover function that preserves the roles of these specialised branches. The crossover point must therefore be selected from a branch of the same type. ADFs have proven to be highly efficient in many GP application areas and outperform normal GP by an order of magnitude. This is particularly the case in problems with well-structured solutions, such as the classic lawnmower test case (Koza, 1994).

Clearly, there is a cost to pay for using ADFs, but this is primarily the additional user input required in designing the initial function set and extra parameters required.

4.5 Summary

Genetic programming is an interesting methodology within EA and has proved to be exceptionally powerful across a broad range of technical problems. By evolving complete functional program sequences GP raises the level of abstraction such that a more direct encoding of applications is possible.

As with all EA methods, however, GP is a computationally intensive process. Parallel processing approaches are one route by which the necessary power and memory for more complex GP applications may be obtained. Recent work in peer-to-peer distributed computing platforms may also be a cost-

effective way of accessing the required resources. (Chapter 6 includes an example of a distributed GP system developed by the author.)

Further work is also required in understanding the role of introns in GP and in analysing which structural representations (tree, graph, linear, others) are most useful.

Further Reading

Banzhaf W., Nordin P., Keller R.E. & Francone F.D., *Genetic Programming, An Introduction, On the Automatic Evolution of Computer Programs and Its Applications*, San Francisco, Morgan Kaufmann, 1998. (Highly recommended)

Kinnear K.L.,(ed.) *Advances in Genetic Programming*, MIT Press (A Bradford Book); ISBN: 0262111888534, 1994.

Koza J.R., *Genetic Programming*, Cambridge, MIT Press, 1992.

5
Engineering Examples Using Genetic Algorithms

There are more things in heaven and earth, Horatio,
Than are dreamt of in your philosophy.

Shakespeare, *Hamlet*

5.1 Introduction

This chapter presents two real-world applications utilising GA techniques. The aim of this section is to demonstrate how evolutionary algorithms can automate the design process of complex multiparameter engineering problems. As outlined in Chapter 1, the solution of a real-world search or optimisation problem is invariably a nontrivial exercise. The two example applications presented in this chapter were selected to illustrate the broad applicability of EA methods and also because they make good use of graphical interfaces to illustrate their operation. The subject areas are also of potential interest to a wide audience. In particular, the robotics example is a useful testbed for a number of machine learning algorithms.

The source code and documentation for both problems are available from the author's web site, at http://www.cybernetics.org.uk.

5.2 Digital Image Processing

The art of manipulating digitised images is now a major commercial venture with applications ranging from simple photographic enhancement in desktop packages, to complex multidimensional visualisation graphics in medicine, and up to rendering of visual effects in video streams. Many computer users are now familiar with using photo-editing packages to retouch family snapshots, by removing defects, and colour or lighting distortions. These are all examples of digital image processing (DIP).

R. Ghanea-Hercock, *Applied Evolutionary Algorithms in Java*

5.3 Basics of Image Processing

Assuming we have acquired an initial image and digitised it, we now effectively have an array of numeric values, each representing the colour and intensity of the corresponding pixels. DIP is simply the process of applying a set of mathematical operations to these numerical values, such that the resulting pixels possess the required colour and intensity values.

5.3.1 Convolution

Convolution typically utilises a window of some finite size and shape, which is scanned across an image, one pixel at a time. The output pixel value is the weighted sum of the input pixels within the window, where the weights are the values of the filter assigned to every pixel of the window itself. One requirement is that the sum must equal 1.0 if the brightness of the original image is to be preserved. The window with its weights is called the *convolution kernel*. The kernel is a linear operator that determines the proportion of each original pixel colour that is used to determine the new pixel value.

The kernel is overlaid on each input pixel from the original image and the convolution operator is then applied. The following example is the simplest case in which no effect is applied to the resulting pixel. (Day & Knudsen, 1998, provides a nice introduction to the use of the Java 2D API and some common filtering operations based on convolution operators.)

```
0.0  0.0  0.0
0.0  1.0  0.0
0.0  0.0  0.0
```

Example of a 3 x 3 neutral convolution kernel.

An example of a kernel that applies a sharpening effect to an image is

```
0.0  -1.0  0.0
-1.0 5.0   -1.0
0.0  -1.0  0.0
```

A kernel for brightening an image can be created by simply adding a small constant to each pixel:

```
float b = 0.12f;
float[] brighten = {
b, b, b,
b, b, b,
b, b, b };
```

5.3.2 Lookup Tables

An alternative DIP operation is to use a *lookup table* method. In this case the original pixel values from an image are translated directly into new pixel values by mapping them into values stored in a table of numeric values. Since any colour is composed of red, green, and blue values, we need three tables in order to specify a given colour.

The length of the tables will be determined by what colour depth we are using. A common colour depth is from 0 to 255, although modern applications will often work on true colour images with a depth equal to 16 million colours. (However, unless the reader has access to a *very* fast computer, 256 colours are quite adequate for researching EA methods in image processing!)

Thresholding

A common operation required in feature extraction from a digital image is thresholding. The input to a thresholding operation is typically a grayscale or colour image, with the output being a binary image representing the segmentation. Darker pixels correspond to background, and brighter pixels correspond to the foreground. In its basic form the segmentation is set by a single parameter known as the *intensity threshold.* In a single pass, each pixel in the image is compared with this threshold. If that pixel's intensity exceeds the threshold value, then the pixel is set to some specified intensity value in the output. If it is less than the threshold, it is usually set to black (Fisher et al., 2000).

Histograms

The histogram of an image refers to a graph showing the number of pixels in an image at each different intensity value found in that image. For a digital image with gray levels in the range [0..L-1] the histogram is a discrete function:

$$p(r_k) = n_k/n \qquad (5.1)$$

where r_k represents the kth gray level, n_k is the number of pixels in the image corresponding to that gray level, and n is the total number of pixels in the image. The function $p(r_k)$ therefore gives the frequency for a given gray level r_k, as shown in Figure 5.1 (Gonzalez & Woods, 1992, p. 171).

In an 8-bit grayscale image there are 256 different possible intensities, and so the histogram will graphically display 256 numbers, showing the distribution of pixels among those grayscale values. Colour images can also generate individual histograms of red, green, and blue channels. A comparison of colour histogram arrays is used in the example application in Section 2.3.

The process of generating histograms from an image is relatively straightforward and involves scanning the image in a single pass and storing a running count of the number of pixels found at each intensity value.

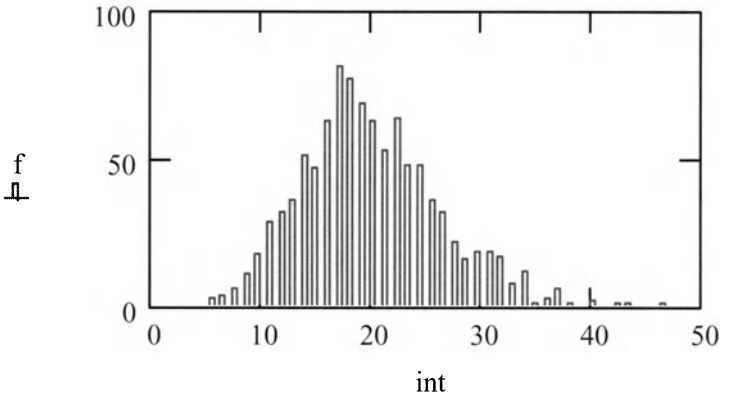

FIGURE 5.1 Example histogram function, for a low-contrast image (i.e., the distribution is centered on a cluster of values.)

Histograms are very useful structures in DIP; for example, they can help decide what value of threshold to use when converting a grayscale image to a binary one by thresholding. If the image is suitable for thresholding, then the histogram will be *bimodal*, that is, the pixel intensities will be clustered around two well-separated values. A suitable threshold for separating these two groups will be found somewhere in between the two peaks in the histogram (Fisher et al., 2000).

5.4 Java and Image Processing

Java is an excellent language for creating DIP applications, thanks to its object-oriented design. Some very useful introductory resources are available online which demonstrate the fundamentals of DIP using Java. For example, the Image J application offers full source code for implementing a wide range of DIP filters and operators (from: http://rsb.info.nih.gov/ij/).

A couple of very good online introductions to the field are available at www.dai.ed.ac.uk/HIPR2/ and www.ph.tn.tudelft.nl/Courses/FIP/. The Sun Java 2D tutorial is also available online at java.sun.com/docs/books/tutorial/2d

5.4.1 Example Application – VEGA

The example problem domain is an image processing application, in which a test image is filtered with a sequence of filter operations in order to extract a target feature pattern. It is therefore an image analysis problem, in which an EA is used to discover useful combinations of DIP operations (see Poli, 1996, for related work in GP). The original project name was Visual Evolution by Genetic Algorithms, hence VEGA.

In this application we presume that a library of relevant filtering operations is available and that the task is to learn a suitable sequence of operations that in combination will transform a given input image into a desired output form. (An EA could also be used to evolve the filtering operations if required.) The goal was therefore to manipulate an input image using a set of filter operations, in order to match a target image. The application test data was an astronomy image, which contained some features of interest to highlight. The set of image operations available to the GA included the following filter operations: Blur kernel, Edge detection kernel, Sharpen kernel, Brighter and darker operations, twelve colour lookup table transforms, and a null operation.

5.4.2 Operator Search Space

A useful question to ask at this point is how complex is the problem being addressed? Given a library of N possible image operators and a genome length of m genes, then we have N^m possible operator combinations. The values used in this application were typically $N = 55$ operators and utilised values of m between 10 to 30 genes in length (with 8 bits/gene). The possible sequences of operators and the resulting image transform are therefore very large and will grow rapidly as a function of genome length.

5.4.3 Implementation

The system is implemented using the Java JDK 1.2 (or later), which contains a useful library of basic image manipulation functions. The Java 2D library includes classes for creating and modifying images, as well as image producers, consumers, and filters for configuring image processing. The following is a list of some of the image classes that are part of the Java 2D API.

- java.awt.image.AffineTransformOp
- java.awt.image.BandCombineOp
- java.awt.image.BandedSampleModel
- java.awt.image.BufferedImage
- java.awt.image.BufferedImageFilter
- java.awt.image.BufferedImageOp
- java.awt.image.ByteLookupTable
- java.awt.image.ColorConvertOp
- java.awt.image.ColorModel

- java.awt.image.ComponentColorModel
- java.awt.image.ComponentSampleModel
- java.awt.image.ConvolveOp
- java.awt.image.DataBuffer
- java.awt.image.DataBufferByte
- java.awt.image.DataBufferInt
- java.awt.image.DataBufferShort
- java.awt.image.DirectColorModel
- java.awt.image.IndexColorModel
- java.awt.image.Kernel
- java.awt.image.LookupOp
- java.awt.image.LookupTable

Figure 5.3 shows the screen interface for the application with display panels for graphs of fitness values, and the test and target images. The image transform process is based on a keyed representation and a translation mechanism. An integer representation scheme was selected for the chromosome of each individual, which are translated into instances of the actual image operators via a hashtable. This avoids the need to directly encode the elements of the image operators in a binary format. Parameters for the operators could also be encoded into the genome, which would increase the resolution achievable, by the sequence of operators. (This is not implemented in the source code and is left as an exercise for the reader. The issue of adaptive parameterisation is considered under the advanced EA topic section in Chapter 6.)

The image application process uses the following steps, as indicated in Figure 5.2:

1. Select a set of image operators that map onto the specific image processing task.
2. Assign a key to each operator. In the current system a simple integer value is assigned to each operator.
3. Using a genetic algorithm, evolve sequences of the keys.
4. Each key sequence is translated into instantiated image operators via a hashtable.
5. The constructed set of operators is tested for fitness against the specified task and the fitness value is assigned to the genome.
6. Process repeats using normal GA selection criteria.

The fitness function for this application was defined as the absolute difference between the pixel red, blue, and green components of the evolved image and the target image. A preprocessing stage involved translating each individual genome into an image operator sequence in order to create the evolved image. The comparison was then made by taking the red, blue, and

green intensity histograms of the test and target images, followed by measuring the absolute difference between each histogram value.

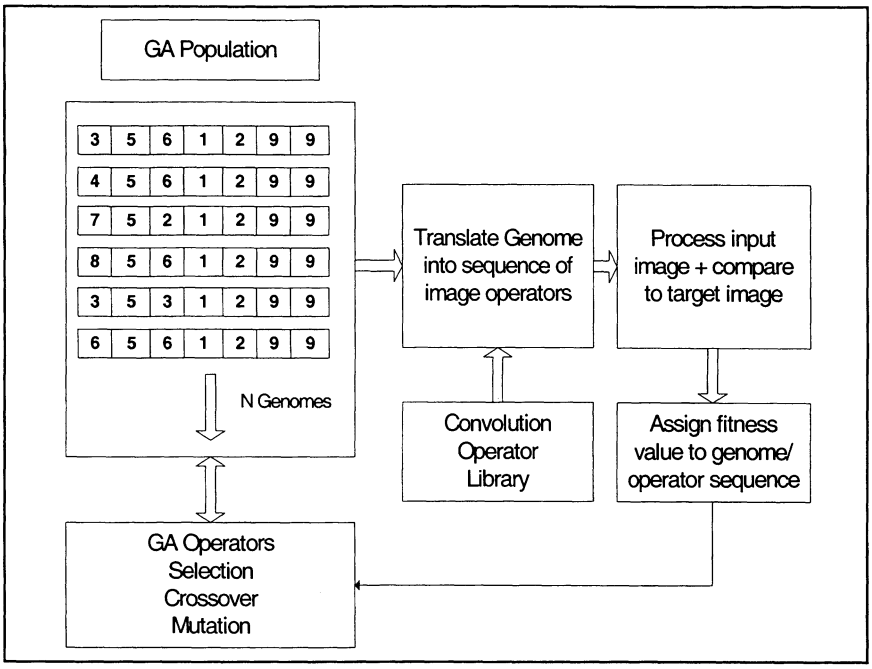

FIGURE 5.2 Mapping process from genome to phenotype, for the DIP application.

BufferedImage is the key class in the immediate-mode imaging API. This class manages an image in memory and provides methods for storing, interpreting, and rendering the pixel data. A `BufferedImage` can be rendered through either a `Graphics` or a `Graphics2D` rendering context.

A `BufferedImage` is essentially an `Image` with an accessible data buffer. A `BufferedImage` has a `ColorModel` and a `Raster` of image data. To filter a `BufferedImage` using one of the image operation classes, you perform the following steps:

1. Construct an instance of one of the `BufferedImageOp` classes: `AffineTransformOp`, `BandCombineOp`, `ColorConvertOp`, `ConvolveOp`, `LookupOp`, or `RescaleOp`.
2. Call the image operation's `filter` method, passing in the `BufferedImage` that you want to filter and the `BufferedImage` where you want to store the results.

```
//Image Comparison code

public class compareImage
{
BufferedImage image1,image2;
histogram hist,hist2;
ImageProducer ip,ip2;
int intensity[] = new int[256];
int intensity2[] = new int[256];
int histRed []; int histGreen[]; int histBlue[];
int histRed2[]; int histGreen2[]; int histBlue2[];
int temp [];
static int level = 0;
Image HistIm;
//constructor
public compareImage()
{}
public void passImages(BufferedImage im1, BufferedImage im2)
{
image1 = im1;
image2 = im2;
}

public void processImages()
{
ip = image1.getSource();
hist = new histogram (ip);
ip.startProduction (hist);
ip2 = image2.getSource();
hist2 = new histogram (ip2);
ip2.startProduction (hist2);
}

public int getDifference ()
{
histRed = hist.getRed();
histGreen = hist.getGreen();
histBlue = hist.getBlue();
histRed2 = hist2.getRed();
histGreen2 = hist2.getGreen();
histBlue2 = hist2.getBlue();
```

```
int difference=0;
int differenceRed=0;
int differenceGreen=0;
int differenceBlue=0;
for (int i=0;i<histRed.length;i++)
differenceRed += (Math.abs(histRed[i] - histRed2[i]));
for (int i=0;i<histGreen.length;i++)
differenceGreen += (Math.abs(histGreen[i] - histGreen2[i]));
for (int i=0;i<histBlue.length;i++)
differenceBlue += (Math.abs(histBlue[i] - histBlue2[i]));
difference = (differenceRed+differenceGreen+differenceBlue)/
3;

return difference;
}

}

class histogram implements ImageConsumer {
ColorModel cm;
private int hRed[]; int hGreen[]; int hBlue[];
Hashtable properties;
ImageProducer producer;
boolean finished;
public final static int COMPLETESCANLINES = 4;
public final static int IMAGEABORTED = 0;
public final static int IMAGEERROR = 0;
public final static int RANDOMPIXELORDER = 1;
public final static int SINGLEFRAME = 16;
public final static int SINGLEFRAMEDONE = 0;
public final static int SINGLEPASS = 8;
public final static int STATICIMAGEDONE = 0;
public final static int TOPDOWNLEFTRIGHT = 2;

public histogram (ImageProducer ip) {
producer = ip;

finished = false;
hRed = new int[256]; hGreen = new int[256]; hBlue = new
int[256];
}

public void imageComplete (int status) {
```

```
producer.removeConsumer (this);
finished = true;
}

public void setPixels (int x, int y, int w, int h,
ColorModel model, int pixels[], int off, int scansize)
{
int x1, y1, pixelindex, pixel, hr, hg, hb;
for (y1 = 0; y1 < h; y1++)
{
for (x1 = 0; x1 < w; x1++)
{
pixelindex = y1 * scansize + x1 + off;
pixel = model.getRGB (pixels [pixelindex]);
hr = (model.getRed (pixel));
hRed [hr]++;
hg = (model.getGreen (pixel));
hGreen [hg]++;
hb = (model.getBlue (pixel));
hBlue [hb]++;
}}

//Code for the image comparison algorithm:
while(i< genome.length)
{
//convert array short value to key
String key1 = String.valueOf(genome[i]);
int limit = Integer.valueOf(key1).intValue();
BufferedImageOp op = (BufferedImageOp)mOps.get(key1);
tempSubImage = op.filter(tempSubImage, null);
i++;}

//pass test and final evolved image to compare class
compare.passImages(testSubImage, tempSubImage);
compare.processImages();
while(!flag)
{
if (compare.hist2.isFinished () &&
compare.hist.isFinished())
flag = true;}

answer = (float)compare.getDifference(); status = true;

repaint();
```

The test image was a 211 x 211 pixel image with a 256 colour depth. Since the image was symmetric, a subimage slice (30 x 211) was extracted for comparison with the target image in order to reduce the evaluation time of each phenotype (see Figure 5.3). (Remember that evaluation of each individual is the most time-consuming aspect of most EAs.) Using a Pentium PC with dual 300 MHz processors, it requires approximately 25 minutes to evolve a solution match to the target image. For both applications in this chapter the free Java graph-plotting package PtPlot was used, which is available from http://ptolemy.eecs.berkeley.edu/java/ptplot/.

5.5 Spectrographic Chromosome representation

The VEGA GUI also contains a subpanel in the lower right corner that provides a two-dimensional view of the complete chromosome for the current GA population.

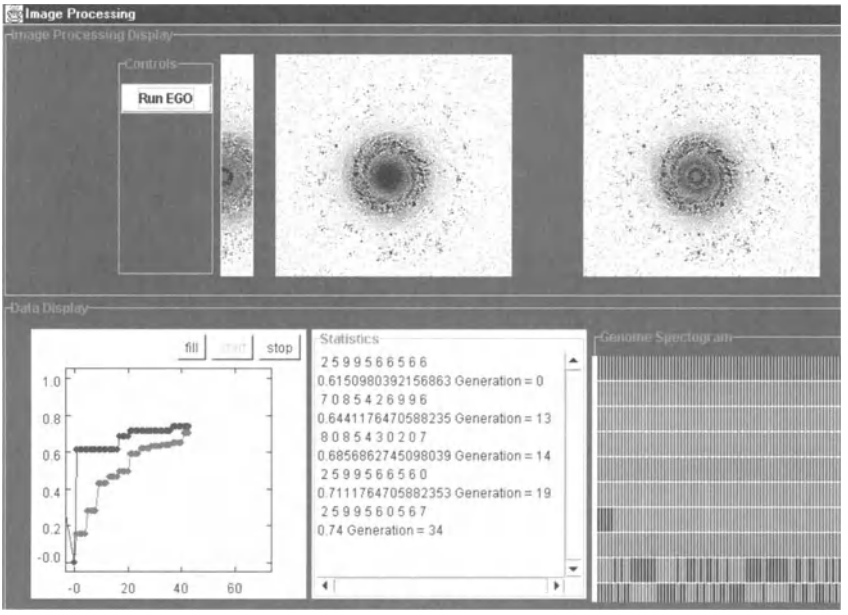

FIGURE 5.3 Screen shot for VEGA software package showing best and average fitness data, and a spectrographic chromosome image (bottom right). The images are (a) processed subimage (top left), (b) current best solution (top centre), (c) sample target image top right.

The purpose of this GUI element is to provide an intuitive visual indication of the relative diversity left in the population, as this can be difficult to deduce from observing the current test image or the fitness data.

The GUI panel utilises a simple colour mapping for each element in the integer chromosomes, which appears as a column of coloured cells. Each individual is then plotted in a table format to provide an intuitive visual display of the genetic variance remaining in the population. This is best understood while running the VEGA package and observing the convergence of the spectrogram into bands of colour as the GA loses diversity over the course of a run.

Clearly, such a visualisation process increases the computing overhead of a GA run; however, the purpose of this example is to illustrate the behaviour of an EA, rather than achieve serious evaluation of an optimisation problem.

5.6 Results

5.6.1 GA Format

The initial state of the test images is shown in Figure 5.3. This example uses a population size of 100 with a mutation rate between 0.005- 0.5. (See Agoston et al., 1998, for an analysis of mutation rates in integer representation schemes.) The GA implementation uses a simple roulette wheel method with fitness-proportionate selection, single-point crossover (with probability 100%), and standard point mutation. Fitness values are scaled from 0.0-1.0 max fitness. This is not a very efficient or elegant set of operators but is more than adequate for this particular application.

As Figure 5.4 shows, the fitness value for the best individual in the GA undergoes discontinuous steps as the population converges on a solution. The average fitness values lag the fittest individual while there is genetic diversity in the population, but as the diversity decreases the average value converges to that of the fittest individual in the population.

Since the selection method is based on a roulette wheel method, a scaling rule is applied in order to maintain evolvability of the individuals. However, with any GA application there is no guarantee that a solution will be found before the population has converged to the point that no further improvement is possible.

Comparing Figures 5.5 and 5.6 shows the effect of increasing the population size in the application. In Figure 5.5 the system undergoes rapid convergence, as the relatively low population of 50 individuals is insufficient to generate a solution with this particular set of GA methods.

By increasing the population size to 300 and using the same GA parameters, the system contains a much higher diversity of solutions and successfully generates a match to the target image. Of course, by increasing the population

size we incur a time penalty in terms of the additional evaluation times for each individual genome.

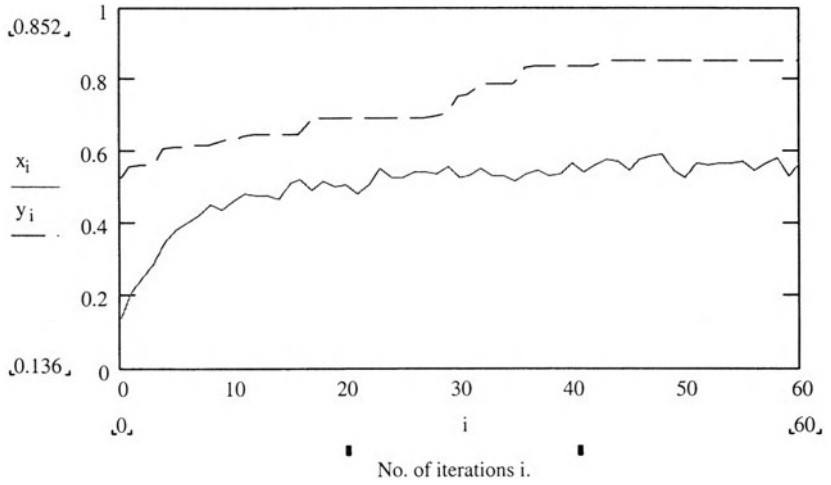

FIGURE 5.4 Fitness values for one run of the VEGA application, with a population size of 100 individuals. x_i curve represents average fitness of the GA population, y_i curve represents fitness of the best individual in each generation, (i).

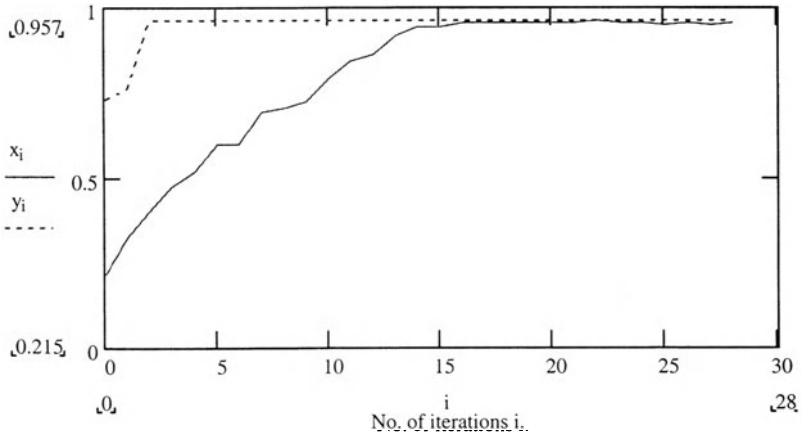

FIGURE 5.5 Fitness values for VEGA application, with a population size of 50. x_i curve represents average fitness of the GA population, y_i curve represents fitness of the best individual in each generation.

Although the best fitness curve indicates the system has attained 100% fitness, this represents a scaled value, and the system was unable to match the target image over several hundred trial runs.

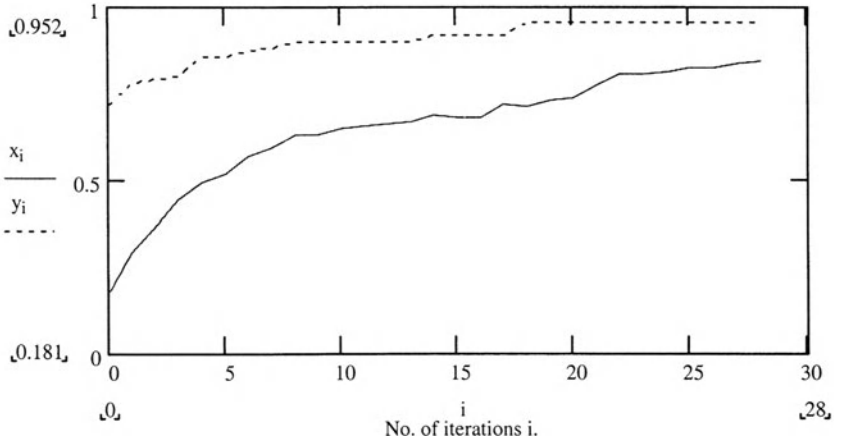

FIGURE 5.6 Fitness values for VEGA application, with a population size of 300. x_i curve represents average fitness of the GA population, y_i curve represents fitness of the best individual in each generation.

VEGA Development

Some obvious developments could be applied to this application, which would significantly extend its capabilities. First, the representation scheme could use floating-point gene values and incorporate parameter values for the image operators. This would improve the resolution of the system and allow smoother transitions within the fitness space.

Second, more computationally efficient operators should be applied if larger genomes are required; for example, by exchanging the roulette wheel method for tournament-based selection.

5.7 Summary – Evolved Image Processing

The use of GA and GP in image processing applications has been well established and represents a useful application domain for demonstrating a range of EA techniques. The scientific and commercial application of advanced DIP techniques is a significant growth area, and evolution-based machine learning

techniques will be increasingly important to this domain. EA methods can help in both high-level tasks such as classifying structures within an image and also for low-level image-analysis problems, e.g., image enhancement, feature detection, or segmentation (Poli, 1996). It would also be interesting to see if the use of Automatically Defined Functions in GP would increase the computational efficiency of evolved image processing; based on the need to create fit subroutines for feature extraction.

A related field, which has attracted some attention, is the use of EA methods to evolve images for artistic purposes. The cover art for this text was also generated using a GP art package (geneticfx). One good example of this is the work of William Latham and his Organic Art package, which allows a user to guide the generation of evolvable images; see: (http://www.artworks.co.uk /index2.htm).

The Bibliography contains a number of references to related papers in the domain of EA applied to image processing.

5.8 Mobile Robot Control

> One, a robot may not injure a human being, or through inaction, allow a human being to come to harm;
>
> Two, a robot must obey the orders given it by human beings except where such orders would conflict with the First Law;
>
> Three, a robot must protect its own existence as long as such protection does not conflict with the First or Second Laws.
>
> Isaac Asimov, The Three Laws of Robotics, *I Robot*, 1950

The history of mobile robots could be defined as the history of robotics in general, as the term "robot" was invented by Karel Capek in 1921 for his play R.U.R and "Robotics" was coined by Isaac Asimov in 1942. Asimov's famous three laws of robotics are quoted above; however, if you are ever working with physical robots, please ignore these laws, as real robots *will* allow you and your limbs to come to harm!

Fortunately this chapter is about designing a controller for a simulated robot that is quite safe. First a review of robotic development is presented, with a detailed discussion of the issues surrounding the control of autonomous robots. This application domain has a close association with the machine learning community as it places heavy demands on the adaptive or learning capabilities of computational systems. The second section presents a simulation tool for a

mobile robot and shows how a GA can be used to evolve the control parameters for such a robot.

In common literature robots have generally been envisaged as autonomous mobile android-like machines. Since these early visions of artificial intelligent beings there has been a gradual evolution of the mechanisms required to realise such complex systems. The process originated with the field of Cybernetics and the development of advanced control theory involving feedback for tracking control mechanisms. Walter's turtles may be the first example of the application of these concepts in autonomous robots (Walter, 1953). Since then there have been two major episodes in the evolution of mobile robots. The first was the development of relatively powerful computers in the 1960s, which made the field of Artificial Intelligence possible, and the second was the emergence of "Reactive" or "Bottom-up" methodology in the mid-1980s (Brooks, 1986). These two methods represented the major philosophical approaches to the theory of mobile robot control and are described in the following sections.

5.8.1 Artificial Intelligence and Mobile Robots

In 1969 the Stanford Research Institute completed the first mobile robot, called "Shakey", which could reason about its surroundings (Nilsson, 1984). This was an important first step in creating an intelligent robot and appeared to be the logical direction to take toward the goal of a fully autonomous machine intelligence. The motivation behind this work was the relative successes achieved in the field of Artificial Intelligence (AI). This term was coined by John McCarthy in 1960 to define the goal of creating a thinking computer, and during the 1960s AI programs were created with the ability to solve symbolic algebra, play chess, and solve problems in logic. The first application of these skills to robotics was in the control of fixed robot arms for the planning of strategies to manipulate objects in a desired sequence. However, the sheer size of existing computers precluded their use on-board a mobile platform. Hence Shakey and the other early mobile robots (such as the Stanford CART project; Morovec, 1983) were all controlled by an off-board computer connected via cable to the mobile platform, which contained the sensors and actuator systems.

5.8.2 Planning

The intelligent behaviour demonstrated by Shakey was in solving a "blocks world" problem, in which several blocks and a ramp had to be positioned such that the robot could reach an assigned target block to push it in some way. The robot had two stepping motors, and a single television camera for acquiring images of its environment, which were sent to the off-board computer. The task was achieved through a reasoning program termed STRIPS (Stanford Research Institute Problem Solver) (Nilsson, 1984). From a specified task, STRIPS constructed a plan based on the actions that the robot could perform, each of which had a precondition and set of consequences. The robot's world model was

constructed in symbolic logic, and generating the plan was simply a matter of theorem proving (it was hoped!). This approach was later defined as the "sense-model-plan-act" strategy or SMPA (Brooks, 1991).

This state-action model was constructed on the premise that the temporal evolution of the system could be analysed as a series of discrete states. *Actions* move the system from one state into the next, and each action is created by the execution of an *operator.* Since the execution process is atomic there is a direct correspondence between *actions* and *operators.* As Gat (1992) points out, this structure aids in analysis but makes the modeling of simultaneous or overlapping actions quite difficult. Since most realistic tasks sequences require parallel processing of multiple tasks, this casts doubt on the ability of traditional planning models to cope with physical robots.

5.8.3 Static Worlds

As previously suggested, the theory of the work was significantly removed from the reality, as Shakey proved to be quite inept at performing the assigned tasks. The robots movement through the environment was very slow, requiring hours to complete a single task and with a very high failure rate in recognising a block or making a correct decision. As Moravec puts it:

> The fault lay not in the STRIPS planner, which produced good plans when given a good description of what was around the robot, but in the programs that interpreted the raw data from the sensors and acted on the recommendations. (Moravec, 1988, p. 15)

This failure was even more apparent as the whole robot environment had been carefully optimised to suit the robot, with even lighting, uniform flat-faced blocks, and a smooth floor. The problem lay in the planners' inability to deal with unexpected outcomes to actions, and the accumulation of errors between its internal world model and the actual environment. Since the only static world is the one within the computer's own mind, this strategy was intrinsically flawed and another 15 years of research up to 1984 proved this to be the case as several groups tried unsuccessfully to create smart mobile robots using the SMPA approach. It was hoped that rapidly advancing computer technology would remove the model uncertainty by providing instant and accurate world knowledge. Unfortunately, the real world is more dynamic than was expected!

More recently several approaches to planning have been developed that attempt to provide for sophisticated conditional plan sequences. Some predictive planning systems (Currie/Tate, 1991; Wilkins, 1988) have dealt with the gap between planned actions and the real environment by using a "Mission Monitor" to record problems with the plan as it is executed and then to modify the plan online. Alternatively, a planner may include conditional actions within its planning process as a backup for possible alternatives when the plan is actually

executed (the "Universal Plans" of Schopper, 1987, is a good example of this approach).

In order to deal with the real-time requirements of controlling a mobile robot, some planning systems attempt to distribute the problem and closely couple the planner to a reactive or local control mechanism. A good example of this approach is Firby's RAP system (Firby, 1989). In his model a predictive planner produces actions that may be varied from a lower level in the system by "competencies", termed Reactive Action Packets. This system was used by Gat (Gat, 1992) as part of his "Sequencer" design in a three-level hierarchy control system for a successful mobile robot explorer. This approach has been summarised by Aylett:

> An abstract model which confines itself to the large scale more static properties present in most human constructed environments need not be updated very often and will support planning at a level which gives an efficient task structure. (Aylett, 1995)

5.8.4 Reactive and Bottom-up Control

> The route is from the bottom up, and the first problems are those of perception and mobility, because it is on this sensorimotor bedrock that human intelligence developed. (Morovec, 1988, p. 17)

In contrast to the robotics work in the AI domain, which has since become labeled as the "Top-down" approach, an alternative philosophy has emerged since 1984 described as the "Bottom-up" method. The pioneering work by Rodney Brooks at MIT defined this route to machine intelligence and represents a fundamentally different approach to the AI model of SMPA. The key bottom-up concepts were defined by Brooks (Brooks, p. 3, 1991) as

1. *Situatedness*: Robots should be situated within the world in terms of their cognitive processes, that is, they don't reason with abstract descriptions of reality. An AI system, however, is often "closed" with no direct interaction with the problem domain.)

2. *Embodiment*: An obvious requirement that the robot should have a physical body, as their actions form a dynamic exchange with the real world, which has an immediate feedback on their own sensations. In contrast, AI systems are generally isolated structures with in-depth competence, such as chess playing.

3. *Intelligence*: Their intelligence is not just a function of the computational processor but comes from the situation in the world, signal transformations

within the sensors and embodiment, that is, it is distributed across the whole robot-world system.

4. *Emergence*: Intelligence emerges from the system's interactions with the world, and the internal dynamic processes of the robot.

The difference between the two methods can best be illustrated by the following diagram, which shows how each method breaks the robot control task down into subtasks. The principal feature of the reactive method is the parallel hierarchy of task achieving modules that is used. This feature enables the robot to handle multiple goals simultaneously and provides the advantages of robustness and extensibility, which were so lacking in the AI method.

This control strategy was also defined as a "Subsumption architecture" as higher layers could subsume or suppress lower layers according to the priority assigned to each layer. These "competence modules" defined by Brooks became identified as "behaviours" as they seemed to correlate with the capabilities of natural animals in responding to real-world environments, and followed the "Behaviourist" school of psychology, in which an agent does not maintain an explicit symbolic representation of the physical world. Example behaviours include wandering, obstacle avoidance, and goal seeking, as shown in Figure 5.7.

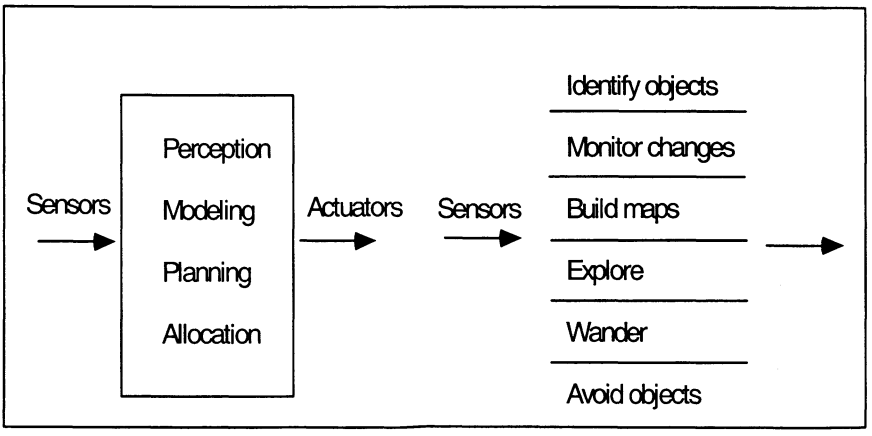

FIGURE 5.7 Architecture of a mobile robot control system showing traditional vertical task decomposition (left) and task achieving behaviours (right) (Brooks, 1986).

Many alternative behaviour-based or reactive architectures have been proposed since Brooks in 1986, for example, Arkin's "schema" system (Arkin, 1990, 1991). In Arkin's system perceptual schemas gather information about the

environment and actions are generated through *motor-schemas*. Each motor-schema is a vector-based function, such as Avoid-static-obstacle, Avoid-past, or Move-to-goal. These form the basis of Arkin's Autonomous Robot Architecture (AuRA). The velocity outputs from each independently active schema is summed, normalised, and sent to the robots actuators for execution.

5.8.5 Advantages of Reactive Control

The early work in reactive or behaviour-based control generated significant interest as researchers demonstrated small sophisticated mobile robots that could operate in unstructured environments in real-time and with vastly less computational resources than traditional AI systems (Brooks, 1986; Arkin, 1990; Connell, 1992). The tight coupling of sensor input to actuators through a set of behaviours resulted in real-time response and a robust ability to handle dynamic changes in the environment. Other advantages of this approach include

- Relatively low cost, due to significantly reduced computation
- High degree of adaptability
- Modular design provides for extensibility
- Well suited to multi-processor architectures

A more fundamental problem, however, occurs in the dynamic interaction of a robot with its environment, which should be viewed as a coupled dynamical system (Beer, 1995). This perspective means that the nonlinear interactions that occur between the robot, its internal processes, and the environment can severely limit our ability to predict its behaviour (Gat, 1994; Ghanea-Hercock & Barnes, 1995).

5.8.6 Alternative Strategies

Recent work by several groups has shown that fuzzy logic is a useful tool for constructing complex multilevel control structures in the area of autonomous robots (Tunstel, 1996; Saffioti & Wesley, 1995; Bonarini, 1996). A fuzzy system can often bridge the gap between low-level behaviours and the necessary reflective task-oriented processes acting on the robot. By intelligently managing the interactions of multiple *primitive behaviours* (Tunstel, 1996) a fuzzy controller raises the basic competence level of an autonomous robot, which can greatly reduce the cognitive workload on a planning system. An example is the ability to recover from situations that commonly trap purely reactive systems, such as box canyons (Watanabe & Pin, 1993). Work by Hoffmann (1997) and Surmann (Surmann et al., 1995) has also demonstrated the significant advantages of fuzzy logic applied to the problem of mobile robot control. In particular they show how a hierarchy of fuzzy rule bases can be constructed that are extremely modular and hence avoids many of the problems associated with increased control parameter search space. (Fuzzy rule bases can be connected

hierarchically with only a linear increase in the number of rules.) Figure 5.8 shows two mobile robots developed by the author (with Barnes & Eustace, 1994) at Salford University. An evolved fuzzy control logic similar to the simulation material presented later in this chapter was successfully tested on these robots. Each robot had five onboard transputers to perform parallel processing of the reactive behaviours and six microprocessors to preprocess sensory data from arrays of infrared, ultrasonic, and tactile sensors. They also frequently broke down, and acted in bizarre ways, just like most autonomous organisms.

These two B12 mobile robots were the basis for extensive research in cooperative multirobot systems. The robots (obtained from Real World Interface, Inc.) were omnidirectional, were controllable via an RS232 interface, and had a comprehensive set of motion commands. Superstructures were also designed and built to house additional sensors, data acquisition, communications, and secondary control hardware. The transputer processors were eventually replaced with a PC computer system – a 75Mhz 486 notebook. The robots both possessed the following sensors:

1. A multifrequency ultrasonic system for obstacle proximity detection (a multifrequency system was used to overcome acoustic interference between each robot.)
2. A self-centering X-Y table that used optical linear encoders to obtain displacement data.

(This allowed each robot to "feel the force" exerted by the other robot.)

They also had translate and rotate drive motor position encoders, and a battery voltage level sensor. They used ultrasound navigation sensors for locating active beacons, with a separate frequency for the obstacle sensors. There was also a half-duplex radio link, which could pass sets of behaviour commands to the robots and allows status information to be returned to a planning system running on a local workstation.

5.9 Behaviour Management

The particular aspect of mobile robot control that this section considers is how to balance the conflicting requirements of the robot in a semistructured environment. The following section describes the reactive control architecture that was utilised on the robots in Figure 5.8.

This example discusses a typical example of the reactive or bottom-up approach to robot control. (The actual application described later uses direct control of the robot via a fuzzy rule base, rather than with BSA, which would add unnecessary complication for the required purpose of demonstrating an evolved controller.)

5.9.1 Behaviour Synthesis Architecture

The Behaviour Synthesis Architecture (BSA) is a reactive control mechanism that has been successfully applied to the control of cooperating autonomous robots for an object relocation task (Barnes & Gray, 1991; Barnes, 1996). The problems of conflict resolution between agents and their behaviours are resolved through a vector synthesis mechanism.

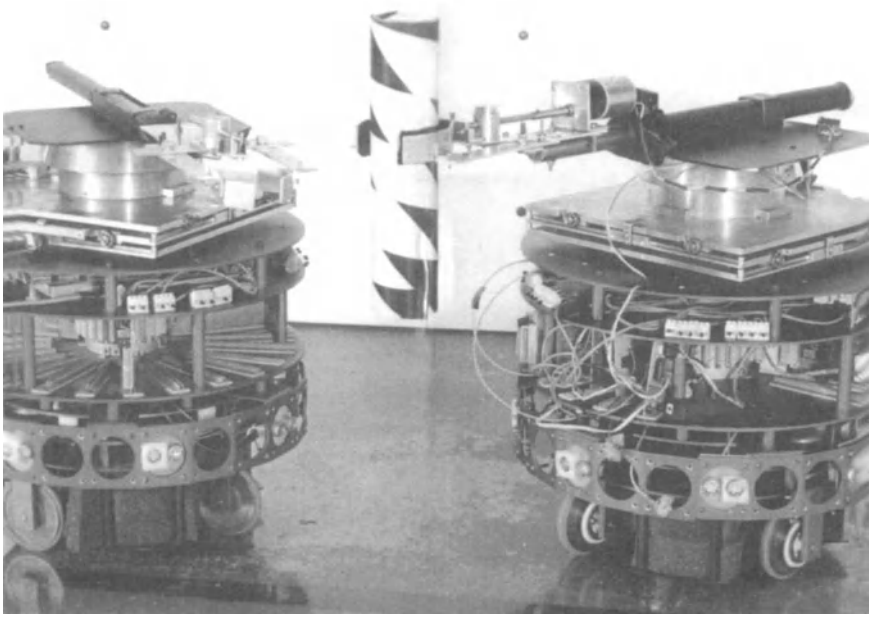

FIGURE 5.8 Fred and Ginger, autonomous mobile robots developed at Salford University, UK.

Each behaviour pattern is defined in terms of a stimulus/response function. The response/stimulus and utility/stimulus functions describe the importance of a behaviour at any particular instance within a given dynamic environment.

One of the advantages of an architecture such as BSA is that individual behaviour patterns can be grouped together to form a behaviour packet and sets of these packets are switched on or off according to sensory pre- and post-conditions by a sequential script. This provided a mechanism to constrict the conflict between behaviours and allow complex task sequences to be specified. It also allows control over the relative *altruistic* or *egotistic* actions of the agents.

A reflective planning agent that could interface to multiple reactive agents was also developed which transmitted scripts to the agents via an RF communication link (Aylett et al., 1995). It also monitored the robots' progress in completing a task and sent new scripts as required.

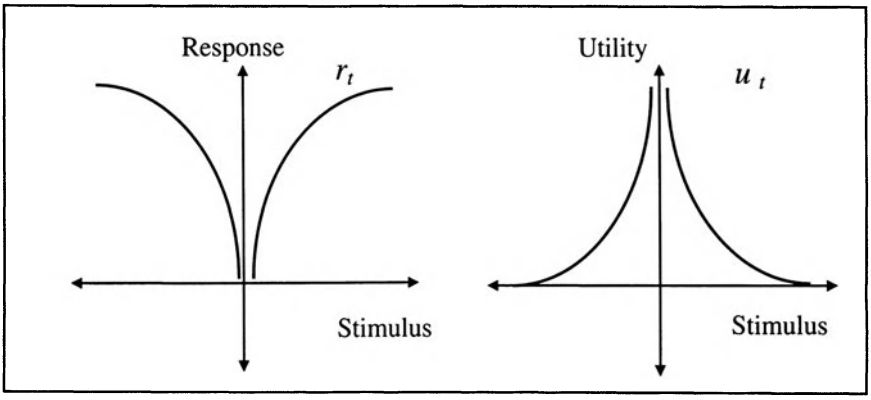

FIGURE 5.9 Example stimulus-response behaviour functions, for response and utility in a robot tracking behaviour, where response is the velocity of one robot in approaching the other, and the utility or importance of doing so increases as the distance between them reduces.

$$bp_t = \{\ r_t = f_r(s_t)\ \}\ \{\ u_t = f_u(s_t)\ \} \tag{5.2}$$

In Eq.(5.2) r_t is the particular motion response at time t and this is a function, f_r, of a given sensory stimulus, s_t. Associated to every response is a measure of its *utility* or importance, u_t.

This quantity is a function, f_u, of the same sensory stimulus and the values of r_t and u_t constitute a vector known as a *utilitor*. Competing utilitors are resolved by a process of linear superposition that generates a resultant utilitor, Ux_t, where

$$UX_t = \sum_{n=1}^{m} u_{t,n} \cdot e^{j \cdot r_{t,n}} \tag{5.3}$$

and m equals the total number of related utilitors generated from the different strategy levels (j is complex part).

Given a resultant utilitor, a resultant utility, uX_t, and a resultant motion response, rX_t are obtained from the following equations:

$$uX_t = \frac{|UX_t|}{m} \tag{5.4}$$

$$rX_t = \arg(UX_t) \tag{5.5}$$

X corresponds to a particular degree of freedom e.g., translate or rotate and the resultant motion response, rX_t, is then executed by the robot. From Eq.(5.3) it can be seen that generating a resultant response from different behaviours within the architecture constitutes a process of *additive synthesis.* The utility associated with each behaviour therefore provides an ideal input with which to adaptively modify a behaviour's contribution to the control of the robot. This higher-level input is taken from a second control layer that aims to balance the total response of the robot in a complex environment.

The process of grouping behaviours into "scripts" (Barnes, 1991) can help decompose the problem and provides a mechanism through which a reflective system may allocate tasks. However, whether a *script* or *sequence* (Gat, 1992) is used to package different behaviours, there remains the problem of inter-behaviour conflict due to multiple concurrent goals.

This problem appears to be intrinsic to all complex adaptive agents and essentially reflects the dynamic nature of the combined agent + world system.

The work by McFarland & Bosser (1993) best defines the nature of this problem and how natural agents may generate solutions based on the balancing of multiple utility functions, which can be visualised as sets of utility "isoclines". We therefore need a clearly distinct third control layer whose function is to manage or balance the interactions between the agents' set of behaviours and/or goals. In addition, it should be responsive to the global task requirements generated by any reflective control layer. The problem can be divided into several distinct categories:

- Conflict between individual behaviours
- Conflict between groups of behaviours
- Conflict between goals
- Temporal goal constraints

5.9.2 Standard Control Methods – PID

One of the most commonly used control techniques is the PID (Proportional, Integral, and Derivative) control system. For this reason it is often taken as a benchmark against which to compare an alternative control system. For a given process with a known transfer function $G_{(s)}$, then a PID system with a transfer function $F_{(s)}$ can direct the output of the process such as to minimise the *set-point* error *e(t)*, from a control input *u(t)*. (It can also be very efficiently encoded in software within embedded systems, which makes it a popular choice in industry.) This is illustrated in the following figure.

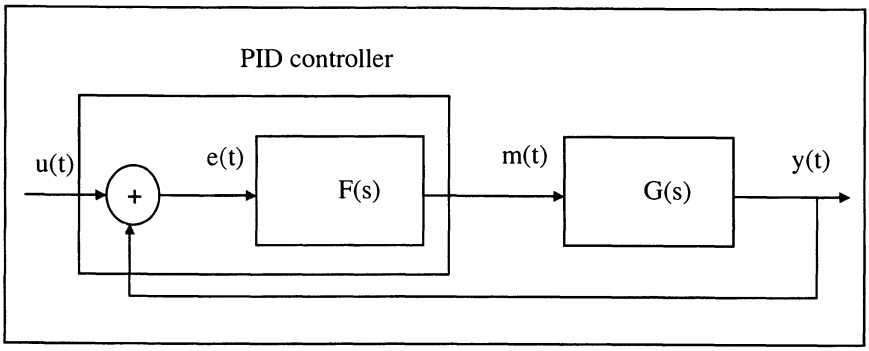

FIGURE 5.10 A standard PID control system (After Borrie, 1986.)

The system is designed by finding suitable values for each gain coefficient and having a sufficient knowledge of the processes characteristics given by $G_{(s)}$. Alternatively, they can be found using empirical methods such as the Reactive Curve Method or Continuous Cycling Method (Borrie 1986). This requirement for knowledge of the underlying process is one of the criticisms of the PID method when compared against equivalent fuzzy controllers. However, for linear systems with few inputs it is a well-tested, efficient tool that can be easily implemented as a simple circuit or computer program.

The action of the PID controller is defined by the equation

$$m(t) = K[e(t) + \frac{1}{\tau_i}\int_0^t e(t)dt + \tau_d \frac{de(t)}{dt}] \tag{5.6}$$

where $m(t)$ is the process input or variable,

K is the proportional gain of the controller,

K / τ_i is the integral gain of the controller, and

$K\tau_d$ is the differential gain of the controller.

An exercise for the reader is to use the GA library and robot simulator to evolve the parameters for a PID mobile robot controller.

5.10 Evolutionary Methods

The problem of designing the interconnection or coupling of multibehaviour control systems is one of searching the enormous parameter space that often emerges as a result of the combinatorial increase in possible connections and parameters as the number of behaviour modules increases. Automated search techniques are therefore extremely useful in designing such systems. The search algorithm is generally run off-line within a simulation of the robot, and the aim is to tune either the parameters or structure of an overall control mechanism. For example an artificial neural network or some direct encoding of the behaviour modules themselves can be used.

Of the possible search methods available, such as simulated annealing, hill-climbing, and genetic algorithms, genetic algorithms and genetic programming are commonly utilised in this field. The following sections describe these two methods and how they may be applied to tuning robotic control systems.

5.10.1 GAs and Robots

As stated, one of the problems in autonomous robot control is the tuning or parameter selection of the behaviours, which act to directly control the robot. The properties of GAs make them well suited to this class of problem, and several groups have used them for this purpose (Cliff & Miller, 1996; Mondada & Floreano, 1995.) Cliff's work illustrates how off-line evolution of the control parameters allows the tailoring of a robot's behaviours to a specific environment, which can significantly reduce the effort involved in designing a set of behaviours.

5.10.2 Inferencing Problem

Part of the robot autonomy problem is also how to merge the low-level numeric processing involved in reactive systems with a knowledge representation of the tasks a user wishes to specify to the robot. Fuzzy logic provides an effective framework for knowledge representation with vague and uncertain data and fortunately confers the property of linking numeric and symbolic processing. It also operates on an intuitive level, which parallels human approximate reasoning mechanisms. It therefore greatly alleviates the symbol grounding problem faced by classical AI (Brooks, 1991) in relation to mobile robot control, as the inferencing mechanism acts on fuzzy sets that are closely coupled to the numeric level of control.

5.10.3 Behaviour Priorities: Natural Agents

The underlying principle is that any agent, when faced with conflicting requirements, will prioritise action according to the most important first, with a continuum of decreasing importance being applied to other behaviours. An obvious hierarchy would begin with personal survival, and kin survival, followed by task accomplishment, such as feeding, etc. However, as McFarland and Boser demonstrate (McFarland & Boser, 1993), an agent should smoothly and intelligently shift its behaviour according to the maximum utility generated by that behaviour. Although due to the dynamic structure of the real world no agent ever has complete knowledge or the ability to make optimum strategy decisions, hence all choices are a tradeoff between probable outcomes. As Figure 5.11 indicates, empirical evidence suggests that many natural agents have a built-in response function for modifying the relative utility of conflicting behaviours, according to some inverse power law.

Similarly with autonomous mobile robots, some form of behaviour selection is necessary, although in this case the priority assigned to behaviours is strongly influenced by external agents, namely the human designer/user. This may be equivalent to the demands of group members on natural agents, as such demands often conflict with the immediate needs of the agent itself. The issue of behaviour selection is therefore not entirely straightforward and must reflect a particular agent's world view/needs.

Adaptation

The primary criterion that separates an autonomous agent from an automaton, is the ability to adapt. However, this quality, as with intelligence, can be difficult to define precisely and is still open to debate.

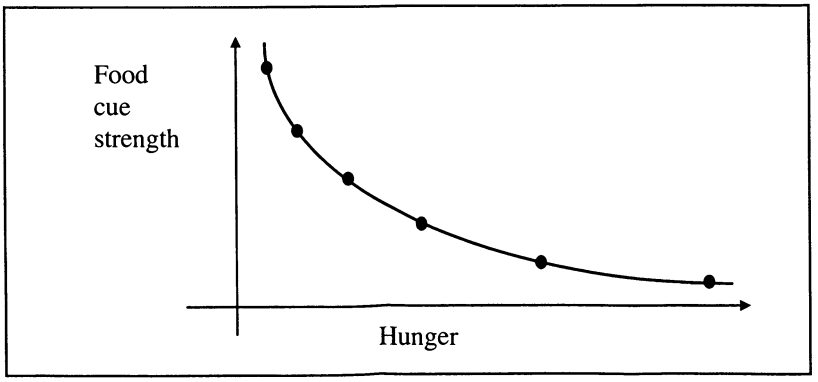

FIGURE 5.11 Motivational isocline of an animal's drive to eat (McFarland & Boser, p. 79.)

A couple of working definitions are offered as a starting point:

> An agent is said to be adaptive if it is able to improve over time, that is, if the agent becomes better at achieving its goals with experience. (Maes, 1994, p. 136)

> A system is capable of adapting and learning if it changes its behaviour so as to continue maximising its intelligence, even if the environment changes. (Steels, 1994, p. 77)

The common feature of most definitions is a change in behaviour in response to environmental change. It is necessary to quote Maes again, as she states:

> there is a continuum of ways in which an agent can be adaptive, from being able to adapt flexibly to short term, smaller changes in the environment, to dealing with more significant and long term changes in the environment. (Maes, 1994, p. 136)

Requirements for Adaptation

Given the need for an agent to be adaptive in order to cope with complex environments, what processes and abilities are therefore necessary in order to support this skill? The minimal requirement must be some form of internal state or memory, which can modify the relative effects of the agents' behaviours, either through suppression or inhibition. In contrast, a purely reactive agent is always driven by its external stimuli, even where behaviours are built into a hierarchy (e.g., subsumption). Gat has also pointed out the need for an intelligent use of internal state in mobile robots (Gat, 1993).

A similar example of this process is the "suppression" and "inhibition" mechanisms developed by Watanabe & Pin (Watanabe & Pin 1993). In this system a mobile robot is controlled by a "Fuzzy Behaviourist" architecture in which behaviours with related outputs can overpower the expression of another behaviour's output or input membership function. This system has proved quite effective in escaping the common deadlock problem in navigation with simple potential field methods.

5.11 Fuzzy Logic Control

This section demonstrates how we can use fuzzy logic to adaptively control simulated mobile robots. (Appendix C contains a more detailed description of fuzzy logic principles, which should be consulted first.) Recent work by several groups has shown that fuzzy logic is a useful tool for constructing complex multilevel control structures. A fuzzy system can often bridge the gap between low-level behaviours and the necessary reflective task-oriented processes acting

on the robot. By intelligently managing the interactions of multiple *primitive behaviours,* a fuzzy controller raises the competence level of an autonomous robot, which can greatly reduce the cognitive workload on a robots planning system. An example is the ability to recover from situations that commonly trap purely reactive systems, such as box canyons. In particular, an agent must be able to suppress behaviours that are no longer producing a useful response, over a timeframe, which is context-dependent. The use of a fuzzy rule base allows a direct linguistic description of the particular relationships the developer requires between any given behaviours. For example, the rule base may specify:

If robot NEAR to obstacle THEN apply LARGE negative feedback to navigate to goal behaviour.

The objective is for the robot to *focus* on its current response while maintaining some awareness of its final goal. This represents a form of "attention span" that allows the mixing of short-and medium-term goals, in a continuous manner (as illustrated in Figure 5.12.)

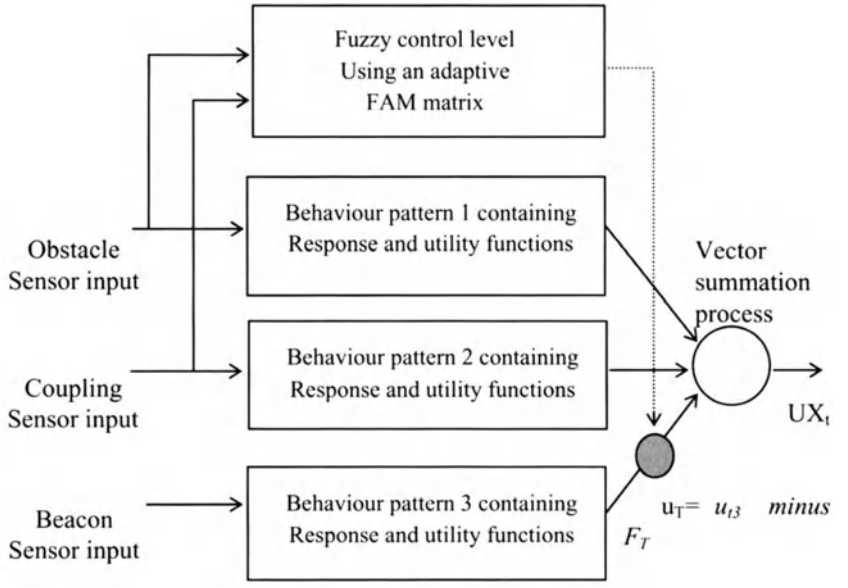

FIGURE 5.12 Hierarchical control system showing relationship between a possible external user's commands, the fuzzy system, and the behaviours inside a B.S.A control architecture. The FAM matrix is controlling the utility of moving toward an infrared beacon in response to inputs from the ultrasound obstacle avoidance, and a force sensor giving feedback from a physical coupling to a second robot, as shown in Figure 5.8.

An example of this is when the robot enters a potential well while navigating toward a beacon (refer to Appendix D for full details of the simulation environment). As the robot approaches an obstacle, the importance of avoiding it increases due to the utility-response generated by the fuzzy rule base turns down the utility of moving toward the beacon. By applying a small bias to one of the rotate behaviours the emergent response is for the robot to follow the perimeter of the obstacle until a free path toward the beacon is found when it returns to navigating in that direction.

As stated a general-purpose fuzzy control system works by encoding an expert's knowledge into a set of rules, which are smoothly interpolated, and the resultant is defuzzified to give a crisp actuation output. Each rule is specified as either a triangular, trapezoid, or some other function (e.g., bell shape) and assigned to some range of input variable. For the experiments performed during this research trapezoid functions were selected as they are computationally efficient, which is a priority when used for real-time control systems, and most groups have reported little difference in control system performance when using more sophisticated functions. Using a FAM representation, the weight for the i_{th} FAM entry was calculated using the minimum rule:

$$w_i = \min\{F_{(i)}(x), F_{(i)}(y)\} \tag{5.7}$$

where x and y represent the input dimensions of the FAM matrix, (see Figure 5.13.)

NL	PL	NL	ZE	PM
NL	PL	ZE	ZE	NL
NL	PM	PL	NL	NL
NL	PM	NL	PL	NL
NL	NL	NL	PM	ZE

FIGURE 5.13 Typical two input FAM matrix with five output fuzzy sets per rule. The shaded cells indicate a typical set of active rules.

Output sets were defined as NL = 0.0, NM = 0.25, ZE = 0.50, PM = 0.75, PL = 1.00. If we define each output fuzzy membership set as

$$B_i = \text{output fuzzy set} \tag{5.8}$$

where the output "universe of discourse" is composed of a finite set of discrete values. Then the total defuzzified response for n output membership sets is

$$u^* = \frac{\sum_{k=1}^{m} c_{(k)} \cdot f_k}{\sum_{k=1}^{n} f_k} \tag{5.9}$$

where $c^{(k)}$ is the peak value of the set LU and f_k is the height of the clipped set CLU and n equals the number of active rules at a given time. This is generally referred to as "height defuzzification". This is in contrast to the "fuzzy centroid" or "centre of gravity" defuzzification scheme, in which the output is a combination of centroids for each overlapping fuzzy membership function. This method generates a unique fuzzy centroid and may make better use of the information in the output distribution. However, by taking the simplified case of assigning a singular discrete value to each output membership set, we greatly reduce the computational requirements in exchange for a small sacrifice in resolution of the output fuzzy action surface. (See Driankov et al., 1996, for detailed comparisons of defuzzification methods in control theory.)

5.11.1 Fuzzy Control of Subsumption Architectures

In order to illustrate the generic nature of this work, a brief consideration of how it may be applied to another behaviour-based architecture will be considered. The seminal work that revived interest in mobile robots was Brook's 1986 paper, which introduced the bottom-up concept using layered reactive control. As Maes has pointed out in her classification of agent architectures (Maes, 1994), subsumption falls in the category of "*hand-built flat networks,*" in which the designer has to solve the whole action selection process.

5.12 Evolved Fuzzy Systems

This section describes the process of automating the design of a fuzzy controller for a simulated mobile robot using a genetic algorithm. Alternative evolutionary search techniques could have been used, such as Genetic Programming (Koza, 1992). (The text by Man Tang and Kwong, 1999, gives a useful related example of applying GAs to evolving fuzzy logic controllers.) The question of whether control systems designed offline in a simulation can be transferred to a physical robot is an interesting arena for current research in the field. There is a strong assumption that simulators usually fail to closely match the full dynamic behaviour of a physically embodied robot (Brooks, 1991). One solution to this

dilemma is that by combining the properties of a fuzzy controller with a hierarchical control system a useful component of the total control mechanism may be evolved in simulation and then transferred to a real robot.

Implementation

The principal difficulty in deploying a fuzzy system is the rapid increase in the possible combinations of rules, along with the number of input and output dimensions. In general, many fuzzy controllers separate higher-dimensional spaces into submatrices of two dimensions, which is possible with such systems (Welstead, 1994). This has led researchers to use search algorithms and artificial neural networks to find an optimum set of rules.

A simple genetic algorithm will be applied to the process of selecting the FAM matrix entries (i.e., output rule values), with manually selected fuzzy input sets and functions. The encoding scheme for genetic optimisation takes each FAM matrix entry and assigns it an integer value in the range 0-5; (see Figure 5.14) these are then strung together to form a single fixed-length chromosome array. A range of 5 output rules is a common choice for a fuzzy control application, as this provides adequate resolution without excessive computational cost. (This set of output rules is defined as a fuzzy associative memory, abbreviated as FAM.) The GA uses a select fittest mechanism and mutation probability was set to a relatively high value of 0.05-0.1%, as relatively small populations were used for this application, that is, less than 200.

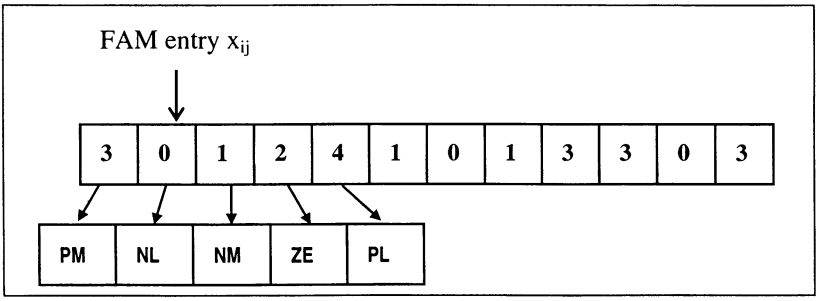

FIGURE 5.14 A GA chromosome showing the encoding for a FAM matrix, using an integer representation.

$$S = P^{(M \cdot N^2)} \tag{5.10}$$

In the example application each FAM entry is represented by an integer that defines the number of possible output fuzzy sets (P). With M inputs and N fuzzy sets per input, the total chromosome has S states (Eq.5.10.) Clearly this will increase rapidly as the number of inputs or sets per input increase, which implies

an obvious increase in computational cost during the evolution phase. (However, due to the modular nature of FAM systems, several could be evolved separately to solve particular aspects of the environment before being combined into a final system.) Each complete FAM matrix is then evaluated against the specified fitness function and the normal processes of fitness-proportionate selection, crossover, and mutation applied.

5.13 Robot Simulator

The basis for this application is a 2D mobile robot simulator written in Java. This simulator, named Rossum's Playhouse (RP1), was designed by Gary Lucas and is available online from http://rossum.sourceforge.net/. (The simulator is also available from the book's web site with the GA and fuzzy logic code included.) It is designed as a general simulation tool for students and researchers interested in developing autonomous robot navigation and control logic and does not model any particular hardware platform. However, it is designed in a very modular style and facilitates the transfer of developed code modules onto a real hardware robot. It was selected as the basis for this application as it offers the essential features of a mobile robot simulator without excessive complexity and also includes good documentation (see Appendix D.)

An alternative Java-based mobile robot simulator that is publicly available is the TeamBots simulator (Tucker Balch, 2000), which can be found at http://www.teambots.org/. This is a well-developed simulation tool, which models specific robot hardware and offers extensive code for researching a number of sensor, navigation, and control algorithms.

The RP1 simulator is a pure Java implementation that provides a number of useful features, including

1. A client-server architecture such that the simulator environment can be run on one computer as a server and the robot control code can run on a separate computer as a client.
2. Event and request interaction model between the client and server. Simplified message structure to facilitate language independence.

An excellent hardware platform for actually building a small mobile robot is available from MIT laboratories, at http://handyboard.com/. This is a micro-controller-based PCB that contains all the functionality required to control and drive a simple mobile robot.

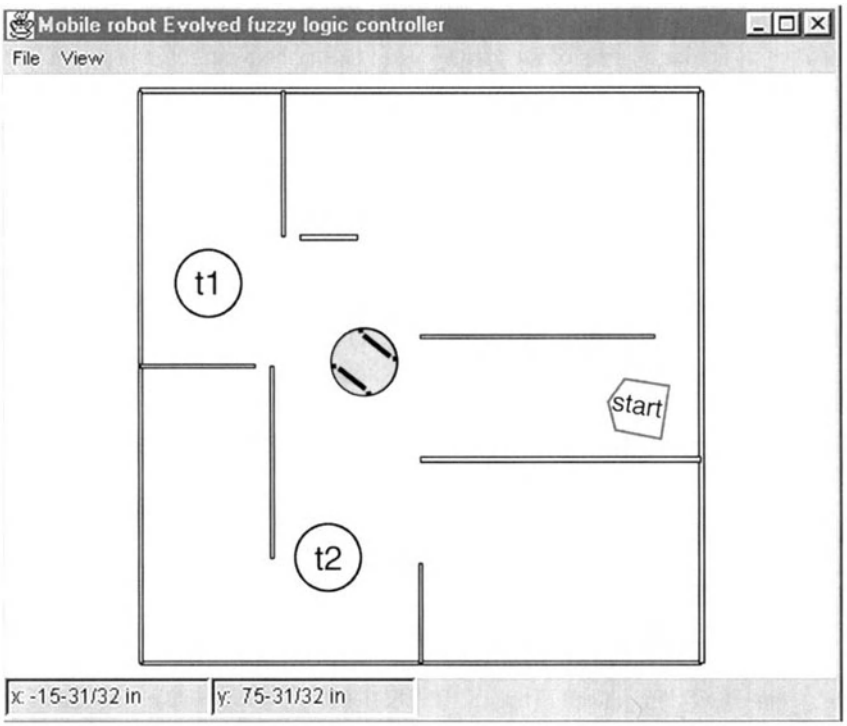

FIGURE 5.15 Example GUI interface to the RP1 robot simulator. t1 and t2 are target beacons, which the robots front sensor can detect. The four sensors on the robot can detect the range and angular position of the walls relative to the axis of the robot.

5.13.1 The Robot Control Architecture

The problem we are addressing is how to automate the process of allocating the output rules for a fuzzy logic robot control algorithm. We can apply the standard GA code already described in the image processing application, as the robot application also requires the GA to find a useful 1D array of integer values. In this case each integer gene is mapped to a value in the 2D matrix of fuzzy logic output rules (or FAM).

The fitness function in this application is clearly how well the robot manages to complete its specified tasks, namely maneuver around the environment and navigate toward a target beacon (t1 in Figure 5.15). A simple fitness function $f(t)$ can be defined in this case as the distance covered by the robot, measured by the

linear range of its target sensor, and a measure of the time taken to perform the task (normalised for 1.0 = max fitness).

$$f(t) = 1 - \left[\frac{rt}{mr} + \frac{t}{mt} \right] \tag{5.11}$$

where rt = minimum final distance between the robot and the target, mr = maximum range to target, t = time spent in a run, and mt is maximum time available for a run. Of course, a range of possible fitness functions could be applied to this problem with equal effect, using other metrics of the robots performance.

However, we also need a set of termination criteria for the robot. Hence if any of the robot collision sensors fire, then the current run is terminated and the robot is returned to the starting point to begin a new run, with a new genome. Similarly, if the robot's obstacle sensors return a value below some threshold value or a time-out criteria is reached, then the run ends.

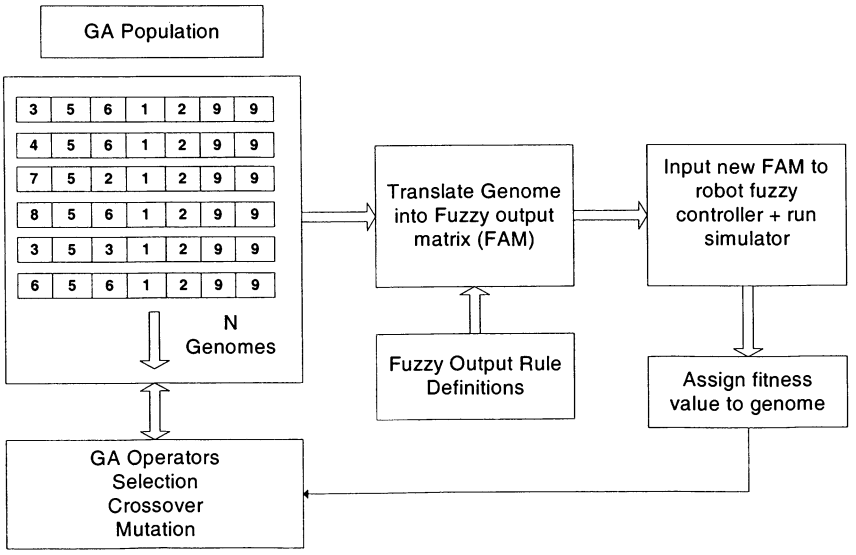

FIGURE 5.16 Interaction between the genetic algorithm, robot simulation, and the evolving fuzzy controller.

The sequence of operations required for these tasks is therefore:

1. Initialise simulation and robot.
2. Initialise genetic algorithm.
3. Run through GA population, and assign each chromosome to a FAM.
4. Start a run of the robot simulation, using the FAM to modulate the strength of the interactions between the conflicting obstacle avoidance and target navigation behaviours, (see Figure 5.16.)
5. If robot fails to complete task, assign a fitness value to the current chromosome based on the fitness function.
6. Repeat from step 3, for N runs or until a specified fitness level is reached.

At this point the fitness value is assigned to the chromosome responsible for the control matrix and a new chromosome is tested. The code for the main control loop is given below.

```
//* Motion control method*//

protected void computeAndSetMotion()
{

//translate calculation
if (targetRange == 0.0)
targetRange = 0.05;//min target limit
double tr = wMax;// - (Math.abs(range0 - range3));

//slow if either range sensor fires

tr = tr - (wMax - (targetRange*wMax));
//scaled by max linear speed

tr = tr + wMin;
//need a net +ve speed to allow approach target

float[] data_array = new float[2];
data_array[0] = (float) (targetRange/2.0);
data_array[1] = (float) (range0/2.0);
graph.showGraph(data_array);

//range sensor rotate calculation
double rangeRight = Math.pow(range0 , 1.0);
//apply a function to amplify sensor
```

```
double rangeLeft =  Math.pow(range3 , 1.0);
//targetBearing is already signed
//target sensor rotate calculation
double targetRight = 0.0;
double targetLeft =  targetBearing;
//targetBearing is already signed +ve/-ve

//calc range and target rotate utility using fam reponse

double famOutput = flControl.evaluateFAM(
Math.abs(rangeRight-rangeLeft) , Math.abs(targetLeft) );
targetLeft  = targetLeft * famOutput;

//calc combined turn from both sensors
double leftTurn = (rangeLeft + targetLeft);
double rightTurn = (rangeRight + targetRight);
tr = tr * (famOutput*2.0);

//execute new drive commands
double turn = leftTurn - rightTurn;
driveRobot(tr, turn);

//end
```

5.13.2 Related Work

In the text by Michalewicz (1999) an example application is described for an evolved path-planning control algorithm in an autonomous mobile robot. In this system the chromosomes contain an ordered list of path node objects. Each path node contains a pointer to the next node, an x- and y-coordinate of an intermediate point along the path and a boolean variable, which indicates if the node is feasible or not. This is a nice example of how a gene can be mapped to a software object with multiple parameters, in contrast to a low-level binary representation. The chromosomes have variable lengths, as the number of nodes required could cover a wide range.

The fitness function used is a more complex version of the one defined in Eq.(5.11), where fitness of a chromosome p is

$$\text{Path Cost(p)} = w_d \cdot dist(p) + w_s \cdot smooth(p) + w_c \cdot clear(p) \tag{5.12}$$

where

w_d, w_s, w_c are normalisation weights,

dist(p) = total path length traveled,

smooth(p) = the largest curvature of p at a knot point in the path, and

clear(p) = largest clearance between all segments of p and the obstacles.

5.13.3 Results

The following results were generated from a series of trial runs of the robot simulator using the described GA to evolve a set of fuzzy output rules for the fuzzy controller. Figure 5.17 shows which sensor inputs are used as inputs to the FAM control matrix and which robot variables are modified by the defuzzified FAM output. The objective is for the robot to acquire a FAM controller, which can successfully adapt its behaviours, as it navigates the environment.

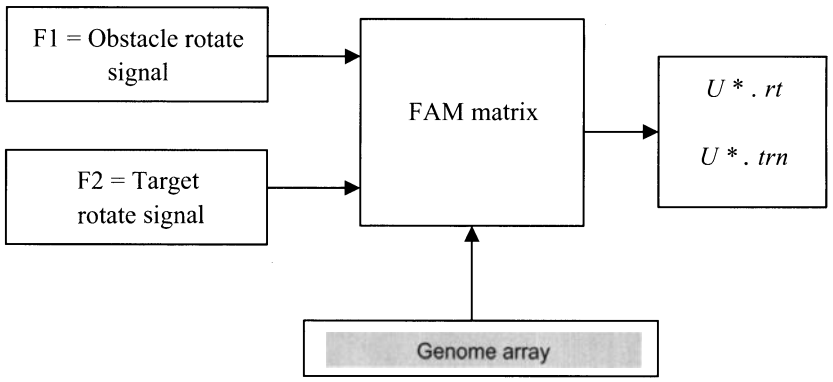

FIGURE 5.17 Functional relationship between the control inputs and robot sensors, where F1 and F2 are the fuzzy input rule variables, U^* is the defuzzified output, *rt* is the rotate to target variable, and *trn* is the robot translate value.

In this configuration the system needs to evolve a FAM that can utilise the input variables F1 and F2 such that the robot adaptively responds to the specific environmental constraints. The target sensor was calibrated with a 360-degree angular range and a linear range value of 2.0 at the starting point of the robot, and falling to 0.0 at the target.

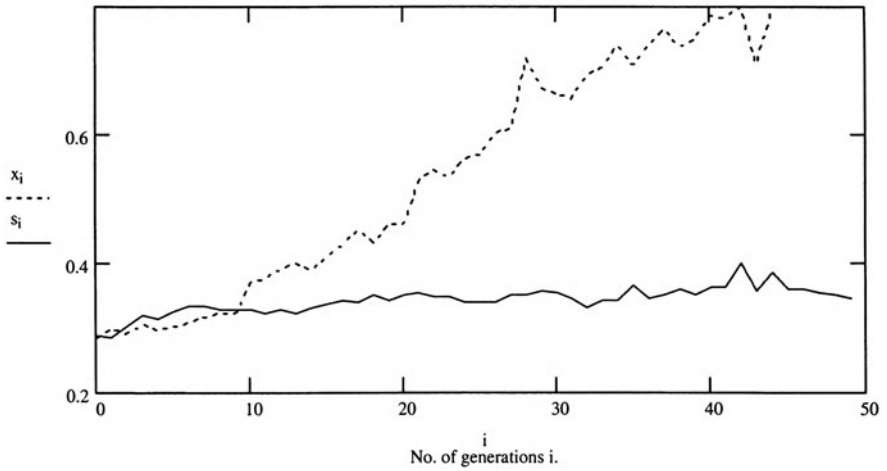

FIGURE 5.18 Example average fitness data for a population of 25 (curve s_i) and a population of 45 (curve x_i), with a genome length of 50 and a mutation rate of 0.05%.

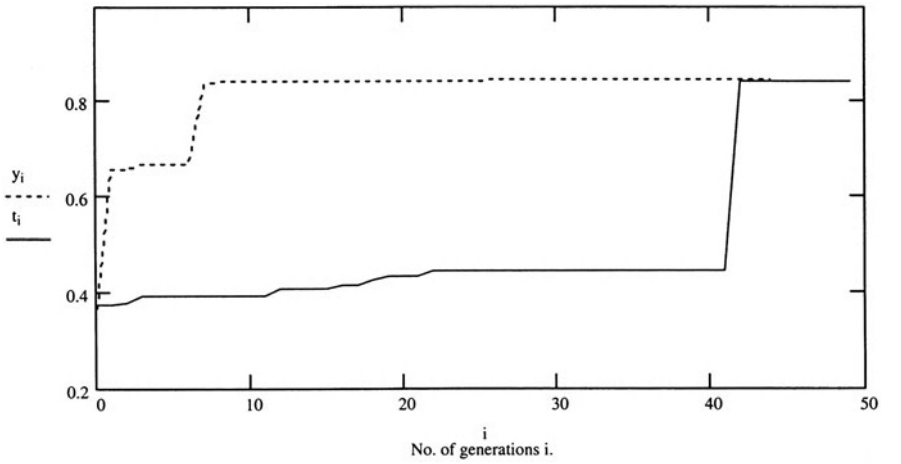

FIGURE 5.19 Example best individual fitness data for a population of 25 (curve t_i) and a population of 45 (curve y_i), with a genome length of 50 and a mutation rate of 0.05%.

A more difficult task can be assigned by limiting the angular range of the target sensor, which is easily defined within the RP1 simulator.

In this case it should reduce the utility of turning toward a target sensor as the robot approaches an obstacle, and simultaneously reduce the translate velocity of the robot.

Figures 5.18 and 5.19 show the effect of having too small a population of genomes, which may fail to provide sufficient genetic diversity for the robot to achieve the required task. In this case the robot is unable to negotiate the first corner in the maze with a fitness of less than 45%. As illustrated it requires significantly more generations with a population of 25 to achieve the same task. Figure 5.20 demonstrates the relative effect of mutation rate on the problem. In this application the lower mutation rate of 0.003% still achieves a solution to the navigation problem and reduces the excessive convergence caused by the high mutation value of 0.5%.

The two dominant rules can be expressed in an IF-THEN rule format

IF obstacle-sensor-input PL AND target-rotate-input PL THEN Feedback-output PL

IF obstacle-sensor-input NM AND target-rotate-input PL THEN Feedback-output PL.

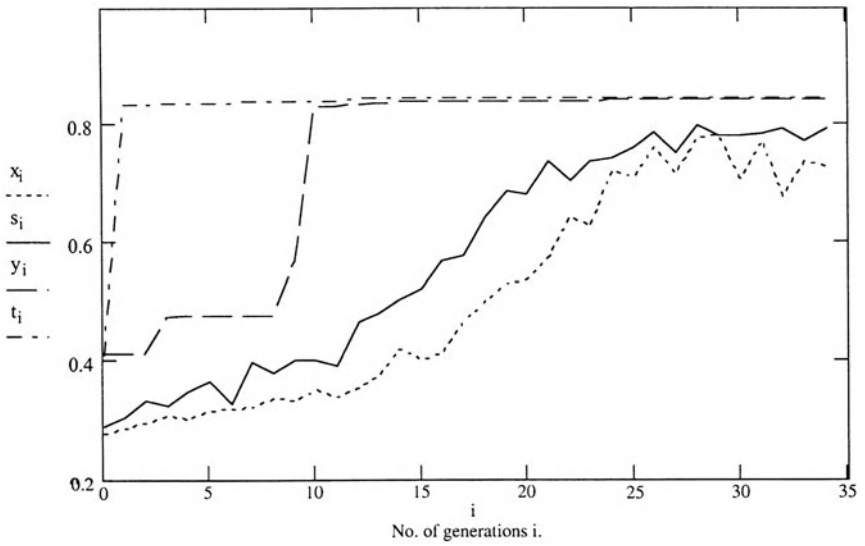

FIGURE 5.20 Illustration of increasing mutation rate on the navigation task (population 45) s_i and t_i represent average and best fitness values for a mutation rate of 0.5%, x_i and y_i represent average and best fitness values for a mutation rate of 0.003%.

We can also extract the fuzzy membership sets for the evolved FAM, as shown in Table 5.1.

	ZE	ZE	PM	PM	PL
	PM	PM	ZE	ZE	PM
x-axis = obstacle sensor input	PM	PM	NM	NM	NL
	ZE	NM	NL	NL	PL
	ZE	NM	NL	NL	PM

y-axis = target rotate sensor input

TABLE 5.1 Evolved FAM fuzzy output rules.

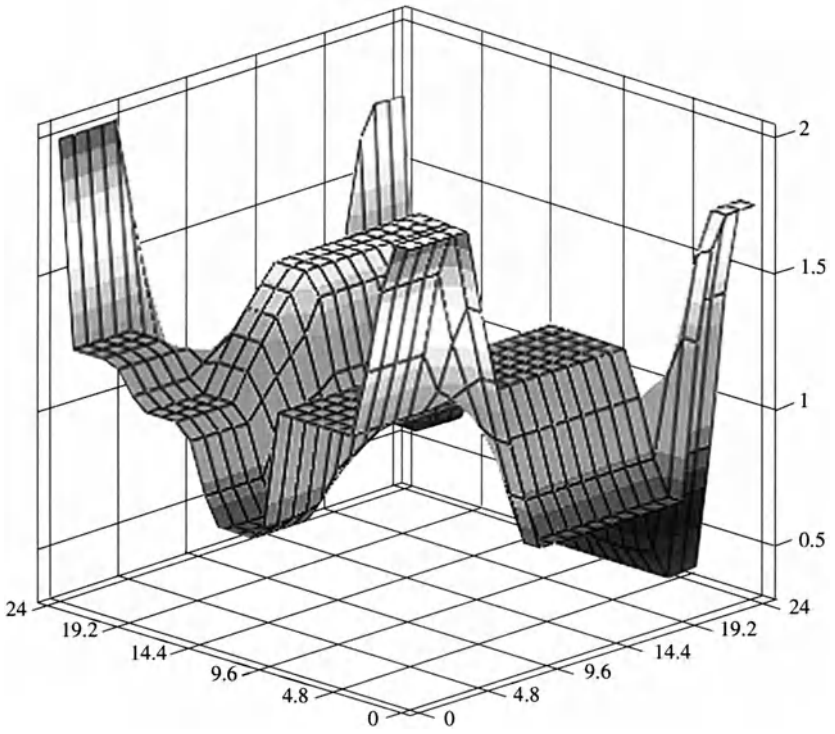

FIGURE 5.21 An evolved FAM action surface for a GA population of 45 individuals. The x-axis represents fuzzy input from the obstacle sensor and the y-axis represents fuzzy input from rotate to target sensor. The z-axis represents the strength of the defuzzified output, used to control the robot's utility of rotating toward the target.

In Figure 5.21 a 3D representation of the fuzzy output array is shown for a successful individual. We can easily interpret this result, as the GA has found a set of fuzzy rules, which apply the necessary feedback to the robot's behaviours. The surface is applying increasing negative feedback on the rotate to target behaviour and translate behaviour, due to increasing obstacle sensor signal input and/or a strong rotate to target signal.

5.14 Analysis

We can see from the results of our evolved fuzzy logic controller that a GA can enable automatic parameter design in a complex control application. In addition, it is a simple process to extract the set of fuzzy output rules, which the GA has discovered, making the whole process transparent to analysis. For example, from Table 5.1 (i.e., final column NL to PL transition) and Figure 5.21 there is a distinct shift in the evolved output rules.

This may be a useful feature the GA has discovered for the particular dynamics of this problem, or it may be due to a parameter deficiency in the GA itself. This particular rule set was repeatedly found for a range of GA parameters, which indicates that it is a useful feature in solving the problem. It is left as an exercise for the reader to experiment with the simulator and verify that this is the case.

It should also be apparent that designing the parameters for a single control FAM could be manually implemented. However, as we increase the number of parallel behaviours used to control the robot, the number of FAMs and their dimensionality rapidly increase such that manual design is extremely difficult.

The next question is how would we improve this application? First, we should consider a more flexible GA method, which would include online parameter adaptation. Second, we could include the fuzzy controller's input rules as part of the evolvable structure, as preset rules may be restricting the solution space. However, as we increase the number of parameters and variables we wish to evolve there is an associated nonlinear increase in the computational cost involved.

Alternatively, a number of researchers have applied artificial neural networks in combination with fuzzy rule systems for robotic control. This approach can offer the best features of each method, the adaptive learning capacity of neural nets, and the efficient inferencing of fuzzy logic (see Bibliography for references).

The next issue is whether a different EA technique would be more suited to this application. For example, GP has been applied to the design of mobile robot controllers (Tunstel & Lippincott, 1996). However, the resulting evolved GP control programs can be very difficult to interpret and reverse engineer. It is left as an exercise for the reader to consider how to use GP to evolve a fuzzy logic controller and maintain the transparency of a fuzzy logic system.

5.15 Summary – Evolving Hybrid Systems

The problems illustrated in this chapter hopefully convey how powerful and flexible EA methods can be in resolving multiparameter design problems. It is important to note that a genetic algorithm is simply one possible EA method that can be applied to these particular applications. The robotic application was covered in some detail, as it is a genuinely complex application in which researchers have applied the full spectrum of EA and other machine learning techniques.

Second, in a real research or industrial application some of the more sophisticated EA techniques available would be applied. Chapter 6 provides a brief overview of the most frequently used advanced EA techniques, and this should be consulted before developing any major application. For example, some form of "parameter control" in which the EA encodes values for its own parameters is frequently used, as it allows the EA to be self-adaptive.

One of the main aims of the second application was also to illustrate how an EA method is often most useful in collaboration with other machine learning techniques, such as neural networks or fuzzy logic. The reader is therefore encouraged to follow the references provided relating to hybrid machine learning methods.

Further Reading

Driankov D. Hellendoorn H., & Reinfrank M., *An Introduction to Fuzzy Control*, Berlin, Springer Verlag, 1996.

Efford N. *Digital Image Processing: A Practical Introduction Using Java*, Addison-Wesley, ISBN: 0201596237, 2000.

Gonzalez R.C. & Woods R.E., *Digital Image Processing*, New York, Addison-Wesley, 1992.

Jones J.L. Flynn A. & Seiger B., *Mobile Robots: Inspiration to Implementation*, paperback 2nd ed., A K Peters Ltd., ISBN: 1568810970, 1998.

McFarland D. & Bosser T., *Intelligent Behaviour in Animals and Robots,* Cambridge, MIT Press, 1993.

Morovec H.P. *Mind Children, The Future of Robot and Human Intelligence*, London, Harvard University Press, 1988.

Walter W.G. *The Living Brain*, London, Gerald Duckworth & co, 1953.

Welstead S.T. *Neural Networks and Fuzzy Logic Applications in C/C++,* New York, Wiley, 1994.

6
Future Directions in Evolutionary Computing

Imagination is more important than knowledge.

Albert Einstein

6.1 Developments in Evolutionary algorithms

As indicated in earlier chapters, all EA methods suffer from some fundamental problems. Prime examples include the computational cost (in processing power and memory requirements), the selection of operators and their parameters is a complex problem in itself, and the complex nature of the EA search space. This chapter considers potential solutions to these issues based on hardware approaches and parallel processing techniques. It also reviews some advanced EA methods that are commonly utilised to address the efficiency of evolution-based search methods.

6.2 Evolvable Hardware

One of the most exciting developments in EA systems has been the development of hardware that directly supports evolution-based search algorithms. In particular, the availability of Field Programmable Gate Array chips (FPGAs) allows real-time configuration of logic circuits that can evaluate a specified function (Thompson, 1996). Work at Sussex University by Adrian Thompson demonstrated that an FPGA device could be automatically configured using a genetic algorithm to solve a signal processing problem. This work opened an entirely new domain for the application of evolutionary algorithms in two important ways: first by showing how reconfigurable hardware could allow improved processing of an EA problem; and second by showing that evolution could exploit the actual physics of a synthetic environment. Thompson draws some fascinating conclusions regarding the power of such techniques:

R. Ghanea-Hercock, *Applied Evolutionary Algorithms in Java*

> A surprising hypothesis is suggested: Even within robust digital design, unconstrained evolution can produce circuits beyond the scope of conventional design rules. Previously it had been assumed that the domain of robust digital design was fully covered by conventional design rules. Given the undoubted utility of robust digital designs, it is an exciting possibility that evolution could explore novel regions of design space, containing circuits that may be better for some applications. (Thompson & Layzell, 2000)

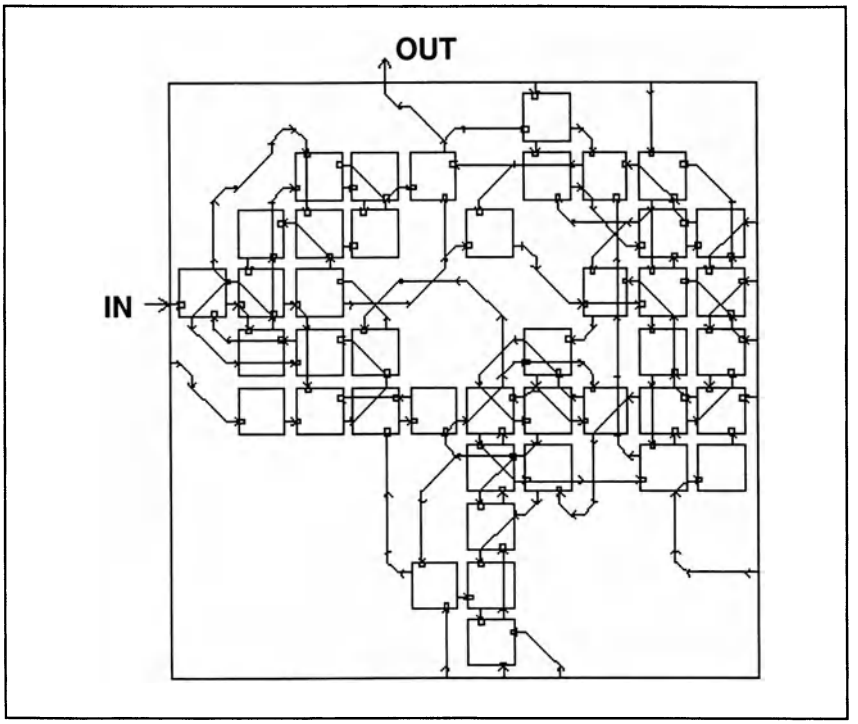

FIGURE 6.1 Example evolved FPGA circuit. (Reproduced with permission, Adrian Thompson, 1996.)

Figure 6.1 is an example evolved circuit design for a tone discriminator circuit. This figure illustrates the complex interconnections, which is typical of such evolved designs. The main difficulty in utilising an EA technique, however, is the sensitivity of the evolved design to the exact environmental conditions under which it was evolved. For example, the resulting circuit is normally sensitive to temperature. Further work by Thompson has aimed to enhance the robustness of

this process by evolving circuits across a range of temperature values (Thompson & Layzell, 2000.)

6.3 Speciation and Distributed EA Methods

Because the solution space of most optimisation problems contains multiple fitness peaks, any EA can experience convergence to a suboptimal solution. A number of methods have been proposed to address this, which rely on dividing the evolving population into smaller subpopulations. The basic idea is to create distinct species of individuals, which then perform a parallel search of each potential local optima. In order to achieve this we need a mechanism to maintain several subpopulations or "niches". This therefore requires a restriction on which individuals are allowed to recombine with each other (i.e., no free sex!). One mating restriction scheme was developed by Deb (1989), based on measures of the phenotypic and genotypic distance between individuals. Hence when a possible mating pair using genotypic separation is selected, the Hamming distance is calculated, (or Euclidean distance in the phenotypic case). If the separation value is less than some threshold value sigma, the individuals are allowed to undergo crossover.

An alternative species method simply appends a sequence of tags to each individual which identify each subgroup (Spears, 1994). Tagging has the advantage that distance metrics do not need to be computed, which therefore saves processing time.

6.3.1 Demetic Groups and Parallel Processing

An alternative method to achieving separate species is to use a distributed processing approach, in which subpopulations of individuals are isolated on separate physical processors or computers. At some point in the evaluation process individuals may be exchanged or collected from each host and evaluated (e.g., Juille & Pollack, 1995; Cant-Paz & Goldberg, 1997). These species or distributed class of EA model are often referred to as *island models* (Gordon et al., 1992; see Whitley & Starkweather, 1990 for a distributed GA design, and Gordon & Whitely, 1993, for a report on parallel GAs).

However, there is a second obvious rationale for taking this approach, which is to access the computational load and memory resources of multiple processors or machines in order to accelerate the evaluation process itself.

There are two basic forms of achieving this kind of parallelism, first by a fine-grained process in which short sequences of computation are evaluated between each synchronisation phase. This is commonly used on a multiprocessor architecture or SIMD (single-instruction, multiple-data) system. The second method is a coarse-grained parallelization, in which longer computational sequences are evaluated on separate machines before being synchronised. This second approach is of growing interest in the machine

learning community as modern intranet environments and the networking capabilities of the Java language greatly facilitate the creation of such parallel systems. A detailed example is presented to illustrate this approach in the following section.

6.3.2 Parallel Genetic Programming with Mobile Agents

This example uses Genetic Programming, which we introduced in Chapter 4 and describes work undertaken by the author at BTexact research laboratories. GP has proved to be a powerful algorithm for the automatic evolution of computer programs. However, as with all evolution-based search mechanisms the computational effort required is large and scales with the size and complexity of the problem. In particular, the variable length of GP chromosomes can lead to the utilisation of very large amounts of memory [although techniques for selecting parsimonious individuals exist (Rush, 1996).] Several groups have therefore investigated the parallel implementation of GP systems, including Koza (Andre & Koza, 1996), in order to access greater computing power than single desktop machines can provide, including:

- Fast serial machines, e.g., Cray and SGI systems
- Fine-grained SIMD machines, e.g., Thinking Machines CM-2

For example, clusters of coupled workstations forming a low-cost parallel workstation cluster have been built by Genetic Programming Inc., which operates a 1000-node Beowulf-style parallel cluster computer, consisting of 533-MHz DEC processors (GP Inc. 2001 URL).

A research platform was therefore designed which adopted an *island* model for the parallelization task, with a coarse-grained division of the population into local subpopulations [or *demes* (Wright, 1943)] residing on each machine in the available network (see Figure 6.2).

6.3.3 Mobile Agents

A simple low-cost method of performing the distribution of the genetic code across some parallel architecture was required. This section describes how mobile agents can provide a solution to this task.

The essential concept in using mobile agents for a distributed parallel processing (DPP) system is that a set of mobile software agents can automatically redistribute the task processes across the network, in order to access the best available computing resources, at any point in time. They can also take care of the management and maintenance of the processes. The mobile agent management method was tested by using a standard benchmark Genetic Programming (GP) problem, namely symbolic regression of a curve (Koza, 1994). (An intentionally simple problem was selected so that the algorithmic structure of the problem did not favour a parallel approach.) Copies of the GP problem were assigned to several agents, which were distributed across the

computing resources of a local area network (comprising four UltraSparcs and two dual-processor PCs running Windows NT). Figures 6.3 and 6.4 shows a set of typical fitness results from this configuration.

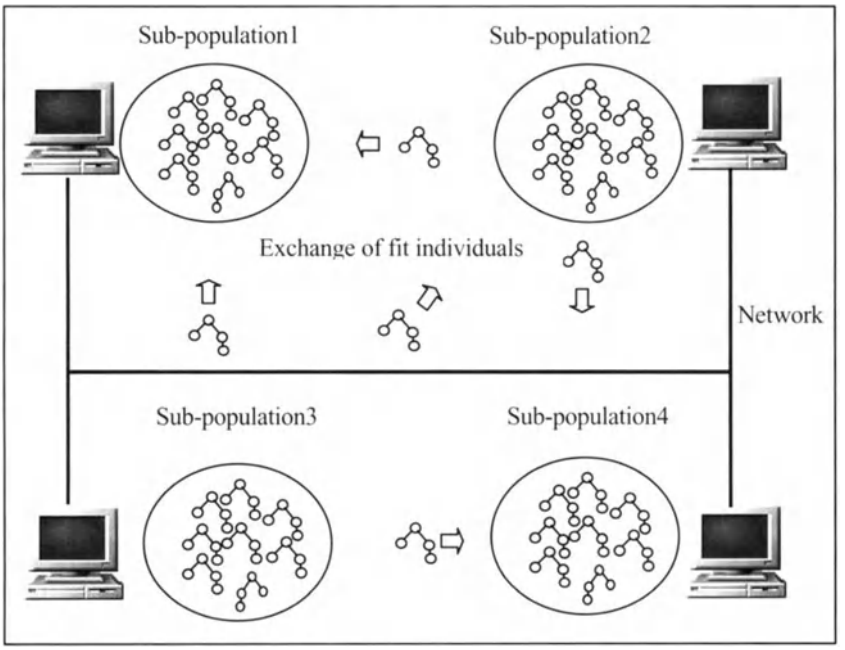

FIGURE 6.2 System design for a distributed GP application.

A mobile agent is a software component that can move between hosts within a computer network. To execute remotely, mobile agents move to a machine running a mobile agent server, which provides an interface to the underlying host machine, and facilities to transmit and receive agents.

Several mobile agent packages currently provide this functionality, including AgentTcl (Kotz, 1997), and Aglets (Clements et al., 1997). The core of an agent server is the virtual machine, onto which mobile agents are loaded and executed. This helps hide from the developer the complexities of moving between, and executing on, differently configured remote hosts. Mobile agent packages also provide facilities for high-level messaging (for interagent communication) and high-level methods for controlling agent behaviour (for instance, moving the agent to a new host).

The system used an exchange of fit individual programs (GP chromosomes) between each agent, which broadcasts optimum solutions at the end of each generation to other agents for insertion into their local "gene pool". The system therefore facilitated the construction of a demetic (or island model) exchange process, in which fit chromosomes can migrate between several subpopulations. The effect is to provide a far larger genetic search space than available on a single machine, while allowing for asynchronous communication between each computer.

The following figures show typical performance data for the distributed GP application. The migration of chromosomes, between the machines, provides an extended gene pool, which allows the system to reach significantly higher fitness levels (for the best individual), as the number of hosts in the system is increased.

The key advantage of this form of parallel processing is that the application domain maps cleanly on to the physical hardware configuration. In addition, the use of Java greatly facilitates this type of distributed system development, due to its built-in networking libraries of code.

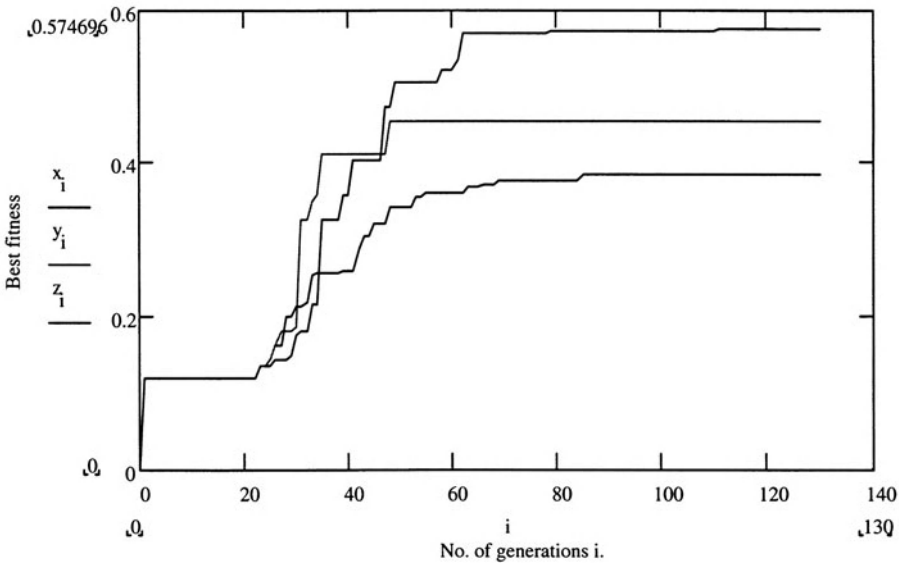

FIGURE 6.3 Best fitness values for a single host machine's population. (x_i and p_i= results for a single machine run, y_I and q_i = two machines, z_i and r_i = three machines.)

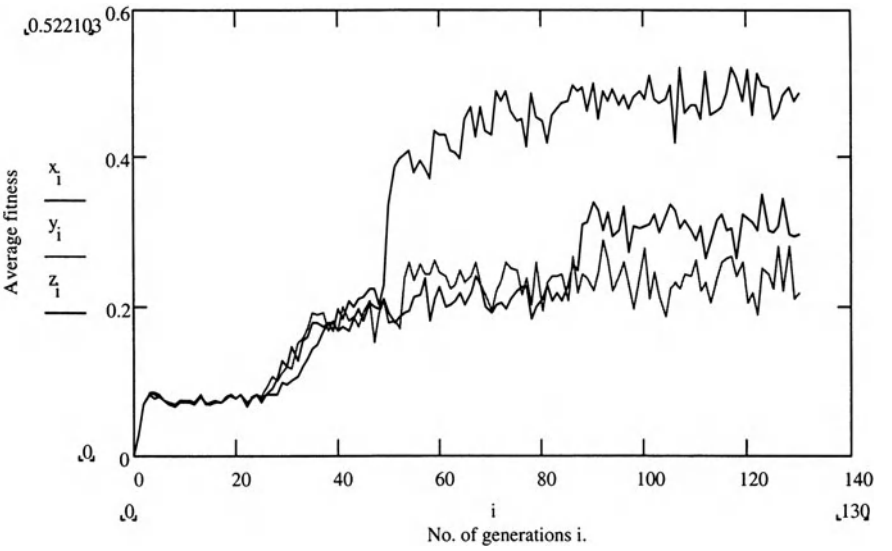

FIGURE 6.4 Average fitness values for a single host machine's population. (x_i and p_i= results for a single machine run, y_I and q_i = two machines, z_i and r_i = three machines).

6.3.5 EA Visualisation Methods

This section describes an often-neglected aspect of EA research and engineering, that is, the visualisation of EA processes, both on- and off-line. Measurements of an algorithm's effectiveness have generally involved the output of some fitness variable, typically that of the best individual and population average. However, the activity of the genetic operators, crossover, mutation, and fitness-proportionate selection, are relatively opaque. This is highlighted by the recurrent debates in the literature regarding their exact roles and effects on the efficiency of a particular evolutionary algorithm. (One example is the exact role of Holland's Building Block or Schema hypothesis.)

Several groups have investigated the need for visualising the processes within evolution-based algorithms (Collins, 1997; Wu et al., 1999.) However, most existing work has focused on offline studies of data generated by an algorithm. This is an important distinction as a key objective of this work was to create an understanding of the dynamic interactions occurring during a run of the particular algorithm. The alternative is to record all the data online and replay the data offline such that the processes of interest can be studied.

The spectrographic method that forms part of the image processing application in Chapter 5 is an example of a simple GA visualisation method. The problem to be solved is how to display the large number of variables typically present in an evolution system, which normally contains hundreds of individual chromosomes or individuals in each generation, each with several parameter variables, such as fitness, scaled fitness, complexity, and length. The normal technique is to plot a fitness curve for the fitness of the current best individual and average fitness of the population. Even plotting just individual fitness would clearly require a large number of graphs. If all the variable data for every individual is displayed, then the quantity of information rapidly becomes unmanageable. This problem is therefore closely related to other data-mining and data visualisation problems. The aim is therefore to display the variable data in a compressed form while retaining maximum information content.

We therefore require a mapping procedure, which compresses the large quantity of numeric data present in a GA. A human user of the system can then be presented with an intuitive visual representation of the data.

The method used in Chapter 5 is to take each genetic variable from an evolution algorithm and map this to an index of a colour component, with red, green, and blue values of intensity, where the colour value is some function of the variable's value.

In the case of a simple genetic algorithm this means translating the individual values of the genome into a graded colour scale. The benefit is a complete two-dimensional display of the genetic diversity in the GA's current population.

By plotting a geometric figure or shape for each individual in the current population with the calculated colour value, a visual image can be constructed which represents the current genetic state of the whole population. The shape of the figure can also encode a fourth variable if required.

In the current version each individual is represented by a vertical narrow bar arranged vertically, where each column of bars represents a single individual. A sequence of columns with one per individual can then represent the whole population, as a two-dimensional grid. This therefore provides a two-dimensional *spectrographic* representation of the data; see Figure 5.3, Chapter 5.

6.4 Advanced EA Techniques

Within the past decade a number of established techniques have been established that are frequently applied in EA applications. These reflect the growing body of theoretical and empirical knowledge in the application of EA techniques. A few of these methods will be discussed in the remainder of this chapter, with the aim of directing the reader to relevant examples and references.

6.4.1 Multiobjective Optimisation

It is a fact of life that we rarely, if ever, achieve all our current goals, since multiple goals of any agent are frequently in conflict with each other. The recommended text in the reading section on advanced evolutionary algorithms contains an excellent introduction to this topic by Fonseca (p. 25, 2000.)

Most real-world applications for EA methods contain conflicting sets of optimisation criteria; hence we require some means for achieving multiobjective optimisation. In the robot example from Chapter 5 it is clear that we would like the robot to take the shortest route to the target, but this is in conflict with the need to safely avoid obstacles en route.

Given some set of competing performance criteria such as resource efficiency, cost, and speed, when no further improvement can be made in one without degrading the quality of another, then the system is said to be Pareto-optimal (see Pareto, 1906; translation, Kelly, 1971).

Fonseca divides approaches to multiobjective optimisation into three categories (Fonseca & Flemming, 1995):

Plain aggregating approaches. Objectives are numerically combined into a single objective function to be optimised. The robot example in Chapter 5 takes this approach.

Population-based non-Pareto approaches. Different objectives affect the selection or deselection of different parts of the population in turn.

Pareto-based approaches. The population is ranked, making direct use of the definition of Pareto dominance.

6.4.2 Methods: Weighted Sum Approach

The set of required objectives for a problem may be defined as $f_1 ... f_n$, which are then assigned positive weights $w_1 ... w_n$. These weights are summed to generate a scalar measurement of cost for every individual. The derived value can then be simply plugged into whatever selection method the user has specified (rank, tournament, etc.)

$$f(a_i) \mapsto \sum_{k=1}^{n} w_k f_k(a_i) \tag{6.1}$$

6.4.3 Minimax Method

This method attempts to minimise the maximum of the set of objectives $f_1 ... f_n$. Alternatively, it may be performed by minimising the maximum weighted difference between the objectives and corresponding goals (Wilson & Mcleod, 1993.) The paper by Wolpert (1995) on the *No-free lunch theorem* discusses

some interesting aspects of this method in relation to evolution-inspired search methods.

6.4.4 Parameter Control

If the reader has found time to experiment with either of the test applications in Chapter 5, will by now understand the difficulty in selecting suitable parameters for an EA application. Choosing good values for the mutation rate, tournament size of selection, population size, or genome length is an intrinsically difficult task. A number of researchers have therefore investigated the possibility of automating the selection of some critical parameter values during the run of an EA (Grefenstette, 1986). Such methods typically use a measurement of the effectiveness of the current parameter set as feedback to modify the parameters online. A simple example would be to monitor the rate of improvement in the global fitness and to reduce the mutation rate as the algorithm converged on particular optima. Holland suggested a simple time-dependency of mutation rates in his original work on the GA (Holland, 1975).

6.4.5 Diploid Chromosomes

One interesting variance between biological evolution mechanisms and those used in most evolutionary algorithms is that organisms normally use a diploid chromosome structure, while EAs use a haploid design. First, however, we need a basic definition of the term "diploid:"

> Pertaining to homologous chromosomes, where each chromosome number has a pair of chromosomes, such as the 23 pairs in humans totaling our 46 chromosome compliment. http://www.biology-online.org

A diploid chromosome structure is therefore composed of a pair of chromosomes. One advantage this offers is to allow memory to be incorporated into the individual's chromosome structure. In an EA context the choice between the two values requires some form of dominance function. Lewis et al., (1998) give a useful comparison of the relative merits of haploid and diploid designs. The main proposed application area for a diploid EA is when the fitness function is time-varying. Hence the additional information stored in the chromosome can provide a secondary source of diversity within the population. In addition, the diploid structure may enable the retention of long-term memory of past solutions, which will be useful if the fitness landscape is time-dependent. However, very little experimental work has yet been performed in this area.

6.4.6 Self-Adaptation

A related method to parameter control is to use self-adaptation, that is, to enable the evolutionary process to modify the control parameters during a run of the algorithm (Bäck 1992). In this case the parameters are encoded into the genome

of each individual in addition to the function defining genes. One example is the self-adaptation of mutation parameters by Bäck (1992). Bäck applies mutation and selection operators to the mutation rate itself within a binary chromosome representation. Hence the mutation parameters can evolve in parallel with the object variables. However, it is important to note that no user-defined fitness criterion is applied to the evolution of the mutation parameter. It is the indirect feedback from the success of the functional chromosome that drives the evolution of the control parameter, namely, mutation.

Such methods of self-adaptation have proved to be very robust and effective across a range of EA methods. It has been pointed out that natural evolution also exploits self-adaptive mechanisms:

> This robustness of the method clearly indicates that a fundamental principle of evolutionary processes is utilized here, and in fact it is worth mentioning that the base pair mutation rate of mammalian organisms is in part regulated by its own genotype by repair enzymes and mutator genes encoded on the DNA. The former are able to repair a variety of damage of the genome, while the latter increase the mutation rate of other parts of the genome. (Bäck, 2000, p. 208).

6.5 Artificial Life and Coevolutionary Algorithms

The field of Artificial Life developed as an attempt to construct life in the computational realm (Langton, 1987). It does not appear to have succeeded in this yet, in part because of the extreme difficulty of defining what is meant by life. But Artificial Life has been successful in exploring some aspects of life-as-it-could-be as opposed to life-as-we-know-it (Langton, 1995). In doing so it has tended to a synthetic approach rather than the analytic approach taken in biology and most other sciences. The principal phenomenon of interest is that of emergence, where overall properties of a system arise from the interaction of many subunits. As computer systems have increased in capacity to deal with complicated systems in realistic time scales, Artificial Life systems have allowed advances in evolutionary computation, the modeling of complex systems, and the simulation of ecological and behavioural systems. Although it has resulted in effective models of some biological systems, it has tended to be ignored by many biological researchers, generally due to the difficulty of testing analytical hypotheses in the systems Artificial Life researchers generate.

Although Artificial Life as a "discipline" has been active for over a decade, its unconventional origins perhaps explain why it cannot be described as having reached a stage of maturity. Recent conferences in the field (Bedau et al., 2000) show an enormous diversity of models taking inspiration from a wide range of biological phenomena. This suggests great potential for innovation, although in a rather haphazard and undirected way. Because of the diverse sources of

inspiration that Artificial Life systems draw upon, it is difficult to point to a set of characteristics that all Artificial Life systems will have, unlike the major evolutionary computational systems discussed above. Perhaps for this reason, Artificial Life has been a weaker source of applications than evolutionary systems per se. Nevertheless, many of the other innovations mentioned in the following sections can be argued as falling within the field of Artificial Life, as well as software approaches to bio-analogous computing, whether or not they include any evolutionary component.

Ecological Systems

Living organisms exist in a web, or network, of interactions with other organisms. The form of these interactions is very familiar from everyday life; they include feeding, cooperation, and competition. Similarly, computing and telecommunications technology is typically organised into networks, with different types of interaction between devices in the network; it is therefore not surprising that comparisons have been drawn between nature and technology in this way.

Ecology is the scientific discipline concerned with the study of living organisms in their environment, including the interactions between them (Cherrett, 1989). Living creatures are organised into hierarchical structures of similar entities, of which the biosphere is the largest unit, within which are placed, successively, ecosystems, communities, populations, and individuals.

Analogies between computational systems and natural ecosystems have led to the concept of "computational ecology" (Huberman, 1988), where the dynamics of interacting networks of computers and associated devices are considered as an entity, rather than focusing on the performance of individual machines. This approach seems appropriate, given the great interdependence of computing devices and the communications systems that link them. The related concept of "information ecosystem" (FET, 1999) emphasises the information being transmitted between networked devices, rather than the devices themselves, but reinforces the dynamic nature of information flow.

Ecology has inspired computer systems in other ways. One of the directions of the field of Artificial Life has been in the development of computer systems inspired by ecology. An early example was the Echo system developed by Holland (described in Holland, 1995). This system models a 2D space in which agents move around. Sources of food are placed in the space, that agents must find, and the agents also have the capability of interacting with each other in several ways (mating, trading, or fighting).

Developmental and Cellular Systems

Complex living organisms such as humans develop from very simple single-cell origins into complicated multicellular creatures (Gerhart & Kirschner, 1997). Development is in many ways a very appropriate source of computational

algorithms, because it incorporates substantial flexibility along with robustness, which means it is relatively stable to perturbation while providing the means to generate complex structures.

Development has been used in a variety of telecommunications applications (Tateson, 2000). An example (Tateson, 1998) uses a process found in the development of the bristles of the fruit fly *Drosophila* to allocate channels for a cellular telecommunications network. The appropriate pattern of bristles in the adult *Drosophila* is arrived at during development by mutually inhibitory interactions among neighbouring cells. Cells that develop bristles inhibit bristle development in neighbouring cells. This process of inhibition of neighbouring cells can be applied to the prevention of interference in mobile telephone networks and is currently being developed for implementation in real-world mobile networks.

The above discussion of aspects of evolutionary computation presumes in general that a single population, possibly subdivided into subpopulations, is used to solve one problem. But what if two or more populations evolve against each other? Their interaction could stimulate more rapid evolution to a solution than if they evolved independently. This is the principle behind coevolutionary algorithms, where problem solving is stimulated by the coevolution of two or more populations. Coevolution has attracted a great deal of attention in biology as a potential mechanism for producing evolutionary innovation (Thompson, 1994), although it often proves more difficult to identify than predicted. Hence it is not surprising that this is an area that has attracted interest for improving the performance and applicability of evolutionary algorithms. Coevolutionary algorithms involve the dependence of the fitness function of one population on the characteristics and interaction with that of another population. Typically, this may be in the form of a competitive interaction between individuals, but it need not be, and the form of interaction between the populations is an active area of research as coevolutionary algorithms are applied in different contexts (Paredis, 2000).

6.6 Summary

The emerging field of evolvable hardware, in FPGA devices, is particularly exciting as it enables the possibility of real-time learning by EA systems. It also opens an entirely new domain of physical evolution in synthetic substrates, which take advantage of the actual physics of electronic systems. Providing such systems can achieve robust solutions, then it could lead to entirely new classes of hardware, which fully exploit the capabilities of the silicon substrate; including their application to evolving analogue electronic devices. (NASA has been particularly interested in the possibilities offered by this technology and has arranged a series of workshops on Evolvable Hardware.)

Second, the development of easy-to-use distributed software platforms has provided the capability to exploit the significant computational resources in a modern intranet environment that allows for large-scale and memory-intensive applications to be developed. Modern networking capabilities, which are intrinsic to Java, have been an essential component in this process (in particular the functionality provided by the socket class). The recent emergence of GRID computing will further enable this process by providing direct access to very large-scale computational resources (see http://www.gridforum.org).

However, it is the development of a deeper theoretical understanding of EA processes that should ultimately lead to a more efficient application of evolution as a machine learning method. An example is the use of self-adaptive parameter optimisation, which has been proven to significantly improve EA performance.

Further Reading

Back T., *Self-adaptation in genetic algorithms*, in Self-adaptation in genetic algorithms. In Varela and Bourgine ed., p. 263 - 271, 1992, Towards a Practice of Autonomous Systems: Proceedings of the First European Conference on Cambridge, Artificial Life, MIT Press, 1992.

Back T., Fogel D.B., & Michalewicz Z. Eds., *Evolutionary Computation 2, Advanced Algorithms and Operators*, Bristol, UK, Institute of Physics, 2000.

Grand S., *Creation: Life and How to Make It*, Weidenfeld; ISBN: 0297643916, 2000.

Levy S., *Artificial Life*, Penguin Books; ISBN: 0140231056, 1993.

Thompson A., *An evolved circuit, intrinsic in silicon, entwined with physics*, Proc.1st Int.Conf. on Evolvable Systems, ICES'96, ed. Tetsuya Higuchi, Masaya Iwata, L. Weixin, pp. 390 - 405, Springer-Verlag, LNCS, Vol.1259, 1997.

Thompson A., *Hardware Evolution Automatic Design of Electronic Circuits in Reconfigurable Hardware by Artificial Evolution*, Springer, ISBN: 3-540-76253-1, 1998.

7
The Future of Evolutionary Computing

> Today the theory of evolution is about as much open to doubt as the theory that the earth goes round the sun.
>
> Richard Dawkins, *The Selfish Gene*

7.1 Evolution in Action

We have demonstrated how evolution-based techniques may be a powerful tool in automated machine learning. However, they are not a panacea for all automated learning domains, and careful selection of the right combination of techniques is critical. For example, artificial neural networks, Bayesian networks, and Case Based Reasoning are equally powerful methods.

As illustrated in Chapter 5, when used in collaboration with other soft computing techniques, EA can generate very productive results. Consequently, hybrid combinations of EA with ANN and fuzzy rule systems are being actively developed by a number of researchers (e.g., Kosko, 1992; Saffioti, 1997).

7.2 Commercial Value of Evolutionary Algorithms

A substantial number of companies are increasingly interested in the commercial application of evolutionary methods. Modern engineering, scientific, and business problems are increasingly complex, with multiple parameters and are often poorly understood at a theoretical level. Hence there is a growing demand for robust adaptive learning and optimisation methods that can resolve such problems. The online European web resource for evolutionary systems provides a database of companies and research groups involved in this area:
http://evonet.dcs.napier.ac.uk/evoweb/resources/evolution_work/index.html

One commercial example is provided from the company Cap Gemini:

Cap Gemini in the Netherlands and KiQ Ltd in the UK have co-developed a system called Omega which uses several genetic algorithm flavours to solve

R. Ghanea-Hercock, *Applied Evolutionary Algorithms in Java*

> marketing, credit and insurance modelling problems. From a customer portfolio of known behaviour, Omega generates a mathematical model which can then be used to predict the behaviour of customers outside the known portfolio. This approach can be applied to credit scoring, targeting mailings, loyalty modelling and fraud detection. Recently a Dutch bank carried out a comparison between Omega and its established credit scoring system based on expert knowledge. Omega yielded a substantial improvement in both quantity (more loans were offered) and quality (credit risk was reduced) of loans both major factors in determining the overall profits of the bank. (http://www.capgeminin.nl/)

Many other application domains are also under commercial development, including (based on the Evonet listing)

- Aerospace, http://www.recherche.enac.fr/opti/
- Business planning and operations research, http://www.quadstone.com/
- Biology and chemistry, http://research.unilever.com/
- Telecommunications, http://www.labs.bt.com/projects/ftg.htm
- Entertainment and media, http://www.attar.com/pages/case_c4.htm
- Finance, http://www.cs.ucl.ac.uk/staff/P.Bentley/
- Manufacturing, http://www.palisade-europe.com/
- Medicine, http://www.mis.coventry.ac.uk/~colinr/cig.html
- Electronics and evolvable hardware, http://domme.ntu.ac.uk/mechdes

It is likely to be the commercial exploitation of EA methods that will lead to the most substantial future developments in the field. Successful evolutionary techniques should therefore replicate and fill the automated machine learning ecosystem!

7.3 Future Directions in Evolutionary Computing

There remains a vast amount of virgin territory to be explored in the realms of evolutionary algorithms and hardware. Current hardware and software can easily support the process of EA. However, the theoretical basis for how to best apply EA to specific engineering domains remains patchy. In particular, there has been an excessive focus on the crossover and mutational operators at the expense of better representational methods for applied EA applications.

The following are a few of the more interesting issues that urgently need experimental and theoretical development.

7.3.1 Alife and Coevolution

> in this place it takes all the running you can do, to keep in the same place.

> Observation to Alice by the Red Queen, in Lewis Carroll's *Through the Looking Glass.*

The ecological and organism-based approach of Alife offers some powerful techniques for enhancing the performance and capabilities of EA, as discussed in Chapter 6. In particular the use of coevolving populations and host-parasite models allows for an open-ended approach (Hillis, 1990). However, while many researchers have advocated the benefits of coadaptive methods, quantitative and theoretical assessment of coevolving populations have been minimal (exceptions are presented in Cliff & Miller, 1995, and Bedau & Packard, 1992). In particular, coevolution may lead to the Red Queen effect (L. van Valen, 1973) in which coadapting populations continuously alter each other's fitness landscape. This may be a useful tool for preventing premature convergence of a population, but it also significantly increases the complexity of the evolving system and raises difficulties in measuring performance.

7.3.2 Biological Inspiration

As indicated in Chapter 3 the bulk of current EA techniques relies on a very simplified model of biological genetics, using haploid crossover and point mutation. While a few researchers have investigated diploid crossover (Smith, 1988), there is still a minimal application of the current knowledge of biological genetic processes. In particular, the understanding of how genes regulate each other's activity needs significant study with respect to artificial evolution. Another fruitful area would be a comparative study of the transcription and translation mechanisms in an artificial evolution context. Similarly, the role that introns play in preserving useful schema is an active research domain (although this has been the subject of greater study in the genetic programming community, where junk genes lead to bloat of the tree representation genomes).

7.3.3 Developmental Biology

As pointed out by Brian Goodwin (Goodwin, 1994), modern biology has focused on the power of genetic descriptions at the expense of the developmental or morphogenetic processes that shape the phenotype of all organisms. Researchers in EA have generally followed the same path as this neo-Darwinian view – that the encoding within the genes completely defines the phenotypic expression.

Some researchers, however, have realised that the developmental forces active during the transformation from genotype to phenotype are also important, and have tried to apply such processes to EA (e.g., Gruau, 1994). The key expression in defining this approach is how to design for *self-guiding growth* in artificial evolution.

It would also be interesting to consider the area of Proteomics, or protein folding. This is a critical and poorly understood phase in the formation of biological organisms and could offer novel insights into artificial phenotype generation.

7.3.4 Adaptive Encoding and Hierarchy

A further area worthy of greater consideration is that of adaptive encoding and encapsulation methods (Mitchell, 1996). Mitchell emphasises the role of self-adapting encoding operators in GA systems and points out that for GAs to be capable of evolving complex structures then open-ended processes of encoding are essential. For example, a detailed description of a hierarchical GA method is given by Man et al., (1999). They incorporate a multilevel set of regulatory genes within a genome, which control whether a particular gene is expressed or not. However, significantly more work is clearly required in this area.

Chapter 6 discussed some of the details surrounding self-adapting EA processes and contains useful references to pursue (for example, Bäck, 2000).

7.3.5 Representation and Selection

Another aspect of EA that demands attention is that of how to select the best representation scheme and operators for specific problem domains. As indicated in Chapter 5, it is possible to use complex objects with internal structure as the base elements of a chromosome. However, there is an absence of theory to guide such a process. Most related work has focused on the tree and graph representations used in Genetic Programming, which is only a small subset of possible complex representation schemes.

7.3.6 Mating Choice

Most current EAs use some form of random mating process to select chromosomes for crossover. One area where research in the Alife field may prove valuable is in developing an understanding of how nonrandom mating techniques can be applied (Mattfield et al., 1994). One example is to use a tagging scheme such that individuals in a population can select mates by tag type, which may indicate useful traits the other individual holds.

7.3.7 Parallelism

As discussed in Chapter 6 the advent of large-scale parallel computing platforms now provides an ideal environment for complex development within EA. In particular, it enables a demetic island model of evolution with a heterogeneous distribution of operators across the island populations. Recent work is also considering the application of evolvable FPGA chips as a fine-grained parallel hardware platform for EA development.

The emergence of the computing Grid infrastructure also provides a coarse-grained platform for very large-scale EA applications to be developed. However, very little experimental work has yet been performed that utilises such platforms. Such systems really need automated mechanisms for allocating EA processes across large-scale networks, and these tools are still in development.

7.4 Conclusion

This text aims to provide the reader with an accessible guide and introduction to developing real-world applications in Java, with a particular focus on the Genetic Algorithm. Significant levels of theoretical detail have been omitted, but pointers are given to the necessary literature where greater depth may be sought.

The impact of EA techniques in computing applications is also increasing. This is due to the continuous expansion in available computing power and to the refinement of the software tools can leverage that power.

To summarise, the availability of the Java programming language has made the process of developing powerful EA software far simpler and more accessible to the student. Java was designed as a stable and simple language, which removes the burden of memory management from the user. This single feature eliminates the most frequent cause of program failure when developing EA code. Further developments in object-oriented software will no doubt contribute to the spread of EA methods into mainstream IT engineering and design.

The author welcomes feedback on the utility of the included Java toolkit for GA experimentation and ideas for improvements to the code. Ideas for development of the robot simulator should be directed to Gary Lucas (see web reference for details).

> It is not the strongest of the species that survive, nor the most intelligent, but the one most responsive to change.
>
> Charles Darwin, *On the Origin of Species*

Bibliography

Ackley D., A Connectionist Machine for Genetic Hillclimbing, Kluwer Academic, 1987.

Agoston E., van der Hauw J.K., and van Hemert J.I., Graph Coloring with Adaptive Evolutionary Algorithms, Heuristics, 4(1), 1998.

Altenberg L., The evolution of evolvability in genetic programming, pp.47 - 74, in K.E. Kinnear, ed., Advances in Genetic Programming, Cambridge, MIT Press, 1994.

Andre D., and Koza J.R., Parallel genetic programming: A scalable implementation using the transputer network architecture, In P.J. Angeline and K.E. Kinnear Jr., eds., Advances in Genetic Programming 2, Cambridge, MIT Press, Chapter 16, 1996.

Ardissono L., Barbero C., Goy A., and Petrone G., An agent architecture for personalized Web stores, Proc. 3rd Int. Conf. on Autonomous Agents, (Agents '99), Seattle, WA, May 1999.

Arkin R.C., Integrating Behavioural, Perceptual, and World Knowledge in Reactive Navigation, Robotics and Autonomous Systems, Vol. 6: p.105 - 122, 1990.

Arkin R.C., The Impact of Cybernetics on the Design of a Mobile Robot System: A Case Study, IEEE Transactions on Systems, Man, and Cybernetics, Vol. 20, (6), Nov./Dec., pp.1245 - 1257, 1990.

Arkin R.C., "Reactive Robotic Systems", Georgia Institute of Technology, Internal report, 1991.

Ashwin R., Arkin R., Boone G., and Pearce M., Using Genetic Algorithms to Learn Reactive Control Parameters for Autonomous Robotic Navigation, Adaptive Behavior, 2(3), 1994.

Aylett R., Coddington A., Barnes D., and Ghanea-Hercock R., A Hybrid Multi-Agent System for Complex Task Achievement by Multiple Co-operating Robots, Multi-Agent Workshop, Oxford, 1995.

Baker J.E., Adaptive selection methods for genetic algorithms, In J.J., Grefenstette, ed., First Int. Conf. on Genetic Algorithms and Their Applications, Erlbaum, 1985.

Bäck T., Fogel D.B., and Michalewicz Z., Evolutionary Computation 2 – Advanced Algorithms and Operators, Bristol, Inst. of Physics, 2000.

Banzhaf W., Nordin P., Keller R.E., and Francone F.D., Genetic Programming – An Introduction, San Francisco, Morgan Kaufmann, 1998.

Barnes D.P., and Gray, J.O., Behaviour Synthesis for Co-operant Mobile Robot Control, Proc. IEE Int. Conf. on Control 91, Vol. 2: 1135 – 1140, 1991.

Bedau M.A., and Packard N.H., Measurement of evolutionary activity, teleology, and life, In C.G. Langton, C. Taylor, J.D. Farmer, and S. Rasmussen, eds., Artificial Life II, Addison-Wesley, 1992.

Bedau M.A., McCaskill J.S., Packard N.H., and Rasmussen S., (eds.,): Artificial Life VII: Proc. of the Seven'th Int. Conf. Artificial Life, Cambridge, MIT Press, Cambridge, 2000.

Bienert P., Rechenberg, I., and Schwefel, H.P., Messung kleiner Wandschub spannungen bei turbulenten Grenzschichten in Ablosenahe, Technical Report of the Hermann Fottinger-Institute for Fluid Dynamics, NUMBER Wi 8/45, 1966.

Bonarini A., Learning dynamic fuzzy behaviours from easy missions, Proc. IPMU 96, Proyecto Sur Ediciones, pp.1223 - 1228.

Bonsma E., Shackleton M., and Shipman R., Eos - an Evolutionary and Ecosystem Research Platform, BT Technology Journal, Vol. 18, (14), 24 - 31, Oct. 2000.

Borrie J.A., Modern Control Systems, London, Prentice Hall, pp.134 - 135, 1986.

Brooks R.A., A Robust Layered Control system for a Mobile Robot, IEEE, J. Robotics and Automation, RA-2, (1), p.14 - 23, 1986.

Brooks R.A., Intelligence without representation, Artificial Intelligence, Vol. 47:139 - 159, 1991.

Cantú-Paz E., and Goldberg D.E., Predicting speedups of idealized bounding cases of parallel genetic algorithms, Proc. of the Seventh Int. Conf. on Genetic Algorithms, San Francisco, Morgan Kaufmann, 1997.

Cherret M., ed., Ecological Concepts, Oxford, Blackwell Scientific, 1989.

Choi A., Optimizing local area networks using genetic algorithms, in J.R. Koza, D.E., Goldberg, D.B., Fogel, and R.L. Riolo, eds., Genetic Programming: Proc. First Annual Conf., pp. 467 - 472, Stanford University, Cambridge, MIT Press, 1996.

Clements P.E., Papaioannou T., and Edwards J.M., Aglets: Enabling the Virtual Enterprise, ME-SELA '97, from, http://www.trl.ibm.co.jp/aglets/index.html, 1997.

Cliff D., and Miller G. F., Tracking the red queen: Measurements of adaptive progress in coevolutionary simulations, F., Mor'an, A., Moreno, J. J., Merelo, and P., Chac'on, eds., Advances in Artificial Life: Proc. Third European Conf. Artificial Life, pp.200 - 218, 1995.

Cliff D., and Miller G.F.,Co-evolution of Pursuit and Evasion: Simulation Methods and Results, from animals to animats 4, editors Maes P., and J. Maja Mataric and Jean-Arcady Meyer and B.P., Jordan and S.W. Wilson, Cambridge, MIT Press, pp.506 - 515, 1996.

Collins T., Using Software Visualization technology to help evolutionary algorithm users validate their solutions, T. Baeck, ed., Proc. Seventh International Conference on Genetic Algorithms (ICGA97), Michigan, pp.307 - 314. Morgan Kaufmann, 1997.

Connell J.H., SSS: A Hybrid Architecture applied to Robot Navigation, Proc. IEEE Robotics & Automation Conf., pp. 2719 - 2724, 1992.

Currie K. and Tate A., O-Plan: The open planning architecture, Artificial Intelligence, Vol. 52, (1), Nov. 1991.

Davidor Y., Genetic Algorithms and Robotics, Singapore, World Scientific, 1991.

Day B. and Knudsen J., Image processing with Java 2D, online report at www.javaworld.com/javaworld/jw-09-1998/jw-09-media_p.html, 1998.

De Jong K. A., An Analysis of the Behaviour of a class of Genetic Adaptive Systems, Ph.D. thesis, Ann Arbor, University of Michigan, 1975.

Driankov D., Hellendoorn H., and Reinfrank M., An Introduction to Fuzzy Control, Berlin, Springer-Verlag, 1996.

Erwin D.H., Lessons from the Past: Biotic Recoveries from Mass Extinctions, The Future of Evolution Colloquium, Mar. 2000, National Academy of Sciences, www.santafe.edu-/sfi/publications/Abstracts/00-12-067abs.html, 2000.

Eustace D., Adaptive Parameter Adjustment of the Behaviour Synthesis Architecture, internal Ph.D. report, University of Salford, 1994.

FET: EU Future and Emerging Technologies proactive initiative on Universal Information Ecosystems, http://www.cordis.lu/ist/fetuie.htm, 1999.

Firby J.R., Adaptive Execution in Dynamic Domains, Ph.D. thesis, Yale University Dept. of Computer Science, 1989.

Fisher R., Perkins S., Walker A., and Wolfart E., online report at www.dai.ed.ac.uk/HIPR2/, 2000.

Fogel L.J., Owens A. J., and Walsh M.J., On the evolution of artificial intelligence, Proc. 5th National Symp. on Human Factors in Electronics, IEEE, 1964.

Fogel D.B., Phenotypes, Genotypes, and Operators in Evolutionary Computation, Proc. IEEE Int. Conf. Evolutionary Computation, Perth, Australia, IEEE Press, pp. 193 - 198, 1995.

Forrest S., Scaling fitnesses in the genetic algorithm, in Documentation for Prisoners Dilema and Norms, Programs that use the Genetic Algorithm, 1985.

Friedberg R. M., A learning machine, part I, IBM Journal, Vol.2: pp.2 - 13, 1958.

Gamma E., Helm R., Johnson R., and Vlissides J., Design Patterns: Elements of Reusable Object-Oriented Software, Reading, MA, Addison-Wesley, 1995.

Gat E., Integrating Reaction and Planning in a Heterogenous Asynchronous Architecture for Controlling Real World Mobile Robots, Proc. Tenth National Conf. on Artificial Intelligence (AAAI), 1992.

Gat E., On the Role of Stored Internal State in the Control of Autonomous Mobile Robots, AI Magazine, Spring 1993, pp. 64 - 73.

Gat E., On the Role of Theory in the Control of Autonomous Mobile Robots, AAAI Symposium on Applications of AI theory to Real World Autonomous Mobile Robots, 1994.

Geist A., et al. PVM: Parallel Virtual Machine A Users' Guide and Tutorial for Networked Parallel Computing, MIT Press Scientific and Engineering Computation, Janusz Kowalik, ed., 1994.

Gerhart J. and Kirschner M., Cells, Embryos and Evolution, Oxford, Blackwell Science, 1997.

Ghanea-Hercock R.A. and Fraser A.P., Evolution of Autonomous Robot Control Architectures, AISB Workshop, Leeds, 1994.

Ghanea-Hercock R.A. and Barnes D.P., Mobile Robot Dynamics: Chaos in Reactive Control Architectures, Proc. IASTED Int. Conf., Cancun, Mexico, 1995.

Ghanea-Hercock R.A. and Barnes D.P, An Evolved Fuzzy Reactive Control System for Co-operating Autonomous Robots, Fourth Int. Conf. Simulation of Adaptive Behaviour, Cape Cod, MA, 1996.

Ghanea-Hercock R. and Barnes D.P., Coupled Behaviours in the Reactive Control of Co-operating Mobile Robots, Int. J. Advanced Robotics, Japan, special issue on Learning and Behaviours in Robotics, April, Vol. 10: (2), 161 - 177, 1996.

Ghanea-Hercock R., Collis J. and Ndumu D., Co-operating Mobile Agents for Distributed Parallel Processing, Mobile Agents Workshop, in Autonomous Agents, Seattle, 1998.

Goldberg D.E., Computer-aided gas pipeline operation using genetic algorithms and machine learning, University of Michigan, Department of Civil Engineering, Dissertation Abstracts International, 44(10), 3174B, 1983.

Goldberg D.E., Genetic Algorithms, in Search Optimization and Machine Learning, Reading, MA., Addison Wesley, 1989.

Goldberg D.E., Korb B. and Deb K., Messy genetic algorithms: Motivation, analysis, and first results. Complex Systems, Vol.3: pp. 493 - 530, 1989.

Goldberg D. K. and Deb K., Comparative analysis of selection schemes used in genetic algorithms, in Foundations of Genetic Algorithms, Morgan Kaufmann, pp.69 - 93, 1991.

Goodwin B., How the Leopard Changed Its Spots, the Evolution of Complexity, London, Phoenix Edition, 1997.

Gonzalez R.C. and Woods R.E., Digital Image Processing, New York, Addison-Wesley, 1992.

Gordon V.S. and Whitley D., Serial and Parallel Genetic Algorithms as Function Opimizers, ICGA-93 conf., the 5th Int. Conf. on Genetic Algorithms, Urbana-Champaign, IL, pp. 177 - 183, Morgan Kaufmann, 1993.

Grand S., Cliff D. and Malhotra A., Creatures: Artificial Life Autonomous Software Agents for Home Entertainment, Millenium Interactive Ltd., www.cyberlife.co, 1998.

Grefenstette J.J, Optimization of Control Parameters for genetic algorithms, IEEE Transactions on Systems, Man, and Cybernetics Vol.16: (1), 122 - 128, 1986.

Grefenstette J.J and Baker J.E., How genetic algorithms work: A critical look at implicit parallelism., in D.J., Schaffer ed. Proc. Third Int. Conf., Genetic Algorithms, pp.20 - 27, San Mateo, CA, Morgan Kaufmann, 1989.

Grefenstette J.J., Conditions for implicit parallelism, in G. Rawlins, ed., Foundations of Genetic Algorithms, Morgan Kaufmann, 1991.

Grefenstette J.J., Deception considered harmful, in L.D., Whitley ed. Foundations of Genetic Algorithms 2, Morgan Kaufmann, 1993.

Gruau F., Neural Network Synthesis using Cellular Encoding and the Genetic Algorithm, Ph.D. thesis, Ecole Normale Superieure de Lyon, 1994.

Harrison G.C., Chess D.M. and Kershenbaum A., Mobile Agents: Are they a good idea?, IBM Internal Research Report, T.J. Watson Research Center, 1995.

Harvey N.R., and Marshall S., The use of genetic algorithms in morphological filter design, J. Signal Processsing: Image Communication, Vol. 8: 55 - 71, 1996.

Hillis W.D., Co-evolving parasites improve simulated evolution as an optimisation procedure, Physica D 42: 228 - 234, 1990.

Hinterding R., Gielewski H. and Peachey T.C., The Nature of Mutation in Genetic Algorithms, Proc. Sixth Int. Conf. Genetic Algorithms, San Francisco, CA, Morgan Kaufmann, L., Eshelman ed., pp. 65 - 72, 1995.

Hoffmann F., Evolving Fuzzy Rules by Genetic Algorithms, Berlin, Germany, 1997.

Holland J., Adaptation In Natural and Artificial Systems, Ann Arbour, University of Michigan Press, 1975.

Holland J., Hidden Order: How Adaptation Builds Complexity, Perseus Books, Reading, 1995.

Hopfield, J.J., Neural Networks and Physical Systems with Emergent Collective Computational Abilities, Proc. National Academy of Sciences, Vol.79: 2554 - 2558, 1982.

Horowitz E., Sartaj S. and Dinesh M., Fundamentals of Data Structures in C++, Computer Science Press, 1995.

Hostetter C., Survey of Object Oriented Programming Languages, www.rescomp.berkeley.edu / ~hossman/cs263/paper.html #tth_sEc4, 2002.

Huberman B., (ed.), The ecology of computation, Amsterdam, North-Holland, 1988.

Juille H. and Pollack J.B., Parallel Genetic Programming and Fine-Grained {SIMD} Architecture, Working Notes for the AAAI Symposium on Genetic Programming, eds. E. V. Siegel and J. R. Koza, pp. 31 - 37, Cambridge, MIT Press, 1995.

Kauffman S.A., The Origins of Order, New York, Oxford University Press, 1993.

Kelly A.M., Manuale di Economia Politica, New York, 1971.

Kinnear Jr. K.E., Generality and difficulty in genetic programming: Evolving a sort, in Forrest S., ed. Proc. 5th Int. Conf. Genetic Algorithms, ICGA093, pp.287 - 294, University of Illinois, San Francisco, CA, Morgan Kaufmann, 1993.

Koch C., Computation and the Single Neurone, Nature, Vol. 385, 16th Jan., 1997.

Kohonen, T., Self-Organization and Associative Memory, Berlin, Springer-Verlag, 1984.

Kosko B., Neural Networks and Fuzzy Systems, A Dynamical Systems Approach to Machine Intelligence, Prentice Hall, 1992.

Kotz D. et al., Agent TCL: Targeting the needs of Mobile Computers, IEEE Computing Online, http://computer.org/internet/icl1997/w4toc.htm, 1997.

Koutsofios E.E, North S.C, Truscott R., and Keim D.A., Visualizing large-scale telecommunication networks and services, Inf. Visualization Res., AT&T Labs., Florham Park, NJ, Proc. Visualization '99, IEEE, pp.457 - 61, 1999.

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

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

Koza J.R., 1,000-Pentium Beowulf-Style Cluster Computer for Genetic Programming, online report at: http://www.genetic-programming.com /machine1000.html, 2001.

King R. C., A Dictionary of Genetics, 4th ed. Oxford University Press, 1990.

Krasner G.E. and Pope S.T., A cookbook for using the model-view controller user interface paradigm in Smalltalk-80, J. Object-Oriented Programming, Vol.1:(3), pp.26 - 49, Aug. 1988.

Lamarck J.B., Histoire Naturelle des Animaux Sans Vertbres, (in French); translated and published as, Zoological Philosophy: An Exposition with Regard to the Natural History

of Animals, (1815), University of Chicago Press, Chicago, IL, USA, 1984.

Lange D., Mobile Agents: Environments, Technologies, and Applications, Proc. PAAM '98, Mar., pp.11 - 14, 1998.

Langton C.G., (ed.), Artificial Life, Adisson Wesley, Redwood City, 1987.

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

Lawler E., Lenstra J., Rinnooy Kan A. and Shmoys D., The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization, New York, Wiley, 1985.

Lea D., Concurrent Programming in Java: Design Principles and Patterns, ISBN 0-201-31009-0, second edition, Addison-Wesley, 1999.

Lewis J., Hart E. and Ritchie G., A Comparison of Dominance Mechanisms and Simple Mutation on Non-stationary Problems, J. Lecture Notes in Computer Science, Vol. 1498: pp.139 - 148, 1998.

Maes P., Modelling Adaptive Autonomous Agents, J. Artificial Life, MIT Press, Vol.1: (2), 135 - 162, 1994.

Maes P., Guttman R., and Moukas A., Agents that Buy and Sell: Transforming Commerce as we Know It, Communications of the ACM, Mar. Issue, online http://ecommerce.media.mit.edu /Kasbah/, 1999.

Man K.F., Tang K.S. and Kwong S., Genetic Algorithms, London, Springer-Verlag, 1999.

Mataric M. and Cliff D., Challenges in evolving controllers for physical robots, Robotics and Autonomous Systems, Vol.19: pp.67 - 83, 1996.

Mathias K.E. and Whitley L.D., Changing Representations during search: A comparative study of delta coding, Evolutionary Computation, Vol.2, 1994.

Mattfeld D., Kopfer H. and Bierwirth C., Control of Parallel Population Dynamics by Social-Like Behaviour of {GA}-Individuals, in Parallel Problem Solving from Nature , PPSN III, Davidor Y. Schwefel H. and Manner R., eds., Berlin, Springer, pp.16 - 25, 1994.

McCulloch W.S. and Pitts W., A Logical Calculus of the Ideas Immanent in Nervous Activity, Bulletin of Mathematical Biophysics, Vol. 5: pp.115 - 133, 1943.

Michalewicz Z., Genetic Algorithms + Data Structures = Evolution Programs, 3rd Ed., New York, Springer, 1999.

Minsky M. and Papert, S., Perceptrons, An Introduction to Computational Geometry, Cambridge, MA, MIT Press, 1969.

Mitchell M. and Forrest S., Relative building-block fitness and the building-block hypothesis, in L. D., Whitley, ed., Foundations of Genetic Algorithms 2, pp.109 – 126, San Mateo, Morgan Kauffman, CA, 1993.

Mitchell M., An Introduction to Genetic Algorithms, Complex Adaptive Systems Series, Cambridge, MA, MIT Press; ISBN: 0262631857, reprint, 1998.

Mondada F. and Floreano D., Evolution and Mobile Autonomous Robotics, in Towards Evolvable Hardware, pp.221 - 249, 1995.

Nehmzow U., Mobile Robotics: A Practical Introduction, London, Springer-Verlag, 1999.

Nordin P., A compiling genetic programming system that directly manipulates the machine code, in K.E., Kinnear, ed., Advances in Genetic Programming, Chap.14, pp.311 - 331, Cambridge, MA, MIT Press, 1994.

Nordin P., Francone F. and Banzhaf W., Explicitly defined introns and destructive crossover in genetic programming, in J.P., Rosca, ed., Proc.Workshop on Genetic Programming: From Theory to Real-World Applications, pp.6 - 22, Tahoe City, CA, 1995.

Nguyen Van and Hailpern, Brent A Generalized Object Model, ACM SIGPLAN NOTICES , Vol.21: (10), ed.: G.R. Wexelblat, Oct. 1986.

ObjectSpace Voyager and Agent Platforms Comparison, www.objectspace.com, 1997.

Paredis J., Coevolutionary algorithms, in: T,Bäck, D,Fogel, Z,Michalewicz, eds., Evolutionary Computation 2: Advanced Algorithms and Operators, pp. 224 - 238. Bristol, Institute of Physics, 2000.

Poli R., Genetic Programming for Image Analysis, in Genetic Programming, Proc. First Annual Conference, Cambridge, MA, MIT Press, ed. J.R. Koza, D.E. Goldberg, David B. Fogel and R.L. Riolo, pp.363 - 368, 1996.

Porto V.W. and Fogel L.J., Evolution of Intelligently Interactive Behaviors for Simulated Forces, Evolutionary Programming VI, P.J. Angeline, R.G. Reynolds, J.R. McDonnell, and R. Eberhart, eds., Berlin, Springer, pp.419 - 429, 1997.

Ray T.S., Netlife - Creating a jungle on the Internet, Online: Digital Territories, Incorporations and the Matrix, Knowbotic Research, ed., Medien Kunst Passagen 3/94, Passagen Verlag, Koeln-Wien 95, ISSN 1019-419-4193, 1994.

Rentsch, Nguyen V. and Hailpern B., A Generalized Object Model ACM SIGPLAN NOTICES, Vol.17: (9), ed. G. R. Wexelblat, Sept. 1982.

Rogers A. and Prügel-Bennett A., Modelling the dynamics of steady-state genetic algorithms, in W. Banzhaf and C. Reeves, eds., Foundations of Genetic Algorithms 5, pp.57 - 68, Morgan Kaufmann, San Francisco, 1999.

Rush J.R., Evolving Cellular Neural Networks for Autonomous Robot Control, Ph.D. thesis, University of Salford, July 1996.

Saffioti A. and Wesley L.P., Hierarchical Fuzzy-Based Localisation in Autonomous Mobile Robots, Proc. IFSA, 1995.

Saffioti A., The Uses of Fuzzy Logic in Autonomous Robot Navigation: A catalogue raisonne, Technical report IRIDIA, University of Brussels, TR/IRIDIA/97-6, June 1997.

Sasaki T. and Tokoro M., Adaptation under Changing Environments with Various Rates of Inheritance of Acquired Characters: Comparison between Darwinian and Lamarckian Evolution, SEAL, pp.34 - 41, 1998.

Schoppers M.J., Universal Plans for Reactive Robots in Unpredictable Environments, in Proc. 10th Int. Joint Conf. Artificial Intelligence, IJCAI 87, Milan, Italy, pp.1039 - 1046, 1987.

Smith R.E., An investigation of diploid genetic algorithms for adaptive search of nonstationary functions, TCGA Report No. 88001, Tuscaloosa, University of Alabama, The Clearinghouse for Genetic Algorithms, 1988.

Spears W. M. and DeJong K. A., An analysis of multi-point crossover, in G. J. Rawlins, ed., Foundations of Genetic Algorithms-1, pp.301 - 315, Morgan Kauffman, 1991.

Stroustrup B., What is "Object-Oriented Programming"? (1991 revised version), http://www.research.att.com/~bs/whatis.ps, 1991.

Steels L., The Artificial Life Roots of Artificial Intelligence, J. Artificial Life, Vol. 1: (2), Cambridge, MA, 1994.

Surmann H., Peters L. and Huser J., A Fuzzy System for Real-time Navigation of Mobile Robots, 19th Annual German Conf. on AI, KI-95, pp.170-172, Bielefeld, 1995.

Tackett W.A., Recombination, Selection, and the Genetic Construction of Computer Programs, Ph.D. thesis, University of Southern California, Dept. of Electrical Engineering Systems, 1994.

Tateson R., Self-organising pattern formation: fruit flies and cell phones, in: Parallel Problem Solving from Nature - PPSN V, pp.732 - 741, Berlin, Springer, 1998.

Tateson R., The role of development in computational systems, BT Technol. J., Vol.18: (4), 85 - 94, 2000.

Teller A. and Veloso M., PADO: Learning tree structured algorithms for orchestration into an object recognition system, Technical report CMU-CS-95-101, Pittsburgh Dept. of Computer Science, Carnegie-Mellon University, 1995.

Thompson J.N., The Coevolutionary Process, Chicago, University of Chicago Press, 1994.

Thompson A. and Layzell P., Evolution of Robustness in an Electronics Design, Proc. 3rd Int. Conf. on Evolvable Systems (ICES2000): From biology to hardware, Miller J., Thompson A., Thomson P. and Fogarty T., eds., pp.218 - 228, Springer-Verlag, Series LNCS, Vol. 1801;, 2000.

Tunstel E. and Lippincott T., Genetic Programming of Fuzzy Coordination Behaviors for Mobile Robots, 1st Int. Symp., Soft Computing for Industry, WAC '96, Montpellier, France, pp.647 - 652, May 1996.

Wagner G.P. and Altenberg L., Complex adaptations and the evolution of evolvability, Evolution Vol.50: 967-976, 1996.

Watanabe Y. and Pin. F.G., Sensor based navigation of a mobile robot using automatically constructed fuzzy rules, Proc. ICAR '93 the International Conference on Advanced Robotics, Tokyo, pp. 81-87, 1993.

Watson R.A. and Pollack J.B., Incremental Commitment in Genetic Algorithms, Proc. Genetic and Evolutionary Computation Conference, Vol.1:, Morgan Kaufmann, Banzhaf W., Daida J., Eiben M.H.V., Honavar G., Jakiela M. and Smith R.E., pp.710-717, 1999.

White J.E., Helgeson C.S. and Steedman D.A., System and method for distributed computation based upon the movement, execution, and interaction of processes in a network, General Magic, U.S. Patent 5,603,031, filed 8 July 1993, issued 11 Feb. 1997.

Whitley D. and Starkweather T., GENITOR II: A Distributed Genetic Algorithm, J. Expt. Theoretical, Artificial Intelligence Vol.2: pp.189 - 214, 1990.

Whitley D., Foundations of Genetic Algorithms 2, Morgan Kaufmann, 1993.

Whitley D., A Genetic Algorithm Tutorial, J. of Statistics and Computing, Vol. 4: pp.65 - 85, 1994.

Wilkins D., Practical Planning - Extending the Classical AI Planning Paradigm, San Mateo, California, Morgan Kaufmann, 1988.

Wolpert D.H. and Macready W.G., No free lunch theorems for search, SFI-TR-95-02-010", citeseer.nj.nec.com/wolpert95no.html, 1995.

Wright, S., Evolution and the Genetics of Populations, Vol. 2, Chicago, University of Chicago Press, 1969.

Wu A., De Jong K.A., Burke D.S., Grefenstette J. and Loggia Ramsey C., Visual analysis of evolutionary algorithms, Congress on Evolutionary Computation, 1999.

Appendix A

A.1 Java-based EA Software

A number of Java-based software packages are available which provide libraries of code for genetic algorithms, genetic programming, and a range of other EA techniques. The following links are a few of those currently available, which may be useful to the student interested in further work.

ECJ – ECJ is a research EC system written in Java. It was designed to be highly flexible, with nearly all classes (and all of their settings) dynamically determined at run-time by a user-provided parameter file. All structures in the system are arranged to be easily modifiable. Even so, the system was designed with an eye toward efficiency; ECJ may make you reconsider biases about Java and slowness. (Luke S. 2001) http://www.cs.umd.edu/projects/plus/ec/ecj/.

EOS – Eos is a software platform for evolutionary algorithms (EA) and ecosystem simulations. EAs are algorithms that are inspired by evolution in nature. They can be applied to hard optimisation problems. Ecosystem simulations mirror natural ecosystems. They can be used to create complex adaptive systems. Eos provides an extensive library of algorithms and structures related to both fields. A licence can be obtained by contacting the author Erwin Bonsma (erwin.bonsma@bt.com) or via the BTexact web site at http://www.labs.bt.com/projects/eos.htm.

GA Playground – The GA Playground is a general-purpose genetic algorithm toolkit where the user can define and run his own optimization problems. http://www.aridolan.com/ga/gaa/gaa.html.

JGProg – Java Genetic Programming is an open-source pure Java implementation of a strongly typed Genetic Programming experimentation platform. http://jgprog.sourceforge.net/.

A.2 C/C++ based EA Software

Evolve – is a freely available, stack-based genetic programming environment. It is an MFC application written in C++. http://www.digitalbiology.com/.

GAlib – contains a set of C++ genetic algorithm objects. The library includes tools for using genetic algorithms to do optimization in any C++ program using any representation and genetic operators. The documentation includes an extensive overview of how to implement a genetic algorithm as well as examples illustrating customizations to the GAlib classes. //lancet.mit.edu/ga/.

Genetic Server and Genetic Library – provides general-purpose APIs for genetic algorithm design. Genetic Server is an ActiveX component that can be used to easily build custom genetic applications in Visual Basic. Genetic Library is a C++ library that can be used for building custom genetic applications in Visual C++. http://www.nd.com/products/genetic.htm.

Genetic Adaptive Systems LAB (GASLAB) – http://gaslab.cs.unr.edu/.

A.3 General Evolution and Robotics References

The following is an eclectic list of web references that may be of interest to anyone who has managed to reach this point in the text and has an interest in evolution or robotics (as most sane people do!).

Agentlink – An extensive reference source for software agent research, http://www.agentlink.org/.

BTexact Technologies – Future Technologies Group. This is the research group of the author which undertakes a wide range of nature-inspired computer research projects. http://www.labs.bt.com/projects/ftg.htm. Also a very cool place to work and the coffee is excellent!

Alife Web Site – Very useful page dedicated to Alife, maintained by Ariel Dolan. A web-oriented artificial life site: alife, genetic algorithms, and cellular automata experiments written in cross-platform web languages (java, tcl/tk), with free source code, http://www.aridolan.com/index.html.

Complex Systems Web site – http://www.brint.com/Systems.htm.

Steve Grand – Read about the Lucy robot project, http://www.cyberlife-research.com/.

Evalife – A European Alife and complex systems page with a Java GA package, http://www.evalife.dk/.

Evo – An evolutionary toolkit built on the Swarm system, http://omicrongroup.org/evo/.

COGS – Major UK academic centre for the study of Complex Systems and EA subjects, within Sussex University, http://www.cogs.susx.ac.uk/.

Evonet – The European Network of Excellence in Evolutionary Computing. A useful reference for EA related research http://evonet.dcs.napier.ac.uk/

UK Alife – General Alife link page, http://www.alife.co.uk/.

Khepera Robots – The home page of the Khepera robot, a popular desktop mobile robot used by many universities, www.k-team.com/robots/khepera/.

In addition, an excellent full simulator of the Khepera written in Java is freely available from Wright State University, at http://gozer.cs.wright.edu/classes /ceg499/sim/sim.html

Out of Control – Online copy of Kevin Kelly's text on complex stuff, http://www.well.com/user/kk/OutOfControl/ , but buy the book – it is very good.

Repast – The best Java-based multiagent simulator, in my opinion. Ideal for studying complex multiagent behaviour, e.g., swarming or group formation. http://repast.sourceforge.net/.

Madkit – Another Java multiagent simulator currently available (really depends on what kind of agent you want to model), http://www.madkit.org/.

MultiAgent.com – General resource for agents, http://www.multiagent.com/.

Santa Fe Institute – A major center for all aspects of research in Complex Adaptive Systems. They also have a very useful collection of online publications and working papers, http://www.santafe.edu. A very cool place to think and meet smart people.

Zurich Alife Group – A very useful site for Artificial Life, including a free online textbook for Alife, http://www.ifi.unizh.ch/ailab/teaching/AL00.html.

A.4 Java Reference Guides

Thinking in Java – 2nd ed., by Bruce Eckel – Complete online free guide to Java, available from http://www.creatlon.net/Eckel/.

A.5 Useful References

Harnessing Complexity – Organizational Implications of a Scientific Frontier, Robert Axelrod and Michael Cohen, The free press, New York, 1999. An excellent guide to the application of Complex Systems.

Understanding Nonlinear Dynamics – Daniel Kaplan and Leon Glass, Springer, New York, 1995. - The best introduction to Chaos and nonlinear systems I have yet found, after a decade of browsing libraries and bookshops.

Appendix B

A Genetic Algorithm Example, and the GPSYS GP Library

B.1 Basic Genetic Algorithm

This appendix describes the basic genetic algorithm code used to develop the applications in Chapter 5 and the design methodology used in the example applications. A brief introduction to object-oriented design in Java is also included with an example application to illustrate the use of OO design.

For comparison an overview of the GPSYS Java GP package is provided, as this provides a very clearly constructed and OO package for GP development (written by Adil Qureshi at UCL). It also includes an example application for evolved mobile robot control, which can be compared with the application in Chapter 5.

B.2 Simple Java Genetic Algorithm

The following Java classes provide the basis for a very simple genetic algorithm. The aim is to make the steps a standard GA requires as transparent as possible, and not to provide a research grade piece of software. I would strongly recommend the Eos platform from BT Future Technologies group if a serious evolutionary library is required (available by contacting the research group via the author or Erwin Bonsma at erwin.bonsma@bt.com). Section 1.4 covers an Eos application based on the traveling salesman problem, which is designed to illustrate a more complex EA development process.

In this simple GA the first class individual acts as a generic object representation of each chromosome in a GA population. An individual contains all the data required to define a specific chromosome and stores the current fitness values associated with the individual.

The actual chromosome is stored within a Java vector object. By selecting this design we have the choice of using any basic Java object as a gene

representation. For example, the vector could hold a sequence of floats, integers, doubles or any other objects with which we want to compose the chromosome. In this case we use doubles as an example, which is the selected representation scheme in the example applications.

The following GA example code tries to match a one-dimensional set of double values generated from a simple mathematical function (sin x) and prints the output to screen.

individual.java

```
/**Individual class, defines a single individual chromosome
**/

import java.util.Observable;
import java.util.Hashtable;
import java.util.Vector;
public class individual implements Cloneable{

/**constructor*/
public individual() {
chromosome = new Vector();
//dynamically resizable array of genes

fitness = 0.0; //raw fitness value
scaledFitness = 0.0; //relative fitness value
}
public double fitness;
public double scaledFitness;
public Vector chromosome;

/**method to return full copy of this individual */
public individual copy()

{
individual newInd = new individual();
newInd.chromosome = (Vector)this.chromosome.clone();
newInd.fitness = this.fitness;
newInd.scaledFitness = this.scaledFitness;
return newInd; }

} /**end class**/
```

genetic.java

```
/**Main genetic algorithm class. This version tries to match
the contents of a 1-D array of double numeric values. **/

import java.util.*;
public class genetic
{
double sumFitness;
double bestFitness;
double averageFitness;
individual population[]; //array of current individuals
individual newpopulation[]; //next generation of individuals
individual bestIndividual;
int popSize; //number of individuals
double alleleSize; //range of the target numeric values
int maxLength; //length of chromsome
int minLength; //same as above for fixed length chromsomes
int generation=0;
Random random;
static final int seed = 93404;
public double testarray[];
public genetic()
{
random = new Random(seed);
popSize = 120;
maxLength = 16;
minLength = 16;
testarray = new double[maxLength];
sumFitness = 0.0;
bestIndividual = new individual();
alleleSize = 99.0;
population = new individual[popSize];
for (int i=0;i<popSize;i++)
{
population[i] = new individual()
}

population = initialisePopulation(population);
}

public individual[] initialisePopulation(individual pop[])
{
```

```
int g=0;
double gene = 0.0;
for (int i=0;i<pop.length;i++)
{
g = (int)(random.nextFloat() * maxLength);

if (g <= minLength) g = minLength;
for (int j=0;j<g;j++)//fill chromosome for this ind
{
gene = Math.abs((random.nextFloat() *   alleleSize));
pop[i].chromosome.addElement(new Double(gene));
}
}
return pop;
}

/**Main method will loop until program is halted,
alternatively could set to end after a fixed number of runs
or a minimum fitness value is reached. */

public void evolve()
{
while(true)
{
for(int i=0;i<population.length;i++)
{
double testcase [] =
convertChromosome(population[i].chromosome);
population[i].fitness = evaluate(testcase);
}

sort(0, population.length - 1);
normalizeFitnessOfPopulation();
select();
generation++;
}
}

/** Re-scale fitness of all individuals and print results of
current best individual.  */

public void normalizeFitnessOfPopulation()
{
double minFitness = 10000.0;
```

```
sumFitness = 0.0;
for (int i = 0; i < population.length; i++)
{
if( population[i].fitness < minFitness)
minFitness = population[i].fitness;

if (population[i].fitness > bestFitness)
{
bestFitness =  population[i].fitness;
bestIndividual = population[i];
printStats(bestIndividual);
}
}

for (int i = 0; i < population.length; i++)
{
population[i].scaledFitness = population[i].fitness -
minFitness;
}

for (int i = 0; i < population.length; i++)
{
sumFitness += population[i].scaledFitness;
}
}

void printStats(individual ind)
{
double test[];
test = convertChromosome(ind.chromosome);
for(int i=0;i<test.length;i++)
{
double a = test[i];
System.out.print(a + " ");
}

System.out.println("Current best genome");

for(int j=0;j<test.length;j++)
{
double t = testarray[j];
System.out.print(" ");
}
```

```
System.out.println("Target genome");
System.out.println(ind.fitness);
}

/** Applies the GA processes of fitness proportionate
selection, crossover and mutation to the population. */

void select()
{
int index=0;
int step=0;
int mutate=0;
newpopulation = new individual[popSize];
individual ind1;
individual ind2;
individual children[];
for(int i =0;i < population.length; i++)
newpopulation[i] = population[i].copy();
index = 2;
while(index < ((popSize/2)-1))
{
step = index + (popSize/2); //replace bottom half
ind1 = selectIndividual();
ind2 = selectIndividual();
children = crossOver(ind1,ind2);
newpopulation[step] = children[0].copy();
index++;
newpopulation[step] = children[1].copy();
index++;
mutate++;
if (mutate% 3 == 0)
{
int k = (int)(random.nextFloat()*popSize-1);
newpopulation[k] = mutate(population[k].copy());
}
}

//copy newpopulation into the current population

for(int i =0;i < population.length; i++)
{
population[i] = newpopulation[i].copy();
}
}
```

```
/** Returns an individual based on a Roulette Wheel
selection method, see chapter 3. */

individual selectIndividual(){

double sumOfFitness = 0.0;
int index = 0;
individual temp = new individual();
float ran = random.nextFloat();
double limit = ran * sumFitness;

while ((index < population.length) && (sumOfFitness <limit)
{
sumOfFitness += population[index].scaledFitness;
index++;
}
temp = population[index -1].copy();
return temp;
}

/** Swap segments of two selected parent individual
chromosomes. */

individual[] crossOver(individual parent0, individual
parent1)
{
individual children[] = new individual[2];
individual longest;
individual shortest;
int min = 0;
int max = 0;
min = (int)Math.min(parent0.chromosome.size(),
parent1.chromosome.size());
max = (int)Math.max(parent0.chromosome.size(),
parent1.chromosome.size());

int cut = (int)(random.nextFloat() * min);
//assumes chromsome may be variable length
if (parent0.chromosome.size() >= parent1.chromosome.size() )
{
longest = parent0.copy();
shortest = parent1.copy();
}
else
```

```
{
longest = parent1.copy();
shortest = parent0.copy();
}

for(int i=cut;i<max;i++)
{
if (i <= (min-1))
{
Double a = (Double)parent0.chromosome.elementAt(i);
Double b = (Double)parent1.chromosome.elementAt(i);
longest.chromosome.setElementAt(b,i);
shortest.chromosome.setElementAt(a,i);
}

else
{
Double a = (Double)longest.chromosome.elementAt(i);
shortest.chromosome.addElement(a);
}
}

for(int i=min;i<max;i++)
{
int len = longest.chromosome.size();
longest.chromosome.removeElementAt(len-1);
}

children[0] = shortest.copy();
children[1] = longest.copy();
return children;
}

/** Returns a new individual from a mutation applied to an
existing individual. */

individual mutate(individual mutant)
{
int m,r=0;
double g = 0.0;
Double gene;
float f = random.nextFloat();
r = (int)( f * mutant.chromosome.size() );
g = ((random.nextFloat())) * alleleSize;
```

```
gene = new Double(g);
mutant.chromosome.setElementAt(gene,r);
return mutant;

}

/** An optional method, to allow alternative primitive
representation within the chromosome. */

public double[] convertChromosome(Vector chrom)
{
Double val;
double out[]= new double[chrom.size()];
for(int i =0;i<chrom.size();i++)
{
val = (Double)chrom.elementAt(i);
out[i] = (double)val.doubleValue();//select ints or double
}
return out;
}

/** Do the comparison between the target array of doubles
and the current individual being tested for fitness. */

double evaluate(double[] genome)
{
double answer=0.0;
boolean flag=false;
short[] shortGenome = new short[maxLength];

// fill the test array, see sinc function in chapter 3
for(int i=0;i<genome.length;i++)
{
testarray[i] = Math.abs(10 * Math.sin(i));
}
for(int i=0;i<genome.length;i++)
{
answer += Math.abs(testarray[i] - genome[i]);
//measure the difference between each target element and the
individual array
}
```

```
return 1.0 - (answer/genome.length);
//scale as 1.0 = fittest individual
}

/** Simple quick sort algorithm, to arrange the population
by fitness. */

void sort(int low, int high) {
int index1, index2;
double pivot;
individual temp;
index1 = low;
index2 = high;
pivot = population[(low + high) / 2].fitness;
do {
while (population[index1].fitness > pivot) {
index1++;}
while (population[index2].fitness < pivot) {
index2--;
}

if (index1 <= index2) {
temp = population[index2];
population[index2] = population[index1];
population[index1] = temp;
index1++;
index2--;
}

} while (index1 <= index2);
if (low < index2) {
sort(low, index2);
}

if (index1 < high)
{
sort(index1, high);
}}

/** Main program method, creates an instance of the class
genetic. */
public static void main(String args[])
{
genetic test = new genetic();
```

```
test.evolve();
}

}//end class  genetic
```

If you have any Java JDK above 1.1 installed then you should be able to just copy this text into a file, compile it using the javac command line compiler, and run in a command window. (Also available at the book web site, http://www.cybernetics.org.uk.) By modifying the GA parameters in this simple example, such as mutation rate, population size, and representation scheme, the student can gain useful insight into the relative importance each plays in the performance of a GA.

This basic GA class is clearly not the most realistic or sophisticated GA available. However, the intention was to present a class that is very intuitive and easy to understand. A number of useful additions are suggested below.

Exercises

- Modify the genetic class to match a two-dimensional array of integer or double values.
- Enable a binary genome representation.
- Provide a choice of selection operators, e.g., tournament selection.
- Enable variable-length genomes. The use of a Java vector to store the chromosome should make this a simple exercise. Also try using the ArrayList object as a storage mechanism and compare the performance with the Vector object.
- Provide a choice of multipoint crossover schemes.

B.2.1 Vectors and Arraylists

In the preceding GA code we make use of the Java vector class to hold sets of primitive objects. Vectors are resizable and easy to use; however, there is an alternative class called the ArrayList. There are several points to consider in choosing which Java array class to use in an evolutionary algorithm.

Synchronisation
`Vectors` are synchronised. Any method that touches the `Vector`'s contents is thread-safe. `ArrayList`, on the other hand, is unsynchronised, i.e., not thread-

safe. Hence synchronisation will incur a performance hit. If you don't need a thread-safe collection, use the `ArrayList`.

Data Growth

Internally, `ArrayList` and `Vector` assign their contents using an `Array`. Hence as you insert an element into an `ArrayList` or a `Vector`, the object will expand its internal array if it runs out of room. A `Vector` defaults to doubling the size of its array, while the `ArrayList` increases its array size by 50%. Either can therefore incur a large performance hit while adding new elements.

Alternatively, you could use a simple Java array in place of either `Vector` or `ArrayList`, especially for any performance-critical code. By using an array you can avoid synchronisation, extra method calls, and suboptimal resizing. Just be careful with indexing of the array elements.

B.3 Application Design

For both of the applications in Chapter 5 a graphical user interface was required. A common approach to designing such an application, however, is to just lump together the functional and interface code. However, even for relatively small projects this is to be strongly discouraged as it makes debugging and future development vastly more difficult.

In contrast a useful methodology, particularly for designing medium- and large-scale software projects, is to use design patterns. The basic idea for a design pattern is to create a set of generic templates, which represent common collections of interacting objects (Grasner, 1988). This is an intuitive process and simply requires a designer to consider how to group objects according to functionality. The most common example of such a pattern is the model-view-controller design pattern (MVC), illustrated in Figure B.1. Using design patterns is the best way to take advantage of the features of an object-oriented language like Java or C++, although it was originally developed for the Smalltalk language.

This pattern is commonly used as it offers a neat way to integrate an application with a user interface. The parts comprise a data model that contains the computational elements of the application, a controller that regulates interactions between a user and the data model, and the viewer itself.

Each component in the pattern may contain a large number of subpackages and specific objects. The key benefit is that changes can be made to the code in say the data model without having a direct impact on the code of the view or controller. The interaction between each component is normally mediated by event processes, i.e., actions occurring in the data model generate events that

notify the controller and/or viewer, which then updates its view of the corresponding data.

The details of selecting and creating design patterns are beyond the scope of this text and the reader is referred to the following reference (Lea, 1999). In particular, a free online guide to using Java design patterns is available from http://www.patterndepot.com/put/8/JavaPatterns.htm.

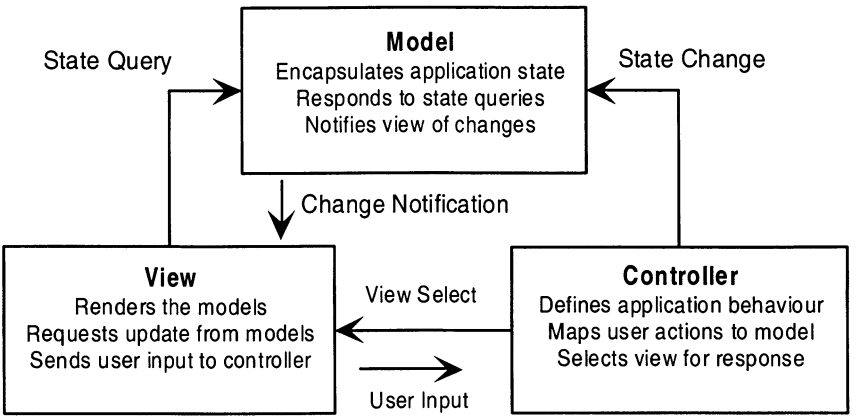

FIGURE B.1 Example Model View Controller design pattern. See for reference: http://java.sun.com/blueprints/patterns/j2ee_patterns/model_view_controller/.

B.4 Eos: An Evolutionary and Ecosystem Research Platform

Authors: Erwin Bonsma, Mark Shackleton & Rob Shipman

The Eos platform supports research and rapid implementation of evolutionary algorithms, ecosystem simulations and hybrid models. It also supports fast prototyping of industrial applications using these technologies. A large and rapidly growing library of evolutionary algorithm types and options is provided that together with a flexible configuration system allows a "plug-and-play" construction of novel algorithms. Support for ecosystem models includes classes for multiple types of physical space (n-dimensional discrete or continuous Cartesian space, graph space), complex interactions between entities, and movement of individuals between populations.

The flexibility of the Eos platform is designed to provide a powerful environment for developing new algorithms and architectures. Eos is

implemented in Java for portability and to allow easy extension of the core functionality. It supports transparent distribution of evolutionary and ecosystem implementations across multiprocessor computer clusters. In this overview we describe the architecture and functionality of the Eos platform and illustrate its use by way of a number of example applications.

B.4.1 Introduction

For a truly flexible R&D software platform it is desirable to provide a framework that in an integrated manner supports both the "traditional" types of evolutionary algorithm *and* aspects of ecosystems. The Eos platform that we describe here has been designed to support precisely such systems. It has also been developed so as to permit rapid extensions of the core components and prototyping of new algorithms and applications.

It is important to note that there are a growing number of evolutionary algorithm toolkits available. Some toolkits such as SWARM do place an emphasis on agent-based systems more reminiscent of ecosystems, but these have less intrinsic support for evolutionary computation. Eos seeks to address both evolutionary algorithms and ecosystem simulations, together with hybrid systems encompassing both aspects, in a unified framework.

In the following sections the design of the Eos software platform, its key classes, and its flexible configuration system are discussed in some detail. Some relatively simple examples are then given to help clarify the sort of algorithms and systems that are supported by Eos, and the typical ways in which these can be implemented.

B.4.2 Design overview

The design goals of the Eos platform demanded the following functionality:

1. Support for the "standard" evolutionary algorithm types
2. Support for ecosystem models
3. Support for the development of hybrid and experimental algorithms
4. Rapid prototyping of applications
5. Easy extension of the core functionality

The latter three aims immediately suggested that an object-oriented design was appropriate. Ease of programming and debugging were considered of paramount importance too, and as a result Java was chosen as the language used to implement the platform. Whilst Java implementations typically do not execute as fast as some other languages such as C++, this is becoming less of an issue as Java virtual machine implementations continue to improve. The superiority in terms of faster prototyping and debugging was therefore considered more important than raw efficiency of execution.

To satisfy the above design goals, Eos fully exploits the object-oriented capabilities of the Java language. Next to basic mechanisms such as inheritance

and encapsulation, a wide range of design patterns [12] is used to ensure that the Eos system is flexible and powerful. For example, the Prototype design pattern together with a polymorphic software design form the base of a "plug-and-play" architecture that facilitates the incorporation of new functionality and speeds up the development of applications.

The requirement to support both evolutionary algorithms *and* ecosystems in a single framework presented certain challenges in the design of Eos. Traditional evolutionary algorithms, such as genetic algorithms, are in essence centralised in nature. Fitness is usually normalised relative to all other individuals in a population, requiring a centralised (and synchronised) calculation. In contrast, ecosystems are by definition distributed, with the dynamics depending on the interactions occurring between individuals within populations. By supporting such distributed operations it is also possible to support certain evolutionary algorithms within which the fitness is implicitly encoded in local interactions.

The heart of each simulation is a flexible "update" mechanism that is suitable for both types of system. It exploits the fact that the elements in the simulation (*entities*) are structured hierarchically. An update "heartbeat" is initiated at the highest level, the environment. By default it propagates down through the hierarchy with the environment update invoking population updates, which in turn invoke subpopulation and individual updates. Every such entity in the system has an associated update method, which by default updates all the entities it contains. Most importantly however, the update method can be overridden to change or extend its functionality. Evolutionary algorithms are implemented by extending the default update behaviour at the population level. All individuals are still updated individually (each evaluating its genome), but additional code performs the centralised fitness calculation and the resulting selection and replacement.

Another key aspect of Eos is its support for *interactions*. These can be registered on any entity in the system as occurring between the entities that it contains. This provides a very flexible mechanism, which is especially relevant to ecosystem simulations. For example, the Echo [13] implementation described later employs interactions to perform local activities such as fight, trade, mate, etc. In an "information ecosystem" [14] an interaction may involve a transfer of information. Alternatively, in an island-based evolutionary algorithm an interaction between populations may involve individuals transferring (moving) between subpopulations.

In addition to the highly object-oriented design of Eos, there are important implementation aspects that make Eos particularly useful for simulation and experimentation. Given that both evolutionary algorithms and ecosystem simulations can be computationally intensive it is very desirable that, whenever possible, multiple computers can be used to distribute the workload. For this reason there is explicit support built into Eos to allow simulations to be distributed over multiple processors. This is entirely transparent to the user, thus

obviating the need for a user to worry about implementation details. Two mechanisms are used internally within Eos to support this parallelism: Java RMI (remote method invocation) is the most efficient, while optional use of the Voyager distributed agent toolkit [15] is the most general. We have used these mechanisms to transparently distribute Eos applications over a 36-processor Beowulf [16] cluster of PCs running Linux.

B.4.3 Key Classes

The previous section described some of the important components and features of the Eos platform. In this section we go into more detail and examine some of the implementation details by looking at a few key classes.

One of the important classes in Eos is `Entity`. Environments, populations, and individuals are all subclassed from `Entity` as they have several features in common. All entities can be updated, can implement the Observer and Visitor design patterns [12], can calculate statistics, and provide ways of arranging entities hierarchically. One advantage of having a shared base class is that it avoids unnecessary code duplication. It also enhances polymorphism by making it possible to operate on entities in general, without distinguishing between the various types of entity. This is, for example, useful if you want to output the state of the simulation by recursively visiting all entities.

The `Prototype` interface and `Params` class are two other essential Eos components. The `Prototype` interface is used by the Eos core to create new objects, possibly of a class that is unknown within the core. The `Params` class is the base class for all parameter classes and is used to configure and initialise objects created from prototypes. Together they provide the plug-and-play capability of Eos, so that user classes can effortlessly be used in combination with the Eos core. The configuration system, discussed in the next section, is also built around these two classes.

Other classes and interfaces worth mentioning are those that implement the Eos *space framework*. The concept of space is useful in ecosystem applications, which typically use spatial environments, but also in problem-solving evolutionary algorithms. Distance measurements, neighbourhood retrieval, and movement are essential aspects of many real-world problems. These capabilities, together with a general concept of space, are defined by interfaces provided by a dedicated space package in the core. Three main interfaces are: `Space` (which supports positions and locations); `MotionSpace` (which supports directions and movement), and `DistanceSpace` (which supports distance and neighbourhood). By using interfaces Eos' space support is not tied into one particular type of space. Simulations can even abstract from the type of space that they use. The range of spaces that is covered is broad and includes discrete Cartesian space, continuous Cartesian space, and graph space. Several carefully optimised space implementations are already available, and more will follow shortly.

Eos also includes general base classes for genomes, mutation, recombination, selection, and replacement, which are all ingredients of evolutionary algorithms. For each base class there are various subclasses, each corresponding to a particular implementation. For example, in the case of selection strategies there are subclasses for roulette wheel selection, tournament selection, breeder selection, and random selection. Eos currently explicitly supports generational and steady-state Genetic Algorithms as well as Evolution Strategies [17]. However, the large library of classes Eos provides can be used as a basis for implementing any type of evolutionary algorithm. It is of course possible to use your own specialised genomes, operators, and strategies. For example, the information filtering application described in [18] uses a hybrid genome and operators. These are specifically designed for the application but are still constructed from genomes and operators provided by the core.

B.4.4 Configuration

Ease of configuration is a particularly important goal for software platforms that are used for the development of hybrid algorithms, research experiments, and rapid prototyping of applications. The design and implementation of such systems are usually iterative, with the results from one phase of the research affecting design choices made at the next. Consequently significant effort has been put into making configuration changes in Eos simple and straightforward.

Eos applications are dynamically configured from a set of parameters. It is possible to experiment with a wide range of different configurations without even having to recompile the program code. The parameter system can be used to control factors as simple as mutation rates, right through to changes that affect the entire structure of the application and the classes of algorithm it employs.

Every component in an Eos simulation has its own set of parameters. New components, which extend existing ones, will inherit existing parameters and can modify how they are handled. They can override default values with their own and impose specific restrictions on the values that their parameters are permitted to take. Naturally they can add extra parameters of their own.

Parameters within Eos are structured hierarchically and mirror the structure of the simulation. It is therefore possible, by changing the hierarchy of certain parameters, to change the hierarchical relationship of populations and subpopulations, as well as their number and type. An example configuration is shown in Figure B.2.

What parameters you can specify depend on the specific configuration you choose to use. For example, when you use an evolutionary algorithm you can specify the genetic operators, their probabilities, the selection and replacement strategies, etc. All parameters have sensible default values so that you can develop new applications quickly and easily. To help users configure the details of their applications, a graphical frontend has recently been produced for Eos, which makes it easy to create and edit configuration files.

```
environment-params = {
 max-generations = 200;
 population-params = { num-subpop = 3;
1.   subpop-type-*=
                "core.ea.EaPopulation";
  subpop-params-* = {  size = 50;
   individual-type = "HostIndiv3";
   individual-params = {
    virulence-coefficient = 0.5;
    genome-params = {
     x-length = 12;y-length = 12;
    };
       };// etc...
```

FIGURE B.2 Example configuration illustrating how parameters of a simulation can easily be specified and modified.

B.4.5 Illustrative Example Systems

The description of Eos has so far concentrated on the rationale and motivation for developing the platform and its overall design. In this section we illustrate the flexibility and use of Eos by way of three relatively simple example systems that have been developed using Eos. These systems have been deliberately chosen to span a range of underlying architectures:

- evolutionary optimisation algorithm with explicit fitness;
- distributed ecosystem architecture with implicit fitness;
- multiple coevolutionary populations applied to function optimisation.

These simple systems are described for two reasons. First, they demonstrate the flexibility of the Eos architecture and illustrate how it can easily be configured to model a diverse range of problems. Second, a range of example system implementations can help clarify how Eos is brought to bear in tackling different scenarios.

B.4.6 Aerial Placement for Mobile Networks

Figure B.3. shows an application that optimises the placement of aerials for mobile telecommunications. It tries to cover the given region as effectively as possible, while minimising areas of overlap and the total costs. A real-valued genome is used to encode the coordinates and radii for each solution. The

genome, as well as the Evolution Strategy-specific operators that are applied to it, are all part of the Eos platform. Therefore the only application-specific code required to evolve solutions for this problem is code to evaluate the quality of each solution. The Observer design pattern, which is built into Eos, made it easy to finish the application by wrapping a dedicated graphical front end around the actual simulation.

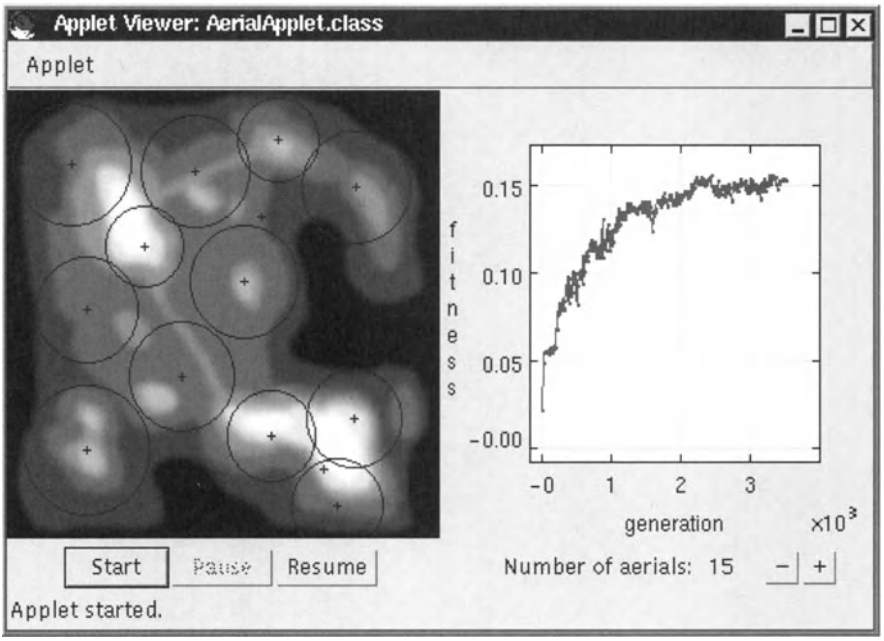

FIGURE B.3 Optimising the placement of aerials for a mobile network using an Evolution Strategy algorithm and an explicit fitness function.

B.4.7 An Ecosystem Simulation Based on Echo

Eos is also suited for running ecosystem-based simulations. An application based on the Echo ecosystem [13] has been developed to demonstrate this capability. The simulation consists of a spatial environment inhabited by various individuals. Individuals can interact (fight over resources, trade resources, mate, etc.), and move in space. This has been implemented using Eos' interaction and space framework. One advantage of using the space framework is that it is very easy to run the simulation in 3D or an entirely different type of space. It does not require a single change in the simulation core, only the visualisation would need to change.

Note that there is no notion of an explicit fitness function. Reproduction is implicitly controlled by how successfully each individual interacts with the environment and its fellow individuals.

B.4.8 Coevolutionary Function Optimisation

The application in Figure B.4 illustrates how Eos can be used to experiment with interesting variations on basic evolutionary algorithms.

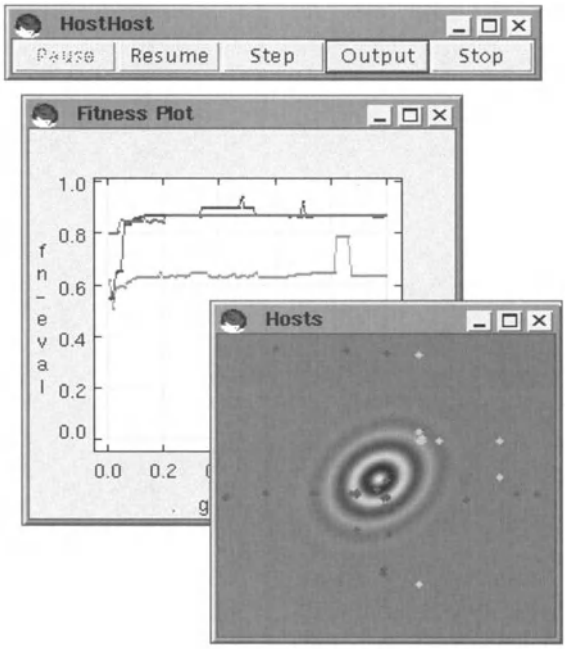

FIGURE B.4 A coevolutionary approach to function optimisation, illustrating how Eos can be used to evolve multiple populations simultaneously.

It uses a coevolutionary approach to find the highest point in a two-dimensional landscape; a function optimisation problem. Instead of operating on a single population, the application simultaneously evolves three populations. Individuals from different populations interact with (attack) individuals in the other populations. The idea is that the constantly changing selective pressure helps to prevent that populations converge prematurely. Eos' modular design and flexible entity hierarchy meant that such a nonstandard evolutionary algorithm could nevertheless be developed quickly. The short implementation

span is a real benefit, as often the only way to get an idea of how novel algorithms might perform is by implementing them.

B.4.9 Telecommunications Research Using Eos

The previous sections have described the core functionality of the Eos platform and presented some simple examples illustrating how Eos can be used to implement diverse simulations and applications. While it is desirable to have a general and flexible platform, we are particularly interested in how Eos might be used to solve problems in the telecommunications domain. Consequently several extensions have been developed that specifically support aspects of telecommunications problems.

In the following subsection the current support for telecommunications research is described. This is then further illustrated by its use in the NetGrow tool, which is a system that evolves planning rules for network growth.

Extensions Supporting Telecommunications

One version of physical space that Eos supports is "graph space." This is not strictly an extension, as it is provided in the core Eos classes, but it is extremely useful for telecommunications applications that manipulate network representations. It consists of nodes connected by edges, which may be unidirectional or bidirectional. It is based on the Eos space framework and therefore includes methods that support neighbourhood relationships and distance (e.g., hop count). This class can even be used in tandem with a Cartesian space model to simultaneously incorporate geographical aspects into an application.

Many telecommunications network management and design problems (which are appropriate to tackle with evolutionary techniques) require the simulation of a communications network in order to assess the "quality" of a given solution. Consequently a generic network simulator has been developed for use with the core Eos platform. This is a highly extensible system, permitting subcomponents to be combined in numerous ways. For instance, nodes, links, and capacities can all be manipulated and routing tables can be generated that determine routing strategies in the network.

The network simulator can be combined with another system that extends the core Eos functionality – the "traffic generator." This uses a powerful and flexible event-driven model and schedule in order to allow simulated traffic to be provided to the network simulator. Traffic can be generated that is subject to probability distribution functions or alternatively can be generated by deterministic algorithms. This is a very powerful system, but it is not appropriate to describe it further here.

In the next section we describe how Eos, together with the above extensions, is being used in a network growth research application.

B.4.10 NetGrow: A Telecommunications Application

When growing networks to satisfy increasing demand, designers and planners often use a set of rules that dictate how the network should be grown based on the current conditions and expected future demand. The quality of the resulting network is thus heavily dependent on the quality of these planning rules. A tool is being developed using Eos that aims to aid in planning network growth through evolving high-quality planning rules.

A core component of the tool is the Eos network simulator, which allows the simulation of the current network. The simulation makes use of both graph space and Cartesian space. The former allows the logical topology of the network to be defined. The latter allows objects to be placed at precise geographical locations and thus provides information regarding the geographical proximity of nodes and equipment; information that may be required to make decisions about the placement of new hardware. Information gathered from this simulation in conjunction with the demand generated by the traffic generator triggers certain planning rules, which have the effect of adding or modifying hardware. The quality of the resulting network is calculated, taking into account hardware costs and the capability of the network to handle the demand that is ascertained using the network simulator. This measure of quality is used to assign a fitness value to the rule set that produced the network.

The core functionality of Eos is used directly in order to evolve the rule sets. Each rule set is defined as an individual, and these individuals are grouped into a population. The type of evolutionary algorithm, mutation operator, and recombination operator are selected from the Eos libraries. The user interface developed for NetGrow allows control and configuration of the system in conjunction with the parameter configuration mechanism that is a core component of Eos. NetGrow also uses Eos' observer support to connect the user interface to the actual simulation code. Further details of this research are presented in another paper in this journal [19].

B.4.11 Eos Summary

The Eos evolutionary and ecosystem R&D platform has been described in some detail. The overall object-oriented design of the system has been presented, together with the motivations for developing the platform. The flexible nature of Eos has been demonstrated in the context of three diverse example applications and a more comprehensive, telecommunications-related application. The powerful object-oriented design, modular configuration system, and large library of structures and existing algorithms make the system very suitable for experimenting with novel algorithms as well as quickly developing full-featured applications.

Eos is already in use by a number of people within BT Labs for both longer-range research, and more applications-oriented problems, across a broad range of domains. In addition, it has been licensed (without charge) to a number of our

research collaborators and interested parties in academia. We consider the feedback provided by such users to be extremely valuable in helping improve the software platform over time, and incorporating new algorithms and features.

B.5 Traveling Salesman Problem

This section uses a classic search and optimisation problem to demonstrate the abilities of an evolutionary algorithm.

In the Traveling Salesman problem a fixed number of cities are placed on a square plane. The traveling salesman wants to visit all the cities and then return back home. The task is to find the shortest path that he should follow, given the number of cities and the distances between them.

Unfortunately, the number of possible tours increases enormously as the number of cities increases. For example, with 5 cities, there are 4! (=24) possible tours that must be evaluated to find the minimum-length closed tour. For 10 cities, there are 362,880 tours to be evaluated to find the minimum-length tour! If we double the number of cities again to 20 cities, we have 1.216 times 10^{17} possible tours and for 30 cities, we have 8.841 x 10^{30} possible tours. Now, assume a hypothetical machine that can compute the length of a tour in one instruction and that the machine computes 1000 MIPS, evaluating all possible tours of a 30-city problem would take this machine 280 billion millennia or approximately 10,000 times the estimated age of the Universe! The Travelling Salesman problem (TSP) therefore falls into the category of NP-complete problems. Such problems are considered to have no solution in real time, although by approximating the solutions to these problems to within specified tolerances approximate solutions may be obtained.

(For a comprehensive analysis of the TSP see the text by Lawler et al., 1985. Alternatively a useful online report on TSP can be found at www.lips.utexas.edu/~scott/ta/project9/TSPReport.htm).

B.5.1 EOS Traveling Salesman Problem

The EOS Java toolkit includes a demo application based on the TSP. In the TSP application each individual simply has a sequence genome that encodes a possible order (sequence) in which it is possible to visit all cities.

The application demonstrates various features of Eos:

- It illustrates how sequence genomes can be used. For the TSP, sequence genomes are a natural way of representing solutions. The operators that are applied to it are specific to sequence genomes that are guaranteed to always produce valid sequences. The EA also uses multiple mutation operators simultaneously. This is very useful in this case, as each operator implements a different way of making "obvious" improvements to a given solution.

- It demonstrates how simulation wrappers can be used for visualisation. It is easy to control how each is configured and you can even entirely disable some or all visual components. You can control this entirely from the simulation wrapper parameter file (wrapper_parameters.txt), in the Eos package.
- It uses Evolutionary Activity statistics.

To run the TSP application, first download the Eos library and follow the installation instructions (i.e., set the computers classpath to the download directory), then simply type at a DOS command prompt:

```
java com.bt.eos.app.tsp.TSPApp parameters.txt
wrapper_parameters.txt
```

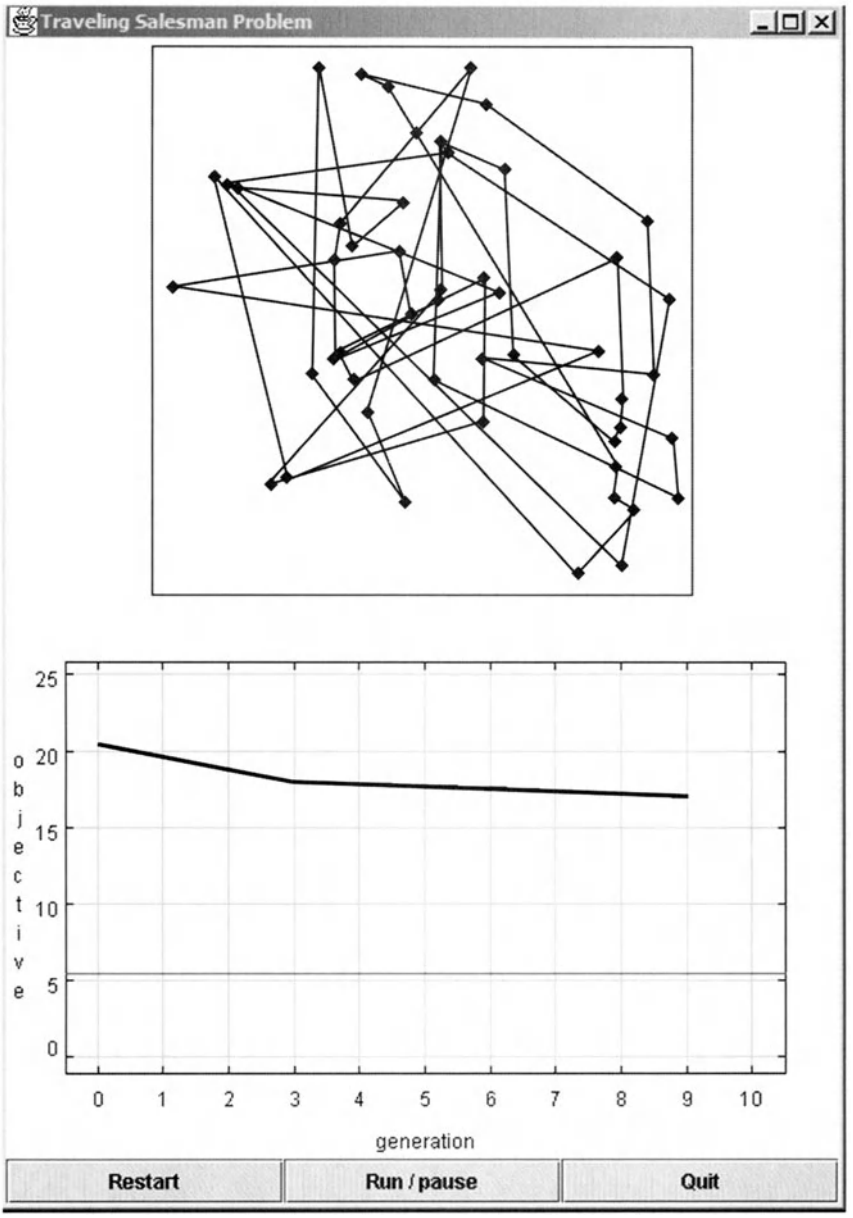

FIGURE B.5 Screenshot of the Eos TSP problem. Lower curve shows the current best fitness objective response for a completed run.

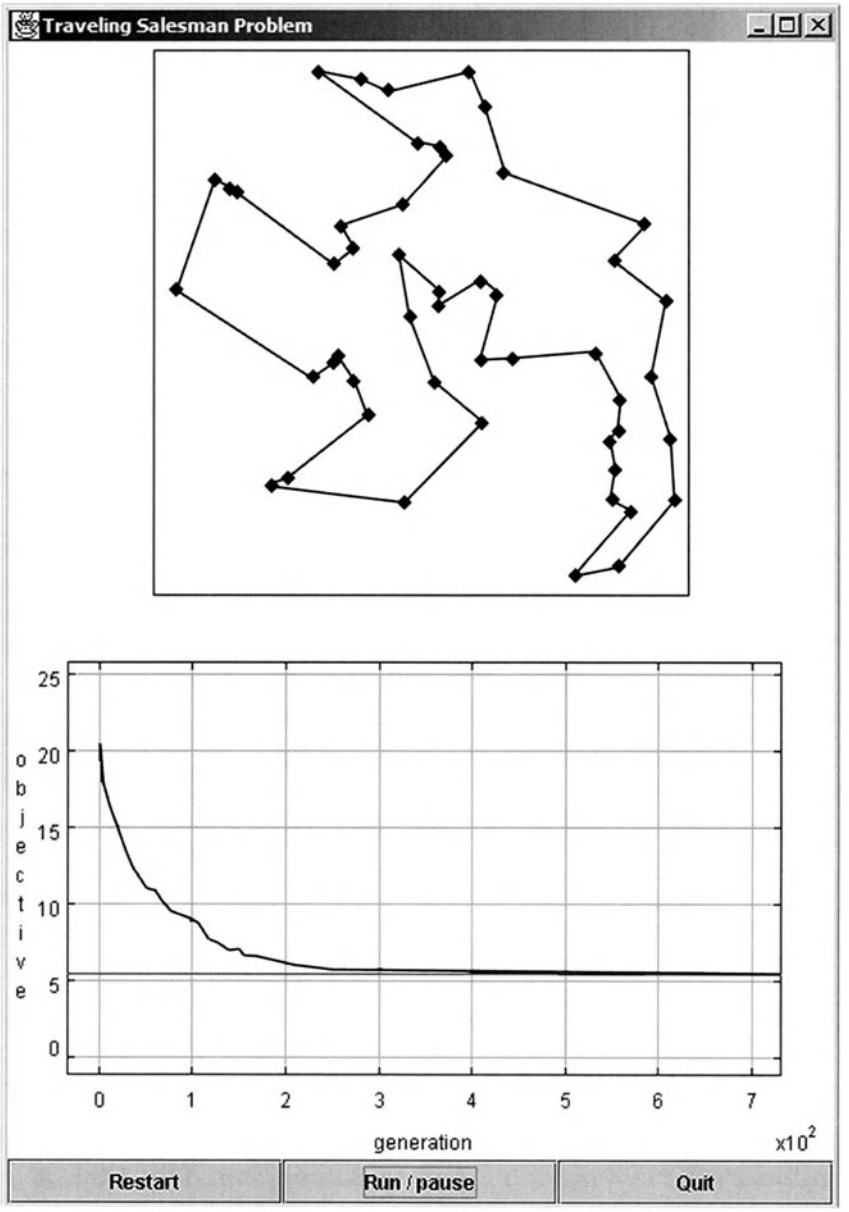

FIGURE B.6 Screenshot of the Eos TSP problem. Lower curve shows the current best fitness objective response per generation. After 30 generations a solution has been acquired.

There are a large number of alternative Java implementations of the TSP using EA. A quick search on Google or any search engine for "TSP + GA" will return a large selection.

B.6 Genetic Programming

This section contains a brief overview of the Java Genetic Programming library Gpsys, created by Adil Qureshi. He has prepared the following overview of the library of code. The full package is available online from http://www.cs.ucl.ac.uk/staff/A.Qureshi/gpsys_doc.html_. The package comes bundled with two example problems. The first is the *lawnmower problem*. Some of the major classes are described briefly below.

Run the program by typing

```
"java gpsys.lawnmower.LawnMower 0 100 100"
```

LawnMower Class: The first parameter (0) represents the "rngSeed". The second parameter (100) represents the initial population. The third parameter (100) represents the number of generations to be produced. If the rngSeed == 0, then the current time is used as the seed for subsequently produced random numbers. The LawnMowerGPParameters are then set (see below). Finally, a new instance of GPsys is created and its "evolve" function is called with the GPparameters as an argument. The LawnMower Class implements the GPObserver class, which monitors events such as "generationUpdate", whereby it writes a status report to the console window.

LawnMowerGPParameters Class: The probability of a mutation is set at 0.05. The tournament size (for determining the fittest to reproduce) is 7. A lawn of size 8 x 8 is constructed, and an instance of the mower class is also produced. Finally, the Chromosome Parameters are set. Lastly, the fitness is calculated (see below).

LawnMowerChromosomeParameters Class: Each of the LawnMowerChromosomeParametersADF# classes inherits from the Chromosome Parameters class. Consists of a max Depth (max depth of gene tree), the set of functions and set of terminals, among others. Two types of functions, which determine lawnmower movement behaviour, are Vector2Mod8 and Frog. The former allows the lawnmower to move up or down/left or right, and the latter allows the lawn mower to leap to a distant square on the lawn. Terminals include instructions such as Left (turn the mower left by 90 degrees) and Mow (move the mower forward and cut the grass on that square).

Gene Class: A gene is a node in a Genetic Program Tree. It can be either a function or a terminal. Hence, there are GeneFunction and GeneTerminal classes.

Chromosome Class: References the root gene of its gene tree and implements the different Genetic Operations such as Mutation and Crossover.

LawnMowerFitness Class: The fitness function takes into consideration two different values: the amount of lawn left uncut when the program terminates and the complexity value of the program. The fitness and complexity values are represented in the "Individual" class.

Individual Class: Each individual represents one program from the pool, which is supposed to try to solve the problem. Each individual consists of a set of chromosomes, a fitness value, and a complexity value. Constructors for creating individuals through mutation and crossover are available.

Population Class: Consists of all individuals from a generation. The fittest individual is kept track of, as well as the generation number, average fitness, and average complexity, among other information.

B.6.1 Observations from Running GPsys on the Lawnmower Problem

One classic example from John Koza utilises GP to evolve the controller for a simulated robot lawnmower. This example is included in the Gpsys package and is included here for comparison with the GA-based robot application in Chapter 5. An exercise for the reader is to use Gpsys to evolve a navigation controller for the Rossum robot simulation. Running the Gpsys lawnmower program with an initial population of 10 individuals, for 5 generations, does not guarantee a successful program being produced. The programs evolve slowly as seen in the following example:

best individual of generation 0 =

ADF[0] : (AddVector2Mod8 (Frog (ADF2 (Frog Mow))) (AddVector2Mod8 (ADF2 (Frog (4,1))) (Frog (Frog Left))))

ADF[1] : (ProgN (ProgN (AddVector2Mod8 (AddVector2Mod8 Left (1,0)) (ProgN (7,0)

(7,6))) (ProgN (ProgN (5,1) Mow) (ProgN Left Left))) (AddVector2Mod8 (AddVector2

Mod8 (AddVector2Mod8 Left (5,1)) (AddVector2Mod8 (5,3) Left)) (AddVector2Mod8 (A

ddVector2Mod8 Mow (2,7)) (AddVector2Mod8 (3,3) (3,4)))))

ADF[2] : (Frog (AddVector2Mod8 (Frog (Frog (1,0))) (Frog (ProgN ADF2Arg0 Left)))

) Fitness : Fitness(55,52)

Complexity : 52

Actually, for a set of runs, each program that evolved given those constraints finished before the lawn was completely mowed.

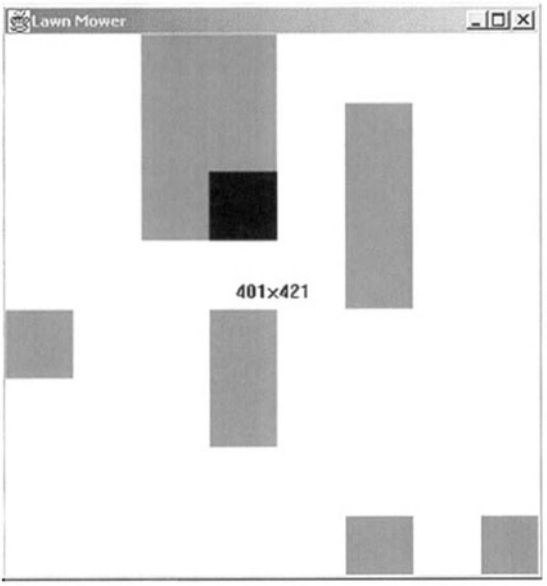

FIGURE B.7 Lawnmower scene from a "fittest" program in action; the black square represents the lawnmower.

The best program of a generation is printed to the console window for each generation evolved.

best individual of generation 1 =

ADF[0] : (AddVector2Mod8 (Frog (ADF2 (Frog ***(ProgN ADF1 (ADF2 (4,0))***))))) (AddVector2Mod8 (ADF2 (Frog (4,1))) (Frog (Frog ***ADF1***))))

ADF[1] : (ProgN (ProgN (AddVector2Mod8 (AddVector2Mod8 Left (1,0)) (ProgN (7,0)

(7,6))) (ProgN (ProgN (5,1) Mow) (ProgN Left Left))) (AddVector2Mod8 (AddVector2

Mod8 (AddVector2Mod8 Left (5,1)) (AddVector2Mod8 (5,3) Left)) (AddVector2Mod8 (AddVector2Mod8 Mow (2,7)) (AddVector2Mod8 (3,3) (3,4)))))

ADF[2] : (Frog (AddVector2Mod8 (Frog (Frog (1,0))) (Frog (ProgN ADF2Arg0 ***(1,0)***))

))

Fitness : Fitness(47,55)

Complexity : 55

An example of a program that gets the job done (i.e., a program that has evolved to the point that it can successfully mow the whole lawn) is seen below:

best individual of run =

ADF[0] : (ADF2 (ADF2 (ProgN (AddVector2Mod8 ADF1 (ProgN (AddVector2Mod8 ADF1 (Pr

ogN (AddVector2Mod8 ADF1 ADF1) ADF1)) (ADF2 ADF1))) (ADF2 ADF1))))

ADF[1] : (ProgN (ProgN (AddVector2Mod8 (ProgN Mow Left) (ProgN (4,1) (2,3))) (AddVector2Mod8 (AddVector2Mod8 Left Left) (ProgN Mow Mow))) (ProgN (ProgN (AddVector2Mod8 Mow Left) (ProgN (2,1) Mow)) (AddVector2Mod8 (ProgN Mow Mow) (AddVector2

Mod8 (2,5) Mow))))

ADF[2] : (AddVector2Mod8 (Frog (AddVector2Mod8 ADF1 (Frog ADF1))) (ProgN (Frog (

AddVector2Mod8 ADF1 ADF1)) (ProgN (Frog Left) Left)))

Fitness : Fitness(0,63)

Complexity : 63

One obvious result is that the GP output and method for solving the problem are not easily understood. This highlights a common problem in almost all evolutionary algorithms, i.e., decomposing the final genome expression into the set of rules that has solved the problem. This issue is one factor for the reluctance of some engineers to trust the output from an EA system. In comparison a key advantage of fuzzy logic is that the inferencing process is transparent to analysis (you can observe which rules are firing as the fuzzy rule set is used).

GPSYS Feature Summary

The following list summarises the key features of the Gpsys library.

- Both a Steady State and memory-efficient Generational GP engine
- ADF support
- Strongly typed
- Supports generic functions and terminals
- Has many built-in primitives
- Includes indexed memory
- Exception support
- Save/load feature
- Can save/load current generation to/from a file
- Data stored in GZIP compression format to minimise disk requirements
- Uses serialisable objects for efficiency

- Fully documented
- Commented source code
- Javadoc-generated documentation for classes
- Example problems
- Lawnmower (including GUI viewer)
- Symbolic regression
- Totally parameterised
- Fully object-oriented and extensible
- High performance
- Memory efficient

Eos References

[1] Marrow P., Nature-inspired Computing, BT Telecommunications Journal, Vol.18(4), 2000.

[2] Bäck T., Fogel D., and Michalewicz Z., Handbook of Evolutionary Computation, eds., Bristol, Institute of Physics, 1997.

[3] Heitkötter J. and Beasley D., eds., The Hitch-Hiker's Guide to Evolutionary Computation: A List of Frequently Asked Questions (FAQ), USENET: ftp://rtfm.mit.edu/pub/usenet/news.answers/ai-faq/genetic/, 2000.

[4] Costa J., Lopes N., and Silva P., Java Distributed Evolutionary Algorithms Library, http://laseeb.ist.utl.pt/sw/jdeal.

[5] Luke, S. ECJ: A Java-based Evolutionary Computation and Genetic Programming Research System, www.cs.umd.edu/projects/plus/ec/ecj/.

[6] Großmann M., et al. GENOM: An Environment for Optimization Methods, http://www.informatik.uni-stuttgart.de/ifi/fk/genom/.

[7] GeNeura Team, EO Evolutionary Computation Framework, http://geneura.ugr.es/~jmerelo/EO.html.

[8] Holland J.H., Adaptation in Natural and Artificial Systems, Ann Arbor, The University of Michigan Press, 1975.

[9] Goldberg D.E., Genetic Algorithms in Search, Optimisation and Machine Learning, Reading, MA, Addison-Wesley, 1989.

[10] Swarm Development Group, Swarm, http://www.swarm.org.

[11] Burkhart R., The Swarm Multi-Agent Simulation System, position paper, OOPSLA '94 Workshop, The Object Engine, 1994.

[12] Gamma E., Helm R., Johnson R., and Vlissides J., Design Patters: Elements of Reusable Object-Oriented Software, Reading, MA, Addison-Wesley, 1995.

[13] Holland J.H., Hidden Order: How Adaptation Builds Complexity, Reading, MA, Perseus Books, 1995.

[14] Future and Emerging Technologies, Proactive Initiative 1999: Universal Information Ecosystems, http://www.cordis.lu/ist/fetuie.htm.

[15] ObjectSpace, Voyager ORB, http://www.objectspace.com/products/prodVoyager.asp.

[16] Wollesen E.A., Krakowiak N., and Daida J.M. Beowulf Anytime for Evolutionary Computation, in Late Breaking Papers, GECCO 1999, pp.298-304. Orlando, FL, GECCO 1999.

[17] Rudolph G., Evolution Strategies, in [2], 1997.

[18] Bonsma E., A Non-discrete Approach to the Evolution of Information Filtering Trees, BTTJ, Vol.18(4): 2000.

[19] Shipman R., Coupling Developmental Rules and Evolution to Aid in Planning Network Growth, BTTJ, Vol.18(4): 2000.

[20] DiBona C., Ockman S., and Stone M., eds., Open Sources: Voices from the Open Source Revolution, Sebastopol, CA, O'Reilly, 1999.

[21] Dept., of Electrical Engineering and Computer Sciences, University of California at Berkeley, Ptolemy Project, ptolemy.eecs.berkeley.edu /java/.

Appendix C

Fuzzy Logic Systems

C.1 Fuzzy Logic

Fuzzy logic (Zadeh, 1965) is a flexible machine learning method, which has enjoyed wide popularity in engineering as an advanced control and AI technique. This appendix outlines the theoretical background of fuzzy sets before detailing how they can be used in control systems and why they are particularly suited for use in mobile robots controllers. A primary reference in the field is Driankov et al., (Driankov, Hellendoorn, and Reinfrank, 1996). This reflects the quality of their text as a convenient source of material but also the fact that it is a unique and groundbreaking work in introducing fuzzy techniques to the established field of control engineering and gives an unbiased view of the uses and limits of fuzzy control systems.

Since Lotfi Zadeh's original work on Fuzzy Logic (Zadeh, 1965), intense debate has surrounded the subject with vigorous defenders and opponents on both sides. Fortunately for those seeking a working method of controlling complex systems, these debates are largely irrelevant, as it is surprisingly easy to construct and operate a fuzzy logic system. (The reader should be assured that it really is very easy to create a basic working Fuzzy Logic control system for a simulated model. It is not necessary to understand the underlying theory, although of course it helps!) This material supports the application in Chapter 5, which contains details of how a fuzzy logic system can be used to control co-operating mobile robots, and how genetic algorithms enable the automation of the design process.

The reference by Kosko (1992) provides a useful introduction to the theory of Fuzzy Logic and how it may be combined with neural network techniques.

C.2 Fuzzy Set Theory

The basis of fuzzy logic is the concept of *membership*, which defines to what degree an element belongs to a fuzzy set. This is in contrast to the classical notion of sets where an element either is or is not a member of a set (defined as a *crisp* set). Hence for a crisp set C, defined on a universe U, then for an element u of U we can say $u \in C$ or $u \notin C$. Using a fuzzy set F there is a characteristic function termed the *membership* function which assigns to each $u \in U$ a value from the unit interval [0,1]. We can then formally define the membership function of a fuzzy set F:

$$\mu_F: U \rightarrow [0,1] \tag{C.1}$$

As an example consider the property tall applied to the set "people." This property is a relative term and can be defined in terms of fuzzy sets. Three suitable fuzzy sets would be "short", "average", and "tall." Figure C.1 shows this as a graph where the *y*-axis defines the degree of membership of a fuzzy height set. Hence a height of 1.7 m belongs to the set average to degree 0.5 and to the set short by degree 0.3. The shape of the membership function can take several forms, such as trapezoid, triangular, or bell, as in Figure C.2.

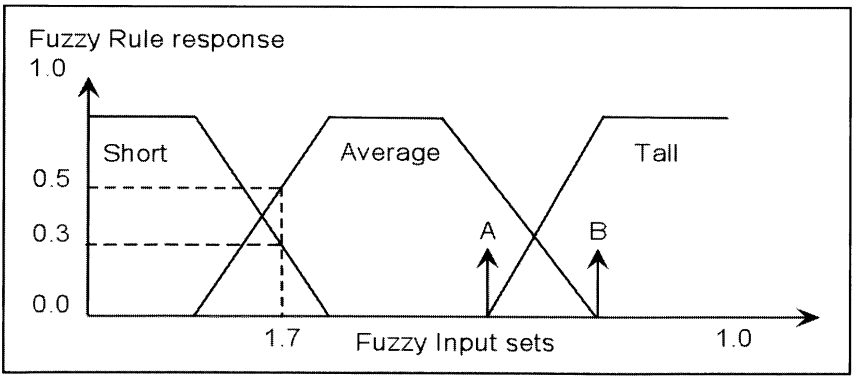

FIGURE C.1 Example of fuzzy sets defining human height (A and B define overlap limits).

The most common practical shape is the trapezoid form as it is computationally efficient. It is important in most control applications that we use

membership functions, which satisfy the "width condition." In this case the end-point of one function must fully overlap with the start point of an adjacent function as at points A and B. If the functions are asymmetrical at point A, then discontinuities can occur in the control output (hence leading to loss of control in the system).

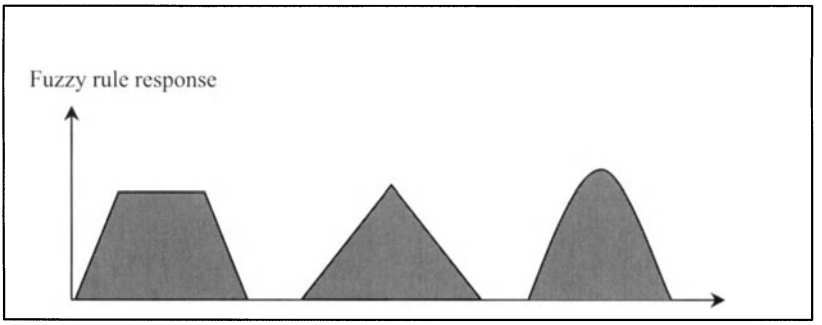

FIGURE C.2 Examples of common fuzzy set membership functions; trapezoid, triangular, and Gaussian or ellipsoid.

C.2.1 Fuzzy Operators

Having defined a system of fuzzy sets it is necessary to establish a formal logic with which to operate on them. By extending classical logic operations, Zadeh proposed that

$$\forall x \in X : \mu_{A \cap B}(x) = \min(\mu_A(x), \mu_B(x)) \tag{C.2}$$

$$\forall x \in X : \mu_{A \cup B}(x) = \max(\mu_A(x), \mu_B(x)) \tag{C.3}$$

$$\forall x \in X : \mu_{A'}(x) = 1 - \mu_A(x) \tag{C.4}$$

It is therefore possible to build connectives from logical operators such as the classical logic operators AND, OR, and NOT.

C.2.2 Linguistic Variables

A critical concept in manipulating fuzzy sets and operators is the idea of a *linguistic variable.* Zadeh defined this as

> By a linguistic variable we mean a variable whose values are words or sentences in a natural or artificial language. For example, Age is a linguistic variable if its values are linguistic rather than numerical. i.e. young, not young, very young...etc. (Zadeh, 1975).

This provides the basis for knowledge representation in approximate reasoning and it is usual to have a set of symbolic names associated with each linguistic variable:

$$\langle X, LX, X, Mx \rangle \tag{C.5}$$

where LX is the set of linguistic values that X can take. Mx is a semantic function that assigns a meaning to a linguistic value using the quantitative elements of X. A more common expression is to use U for *universe of discourse* rather than X. For example, in a fuzzy knowledge-based controller the error or change of error can be the set:

{NL,NS,NM,ZE,PM,PL} where NS = negative small, NM = negative medium, ZE = zero (often an average point), PM = positive medium, and PL = positive large. This scheme is adopted in the example application of Chapter 5 and is shown in Figure C.2. The GA used in the application is basically searching for the values to assign to each of the cells in the output array, which may be in the range NL to PL (for 5 membership sets).

C.2.3 Fuzzy IF

Based on the framework of fuzzy logic and sets we can process information using standard if-then relationships. In terms of fuzzy logic we can express a fuzzy production rule as

if (fuzzy proposition X) then (fuzzy proposition Y)

In a control system, for example, the production rule links the process state and control output variables; for example, given two input variables b,c and one output variable u, then we can state

if b is NS AND c is PL THEN u is NM

that is the fuzzy causal relationship between inputs and outputs can be stated as

If the current value of input b is negative small and value of input c is positive large then apply a negative medium shift in control output u.

This method of using fuzzy logic within a rule-based system comes from the pioneering work of Mamdani (Mamdani, 1974) in fuzzy control systems.

C.2.4 Fuzzy Associative Memories

A powerful representation scheme for describing fuzzy systems has been proposed by Kosko (1992) in which multidimensional fuzzy sets can be mapped onto each other. In this method fuzzy sets are viewed graphically as points in the unit hypercube, where continuous fuzzy systems behave as associative memories that map groups of close inputs to close outputs. These are defined as *fuzzy associative memories*, or FAMs. The simplest case of a FAM encodes the FAM rule association (A_i, B_i), that is, it associates the n-dimensional fuzzy set B_i with the n-dimensional fuzzy set A_i. For most practical systems, however, a simplified binary input-output scheme is used.

A classic example of applying the FAM scheme to a control problem is the inverted pendulum on a moving cart problem. The speed of a drive motor must be adjusted to drive a cart such that a balanced inverted pendulum remains upright. There are two input fuzzy state variables, the angle φ of the pendulum to the vertical and the angular velocity $\Delta\varphi$. The control output fuzzy variable is the motor current v_t. The real number line R is the universe of discourse for each fuzzy variable, each of which is divided into five overlapping fuzzyset values, such as NS, NM, ZE, PM, and PL.

Once the membership functions have been assigned we can proceed to build the relationships between the input and output fuzzy sets. Each FAM rule is a triple set, which describes how to modify the control variable for specific values of the pendulum states. The possible set of FAM rules can then be defined as a N by N matrix, where each cell entry is a linguistic fuzzy set.

An interesting design point is to what degree should the fuzzy sets overlap. If the overlap is too great, then there is little distinction between the fuzzy sets, while too little will generate a bivalent control structure with poor performance (Welstead, 1994). Kosko suggests an overlap value of approximately 25%, although Driankov (Driankov et al., 1996) suggests that the critical parameter in terms of smooth control is whether the membership functions are symmetrical with respect to each other. (In the example code both criteria were used, and the effect of not using symmetric overlap in the membership functions is significantly reduced performance.)

A useful property of this method is that the binary FAM matrix can be displayed as a geometric surface in the input-output product space sometimes called the *action surface*. Chapter 5 gives one example of such a surface, generated from the evolved mobile robot fuzzy control system.

NL	PL	NL	ZE	PM
NL	PL	ZE	ZE	NL
NL	PM	PL	NL	NL
NL	PM	NL	PL	NL
NL	NL	NL	PM	ZE

FIGURE C.3 The typical FAM matrix representation of the pendulum control problem, with $N = 5$. Each shaded cell represents a rule that is being repeatedly fired by the inferencing mechanism, hence in this case the rules that are responsible for keeping the pendulum at zero degrees with minimal angular velocity are most active.

In operation the binary FAM system inference procedure activates the antecedent rules of each matrix entry to generate the resultant fuzzy output at each point. In practice a *min* or *max* product rule is used to inference between fired fuzzy rules. To create the output set we can use product inferencing again

$$\forall x: \mu_{out}(x) = \max(\mu_{out1}(x), \mu_{out2}(x), \ldots.) \tag{C.6}$$

The next stage is to convert the fuzzy output set back into a crisp value, in this case motor current value v_t.

C.2.5 Fuzzy Control systems

A complete fuzzy control system is illustrated in Figure C.4, which is composed of several distinct modules. It is important to realize that a large number of design options exist based on this generic method. A number of additional features are required for an industrial control system, such as stability analysis of the final fuzzy rule set (Driankov et al 1996.)

These are common features to most fuzzy-based control systems that may be classified in general as knowledge-based controllers.

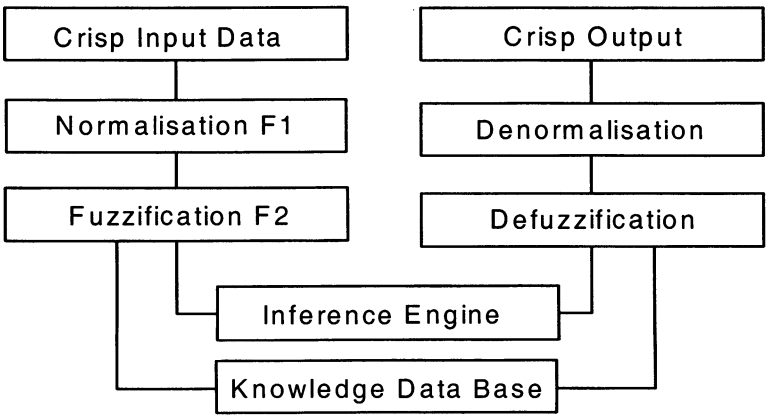

FIGURE C.4 Typical stages in a fuzzy controller, (After Driankov et al., pp.104, 1996.)

The modules functions are

F1: Performs an input normalisation to map the physical values into a normalised universe of discourse.

F2: Performs the fuzzification procedure. This may be composition-based inference or individual rule firing-based inference.

Knowledge Base: This is composed of a database and a rule base. The database provides the information needed for defining the fuzzy sets, physical domains, membership functions, and any scaling factors. The rule base encodes the experience of a human expert within the set of if-then rules.

Inference Engine: The function of the inference engine is to compute the overall value of the control output variable based on the individual contribution of each rule in the rule base.

Defuzzification Module: Translates the set of modified control fuzzy output values into a crisp value.

C.2.6 Defuzzification

There are several common methods used in translating the combined output of the fuzzy output sets into a single discrete value representing the new control variable. In general, they involve either taking: the center of area of the set, or

taking the average of the elements that are members with weighting due to degree of membership.

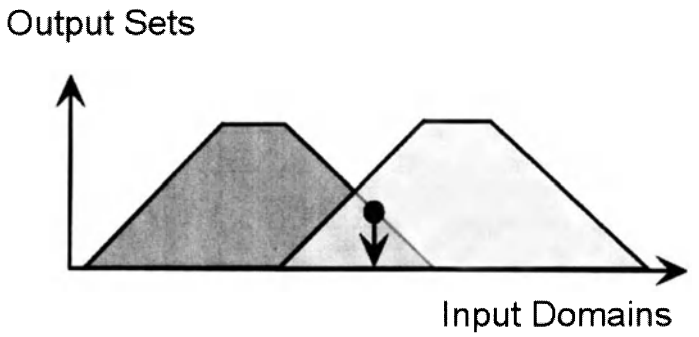

FIGURE C.5 An example of a center-of-area defuzzification method.

The most frequently used defuzzification methods are

- Center-of-area/gravity
- Center-of-sums
- Height defuzzification

This method works by finding the center of the area below the combined output membership functions, and only takes account of the overlap area once.

$$u^* = \frac{\sum_{i=1}^{l} u_i \cdot \max \mu_{CLU(k)}(u_i)}{\sum_{i=1}^{l} \max \mu_{CLU(k)}(u_i)} \tag{C.7}$$

where u^* is the crisp output value from the defuzzification operation, and CLU is the value for the control output after the firing of a rule. A faster method is the center-of-sums, which ignores any overlap between the output sets.

The method illustrated in Figure C.6 using height defuzzificiation is the simplest and fastest method available and ignores both the shape and support of the membership sets and simply uses the weighted peak of each set.

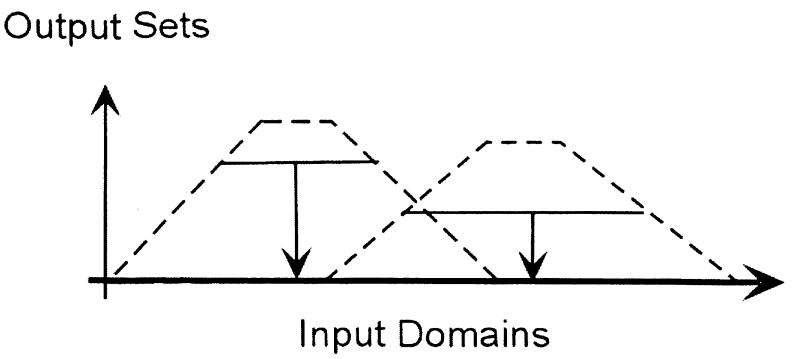

FIGURE C.6 An example of the height defuzzification method.

$$u^* = \frac{\sum_{k=1}^{m} c(k) \cdot f(k)}{\sum_{k=1}^{n} fk} \tag{C.8}$$

where $c^{(k)}$ is the peak value of the set LU and $f^{(k)}$ is the height of the clipped set CLU . As an example, if we assume a pair of input values for the domains of the angle φ of the pendulum to the vertical and the angular velocity $\Delta\varphi$, of $\varphi = 0.4$ and $\Delta\varphi = 0.8$, then the active or fired rules are

$$\begin{aligned} w_1 &= \min\{ZE(0.4), ZE(0.8)\} \\ w_1 &= \min\{0.2, 0.3\} = 0.2 \end{aligned} \tag{C.9}$$

where 0.2 and 0.3 are the degree of membership of each value in the given set. The process is repeated for each rule activated by the particular inputs and the crisp output is found by summating the product of each weight as in Eq. (C.10).

$$Output = \frac{(w_1 \cdot NL + w_2 \cdot NM + ... w_n \cdot PL)}{\sum_{i=1}^{n} w_i} \tag{C.10}$$

In Chapter 5 the specific fuzzy system used in this work is presented and this uses the height defuzzification method, as this has the advantage for real-time control of being very computationally efficient compared to the other methods.

C.2.7 Fuzzy Applications

As stated at the start of this chapter fuzzy logic-based systems have been extensively used in the Far East, since the early 1980s. Common applications typically use a dedicated fuzzy VLSI microcontroller as an embedded device; examples include washing machines that monitor the dirt level in the wash and adjust their cycle automatically, autofocus systems in video cameras, and transport braking systems (as in the Sendai city underground railway). Driankov (Driankov, Hellendoorn & Reinfrank, 1996) outlines the principal reasons why fuzzy logic may provide improved performance over conventional control techniques, such as

Implementing expert knowledge for a higher degree of automation:
Fuzzy logic is ideally suited to industrial process control where the control cycle time may operate over an extended time period, and has been used in paper and cement production systems.

Robust nonlinear control:
Fuzzy systems can cope with major parameter changes and widely varying load conditions.

Reduction of development and maintenance time:
Fuzzy systems are generally very simple to implement, even when the designer has no detailed knowledge of the formal theory behind fuzzy logic.

Marketing and patents:
Less obviously the use of a fuzzy system may allow a company to bypass existing patents for a control system.

For these reasons and the advantages listed below many Western manufacturers are beginning to adopt fuzzy systems. A further reason is the availability of complete software and hardware tools such as the Motorola 6812 fuzzy microcontroller.

C.3 Limitations of Fuzzy Control

As with any complex system fuzzy logic has its limitations, and contrary to some advocates' belief, it is not a panacea for all problems. If a problem is already well defined mathematically and operates within known parameters, then standard control algorithms probably exist which are highly efficient and

well optimised. In addition, the nonlinear nature of most fuzzy controllers makes mathematical analysis of control factors such as stability or existence of singularities extremely difficult to predict (although Driankov's work is important as it attempts to address these issues in a practical way).

C.3.1 Advantages of Fuzzy Systems

The advantages of fuzzy control systems can be divided into theoretical and practical issues. First, arguments for fuzzy control include

- They permit the incorporation of linguistic knowledge into a control architecture.

- It is a model-free system; hence no underlying mathematical model of the control process or plant is required.

- It provides an efficient nonlinear control mechanism, which is justified by the universal approximation theorem.

Practical reasons for fuzzy control include

- Ease of implementation
- Simple to design, compared to an equivalent conventional controller for the same system (particularly true for nonlinear systems)
- Cheap to develop, with well-established software packages available, and fast to prototype new code if required
- Possible to implement as a VLSI chip which are now widely available (in contrast to the possibility of hardware ANN systems).

Hence, there are powerful reasons for using fuzzy control in many systems and only a lack of knowledge is preventing a more widespread use of this technique.

Robots and Fuzzy Control

One of the first applications of a fuzzy control system in mobile robots was by Sugeno & Nishida in 1985, in which a model car was guided along a track with a single rotating ultrasound sensor measuring the distance to a wall. More recent work has demonstrated that fuzzy control is quite capable of controlling many parallel behaviours, including navigation, obstacle avoidance, and beacon location (Saffioti, 1995, and Surmann et al., 1995). Chapter 5 illustrates the capabilities of a fuzzy application in which a fuzzy controller acts as an adaptive parameter controller for a behaviour-based mobile robot.

C.4 Summary

Fuzzy logic has been shown to provide a powerful and flexible mechanism for processing numeric information using linguistic rules and offers significant advantages over conventional control methods. However, the design of the parameters and rule structure for a real-world application can be time-consuming and knowledge-intensive. It is therefore a suitable domain in which to apply evolution-based search techniques.

Further Reading

Driankov D., Hellendoorn H. & Reinfrank M., An Introduction to Fuzzy Control, Berlin, Springer Verlag, 1996.

Kosko B., Neural Networks and Fuzzy Systems, A Dynamical Systems Approach to Machine Intelligence, Prentice Hall, 1992.

Appendix D

Rossum's Playhouse (RP1)
User's Guide for Version 0.48

Introduction

This appendix is a shortened version of the user guide, which is available for the Rossum's Playhouse (RP1) robot simulation. The full guide is available from http://rossum.sourceforge.net/sim.html. Many thanks go to Gary Lucas for his work in creating the excellent RP1 simulation and for allowing the reproduction of this version of the user guide in this book. The reader is encouraged to take part in the ongoing development of this simulator.

System Overview

Rossum's Playhouse (RP1) is a two-dimensional mobile-robot simulator. It is intended to be a tool for developers who are building robot navigation and control logic. RP1 does not model any particular robot hardware, but provides components that can be configured and extended to represent the platform of choice.

Use and License

The RP1 source code is freely available and may be copied and distributed according to the stipulations of the GNU General Public License (GPL). The full

text of the GNU General Public License is provided in the text file "gpl.txt" which is included in the RP1 software distribution.

Programming Language and Run-Time Environment

All source code for Rossum's Playhouse is written in Java. All run-time code currently available is in the form of compiled Java class files. Of course, Java is not the "language of choice" for many robot developers. Keeping the system language-independent was the single most important element in its design. The techniques used to support this goal are described in the section below.

At present, however, all code for the RP1 simulator and the tools that facilitates the construction of RP1 robot models are written in Java. It is likely that tools supporting C and C++ programs will become available in the future.

To run the Rossum's Playhouse simulation, you will require a system with either the Java Run-Time Environment (JRE), Java Development Kit (JDK), or one of the many Integrated Development Environments (such as Symmantec's Café or Inprise's Jbuilder). Versions of the JRE or JDK for Sun Solaris or Windows architectures can be downloaded from the Internet from Sun's Java site http://java.sun.com/products. Other equipment manufactures have their own versions of the JDK. Rossum's Playhouse currently requires version 1.1.8 or better.

Rossum uses Java packages to group related classes. A package is, roughly, the Java equivalent of a "library." A package is a directory or an archive containing multiple classes and related data files. Java packages have a close relationship with the file directory structure. All the class files for a particular Java package must be stored in a directory with the same name as the package. For example, the RP1 software distribution provides a package called "rossum" which includes general tools for building RP1 client applications. All the class files for that package are stored in a directory that is also called "rossum."

Top-Level Directory Files and Hierarchy

The top-level directory of the version 0.4x RP1 Software Distribution is organized as shown below.

Server.class	The binary for the main simulator (source code Server.java also included)
FireFighter.bat	A DOS-style .bat file that permits running the Fire-Fighter demo using Windows Explorer or similar

	interface (will work as a Unix script as well)
LineTracker.bat	A DOS-style .bat file that permits running the Line-Tracker demo using Windows Explorer or similar interface (will work as a Unix script as well)
gpl.txt	The statement of the GNU General Public License (GPL)
RP1.ini	A Java-style properties file defining options for the RP1 simulator (Server)
rossum	A directory (Java package) containing the source and compiled classes for the rossum package, a set of general tools used by both clients and servers including the **RsProtocol, RsClient, and RsConnection** classes
simulator	A directory (Java package) containing the source and compiled classes for the simulator package
clientzero	A directory containing the source and compiled classes for the demonstration clients
demozero	A directory containing the source and compiled classes for the GUI-based client which extends the functionality defined in the clientzero package
linetracker	Line-following robot source and compiled classes
FloorPlan	A directory used to store definition (floor-plan) files for simulated environments

Units of Measure

In the external user interfaces, the RP1 simulator and related applications are free to display distance and angular measures in whatever units the user pleases. Internally, distance is always measured in meters. Angles are always measured in radians, *counter*-clockwise from the x-axis. Time is measured in seconds when using real-valued specifications, but is specified as milliseconds when integral quantities are used. In general, the minimum resolution of time in the simulator is the millisecond.

The Client-Server Architecture

RP1 is based on a client-server architecture. Anyone who has ever operated a web browser has experienced a client-server architecture. The browser, a client program resident on one computer, connects to a server, which is resident on

another. The two programs exchange data via network-based communications. Often, more than one client may connect to the server simultaneously.

Though its implementation is quite different than that of a web browser/server, the relationship between RP1's client/server components is analogous. In RP1, the server is the simulator and the virtual world that it provides. The clients[1] are the robots that occupy that world. The server provides the virtual hardware for those robots. It models their bodies and physical interactions with their simulated environment. It responds to their commands to change position or begin a movement and provides them with sensory feedback. But it does not control the robots. Control comes from the clients. In a sense, the clients provide the "brains" for the robot, while the server provides them with a "world."

Network/Local Connections Versus Dynamically Loaded Clients

The RP1 client and server can run in two different modes:

1. As two individual programs, potentially written in different languages and running on different computers (if the programs run on different computers, the client establishes a *network* connection; if they run on the same computer, the client establishes a *local* connection).
2. Using a *Dynamically Loaded Client (DLC)* that is loaded into the server at run-time[2]

Each of these approaches has its own advantages and disadvantages. If you implement the client and server as separate programs, it does permit you to run a client that is not written in Java.[3] It also provides better isolation between the client and the server for debugging purposes. If the client is crashing, it does not affect the server (and vice versa). And, again, it even allows you to run a client on a different machine (and even a different *kind* of machine) than the server.

---

[1] In the current implementation, the server can handle only a single client at a time. Support for multiple clients is planned for future development.

[2] The ability to support dynamically loaded clients was introduced in Revision 0.48.

[3] An C++ API is currently under development. No release date is available at this time.

Why a Client-Server Architecture?

Language Independence

The primary motivation for adopting a client-server architecture was language independence. Although the RP1 simulator is written in Java, few developers implementing robotic software are currently working in that language. The division of the simulation into separate entities, a client and a server, allows the two components to exist in different environments. Client programs are free of any dependency on Java. They may be written in a number of different languages and may even run on a different computer than the simulator/server. The only system requirement is that the platform of choice be able to handle elementary data types and network connectivity.

Quicker Development for User Implementations

The client-server architecture simplifies user implementations by separating their code from the simulator software. Code written for independent clients tends to be more modular and more easily extracted from the simulator. The overall size of client executables is reduced, thus permitting faster development, testing, and modification. And because the client and server are implemented as separate entities, it becomes easier to identify the source of problems. If the client crashes, the simulator continues to run and can accept new connections without delays for start-up times.

Extensibility

Even a full-featured simulator cannot meet every user need. Eventually, there will be some requirement that falls outside the range of the simulator. The client-server architecture used for RP1 allows users to easily develop interface layers in their applications to allow them to extend the capabilities of the simulation. Because the client-server architecture provides the user with more control over their own code than they would in an integrated simulation, the architecture allows them more freedom in building their applications. Users are free to import tools, adapt existing solutions, and code the algorithms without the restrictions that other architectures might impose.

Client-Server Communications

Of course, for a client-server architecture to work, there must be some mechanism for the client and server to talk to each other. We also mentioned that there were two modes for running the client and server modules. If the modules are run as separate programs (or as Java applications running in

separate Java Virtual Machine sessions), RP1 uses a network-based communication mechanism known as the "socket." If the client module is loaded dynamically, it can still use socket communications, but might benefit from a more efficient mechanism known "piped data streams" or "pipes." Pipes are a means of allowing one part of a program to talk to another part as if they were separate entities.[4] Because pipes are restricted to a single program, they can bypass the overhead associated with socket communications.

Network and Local Connection Issues

When running as separate programs, RP1 clients and servers communicate using the same protocol that is used in many Internet applications, including net browsers and email applications. This protocol, which is called TCP/IP (or, in informal settings, Internet protocol), provides reliable connections between the client and server and permits them to communicate without loss of data. While TCP/IP is widely used for Internet communications, it is also suitable for exchanges between programs running on the same computer. When a client connects to a program running on the same computer, it is said to be making a connection to the *local host*, thus making a *local connection* rather than a *network connection*.

An excellent discussion of TCP/IP network communications can be found in *TCP/IP Network Administration* by Craig Hunt, published by O'Reilly and Associates.

When a TCP/IP client attempts to connect with a server, it does so by specifying a hostname or Internet address. Hostnames are generally human-readable strings. Before they can be used by the computer, they need to be resolved into an IP address. An IP address is a single integer value, which gives a unique address for every computer on the associated network (and, perhaps, the Internet itself). For human purposes, IP addresses are typically expressed as a set of four values such as 198.186.203.33.

Any particular host may be running several servers (or "services"). So in addition to the host address, a client wishing to connect to a particular server will require something to indicate which one it wants. This specification is accomplished through the use of a port number. A port number is an arbitrary two-byte integer value. Service are assigned to port numbers according to convention and accepted practice. For example, http (worldwide web) connections user port number 80. The POP3 post office protocol (used for

---

[4] In Java, pipes are actually a method of allowing different *threads* to exchange data and, in fact, may result in deadlock if used to communicate within a single thread.

email) uses port 110. The file transfer protocol (ftp) uses 21, etc. Port values between 0 and 1023 are commonly reserved for system applications (such as ftp, telnet, timed, etc.) and should not be used for other applications.

In the current revision of Rossum's Playhouse, details such as port assignment and host are specified in the properties file "rossum.ini" which is described below. In the version supplied with the Version 0.4x distribution, the following values are supplied:

host: 127.0.0.1, port: 7758

The IP address 127.0.0.1 is called the "loopback address." A loopback address allows a machine to make a virtual connection to itself. It allows a client to look for the server on the same machine as the one on which it is running. If you wish for a client to communicate with a machine other than its own host, you will have to change this IP address. If your network is configured properly, the RP1 software will also accept hostnames as entries in this field.

The port number is read from "rossum.ini" and reserved for client connections when the server starts up. There is nothing special about the value 7758, we simply chose a port number that we thought would not be used by another application. There is a small, but nonnegligible, probability that there will be another process running on your system that allocates this port number. In that case, there will be a conflict and the simulator will be unable to run. The conflict will be noted in the RP1 run log. To resolve the problem, simply choose a different port number and restart the simulator.

Communication via Events and Requests

Once the server (simulator) is running, it may accept connections from clients at any time using the TCP/IP protocol. TCP/IP is a low-level protocol sufficient for establishing connections between clients and servers, but most applications require that a higher-level protocol be implemented "on top" of TCP/IP to support the exchange of meaningful data. Rossum's Playhouse implements a set of conventions known as the "RP1 protocol."

In the RP1 protocol, most data is exchanged between a client and server in the form of messages. Messages sent from the client to the server are described as "requests." Messages from the server to the client are "events." A typical request might be something like "rotate the left wheel ½ turn at 20 rpm." A typical event might be "sensor 1 detected a light source 5 degrees off its central axis." The choice of these terms reflects the realities of controlling an actual robot. Although an operator might "want" a robot to move in a particular way, physical issues (such as wall collisions or power limitations) might make it

impossible for the system to comply. Therefore, messages sent to the robot are described as "requests" rather than "instructions" or "commands." Similarly, the choice of the word "event" reflects the probability that conditions might occur in the simulated world that are unpredictable and require the client to respond without warning. In general, you will not need to know the details of the RP1 protocol. It is handled by the rossum API that provides a simple interface for exchanging data.

Keeping the RP1 Protocol Language-Independent

The importance of permitting clients to be written in languages other than Java was noted above. In the existing RP1 code, messages are implemented (quite naturally) as Java classes. Java provides an elegant method for transmitting classes between different applications ("applications" are the Java equivalent to "programs"). But this method, called "object serialization," requires a special data encoding that is not easily implemented in other languages.

Configuration Elements and Properties Files

All RP1 client applications depend on at least one properties file, rossum.ini, which specifies the network communications port that clients can use to connect to the Server. The Server, of course, also needs this information so that it knows on which port to establish its service. The rossum.ini file is treated as a Java "resource" and is stored as part of the rossum package (recall that Java packages are equivalent to directories). Because it is treated as a resource, you do not have to worry about file path. As long as Java can find the rossum package, it will be able to find the rossum.ini resource (and if it can't find the package, you won't be able to run the application anyway).

The information in the rossum.ini file is used to populate the elements of a Java class known as RsProperties. The RsProperties class is derived from Java's standard Properties class. In Java, Properties can be used to read a file, which specifies a set of values using a simple syntax that resembles a traditional assignment statement:

```
# rossum.ini -- fundamental specifications for all RP1 applications.
port=7758
hostName=127.0.0.1
logFileName=rossum.log
logToFile=false
logToSystemOut=false
logVerbose=false
```

Properties Specifications from rossum.ini

Java's Properties class provides methods for getting the strings associated with each properties name. Because RsProperties extends Properties, it inherits all Properties methods. For example,

```
RsProperties rsp  = new RsProperties();

String       name = rsp.getProperty("hostName");
```

The "port" and "hostName" Properties

RsProperties also extends Properties by adding fields that are relevant to RP1. The most prominent of these is the "port" value mentioned above. It also supplies the host specification for RP1 clients. The Server will always need the port specification. The clients will always need the host specification (to find the Server). All RP1 applications that use RsProperties read the rossum.ini file and, except where you override the specifications (discussed below), all RP1 applications will use the port value assigned in rossum.ini. This feature makes it possible to reassign the port value for the entire family of RP1 applications by modifying a single file.

You may modify the rossum.ini file as you see fit, but exercise caution in doing so. The Java Properties syntax is very fussy. It is case-sensitive and does not tolerate embedded white space characters. All specifications must be completed in a single line.

Overriding Properties

The rossum.ini file also includes specifications for logging. Typically, these are overridden by loading data from additional .ini files. For example, suppose a client application wishes to use the specifications from a data file. It could do so by invoking either the load() method from the Java Properties class, or the loadFromFile() method defined by RsProperties:

```
RsProperties rsp = new RsProperties();

rsp.loadFromFile("client.ini");
```

The loadFromFile method looks for files from either the current working directory or, if specified, from a fully qualified file pathname. The example

above was simplified a bit for purposes of clarity. Consider what would happen if the "client.ini" properties file was not found or if it contained syntax errors. Java would detect the error and *throw an exception.* In order for the example code to compile, Java requires that it *catch* the exception from the loadFromFile() method:

```
RsProperties rsp = new RsProperties();
try{
rsp.loadFromFile("client.ini");
}catch(RsPropertiesException e){
System.err.println("Error reading client.ini "+e.toString();
}
```

Loading RsProperties Files as a Resource

The loadFromFile() example above reads properties from a file found in the "file path." The default file path is whatever directory you happen to be in when you launch Java. An alternate file path may be specified through the argument to loadFromFile().

You may find it convenient to keep your properties files bundled up with your class files as a *resource* in the same manner as the rossum.ini file. You can do so by using the RsProperties loadFromResource() method. Pass the method an object belonging to a class defined in the same directory as your properties file. RsProperties will use that object to obtain the path to the properties file.

```
Example object  = new Example();

// an object is created
RsProperties rsp = new RsProperties();
rsp.loadFromResource(object,"client.ini");
//throws an RsPropertiesException
```

In another variation, the example below creates extends RsProperties and load the specifications as part of the constructor:

```
public class ExampleProperties extends RsProperties {
public ExampleProperties()   // the constructor
{
super();// RsProperties is the "super class"
// super(); invokes its constructor
try {loadFromResource(this, "client.ini");
}catch(RsPropertiesException e){
```

```
// handle this according to your own needs}}
}
```

The Server

The main Java application for Rossum's Playhouse is named Server. The RP1 Server is a multi-threaded application that manages both the simulation functions and client communications. The server has an optional GUI that depicts the robot simulation as it navigates its virtual landscape. In Revision 0.4x, the GUI is quite primitive and has potential for considerable refinement in future versions.

Server Properties Files

Upon start-up, the server reads two Java-style property files. The first is the "rossum.ini" file that was discussed above. From this file, it obtains the port specification for accepting TCP/IP connections (the host specification is irrelevant to the server). It then reads the file "server.ini" which supplies server-specific settings.

Accepting Clients

The simulator accepts clients via TCP/IP connections as described above. By design, the system should be able to accept multiple clients. At present, it can accept only one at a time. This limitation is due to certain key elements that are, as yet, unimplemented. Once a client disconnects from the server, the server is immediately ready to accept new connections.

Normally, the simulator should be ready to accept connections a few seconds after start-up. On earlier releases of Windows95, there appears to be delays of up to a minute related to problems with windows establishing the ability of the server to establish the port for client connections.

Internal Architecture

The Threads

One of the strengths of the Java language is the ease with which it can be used to code multithreaded applications. In the RP1 simulator, the following threads are implemented:

Scheduler Thread

The scheduler thread is the main simulation thread. This thread creates the simulator's virtual clock and also performs all the computation and modeling required to drive the simulation

AWT Thread

The AWT thread is launched by Java's graphics environment, the Abstract Window Toolkit (AWT). The implementation makes a very straightforward use of the Java tools.

Client-Listener Thread

The client-listener thread accepts connections from client applications, creates client objects, and launches new client monitor threads. Optionally, you may disable this thread if you are using Dynamically Loaded Clients.

Client Threads

The Client Thread services *incoming* communications from clients. Each client is assigned a unique thread. When network connections are enabled, Client Threads are launched by the Client-Listener Thread when a new client connection is accepted. When Dynamically Loaded Clients (DLC's) are used, Client Threads are launched by the Scheduler Thread shortly after start-up.[5]

The Scheduler

The scheduler is the heart of the simulator. Essentially, the scheduler implements a task queue not much different from a classic printer queue or batch-processing queue. In the simulator, events are treated as tasks and kept in a list sorted by time. The tasks are processed serially based on their time values. Time is treated in discrete intervals with a resolution of one millisecond. Consider the following list which shows the potential state of the queue at some time.

time 0.000 start robot motion for 10 seconds
time 0.100 evaluate robot position and disposition
time 0.200 evaluate

[5] Dynamically Loaded Clients are launched by a task, which is added to the scheduler shortly after start-up. Future revisions of RP1 will feature a browser-style interface that will allow the user to launch a DLC interactively. The class SimClientLauncherTask was designed with this use in mind.

time 0.300 evaluate
time 0.301 halt-motion due to collision with wall

At time 0.000 the client requested a robot motion with a duration of 10 seconds. The simulator determined that after just 0.301 seconds, the motion would result in a wall collision, and so scheduled a halt task at that time. It then queued up 5 tasks as shown in the table. The spacing of 1/10th second between tasks reflects the default SimulationFrameRate of 10 frames per second (see above).

As tasks are performed in sequence, the internal clock is adjusted to the time of the task. In most cases, individual tasks require far less time than the interval allotted. To simulate real-time behavior, the scheduler often introduces waits during which the scheduler is idle and other processes can be completed. In situations where the simulator is instructed to run at faster than real time, the waits are shortened or removed entirely.

At each evaluation, the simulator has the potential to send back information to the client (based on the results of the evaluation). Suppose that at 0.200 seconds, a proximity sensor detected the wall and the client sent a motion-halt request to the simulator. The client thread would create a task to perform a motion-halt operation and would place it at the *head* of the queue in a priority mode. When the halt task was serviced, all remaining motion tasks would be removed from the queue causing the robot simulacrum to halt.

The description of the task queue is algorithmic rather than practical. Certain real-world considerations complicate it somewhat. For example, at the default SimulationFrameRate, a one-minute motion requires 600 tasks. Queuing up so many tasks is not a good use of memory or processor cycles. This is especially true when the sequence might be cut short by a client request. So, typically, the simulator queues up only one motion task at a time. Motion task objects have the ability to "recycle" themselves so that they are placed back on the queue (with an adjusted time value) when they complete.

The virtual clock is coupled to the passage of real time. If a client or some other process requests the "simulation time" while the scheduler is resting between tasks, the scheduler will consult the system clock to derive a reasonable value.

Finally, a word on the Java thread scheduling mechanism. Under Windows architectures, threads are allocated a 50-millisecond slice of time to perform various operations. This approach has consequences for applications such as the simulator that depend on timing considerations. For example, consider the case where the code execute two requests for the system time separated by a 5-millisecond wait:

```
long time0, time1, deltaValue;
time0 = System.currentTimeMillis();  // current time in milliseconds
wait(5);  // wait 5 milliseconds
time1 = System.currentTimeMillis();
deltaValue = time1-time0;
```

The delta value for the times will probably be close to 50 milliseconds, rather than the 5 milliseconds you might expect. The reason for this is that when the wait is executed, control may be transferred to another thread. Even though the 5-millisecond wait expires, as much as 50 milliseconds may pass before the scheduler returns control to the waiting thread.

Due to this scheduling mechanism, the simulator will often run at slower-than-real-time speeds when we chose SimulationFrameRates of greater than 10 Hertz. This is not necessarily a bad thing. It does produce a more accurate model and also washes out some of the timing considerations due to the overhead for client-server communications.

The Floor-Plan

One of the first things the server does on start-up is to load data for the simulated environment. This data is stored in files called "floor plans." Floor plans include data describing the physical layout of walls and other features. Floor-plan data is encoded in textfiles. At start-up, the Server consults the simulation property file to obtain the name of the desired file. It then reads and parses the data in that file to create a virtual landscape for modeling.

Syntax and Semantics

The specification of floor plans is based on an very simple grammar. The grammar supports two kinds of statements: specifications and declarations. Specifications are used to supply parameters such as what system of units is to be used for the floor plan or what geometry is to be used for a particular object. Declarations are used to create particular objects such as walls. A specification is a simple statement given in the form:

```
    specification:  parameter;
or
    specification:  parameter[1], ..., parameter[n];
```

```
while declarations have a more complex syntax
```

```
objectClass  objectName {
specification[1..n];
}
```

and may contain one or more specifications. Revision 0.4x supports three kinds of object declarations: walls, targets, and placements.

Specifications

The units specification can be either meters, feet, or kilometers. Internally, all units are converted to meters, but the system specified in the floor plan is used for display purposes. The caption specification is used for labeling the main application window (Java frame). No other general floor-plan specifications are supported at this time.

Walls

In RP1, walls are simple barriers. They are specified as a set of two endpoint coordinates and a wall thickness. Walls must include a geometry specification with 5 real-valued parameters:

x-coordinate of first endpoint
y-coordinate of first endpoint
x-coordinate of second endpoint
y-coordinate of second endpoint width

All geometry parameters are assumed to be in the specified units system (by default, meters). If a robot simulacrum collides with a wall, all motion will be halted and a RsMotionHalted event will be sent to the client indicating the cause of the halt. Robot simulacrums can detect walls using the range-sensor component included in the standard tool set.

Targets

Targets are meant to model point sources of light or infrared radiation. Essentially, they give the robot a point target that it can detect and identify. The robot component objects include a "target sensor" that can be detect targets and generate sensor events (see "Communication via Events and Requests" above). The target geometry is specified as

x-coordinate of target
y-coordinate of target
radius for surrounding circle (purely for human reference)

Other specifications include a label, the color, and the thickness (width) of the line used to draw the surrounding circle.

Building a Virtual Robot

Introduction

In the Rossum's Playhouse simulation environment, the way to build a virtual robot is to write a client application. This section tells you how to do it. The point of Rossum's Playhouse is to help the user to test logic that will eventually be integrated into an actual robot. Naturally, doing so requires some method of configuring the simulated robot so that its characteristics follow those of the actual hardware being studied. It does so by providing a tour of "ClientZero," one of the two demonstration client applications that was provided with the RP1 software distribution. ClientZero is a simple application that was written expressly for the purpose of providing an example for developers implementing a client.

Thinking about Client Design

Before we talk about the demonstration clients, we want to make an important point about client design in general. We think that the code for a well-designed client ought to have a life of its own, separate from the simulator. We've already mentioned our hope that the code that you test on the simulator can be ported to an actual platform with little modification. Even if your target is not actual hardware (perhaps you are writing a general-purpose navigation tool kit), you should strive to minimize dependencies on the simulator.[6] The more successful you are at doing this, the more useful your code will be to yourself and to other developers. Of course, the means of accomplishing this goal are not at all obvious. Certainly, there is no one "right" way to design a client application.

The Demonstration Clients

The two demonstration clients supplied with the RP1 software distribution are called ClientZero and DemoZero. ClientZero is a very simple application that performs three basic operations: it connects to the server, it supplies a body

[6] This may be the only time you ever hear a software vendor saying, "whatever else, don't lock yourself into our product."

definition, and then it interacts with events received from the simulator. It has no user interface, but does keep a log of its interactions with the server. All the source code for ClientZero is stored in the Java package clientzero (recall that a Java package corresponds directly to a system directory). DemoZero adds a graphical user interface (GUI) to ClientZero, but does not otherwise extend its functionality. In fact, the DemoZero client is built on the ClientZero code. It does not change the underlying logic of the "robot's brains." It simply adds features (the GUI) that are useful to the user running a simulation.

In doing so, it illustrates a point. You may add things to your simulator (client) code that do not have to go into your target robot. DemoZero piggybacks a GUI on top of the ClientZero control logic, but does not alter the ClientZero behavior. Actual clients might extend their robot logic by adding user controls, performance analysis, or better simulation logging. Remember, though, that an important goal of the simulator is to be able to reuse the code that you test in the simulation environment in an actual platform. The key to being able to do this is to separate those components that are specific to the simulator from those which go into both the RP1 client and the real-world robot. The DemoZero GUI does not interfere with the ClientZero logic; it simply provides the user with some auxiliary controls.

Life Cycle of the Demonstration Clients

Both demonstration clients follow the same life cycle:

1. Initialize its session with the server (establish network or local connection if necessary)
2. Send body plan to server
3. Register relevant event handlers
4. Enter an event-loop

In the first step, the RP1 client contacts the server and exchanges introductory information. To do so, a client running as an independent client must make a network/local connection. A dynamically loaded client simply accepts the I/O streams provided by the server. In the second and third steps, the client tells the server about the robot it wishes to model and then registers event handlers to manage the communications from the RP1 server (which come in the form of events). It is necessary to send the body plan before registering event handlers because many of the events that come from the server depend on components of the virtual robot.

Finally, the clients enter an event loop in which incoming communications are received, processed, and passed to the client-supplied event-handlers.

The Listing below shows source code for a simplified version of ClnMain.java. We will refer to it in the discussion that follows.

```
package clientzero;
import rossum.*;
import java.lang.*;
import java.io.*;


public class ClnMain extends RsClient implements RsRunnable
{
public static void main(String args[]) throws IOException
{
ClnMain c = new ClnMain();
c.initialize();//throw IOException if unable to reach server
c.run();
}

public ClnMain(){
super();
}

public void initialize() throws IOException {
// to connect to the server, invoke the initialize method

// RsClient, the super class (parent class) of ClnMain

super.initialize();
// create body and send its specification to server
body = ClientZero.build();
sendBodySpecification(body);
// register event handlers -----------------------

addMouseClickEventHandler(new
ClnMouseClickEventHandler(this));

addPositionEventHandler(new ClnPositionEventHandler(this));

addPlacementEventHandler(new
ClnPlacementEventHandler(this));
```

```
addMotionHaltedEventHandler(new
ClnMotionHaltedEventHandler(this));

addTargetSensorEventHandler((RsBodyTargetSensor)(body.getPar
tByName("head")),new ClnTargetSensorEventHandler(this));

addTargetSelectionEventHandler(new
ClnTargetSelectionEventHandler(this));

addTimeoutEventHandler(new ClnTimeoutEventHandler(this));

addPlanEventHandler(new ClnPlanEventHandler(this));

// send a request for a floor plan (used to make a
navigation network)

sendPlanRequest();

// request that the robot be placed on the floor.

sendPlacementRequest("home");

}
```

Listing 4. Source Code for ClnMain (modified for clarity)

How ClnMain Extends RsClient and Implements RsRunnable

When you examine the class definition for ClnMain, note the statement *class ClnMain extends RsClient implements RsRunnable.* The ability of one class to extend another is a hallmark of object-oriented programming languages. ClnMain extends a class RsClient, which is defined in the java package *rossum.* In doing so, it inherits all the capabilities of RsClient.

So what is RsClient? The RsClient class provides an RP1 client with its interface to the simulator environment. In includes I/O channels, methods for encoding and decoding communications, registering event-handlers, and running

the client side of the simulator session. Without some instance of an RsClient object, it would be quite difficult for a client to communicate with the server.[7]

And what about RsRunnable? RsRunnable is an interface. In Java, an interface is a way of specifying that a class will implement a certain set of methods (functions). If a class statement says that it will implement a particular interface, we know that those methods will be available for any instance of that class. Armed with this knowledge, we can use any object of that class, or any object of any other class that implements that interface, interchangeably.[8]

For example, the Java standard Runnable interface implements a method called run(). Because Java knows that any class that implements Runnable includes the run method, Java knows that such a class can be launched as a separate execution thread. RsRunnable is simply an extension of the Java Runnable class. In addition to run(), RsRunnable specifies a few other methods that permit any class that implements it to be treated as a dynamically loaded client.

RsClient implements RsRunnable. It was not strictly necessary to include the reference to RsRunnable in the declaration for ClnMain. When ClnMain inherited the properties of its super class, RsClient, it also inherited any of the associated interfaces. We included the "implements RsRunnable" in the declaration for clarity.

The RsRunnable Interface

To load a class as a dynamically loaded client, the RP1 simulator requires that the class implement four methods. At run time, these methods are invoked in the sequence shown in the table below:

setInputOutputStream	allows the server to bypass the network (socket) based communications by supplying I/O
setLogger	allows the server to supply log-keeping

[7] The alternative to an RsClient implementation would be to study the RP1 protocol and duplicate a lot of its functions. If you are working in Java, there is no sensible reason to do that. If you are working in a language other than Java, it's you, only alternative at this time (please let us know if you'd like to work on an API in an alternate language).

[8] If one of your class definitions specifies that it implements a certain methods, but you forget to include the method in the class, it will not compile. Thus the interface ensures that a class does, in fact, implement all the methods it claims. Of course, determining whether your methods actually work is beyond the scope of the compiler.

	information to the client
initialize	tells the client to perform its main initializations; if I/O is not set, the client should establish a connection to the server
run	from Java's Runnable interface, allows the client to be run as a separate thread; this method is the main run-loop for the client.

The RP1 server always invokes initialize and run, but can be configured to not invoke setInputOutputStream and setLogger. In addition to the methods specified by the RsRunnable interface, RP1 also has a requirement about the constructor:

constructor()	RP1 always invokes a constructor that takes *no arguments*; when you build a dynamically loaded client, make sure that you provide any necessary functionality within such a constructor.

The methods defined by the RsRunnable interface are coded as follows:

```
package rossum;
public interface RsRunnable extends Runnable {

public void setInputOutputStreams(InputStream input,
OutputStream output);

public void setLogger(RsLogInterface logger);

public void initialize() throws IOException;

public void run();
// actually inherited from java.lang.Runnable}
```

Building RsRunnable and RsClient into ClnMain

When we implement an RP1 client, we need to ensure that it includes an instance of the RsClient class so that it may communicate with the server. When we implemented ClnMain, we had two choices. ClnMain could include

RsClient object as an element, or it could inherit all the capabilities of RsClient by extending it.

In earlier version of the demonstration software, ClnMain actually did include an object of class RsClient. In the present implementation, ClnMain extends RsClient. This use of a derived-class follows one of the elegant conventions of object-oriented programming. By extending RsClient, ClnMain inherits all the methods and elements RsClient. Thus, ClnMain can use all the functions and capabilities of the parent class as if they were its own. The approach saves coding and provides a convenient framework for our client implementation.

So why didn't we implement ClnMain as a derived class of RsClient in earlier versions of the demonstration? Well, in earlier versions, we were concerned that by deriving ClnMain from RsClient, the example code would give a false emphasis to the importance of the Rossum environment in building robot software. We thought that if we implemented our main class so that it was "just a derived class of a Rossum fixture" that it would suggest that any client implementation is hard-wired into the RP1 environment. One of the recurring themes in this document is that your robots are more important than our simulation. In our own robotics work, we are careful to separate the code need to write so that we can work with the simulator from that which we need for our robots. Ideally, simulator-interface code like ClnMain should not contain any code related to the real problem of implementing a robot.

DemoMain Implements RsRunnable, But Does Not Extend RsClient

Earlier, we mentioned the DemoZero application that piggybacks a GUI onto ClientZero. Like ClnMain, the DemoMain class implements the RsRunnable interface. It does not, however, extend RsClient (nor does it extend ClnMain, though that would have been an elegant way to implement it). We wrote it as a nonderived class to illustrate the fact that extending RsClient is not the only way to implement a class that can run as a Dynamically Loaded Client. Instead of extending RsClient, DemoMain contains an object of type RsClient that it uses for its communication with the server. To satisfy the RsRunnable interface, it provides simple "pass-through" methods.

The Execution of ClnMain

Anyway, ClnMain inherits all the features of RsClient. RsClient itself implements RsRunnable. We included the statement "implements RsRunnable" in the declaration as a reminder that the interface was part of the class. Since

RsClient already includes all the methods in the RsRunnable interface, we don't have to bother with them except where we have special needs. The first method defined by the class ClnMain is a static main(). This method is a standard Java technique that allows you to launch ClnMain as an application by typing:

```
java clientzero.ClnMain
```

at the command line.[9] Referring to the code, note that the main() method creates an instance of an object of type ClnMain (by using the Java keyword "new"), and then invokes the initialize() method that ClnMain inherited from its super class RsClient. The RsClient.initialize() method is provided as a convenient way of initializing a connection with the server.

When ClnMain is launched from the command line (or a script, or a Windows .bat file), it runs as a separate program from the RP1 simulator. Java invokes the main() method, which in turn invokes the initialize() method that ClnMain inherited from RsClient. The RsClient.initialize() method recognizes that it does not have a connection to the server and sets out to establish one. If it fails (as when the RP1 server is not running), it will throw an IOException, in this case terminating the program. RsClient.initialize() also loads the properties file for ClnMain (which is included in the clientzero package). If the properties files specifies that it should do so, initialize() opens up a log file.

After the initialize() method is completed, ClnMain.main() invokes run(). The run method, which is inherited from RsClient() transfer control to an event-loop, is which the client receives communications (events) from the server and uses them to invoke event-handlers. Control does not return from the event-loop until the connection with the server fails (as when the server shuts down) or when the application is terminated. *Applets*, which are launched from a Browser, behave slightly differently than *Applications,* which are launched directly from the system or command line.

When ClnMain is treated as a dynamically loaded client, the main method is not used. Instead, RP1 loads the ClnMain class and *creates an instance* of that class (creates an object). It then uses that object to invoke the methods that were specified in the RsRunnable interface. The first of these, setInputOutputStreams(), tells the client object that it does not need to establish I/O. The second, setLogger(), suppresses the usual logging for the client and

---

[9] This may be the only time you hear a software vendor saying "don't lock yourself into our product."

redirects it to the main simulator log.[10] Once these preliminaries are complete, it invokes the object's initialize() method. The initialize() method that ClnMain inherited from RsClient is smart enough to check whether I/O and logging have been specified and to use them rather than attempting to set up its own functions.

Uploading the Body Plan

Once the connection is established, the next thing the client needs to do is to give the server a physical description of kind of robot it will be modeling. It does so by creating and uploading a body plan. For demo purposes, the body plan is created using a static method in the class ClientZero.

Registering Event Handlers

The RsClient class, which is used by ClnMain, allows applications to run in an "event-loop," which allows them to communicate with the server. Event loops are a very common feature in many graphics environments. Developers who have had experience with Unix and X-windows will recognize the origins of the RP1 concept immediately. Those who have worked with the Java AWT will see a similar parallel with the ideas of "event listeners" and the Java graphics thread (which is a kind of event-loop).

Of course, a robot control system is *not* the same thing as a computer graphics application. But the event-loop concept used in many graphics applications turns out to be quite adaptable to other purposes. As you examine the code for clientzero, you will see how this is accomplished.

To use the event-loop, clients register "event handlers," which are methods (or "functions") that are invoked when the client receives an event (message) from the server. Once ClnMain invokes the RsClient.run() method, it enters an endless loop in which RsClient receives messages from the simulator/server and invokes the methods that were registered by the client application. Recall that the client communicates to the server by sending it "request" messages and that the server responds by sending the client "event" messages. An event might be something like "the robot hit a wall," or "the robot's sensor detected a light source," or "a timer expired." When these event messages are received, any registered event handlers are invoked

---

[10] Both setInputOutputStreams() and setLogger() can be suppressed using configuration elements in the RP1.ini properties file. This feature is intended for testing purposes (so that a dynamically loaded client can exercise its ability to establish a network connection, or so that it may be configured to keep an independent log file).

Multiple Event Handlers and the RsEvent.consume() Method

When multiple handlers are registered for a particular event, they are invoked in the order in which they were registered. Sometimes, it is useful for one event handler to be able to prevent the event from being passed on to other event handlers. It may do so by invoking the consume() method of the current event.

Running the Event Loop

Once the ClnMain.initialize() method registers event handlers, it sends the RP1 simulator a request for a placement on the floor plan. Up until the time the simulator receives the request, the robot is not visible on the display. Once RP1 receives and processes the request, it places the robot simulacrum on the floor plan and issues a placement event.

ClnMain will not process the incomming events until it enters its event-loop. The event-loop is embodied in the RsClient.run() method. Since ClnMain extends RsClient, it inherits that method. The run() method implements an endless loop in which incoming events are processed and the corresponding event handlers are invoked. The choice of the name "run()" for the event-loop method in RsClient is not an accident. It allows client applications to take advantage of Java's multithreading (multitasking) feature.

How the Demo Clients Work

The demonstration clients take advantage of a feature of the simulator, which has no counterpart in the world or real robotics: the mouse click event. For demonstration purposes, the mouse click is used to tell the robot where to go. When you point the mouse in front of the robot and click the left button, the robot will move toward the position you indicate. If you point the mouse somewhere else, and click the right button, the robot will turn to face the indicated direction. Referring back to the code for ClnMain, note that the first event-handler statement

```
addMouseClickEventHandler(new
ClnMouseClickEventHandler(this));
```

establishes a mouse-click event handler. ClnMouseClickEventHandler extends the class RsMouseClickHandler, which is included in the rossum package. When you click the mouse button, the Server sends an event message to any clients that have registered event handlers of mouse clicks. ClientZero responds

by performing some simple navigation computations and moving toward the click point.

ClientZero is not especially smart. It will go where you direct it and eventually may crash into a wall. When this happens, its motion will be halted and the server will issue an RsMotionHaltedEvent. Motion-halted events are handled by ClnMotionHaltedEventHandler, which extends RsMotion-HaltedEventHandler.

```
addMotionHaltedEventHandler(new
ClnMotionHaltedEventHandler(this));
```

As you navigate the robot simulacrum, you may eventually read a point where it detects a target. When that happens, the Server will send an RsTargetSensorEvent to registered clients. The specification for the target sensor event is a little more complicated than other event handlers. Because a robot can have multiple sensors, you can register multiple sensor-event handlers. To do so, you have to be able to tell the simulator which sensor is associated with which handler.

The RP1 simulator allows you to name body parts when you create them. In ClientZero, we assigned the name "head" to the forward-looking target sensor on the robot. So when we register a target sensor for the robot, we included it in the specifications.

```
addTargetSensorEventHandler((RsBodyTargetSensor)(getBody().g
etPartByName("head")),ClnTargetSensorEventHandler(this));
```

Two other event handlers are established by ClnMain.initialize(): a position-event handler and a placement-event handler. We'll talk about the placement handler first. The placement handler is used to establish a starting position for the robot when we run a simulation. The event is generated on request, and only by request. Note that the last statement in ClnMain.initialize() is:

```
sendPlacementRequest("home");
```

When the Server receives a placement request, it cancels all robot motions and moves it to the named placement. The placement also specifies orientation for the robot. The example floor plan contains only a single placement feature, though any number can be included. Upon placement, an RsPlacementEvent is sent to the client. The placement event contains fields giving the position and orientation for the robot.

Physical Layout of ClientZero

The body plan for ClientZero is established using a method defined in the Java class file "ClientZero.java" which is part of the clientzero package. Referring back to the code for ClnMain.java in listing 2, note the statement

```
RsBody body = ClientZero.build();
```

It produces a simple robot body plan that appears as shown in Figure D.1 below.

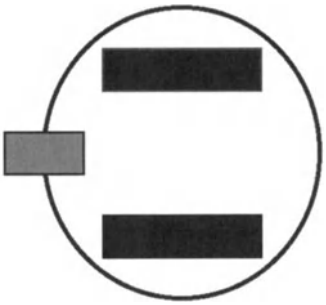

FIGURE D.1 ClientZero body plan.

The first rectangular element on the front of the robot is a target sensor. The dark rectangles within its body are wheels. The design for ClientZero is deliberately simple and cartoonlike. In particular, the wheels are much thicker than might be found on an actual robot.[11] It is possible to code more realistic depictions.

The RsBody and RsBodyPart Classes

Note that the ClientZero.build() method returns an object of type RsBody. RsBody objects act as containers, which hold objects of classes, derived from RsBodyPart. There are a variety of different kinds of body parts, including sensors, wheel actuators, and shapes corresponding to physical components

[11] Thicker wheels provide better traction and stability, but tend to degrade accuracy when computing position during turns if you are using wheel position for dead-reckoning.

(such as chassis, housings, etc.). These are added to the body using calls to the RsBody.addPart() method.

Body parts are rendered in the order they are added to the body. So if two parts overlap, the last one added will be drawn on top of the first. The geometry of the body parts is specified using a Cartesian coordinate system with the origin at a reference point defined as "the center of the robot body." The robot's "forward vector" is defined as the x-axis. When the wheel actuators are added to the robot body, their axle is assumed to be centered on the origin and aligned with the y-axis.

By default, both the line and fill color values are defined to be Color.lightGray. You may adjust this value by using the methods

```
RsBodyPart.setFillColor(Color c);
// Color from java.awt.Color class;

RsBodyPart.setLineColor(Color c);
```

If you prefer not to render a particular feature you may suppress them by using the setFillColor() and setLineColor() methods to assign null color values. Note that the specification of a null color does not affect the interactive characteristics of a body part. Recall that in Java, the definition of classes is hierarchical. Both the RsBody and RsBodyPart classes are derived from the RsComponent class.[12]

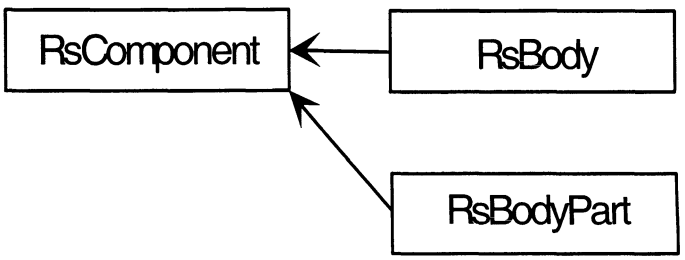

FIGURE D.2 Inheritence for RsBody and RsBodyPart classes.

[12] The arrow notation is used in class diagrams to show inheritance. Remember that a class always "points to the class from which it was derived."

The RsComponent class is of interest only from a Java programming point of view. It provides a method for *cloning* (duplicating) objects which is inherited by RsBody, RsBodyPart, and all classes derived from RsBodyPart. The clone() methods are used internally by the RP1 simulator.

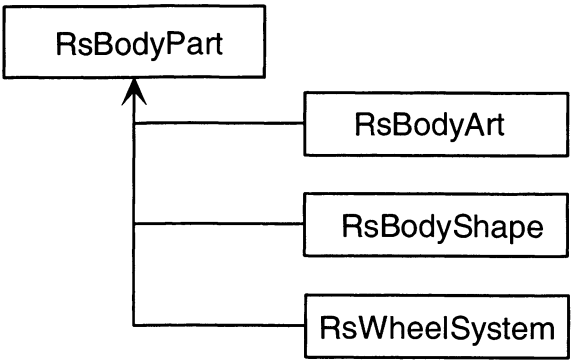

FIGURE D.3 Subclasses of RsBodyPart.

The RsComponent class also implements two Java interfaces, *Cloneable* and *Serializable*. These interfaces are important to Java coding. If they are unfamiliar to you, then you need not be concerned with them at this time. The RsBodyPart class is super class to three major classes as shown in Figure D.3.

RsBodyShape

The RsBodyShape class is the super class for all physically interactive body parts[13]. Most of the visible components of the robot are derived from RsBodyShape. These include all passive components, such as the chassis or any housing elements, as well as active components such as sensors. All objects of classes derived from RsBodyShape will interact with walls, resulting in collision events if contact occurs.

The RsBodyShape allows you to create an arbitrary simple polygon for depiction and modeling (the polygon topologically simple, that is to say "not self-intersecting"). Because the RsBodyShape class derives from RsBodyPart, you may specify both a line and a fill color for depiction purposes.

---

[13] The RsBodyShape class contains a number of vestigial elements that are left over from an earlier implementation and which will be removed from the code before the 1.0 release of the system.

One popular design element for smaller robots such as the Rug Warrior is a circular body plan. The RsBodyCircle class extends RsBodyShape by providing a constructor that allows you to specify the body part using a center and radius rather than a multipoint polygon. RsBodyCircle also overrides the standard paint method defined in RsBodyShape and invokes Java's circle-drawing calls directly in order to create a more pleasing depiction of a circle.

RsWheelSystem

The RsWheelSystem class is used to model the behavior the wheel actuator for the simulated robot. In the present implementation, only one kind of actuator is available: the classic *differential steering* system.

In a differential steering system, two independently controlled and powered wheels are mounted perpendicular to a common axis. A third wheel, usually a caster, is provided for support.[14] Steering is accomplished by varying the speeds of the wheels. When both wheels turn at the same speed, the robot travels straight. When the right wheel turns faster than the left, the robot steers toward the left, and vice versa. When the wheels turn at different, but fixed, speeds, the robot steers a circular path. If the two wheels turn at the same speed but in opposite directions, the robot can execute a pivot. The constructors for the RsWheelSystem class can be found in RsWheelSystem.java.

```
public class RsWheelSystem extends RsBodyPart
{
// constructors
public RsWheelSystem(
double wheelBase,
double radius,
double thickness,
double nStepPerRevolution,
double maxStepsPerSecond);
}
```

Odometry

Many robots depend, at least in part, on odometry to perform navigation. By measuring, or estimating, the distance each wheel turns, it is possible to dead-

[14] In larger, heavier robots, more than one additional wheel may be supplied. Some small robots actually use a skid or even a simple dowel with a rounded end for support.

reckon the position and orientation of the robot. Dead-reckoning is tricky. Small errors can degrade the calculation. Estimations of the robot's orientation are especially prone to error. Some developers replace the simple caster mentioned above with a steering wheel. By reinforcing the steering provided by the drive wheels, errors in orientation computations can be substantially reduced.

At present, the models provided by RP1 assume that the wheels turn at fixed, but independent speeds. For example, you can specify that the left wheel turn at 20 rpm while the right turns at 30 rpm over a period of 30 seconds. Changes in speed are treated as instantaneous. Acceleration is not modeled.[15]

Wheel and Path Calculations

Clearly, the path followed by the robot is dependent on the geometry of the wheels (wheel base, tire radius, etc.) and the velocity at which they turn. These specifications are contained in objects of the RsWheelSystem class. So it is not surprising that RsWheelSystem also provides methods for navigation and specifying robot control requests. Two useful methods for performing computations are

```
public RsMotionRequest getMotionRequest(
boolean useStepMethod,double x, double y, double speed);

public RsMotionRequest getMotionRequestForPivot(
boolean useStepMethod, double x, double y, double speed);
```

These methods are general objects of the RsMotionRequest class. Request classes are discussed below. RP1 clients use the motion-request class to instruct the robot simulacrum to rotate its wheels. The getMotionRequest() method returns a RsMotionRequest that will put the robot on a path to the specified (x,y) coordinates. The getMotionRequestForPivot() returns a request that instructs the robot to face the specified coordiantes.

The coordinates (x,y) in these method calls are given relative to the robots current position and orientation. They refer to the "center point" of the robot that is defined when its body is specified. So a coordinate of $(x,y) = (1.0, 0.0)$ would indicate a point 1 meter directly in front of the robot's center point. A coordinate of $(x,y) = (0.707, 0.707)$ indicates a point 45 degrees to the front and

---

[15] An improved motion model, which does account for acceleration, is a high priority in our development plan.

left of the robot. Note that a coordinate specification of (-1.0, 0) would result in the robot backing up.

The Sensor Classes

The RP1 simulator does not attempt to model specific sensors. In the real world, there are a vast number of different kinds of sensory apparatus, with many different operating parameters. An attempt to model even a fraction of the devices available would be futile. Instead, RP1 provides models for highly abstract sensors. Client applications can use the data provided by the abstract sensors as inputs into their own models. This approach puts the ability to simulate the behavior of real sensors into the hands of the client developer (who probably knows more about the specifics of his system than some guy writing a simulator would anyhow). The abstract sensors include the following:

Target sensor	a sensor that detects point objects, such as visible light or infrared sources
Contact sensor	a "bumper" sensor that detects physical contact with walls
Range sensor	a sensor that detects the range to walls, the resolution of this sensor can be adjusted so that it behaves as a "proximity sensor"

The sensors classes all derive from the RsBodySensor class, which in turn derives from the RsBodyShapeClass. RsBodySensor also adds the notion of a "state" to the body part. Sensors are said to be either "hot" or "cold" depending on whether they have a detection or not. If you run either of the ClientZero demo applications, you will note that the sensor on the front of the robot changes from blue to bright orange when it "goes hot" (encounters a detection). The RsBodySensor class adds two color-related methods to the RsBodyPart class:

```
RsBodySensor.setHotFillColor(Color c);
RsBodySensor.setHotLineColor(Color c);
```

If you examine the code for ClientZero.java, you will note that these methods were used as part of the sensor definition.

Obtaining State Data of a Sensor

Each of the sensor classes provided by RP1 has a number unique parameters describing its state. The particulars of these parameters are described below in

the paragraphs dealing with individual sensor classess. RP1 communicates these values to a client application in the form of a "sensor event."

RsBodyTargetSensor

The RsBodyTargetSensor, which was mentioned above, is used to detect objects of the RsTarget class. RsTarget objects are single-point features that can be added to the simulator's floor plan. They can be used to model visible light or infrared sources.

As noted above, when a sensor changes state, the simulator generates a sensor event. Client applications can apply their own enhanced models to the data in the sensor event. For example, if you were measuring an IR source with a real-world device, you might expect a drop in sensitivity as the source moved away from the detector axis. A dim source, which could be detected at a range of one meter when the sensor was pointed directly at the target, might not register when the sensor was turned slightly. Your client application could add this level of enhanced modeling based on specific knowledge of the devices you were using.

RsBodyContactSensor

The RsBodyContactSensor is used to model pressure-sensitive contact sensors. Typically contact sensors are mounted on bumpers on the outermost edges of a robot chassis. The RsBodyContactSensor has two states: detection and no-detection. When a robot simulacrum begins a motion, any contact sensors are set to the no-detection state. If any contact sensors were previously activated, and thus undergo a state change, a RsContactSensorEvent will be issued with its status field set to false (no-detection).[16] If a collision occurs at the end of a motion, contact-sensor events will be issued for any sensors that are activated. The constructor for a contact sensor is shown below:

```
public class RsBodyContactSensor extends RsBodySensor {
public RsBodyContactSensor(double []point,int nPoint)
```

As you can see, the definition of a contact sensor is a simple closed polygon.

---

[16] Not all motion requests will result in a contact-sensor status change. For example, if the robot has driven into a wall, a request for a forward motion will just press the robot against the wall harder. No movement will occur, so no sensor state change will result.

Events and Requests

In the Rossum's Playhouse environment, an "event" means just what the word suggests. It indicates that an action has occurred. The RP1 simulator generates an event when a robot collides with a wall, when a sensor has a detection, or when the robot begins a movement sequence.

A "request" is the way the client application transmits instructions to the robot simulacrum. The term "request" is used rather than a stronger term such as "instruction" in recognition of the fact that a real-world robot does not always do what the controlling software tells it to do. No matter how much you spin the wheels, the robot won't move if jammed into a corner.

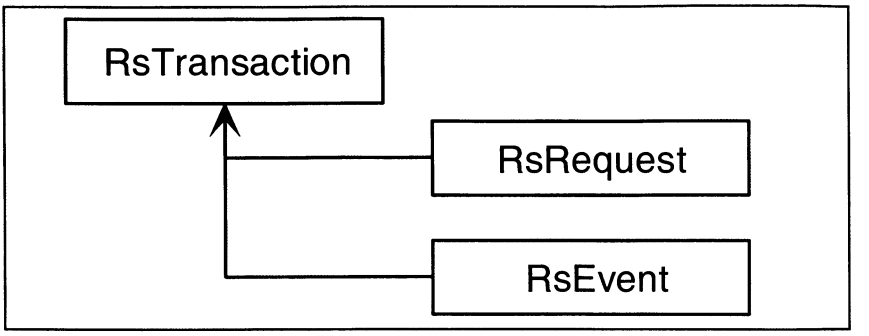

FIGURE D.4 Class inheritence for RP1 messages.

After a client connects to the server and exchanges preliminary specifications (such as uploading the body plan), all communication between the two applications is conducted via messages called "events" (server-to-client messages) and "requests" (client-to-server messages). Recall that the fundamental entity in Java is the "class." The class is a collection of data similar to the C programming languages structure. All messages exchanged between the client and the server are expressed in the form of Java classes of the type RsEvent and RsRequest, both of which derive from the class RsTransaction.[17]

The primary function of RsEvent and RsRequest is to serve as base classes for the various message classes that are derived from them. The sole purpose of RsTransaction is to server as a base class for RsEvent and RsRequest. This hierarchy of definitions creates common relationships between the classes

[17] The choice of the word "transaction" to name the RsTransaction class was a bit of bad luck. In Revision 0.5, we may change this to the more accurate "RsMessage."

allowing them to be treated interchangeably by a number of shared methods and data structures. A method (function) that can process an object of class RsTransaction can process objects of any of the classes derived from RsTransaction. A method that can process an object of class RsEvent can process any of the RsEvent objects, etc.

Index