ARTIFICIAL INTELLIGENCE

Zbigniew Michalewicz

Genetic Algorithms + Data Structures = Evolution Programs

Springer-Verlag Berlin Heidelberg GmbH

Artificial Intelligence

Zbigniew Michalewicz

Genetic Algorithms + Data Structures = Evolution Programs

With 48 Figures

Springer-Verlag
Berlin Heidelberg GmbH

Zbigniew Michalewicz
Department of Computer Science
University of North Carolina
Charlotte, NC 28223, USA

ISBN 978-3-662-02832-2

Library of Congress Cataloging-in-Publication Data
Michalewicz, Zbigniew. Genetic algorithms + data structures = evolution programs / Zbigniew Michalewicz. p. cm. – (Artificial intelligence) Includes bibliographical references and index.
ISBN 978-3-662-02832-2 ISBN 978-3-662-02830-8 (eBook)
DOI 10.1007/978-3-662-02830-8
Computer algorithms. 2. Data structures
(Computer science) 3. Computer programs. I. Title II. Series.
QA76.9.A43M53 1992 005.1-dc20 92-19925

Originally published by Springer-Verlag Berlin Heidelberg New York in 1992
Softcover reprint of the hardcover 1st edition 1992

Typesetting: Camera ready by author
45/3140 - 5 4 3 2 1 0 - Printed on acid-free paper

To the next generation:
Matthew, Katherine, Michael,
Thomas, and Irene

Preface

'What does your Master teach?'
asked a visitor.
'Nothing,' said the disciple.
'Then why does he give discourses?'
'He only points the way — he teaches
nothing.'

Anthony de Mello, *One Minute Wisdom*

During the last three decades there has been a growing interest in algorithms which rely on analogies to natural processes. The emergence of massively parallel computers made these algorithms of practical interest. The best known algorithms in this class include evolutionary programming, genetic algorithms, evolution strategies, simulated annealing, classifier systems, and neural networks. Recently (1–3 October 1990) the University of Dortmund, Germany, hosted the First Workshop on Parallel Problem Solving from Nature [164].

This book discusses a subclass of these algorithms — those which are based on the principle of evolution (survival of the fittest). In such algorithms a population of individuals (potential solutions) undergoes a sequence of unary (mutation type) and higher order (crossover type) transformations. These individuals strive for survival: a selection scheme, biased towards fitter individuals, selects the next generation. After some number of generations, the program converges — the best individual hopefully represents the optimum solution.

There are many different algorithms in this category. To underline the similarities between them we use the common term "evolution programs".

Evolution programs can be perceived as a generalization of genetic algorithms. Classical genetic algorithms operate on fixed-length binary strings, which need not be the case for evolution programs. Also, evolution programs usually incorporate a variety of "genetic" operators, whereas classical genetic algorithms use binary crossover and mutation.

The beginnings of genetic algorithms can be traced back to the early 1950s when several biologists used computers for simulations of biological systems [72]. However, the work done in late 1960s and early 1970s at the University of Michigan under the direction of John Holland led to genetic algorithms as they are known today. The interest in genetic algorithms is growing rapidly — the recent Fourth International Conference on Genetic Algorithms (San Diego, 13–16 July 1991) attracted around 300 participants.

The book describes the results of three years' research from early 1989 at Victoria University of Wellington, New Zealand, through late 1991 (since July 1989 I have been at the University of North Carolina at Charlotte). During this time I built and experimented with various modifications of genetic algorithms

using different data structures for chromosome representation of individuals, and various "genetic" operators which operated on them. Because of my background in databases [122], [126], where the constraints play a central role, most evolution programs were developed for constrained problems.

The idea of evolution programs (in the sense presented in this book), was conceived quite early [101], [133] and was supported later by a series of experiments. Despite the fact that evolution programs, in general, lack a strong theoretical background, the experimental results were more than encouraging: very often they performed much better than classical genetic algorithms, than commercial systems, and than other, best-known algorithms for a particular class of problems.

Some other researchers, at different stages of their research, performed experiments which were perfect examples of the "evolution programming" technique — some of them are discussed in this volume. Chapter 8 presents a survey of evolution strategies — a technique developed in Germany by I. Rechenberg and H.-P. Schwefel [149], [162] for parameter optimization problems. Many researchers investigated the properties of evolution systems for ordering problems, including the widely known, "traveling salesman problem" (Chapter 10). In Chapter 11 we present systems for a variety of problems including problems on graphs, scheduling, and partitioning. Chapter 12 describes the construction of an evolution program for inductive learning in attribute based spaces, developed by C. Janikow [97]. In the Conclusions, we briefly discuss evolution programs for generating LISP code to solve problems, developed by J. Koza [108], and present an idea for a new programming environment.

The book is organized as follows. The introduction provides a general discussion on the motivation and presents the main idea of the book. Since evolution programs are based on the principles of genetic algorithms, Part I of this book serves as survey on this topic. We explain what genetic algorithms are, how they work, and why (Chapters 1–3). The last chapter of Part I (Chapter 4) presents some selected issues (selection routines, scaling, etc.) for genetic algorithms.

In Part II we explore a single data structure: a vector in a floating point representation, only recently widely accepted in the GA community [38]. We talk only about numerical optimization. We present some experimental comparison of binary and floating point representations (Chapter 5) and discuss new "genetic" operators responsible for fine local tuning (Chapter 6). Chapter 7 presents two evolution programs to handle constrained problems: the GENOCOP system, for optimizing functions in the presence of linear constraints, and the GAFOC system, for optimal control problems. Various tests cases are considered; the results of the evolution programs are compared with a commercial system. The last chapter of this part (Chapter 8) presents a survey of evolution strategies and describes some other methods.

Part III discusses a collection of evolution programs built over recent years to explore their applicability to a variety of hard problems. We present further experiments with order–based evolution programs, evolution programs with ma-

trices and graphs as a chromosome structure. Also, we discuss an application of evolution program to machine learning, comparing it with other approaches.

The title of this book rephrases the famous expression used by N. Wirth fifteen years ago for his book, *Algorithms + Data Structures = Programs* [197]. Both books share a common idea. To build a successful program (in particular, an evolution program), appropriate data structures should be used (the data structures, in the case of the evolution program, correspond to the chromosome representation) together with appropriate algorithms (these correspond to "genetic" operators used for transforming one or more individual chromosomes).

The book is aimed at a large audience: graduate students, programmers, researchers, engineers, designers — everyone who faces challenging optimization problems. In particular, the book will be of interest to the Operations Research community, since many of the problems considered (traveling salesman, scheduling, transportation problems) are from their area of interest. An understanding of introductory college level mathematics and of basic concepts of programming is sufficient to follow all presented material.

Acknowledgements

It is a pleasure to acknowledge the assistance of several people and institutions in this effort.

Thanks are due to many co-authors who worked with me at different stages of this project; these are (in alphabetical order): Paul Elia, Lindsay Groves, Matthew Hobbs, Cezary Janikow, Andrzej Jankowski, Mohammed Kazemi, Jacek Krawczyk, Zbigniew Ras, Joseph Schell, David Seniv, Don Shoff, Jim Stevens, Tony Vignaux, and George Windholz.

I would like to thank all my graduate students who took part in a course *Genetic Algorithms* I offered at UNC–Charlotte in Fall 1990, and my Master students, who wrote their thesis on various modifications of genetic algorithms: Jason Foodman, Jay Livingstone, Jeffrey B. Rayfield, David Seniw, Jim Stevens, Charles Strickland, Swarnalatha Swaminathan, and Keith Wharton. Jim Stevens provided a short story on rabbits and foxes (Chapter 1).

Thanks are due to Abdollah Homaifar from North Carolina State University, Kenneth Messa from Loyola University (New Orleans), and to several colleagues at UNC–Charlotte: Mike Allen, Rick Lejk, Zbigniew Ras, Harold Reiter, Joe Schell, and Barry Wilkinson, for reviewing selected parts of the book.

I would like to acknowledge the assistance of Doug Gullett, Jerry Holt, and Dwayne McNeil (Computing Services, UNC–Charlotte) for recovering all chapters of the book (accidentally deleted). I would like also to thank Jan Thomas (Academic Computing Services, UNC–Charlotte) for overall help.

I would like to extend my thanks to the editors of the Artificial Intelligence Series, Springer-Verlag: Leonard Bolc, Donald Loveland, and Hans Wössner for initiating the idea of the book and their help throughout the project. Special thanks are due to J. Andrew Ross, the English Copy Editor, Springer-Verlag, for his precious assistance with writing style.

I would like to acknowledge a series of grants from North Carolina Supercomputing Center (1990–1991), which allowed me to run the hundreds of experiments described in this text.

In the book I have included many citations from other published work, as well as parts of my own published articles. Therefore, I would like to acknowledge all permissions to use such material in this publication I received from the following publishers: Pitman Publishing Company; Birkhäuser Verlag AG; John Wiley and Sons, Ltd; Complex Systems Publications, Inc.; Addison–Wesley Publishing Company; Kluwer Academic Publishers; Chapman and Hall Ltd, Scientific, Technical and Medical Publishers; Prentice Hall; IEEE; Association for Computing Machinery; Morgan Kaufmann Publishers, Inc.; and individuals: David Goldberg, Fred Glover, John Grefenstette, Abdollah Homaifar, and John Koza.

Finally, I would like to thank my family for their patience and support during the (long) summer of 1991.

Charlotte
January 1992

Zbigniew Michalewicz

Table of Contents

Introduction

Again I saw that under the sun
the race is not to the swift,
nor the battle to the strong,
nor bread to the wise,
nor riches to the intelligent,
nor favor to the man of skill;
but time and chance
happen to them all.

The Bible, Ecclesiastes, 9

During the last thirty years there has been a growing interest in problem solving systems based on principles of evolution and hereditary: such systems maintain a population of potential solutions, they have some selection process based on fitness of individuals, and some recombination operators. One type of such systems is a class of Evolution Strategies i.e., algorithms which imitate the principles of natural evolution for parameter optimization problems [149], [162] (Rechenberg, Schwefel). Fogel's Evolutionary Programming [57] is a technique for searching through a space of small finite-state machines. Glover's Scatter Search techniques [64] maintain a population of reference points and generate offspring by weighted linear combinations. Another type of evolution based systems are Holland's Genetic Algorithms (GAs) [89]. In 1990, Koza [108] proposed an evolution based system to search for the most fit computer program to solve a particular problem.

We use a common term, **Evolution Programs (EP)**, for all evolution-based systems (including systems described above). The structure of an evolution program is shown in Figure 0.1.

The evolution program is a probabilistic algorithm which maintains a population of individuals, $P(t) = \{x_1^t, \ldots, x_n^t\}$ for iteration t. Each individual represents a potential solution to the problem at hand, and, in any evolution program, is implemented as some (possibly complex) data structure S. Each solution x_i^t is evaluated to give some measure of its "fitness". Then, a new population (iteration $t+1$) is formed by selecting the more fit individuals (select step). Some members of the new population undergo transformations (recombine step) by means of "genetic" operators to form new solutions. There are unary transformations m_i (mutation type), which create new individuals by a small change in a single individual ($m_i : S \rightarrow S$), and higher order transformations c_j (crossover type), which create new individuals by combining parts from several (two or more) individuals ($c_j : S \times \ldots \times S \rightarrow S$). After some number of generations the program converges — the best individual hopefully represents the optimum solution.

```
procedure evolution program
begin
  t ← 0
  initialize P(t)
  evaluate P(t)
  while (not termination-condition) do
  begin
    t ← t + 1
    select P(t) from P(t − 1)
    recombine P(t)
    evaluate P(t)
  end
end
```

Fig. 0.1. The structure of an evolution program

Let us consider a general example. Assume we search for a graph which should satisfy some requirements (say, we search for the optimal topology of a communication network accordingly to some criteria: cost of sending messages, reliability, etc.). Each individual in the evolution program represents a potential solution to the problem, i.e., each individual represents a graph. The initial population of graphs $P(0)$ (either generated randomly or created as a result of some heuristic process) is a starting point ($t = 0$) for the evolution program. The evaluation function usually is given — it incorporates the problem requirements. The evaluation function returns the fitness of each graph, distinguishing between better and worse individuals. Several mutation operators can be designed which would transform a single graph. A few crossover operators can be considered which combine the structure of two (or more) graphs into one. Very often such operators incorporate the problem-specific knowledge. For example, if the graph we search for is connected and acyclic (i.e., it is a tree), a possible mutation operator may delete an edge from the graph and add a new edge to connect two disjoint subgraphs. The other possibility would be to design a problem-independent mutation and incorporate this requirement into the evaluation function, penalizing graphs which are not trees.

Clearly, many evolution programs can be formulated for a given problem. Such programs may differ in many ways; they can use different data structures for implementing a single individual, "genetic" operators for transforming individuals, methods for creating an initial population, methods for handling constraints of the problem, and parameters (population size, probabilities of applying different operators, etc.). However, they share a common principle: a population of individuals undergoes some transformations, and during this evolution process the individuals strive for survival.

The idea of evolution programming is not new and has been around for at least thirty years [57], [72], [162]. However, our concept of evolution programs is different from the previously proposed ones. It is based entirely on the idea of

genetic algorithms [89]; the difference is that we allow *any* data structure (i.e., chromosome representation) suitable for a problem together with *any* set of "genetic" operators, whereas genetic algorithms use fixed-length binary strings (as a chromosome, data structure S) for its individuals and two operators: binary mutation and binary crossover. In other words, the structure of a genetic algorithm is the same as the structure of an evolution program (Figure 0.1) and the differences are hidden on the lower level. Each chromosome need not be represented by a bit-string and the recombination process includes other "genetic" operators appropriate for the given structure and the given problem.

This is a relatively new direction. Only in 1985 De Jong wrote [43]:

> "What should one do when elements in the space to be searched are most naturally represented by more complex data structures such as arrays, trees, digraphs, etc. Should one attempt to 'linearize' them into a string representation or are there ways to creatively redefine crossover and mutation to work directly on such structures. I am unaware of any progress in this area."

As mentioned earlier, genetic algorithms use fixed-length binary strings and only two basic genetic operators. Two major (early) publications on genetic algorithms [89], [41] describe the theory and implementations of such GAs. As stated in [73]:

> "The contribution of this work [41] was in its ruthless abstraction and simplification; De Jong got somewhere not in spite of his simplification but because of it. [...] Holland's book [89] laid the theoretical foundation for De Jong's and all subsequent GA work by mathematically identifying the combined role in genetic search of similarity subsets (schemata), minimal operator disruption, and reproductive selection. [...] Subsequent researchers have tended to take the theoretical suggestions in [89] quite literally, thereby reinforcing the implementation success of De Jong's neat codings and operators."

However, in the next paragraph Goldberg [73] says:

> "It is interesting, if not ironic, that neither man intended for his work to be taken so literally. Although De Jong's implementations established usable technique in accordance with Holland's theoretical simplifications, subsequent researchers have tended to treat both accomplishments as inviolate gospel."

It seems that a "natural" representation of a potential solution for a given problem plus a family of applicable "genetic" operators might be quite useful in the approximation of solutions of many problems, and this nature-modeled approach (evolution programming) is a promising direction for problem solving in general. Already some researchers have explored the use of other representations as ordered lists (for bin-packing), embedded lists (for factory scheduling problems), variable-element lists (for semiconductor layout). During the last

ten years, various application–specific variations on the genetic algorithm were reported [33], [78], [82], [83], [134], [172], [173], [190]. These variations include variable length strings (including strings whose elements were *if–then–else* rules [172]), richer structures than binary strings (for example, matrices [190]), and experiments with modified genetic operators to meet the needs of particular applications [129]. In [141] there is a description of a genetic algorithm which uses backpropagation (a neural network training technique) as an operator, together with mutation and crossover that were tailored to the neural network domain. Davis and Coombs [29], [36] described a genetic algorithm that carried out one stage in the process of designing packet-switching communication network; the representation used was not binary and five "genetic" operators (knowledge based, statistical, numerical) were used. These operators were quite different to binary mutation and crossover. Other researchers, in their study on solving a job shop scheduling problem [9], wrote:

> "To enhance the performance of the algorithm and to expand the search space, a chromosome representation which stores problem specific information is devised. Problem specific recombination operators which take advantage of the additional information are also developed."

There are numerous similar citations available. It seems that most researches "modified" their implementations of genetic algorithms either by using non-string chromosome representation or by designing problem specific genetic operators to accommodate the problem to be solved. In [108] Koza observed:

> "Representation is a key issue in genetic algorithm work because the representation scheme can severely limit the window by which the system observes its world. However, as Davis and Steenstrup [34] point out, 'In all of Holland's work, and in the work of many of his students, chromosomes are bit strings.' String-based representations schemes are difficult and unnatural for many problems and the need for more powerful representations has been recognized for some time [43], [44], [45]."

Various nonstandard implementations were created for particular problems — simply, the classical GAs were difficult to apply directly to a problem and some modifications in chromosome structures were required. In this book we have consciously departed from classical genetic algorithms which operate on strings of bits: we searched for richer data structures and applicable "genetic" operators for these structures for variety of problems. By experimenting with such structures and operators, we obtained systems which were not genetic algorithms any more, or, at least, not classical GAs. The titles of several reports started with: "A Modified Genetic Algorithm ..." [130], "Specialized Genetic Algorithms..." [98], "A Non-Standard Genetic Algorithm..." [134]. Also, there is a feeling that the name "genetic algorithms" might be quite misleading with respect to the developed systems. Davis developed several non-standard systems with many problem-specific operators. He observed in [37]:

> "I have seen some head-shaking about that system from other researchers in the genetic algorithm field [...] a frank disbelief that the system we built was a genetic algorithm (since we didn't use binary representation, binary crossover, and binary mutation)."

Additionally, we can ask, for example, whether an evolution strategy is a genetic algorithm? Is the opposite true? To avoid all issues connected with classification of evolutionary systems, we call them simply "evolution programs" (EPs).

Why do we depart from genetic algorithms towards more flexible evolution programs? Even though nicely theorized, GA failed to provide for successful applications in many areas. It seems that the major factor behind this failure is the same one responsible for their success: domain independence.

One of the consequences of the neatness of GAs (in the sense of their domain independence) is their inability to deal with nontrivial constraints. As mentioned earlier, in most work in genetic algorithms, chromosomes are bit strings — lists of 0s and 1s. An important question to be considered in designing a chromosome representation of solutions to a problem is the implementation of constraints on solutions (problem-specific knowledge). As stated in [34]:

> "Constraints that cannot be violated can be implemented by imposing great penalties on individuals that violate them, by imposing moderate penalties, or by creating decoders of the representation that avoid creating individuals violating the constraint. Each of these solutions has its advantages and disadvantages. If one incorporates a high penalty into the evaluation routine and the domain is one in which production of an individual violating the constraint is likely, one runs the risk of creating a genetic algorithm that spends most of its time evaluating illegal individuals. Further, it can happen that when a legal individual is found, it drives the others out and the population converges on it without finding better individuals, since the likely paths to other legal individuals require the production of illegal individuals as intermediate structures, and the penalties for violating the constraint make it unlikely that such intermediate structures will reproduce. If one imposes moderate penalties, the system may evolve individuals that violate the constraint but are rated better than those that do not because the rest of the evaluation function can be satisfied better by accepting the moderate constraint penalty than by avoiding it. If one builds a "decoder" into the evaluation procedure that intelligently avoids building an illegal individual from the chromosome, the result is frequently computation-intensive to run. Further, not all constraints can be easily implemented in this way."

In evolution programming, the problem of constraint satisfaction has a different flavor. It is not the issue of selecting an evaluation function with some penalties, but rather selecting "the best" chromosomal representation of solutions together with meaningful genetic operators to satisfy all constraints

imposed by the problem. Any genetic operator should pass some characteric-tic structure from parent to offspring, so the representation structure plays an important role in defining genetic operators. Moreover, different representation structures have different characteristics of suitability for constraint representation, which complicates the problem even more. These two components (representation and operators) influence each other; it seems that any problem would require careful analysis which would result in appropriate representation for which there are meaningful genetic operators.

Glover in his study on solving a complex keyboard configuration problem [63] wrote:

> "Although the robust character of the GA search paradigm is well suited to the demands of the keyboard configuration problem, the bit string representation and idealized operators are not properly matched to the [...] required constraints. For instance, if three bits are used to represent each component of a simple keyboard of only 40 components, it is easy to show that only one out of every 10^{16} arbitrarily selected 120-bit structures represents a legal configuration map structure."

Another citation is from the work of De Jong [48], where the traveling salesman problem is briefly discussed:

> "Using the standard crossover and mutation operators, a GA will explore the space of all *combinations* of city names when, in fact, it is the space of all *permutations* which is of interest. The obvious problem is that as N [the number of cities in the tour] increases, the space of permutations is a vanishingly small subset of the space of combinations, and the powerful GA sampling heuristic has been rendered impotent by a poor choice of representation."

At early stages of AI, the general problem solvers (GPSs) were design as generic tools for approaching complex problems. However, as it turned out, it was necessary to incorporate problem-specific knowlege due to unmanageable complexity of these systems. Now the history repeated itself: until recently genetic algorithms were perceived as generic tools useful for optimization of many hard problems. However, the need for the incorporation of the problem-specific knowledge in genetic algorithms has been recognized in some research articles for some time [5], [58], [59], [81], [175]. It seems that GAs (as GPS) are too domain independent to be useful in many applications. So it is not surprising at all that evolution programs, incorporating problem-specific knowledge in the chromosomes' data structures and specific "genetic" operators, perform much better.

The basic conceptual difference between genetic algorithms and evolution programs is presented in Figures 0.2 and 0.3. Classical genetic algorithms, which operate on binary strings, require a modification of an original problem into appropriate (suitable for GA) form; this would include mapping between potential solutions and binary representation, taking care of decoders or repair algorithms, etc. This is not usualy an easy task.

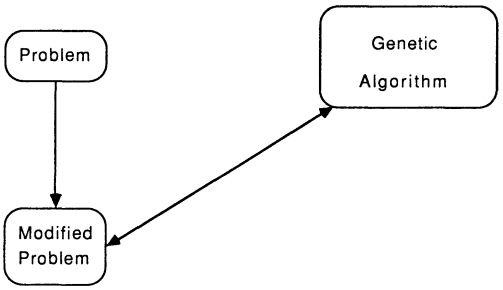

Fig. 0.2. Genetic algorithm approach

On the other hand, evolution programs would leave the problem unchanged, modifying a chromosome representation of a potential solution (using "natural" data structures), and applying appropriate "genetic" operators.

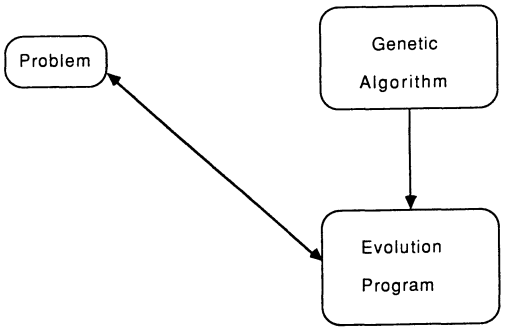

Fig. 0.3. Evolution program approach

In other words, to solve a nontrivial problem using an evolution program, we can either transform the problem into a form appropriate for the genetic algorithm (Figure 0.2), or we can transform the genetic algorithm to suit the problem (Figure 0.3). Clearly, classical GAs take the former approach, EPs the latter. So the idea behind evolution programs is quite simple and is based on the following motto:

> "If the mountain will not come to Mohammed, then Mohammed will go to the mountain."

This is not a very new idea. In [37] Davis wrote:

> "It has seemed true to me for some time that we cannot handle most real-world problems with binary representations and an operator set consisting only of binary crossover and binary mutation. One reason for this is that nearly every real-world domain has associated domain knowledge that is of use when one is considering a transformation of a solution in the domain [...] I believe that genetic algorithms are the appropriate algorithms to use in a great many real-world applications. I also believe that one should incorporate real-world knowledge in one's algorithm by adding it to one's decoder or by expanding one's operator set."

Here, we call such modified genetic algorithms "evolution programs".

It is quite hard to draw a line between genetic algorithms and evolution programs. What is required for an evolution program to be a genetic algorithm? Maintaining population of potential solutions? Binary representation of potential solutions? Selection process based on fitness of individuals? Recombination operators? The existence of a Schema Theorem? Building-block hypothesis? All of the above? Is an evolution program for the traveling salesman problem with integer vector representation and PMX operator (Chapter 10) a genetic algorithm? Is an evolution program for the transportation problem with matrix representation and arithmetical crossover operator (Chapter 9) a genetic algorithm?

In this book we will not provide answers for the above question, instead we present some interesting results of using evolution programming techniques on variety of problems.

As mentioned earlier, several researchers recognized potential behind various modifications. In [38] Davis wrote:

> "When I talk to the user, I explain that my plan is to hybridize the genetic algorithm technique and the current algorithm by employing the following three principles:
>
> - *Use the Current Encoding.* Use the current algorithm's encoding technique in the hybrid algorithm.
> - *Hybridize Where Possible.* Incorporate the positive features of the current algorithm in the hybrid algorithm.
> - *Adapt the Genetic Operators.* Create crossover and mutation operators for the new type of encoding by analogy with bit string crossover and mutation operators. Incorporate domain-based heuristics as operators as well.
>
> [...] I use the term *hybrid genetic algorithm* for algorithms created by applying these three principles."

It seems that hybrid genetic algorithms and evolution programs share a common idea: departure from classical, bit-string genetic algorithms towards

more complex systems, involving the appropriate data structures (Use the Current Encoding) and suitable genetic operators (Adapt the Genetic Operators). On the other hand, Davis assumed the existence of one or more current (traditional) algorithms available on the problem domain — on the basis of such algorithms a construction of a hybrid genetic algorithm is discussed. In our approach of evolution programming, we do not make any assumption of this sort: all evolution systems discussed later in the book were built from scratch.

What are the strengths and weaknesses of evolution programming? It seems that the major strength of EP technique is its wide applicability. In this book, we try to describe a variety of different problems and discuss a construction of an evolution program for each of them. Very often, the results are outstanding: the systems perform much better than commercially available software. Another strong point connected with evolution programs is that they are parallel in nature. As stated in [72]:

> "In a world where serial algorithms are usually made parallel through countless tricks and contortions, it is no small irony that genetic algorithms (highly parallel algorithms) are made serial through equally unnatural tricks and turns."

Of course, this is also true for any (population based) evolution program. On the other hand, we have to admit the poor theoretical basis of evolution programs. Experimenting with different data structures and modifying crossover and mutation requires a careful analysis, which would guarantee reasonable preformance. This has not been done yet.

However, some evolution programs enjoy theoretical foundations: for evolution strategies applied to regular problems (Chapter 8) a convergence property can be shown. Genetic algorithms have a Schema Theorem (Chapter 3) which explains why they work. For other evolution programs very often we have only interesting results.

In general, AI problem solving strategies are categorized into "strong" and "weak" methods. A weak method makes few assumptions about the problem domain; hence it usually enjoys wide applicability. On the other hand, it can suffer from combinatorially explosive solution costs when scaling up to larger problems [46]. This can be avoided by making strong assumptions about the problem domain, and consequently exploiting these assumptions in the problem solving method. But a disadvantage of such strong methods is their limited applicability: very often they require significant redesign when applied even to related problems.

Evolution programs fit somewhere between weak and strong methods. Some evolution programs (as genetic algorithms) are quite weak without making any assumption of a problem domain. Some other programs (GENOCOP, GAFOC, GENETIC-2) are more problem specific with a varying degree of problem dependence. For example, GENOCOP (Chapter 7), like all evolution strategies (Chapter 8), was build to solve parameter optimization problems. The system can handle any objective function with any set of linear constraints. The

GAFOC (Chapter 7) and GENETIC-2 (Chapter 9) work for optimal control problems and transportation problems, respectively. Other systems (see Chapter 10) are suitable for combinatorial optimization problems (like scheduling problems, traveling salesman problems), or for graph drawing problems (Chapter 11). An interesting application of an evolution program for inductive learning of decision rules is discussed in Chapter 12.

It is little bit ironic: genetic algorithms are perceived as weak methods; however, in the presence of nontrivial constraints, they change rapidly into strong methods. Whether we consider a penalty function, decoder, or a repair algorithm, these must be tailored for a specific application. On the other hand, our evolution programs (perceived as much stronger, problem-dependent methods) suddenly seem much weaker. For example, both GENOCOP and GAFOC (Chapter 7) work for large classes of problems. This demonstrates a huge potential behind the evolution programming approach.

All these observations triggered my interest in investigating the properties of different genetic operators defined on richer structures than bit strings — further, this research would lead to the creation of a new programming methodology (in [133] such a proposed programming methodology was called EVA for "EVolution progrAmming"). Roughly speaking, a programmer in such an environment would select data structures with appropriate genetic operators for a given problem as well as selecting an evaluation function and initializating the population (the other parameters are tuned by another genetic process).

However, a lot of research should be done before we can propose the basic constructs of such a programming environment. This book provides just the first step towards this goal by investigating different structures and genetic operators building evolution programs for many problems.

We shall return to the idea of a new programming environment at the end of the book, in the conclusions.

Part I

Genetic Algorithms

1. GAs: What Are They?

Paradoxical as it seemed, the Master
always insisted that the true reformer
was one who was able to see that everything
is perfect as it is — and able to
leave it alone.

Anthony de Mello, *One Minute Wisdom*

There is a large class of interesting problems for which no reasonably fast algorithms have been developed. Many of these problems are optimization problems that arise frequently in applications. Given such a hard optimization problem it is often possible to find an efficient algorithm whose solution is approximately optimal. For some hard optimization problems we can use probabilistic algorithms as well — these algorithms do not guarantee the optimum value, but by randomly choosing sufficiently many "witnesses" the probability of error may be made as small as we like.

There are a lot of important practical optimization problems for which such algorithms of high quality have become available [33]. For instance we can apply simulated annealing for wire routing and component placement problems in VLSI design or for the traveling salesman problem. Moreover, many other large-scale combinatorial optimization problems (many of which have been proved NP-hard) can be solved approximately on present-day computers by this kind of Monte Carlo technique.

In general, any abstract task to be accomplished can be thought of as solving a problem, which, in turn, can be perceived as a search through a space of potential solutions. Since we are after "the best" solution, we can view this task as an optimization process. For small spaces, classical exhaustive methods usually suffice; for larger spaces special artificial intelligence techniques must be employed. Genetic Algorithms (GAs) are among such techniques; they are stochastic algorithms whose search methods model some natural phenomena: genetic inheritance and Darwinian strife for survival. As stated in [34]:

> "... the metaphor underlying genetic algorithms is that of natural evolution. In evolution, the problem each species faces is one of

> searching for beneficial adaptations to a complicated and changing environment. The 'knowledge' that each species has gained is embodied in the makeup of the chromosomes of its members."

The idea behind genetic algorithms is to do what nature does. Let us take rabbits as an example: at any given time there is a population of rabbits. Some of them are faster and smarter than other rabbits. These faster, smarter rabbits are less likely to be eaten by foxes, and therefore more of them survive to do what rabbits do best: make more rabbits. Of course, some of the slower, dumber rabbits will survive just because they are lucky. This surviving population of rabbits starts breeding. The breeding results in a good mixture of rabbit genetic material: some slow rabbits breed with fast rabbits, some fast with fast, some smart rabbits with dumb rabbits, and so on. And on the top of that, nature throws in a 'wild hare' every once in a while by mutating some of the rabbit genetic material. The resulting baby rabbits will (on average) be faster and smarter than these in the original population because more faster, smarter parents survived the foxes. (It is a good thing that the foxes are undergoing similar process — otherwise the rabbits might become too fast and smart for the foxes to catch any of them).

A genetic algorithm follows a step-by-step procedure that closely matches the story of the rabbits. Before we take a closer look at the structure of a genetic algorithm, let us have a quick look at the history of genetics (from [180]):

> "The fundamental principle of natural selection as the main evolutionary principle has been formulated by C. Darwin long before the discovery of genetic mechanisms. Ignorant of the basic heredity principles, Darwin hypothesized fusion or blending inheritance, supposing that parental qualities mix together like fluids in the offspring organism. His selection theory arose serious objections, first stated by F. Jenkins: crossing quickly levels off any hereditary distinctions, and there is no selection in homogeneous populations (the so-called 'Jenkins nightmare').
>
> It was not until 1865, when G. Mendel discovered the basic principles of transference of hereditary factors from parent to offspring, which showed the discrete nature of these factors, that the 'Jenkins nightmare' could be explained, since because of this discreteness there is no 'dissolution' of hereditary distinctions.
>
> Mendelian laws became known to the scientific community after they had been independently rediscovered in 1900 by H. de Vries, K. Correns and K. von Tschermak. Genetics was fully developed by T. Morgan and his collaborators, who proved experimentally that chromosomes are the main carriers of hereditary information and that genes, which present hereditary factors, are lined up on chromosomes. Later on, accumulated experimental facts showed Mendelian laws to be valid for all sexually reproducing organisms.

> However, Mendel's laws, even after they had been rediscovered, and Darwin's theory of natural selection remained independent, unlinked concepts. And moreover, they were opposed to each other. Not until the 1920s (see, for instance the classical work by Četverikov [26]) was it proved that Mendel's genetics and Darwin's theory of natural selection are in no way conflicting and that their happy marriage yields modern evolutionary theory."

Genetic algorithms use a vocabulary borrowed from natural genetics. We would talk about *individuals* (or *genotypes*, *structures*) in a population; quite often these individuals are called also *strings* or *chromosomes*. This might be a little bit misleading: each cell of every organism of a given species carries a certain number of chromosomes (man, for example, has 46 of them); however, in this book we talk about one-chromosome individuals only. (For additional information on *diploidy* — pairs of chromosomes — dominance, and other related issues, in connection with genetic algorithms, the reader is referred to [72].) Chromosomes are made of units — *genes* (also *features, characters, or decoders*) — arranged in linear succession; every gene controls the inheritance of one or several characters. Genes of certain characters are located at certain places of the chromosome, which are called *loci* (string positions). Any character of individuals (such as hair color) can manifest itself differently; the gene is said to be in several states, called *alleles* (feature values).

Each genotype (in this book a single chromosome) would represent a potential solution to a problem; an evolution process run on a population of chromosomes corresponds to a search through a space of potential solutions. Such a search requires balancing two (apparently conflicting) objectives: exploiting the best solutions and exploring the search space [21]. Hillclimbing is an example of a strategy which exploits the best solution for possible improvement; on the other hand, it neglects exploration of the search space. Random search is a typical example of a strategy which explores the search space ignoring the exploitations of the promising regions of the space. Genetic algorithms are a class of general purpose (domain independent) search methods which strike a remarkable balance between exploration and exploitation of the search space.

GAs have been quite successfully applied to optimization problems like wire routing, scheduling, adaptive control, game playing, cognitive modeling, transportation problems, traveling salesman problems, optimal control problems, database query optimization, etc. (see [15], [20], [43], [72], [78], [81], [82], [134], [132], [160], [189], [190]). However, De Jong [43] warned against perceiving GAs as optimization tools:

> "...because of this historical focus and emphasis on function optimization applications, it is easy to fall into the trap of perceiving GAs *themselves* as optimization algorithms and then being surprised and/or disappointed when they fail to find an 'obvious' optimum in a particular search space. My suggestion for avoiding this perceptual trap is to think of GAs as a (highly idealized) simulation of a natural

> process and as such they embody the goals and purposes (if any) of that natural process. I am not sure if anyone is up to the task of defining the goals and purpose of evolutionary systems; however, I think it's fair to say that such systems are *not* generally perceived as functions optimizers".

On the other hand, optimization is a major field of GA's applicability. In [162] (1981) Schwefel said:

> "There is scarcely a modern journal, whether of engineering, economics, management, mathematics, physics, or the social sciences, in which the concept 'optimization' is missing from the subject index. If one abstracts from all specialist points of view, the recurring problem is to select a better or best (according to Leibniz, optimal) alternative from among a number of possible states of affairs."

During the last decade, the significance of optimization has grown even further — many important large-scale combinatorial optimization problems and highly constrained engineering problems can only be solved approximately on present day computers.

Genetic algorithms aim at such complex problems. They belong to the class of probabilistic algorithms, yet they are very different from random algorithms as they combine elements of directed and stochastic search. Because of this, GA are also more robust than existing directed search methods. Another important property of such genetic based search methods is that they maintain a population of potential solutions — all other methods process a single point of the search space.

Hillclimbing methods use the iterative improvement technique; the technique is applied to a single point (the current point) in the search space. During a single iteration, a new point is selected from the neighborhood of the current point (this is why this technique is known also as neighborhood search or local search [112]). If the new point provides a better[1] value of the objective function, the new point becomes the current point. Otherwise, some other neighbor is selected and tested against the current point. The method terminates if no further improvement is possible.

It is clear that the hillclimbing methods provide local optimum values only and these values depend on the selection of the starting point. Moreover, there is no information available on the relative error (with respect to the global optimum) of the solution found.

To increase the chances to succeed, hillclimbing methods usually are executed for a (large) number of different starting points (these points need not be selected randomly — a selection of a starting point for a single execution may depend on the result of the previous runs).

The simulated annealing technique [1] eliminates most disadvantages of the hillclimbing methods: solutions do not depend on the starting point any longer

[1] smaller, for minimization, and larger, for maximization problems.

and are (usually) close to the optimum point. This is achieved by introducing a probability p of acceptance (i.e., replacement of the current point by a new point): $p = 1$, if the new point provides a better value of the objective function; however, $p > 0$, otherwise. In the latter case, the probability of acceptance p is a function of the values of objective function for the current point and the new point, and an additional control parameter, "temperature", T. In general, the lower temperature T is, the smaller the chances for the acceptance of a new point are. During execution of the algorithm, the temperature of the system, T, is lowered in steps. The algorithm terminates for some small value of T, for which virtually no changes are accepted anymore.

As mentioned earlier, a GA performs a multi-directional search by maintaining a population of potential solutions and encourages information formation and exchange between these directions. The population undergoes a simulated evolution: at each generation the relatively "good" solutions reproduce, while the relatively "bad" solutions die. To distinguish between different solutions we use an objective (evaluation) function which plays the role of an environment.

An example of hillclimbing, simulated annealing, and genetic algorithm techniques is given later in this chapter (Section 1.4).

The structure of a simple genetic algorithm is the same as the structure of any evolution program (see Figure 0.1, Introduction). During iteration t, a genetic algorithm maintains a population of potential solutions (chromosomes, vectors), $P(t) = \{x_1^t, \ldots, x_n^t\}$. Each solution x_i^t is evaluated to give some measure of its "fitness". Then, a new population (iteration $t + 1$) is formed by selecting the more fit individuals. Some members of this new population undergo reproduction by means of crossover and mutation, to form new solutions. Crossover combines the features of two parent chromosomes to form two similar offspring by swapping corresponding segments of the parents. For example, if the parents are represented by five-dimensional vectors $(a_1, b_1, c_1, d_1, e_1)$ and $(a_2, b_2, c_2, d_2, e_2)$, then crossing the chromosomes after the second gene would produce the offspring $(a_1, b_1, c_2, d_2, e_2)$ and $(a_2, b_2, c_1, d_1, e_1)$. The intuition behind the applicability of the crossover operator is information exchange between different potential solutions.

Mutation arbitrarily alters one or more genes of a selected chromosome, by a random change with a probability equal to the mutation rate. The intuition behind the mutation operator is the introduction of some extra variability into the population.

A genetic algorithm (as any evolution program) for a particular problem must have the following five components:

- a genetic representation for potential solutions to the problem,
- a way to create an initial population of potential solutions,
- an evaluation function that plays the role of the environment, rating solutions in terms of their "fitness",

- genetic operators that alter the composition of children during reproduction,
- values for various parameters that the genetic algorithm uses (population size, probabilities of applying genetic operators, etc.).

We discuss the main features of genetic algorithms by presenting three examples. In the first one we apply a genetic algorithm for optimization of a simple function of one real variable. The second example illustrates the use of a genetic algorithm to learn a strategy for a simple game (the prisoner's dilemma). The third example discusses one possible application of a genetic algorithm to approach a combinatorial NP-hard problem, the traveling salesman problem.

1.1 Optimization of a simple function

In this section we discuss the basic features of a genetic algorithm for optimization of a simple function of one variable. The function is defined as

$$f(x) = x \cdot \sin(10\pi \cdot x) + 1.0$$

and is drawn in Figure 1.1. The problem is to find x from the range $[-1..2]$ which maximizes the function f, i.e., to find x_0 such that

$$f(x_0) \geq f(x), \text{ for all } x \in [-1..2].$$

It is relatively easy to analyse the function f. The zeros of the first derivative f' should be determined:

$$f'(x) = \sin(10\pi \cdot x) + 10\pi x \cdot \cos(10\pi \cdot x) = 0;$$

the formula is equivalent to

$$\tan(10\pi \cdot x) = -10\pi x.$$

It is clear that the above equation has an infinite number of solutions,

$$x_i = \frac{2i-1}{20} + \epsilon_i, \text{ for } i = 1, 2, \ldots$$
$$x_0 = 0$$
$$x_i = \frac{2i+1}{20} - \epsilon_i, \text{ for } i = -1, -2, \ldots,$$

where terms ϵ_i represent decreasing sequences of real numbers (for $i = 1, 2, \ldots$, and $i = -1, -2, \ldots$) approaching zero.

Note also that the function f reaches its local maxima for x_i if i is an odd integer, and its local minima for x_i if i is an even integer (see Figure 1.1).

Since the domain of the problem is $x \in [-1..2]$, the function reaches its maximum for for $x_{19} = \frac{37}{20} + \epsilon_{19} = 1.85 + \epsilon_{19}$, where $f(x_{19})$ is slightly larger than $f(1.85) = 1.85 \cdot \sin(18\pi + \frac{\pi}{2}) + 1.0 = 2.85$.

Assume that we wish to construct a genetic algorithm to solve the above problem, i.e., to maximize the function f. Let us discuss the major components of such a genetic algorithm in turn.

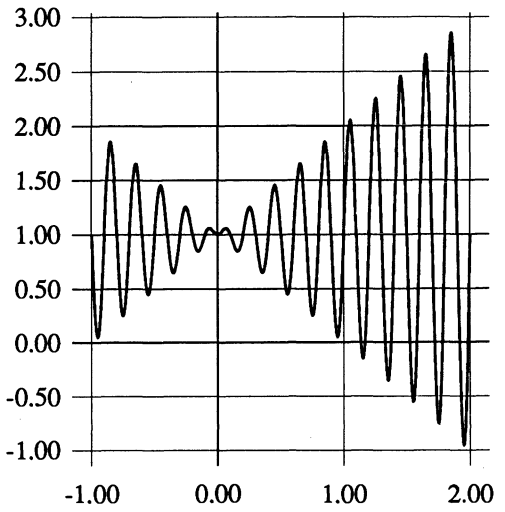

Fig. 1.1. Graph of the function $f(x) = x \cdot \sin(10\pi \cdot x) + 1.0$

1.1.1 Representation

We use a binary vector as a chromosome to represent real values of the variable x. The length of the vector depends on the required precision, which, in this example, is six places after the decimal point.

The domain of the variable x has length 3; the precision requirement implies that the range $[-1..2]$ should be divided into at least $3 \cdot 1000000$ equal size ranges. This means that 22 bits are required as a binary vector (chromosome):

$$2097152 = 2^{21} < 3000000 \leq 2^{22} = 4194304.$$

The mapping from a binary string $\langle b_{21} b_{20} \ldots b_0 \rangle$ into a real number x from the range $[-1..2]$ is straightforward and is completed in two steps:

- convert the binary string $\langle b_{21} b_{20} \ldots b_0 \rangle$ from the base 2 to base 10:

  $$(\langle b_{21} b_{20} \ldots b_0 \rangle)_2 = (\textstyle\sum_{i=0}^{21} b_i \cdot 10^i)_{10} = x',$$

- find a corresponding real number x:

  $$x = -1.0 + x' \cdot \tfrac{3}{2^{22}-1},$$

  where -1.0 is the left boundary of the domain and 3 is the length of the domain.

For example, a chromosome

(1000101110110101000111)

represents the number 0.637197, since

$$x' = (1000101110110101000111)_2 = 2288967$$

and

$$x = -1.0 + 2288967 \cdot \frac{3}{4194303} = 0.637197.$$

Of course, the chromosomes

(00000000000000000000) and (11111111111111111111)

represent boundaries of the domain, -1.0 and 2.0, respectively.

1.1.2 Initial population

The initialization process is very simple: we create a population of chromosomes, where each chromosome is a binary vector of 22 bits. All 22 bits for each chromosome are initialized randomly.

1.1.3 Evaluation function

Evaluation function *eval* for binary vectors $\mathbf{v}$ is equivalent to the function f:

$$eval(\mathbf{v}) = f(x),$$

where the chromosome $\mathbf{v}$ represents the real value x.

As noted earlier, the evaluation function plays the role of the environment, rating potential solutions in terms of their fitness. For example, three chromosomes:

$$\mathbf{v_1} = (1000101110110101000111),$$
$$\mathbf{v_2} = (0000001110000000010000),$$
$$\mathbf{v_3} = (1110000000111111000101),$$

correspond to values $x_1 = 0.637197$, $x_2 = -0.958973$, and $x_3 = 1.627888$, respectively. Consequently, the evaluation function would rate them as follows:

$$eval(\mathbf{v_1}) = f(x_1) = 1.586345,$$
$$eval(\mathbf{v_2}) = f(x_2) = 0.078878,$$
$$eval(\mathbf{v_3}) = f(x_3) = 2.250650.$$

Clearly, the chromosome $\mathbf{v_3}$ is the best of the three chromosomes, since its evaluation returns the highest value.

1.1.4 Genetic operators

During the reproduction phase of the genetic algorithm we would use two classical genetic operators: mutation and crossover.

As mentioned earlier, mutation alters one or more genes (positions in a chromosome) with a probability equal to the mutation rate. Assume that the fifth gene from the $\mathbf{v_3}$ chromosome was selected for a mutation. Since the fifth gene in this chromosome is 0, it would be flipped into 1. So the chromosome $\mathbf{v_3}$ after this mutation would be

$$\mathbf{v_3}' = (11101000001111111000101).$$

This chromosome represents the value $x_3' = 1.721638$ and $f(x_3') = -0.082257$. This means that this particular mutation resulted in a significant decrease of the value of the chromosome $\mathbf{v_3}$. On the other hand, if the 10th gene was selected for mutation in the chromosome $\mathbf{v_3}$, then

$$\mathbf{v_3}'' = (11100000011111111000101).$$

The corresponding value $x_3'' = 1.630818$ and $f(x_3'') = 2.343555$, an improvement over the original value of $f(x_3) = 2.250650$.

Let us illustrate the crossover operator on chromosomes $\mathbf{v_2}$ and $\mathbf{v_3}$. Assume that the crossover point was (randomly) selected after the 5th gene:

$$\mathbf{v_2} = (00000|01110000000010000),$$
$$\mathbf{v_3} = (11100|00000111111000101).$$

The two resulting offspring are

$$\mathbf{v_2}' = (00000|00000111111000101),$$
$$\mathbf{v_3}' = (11100|01110000000010000).$$

These offspring evaluate to

$$f(\mathbf{v_2}') = f(-0.998113) = 0.940865,$$
$$f(\mathbf{v_3}') = f(1.666028) = 2.459245.$$

Note that the second offspring has a better evaluation than both of its parents.

1.1.5 Parameters

For this particular problem we have used the following parameters: population size $pop\_size = 50$, probability of crossover $p_c = 0.25$, probability of mutation $p_m = 0.01$. The following section presents some experimental results for such a genetic system.

1.1.6 Experimental results

In Table 1.1 we provide the generation number for which we noted an improvement in the evaluation function, together with the value of the function. The best chromosome after 150 generations was

$$\mathbf{v}_{max} = (11110011010001000000101),$$

which corresponds to a value $x_{max} = 1.850773$.

As expected, $x_{max} = 1.85 + \epsilon$, and $f(x_{max})$ is slightly larger than 2.85.

Generation number	Evaluation function
1	1.441942
6	2.250003
8	2.250283
9	2.250284
10	2.250363
12	2.328077
39	2.344251
40	2.345087
51	2.738930
99	2.849246
137	2.850217
145	2.850227

Table 1.1. Results of 150 generations

1.2 The prisoner's dilemma

In this section, we explain how a genetic algorithm can be used to learn a strategy for a simple game, known as the prisoner's dilemma. We present the results obtained by Axelrod [6].

Two prisoners are held in separate cells, unable to communicate with each other. Each prisoner is asked, independently, to defect and betray the other prisoner. If only one prisoner defects, he is rewarded and the other is punished. If both defect, both remain imprisoned and are tortured. If neither defects, both receive moderate rewards. Thus, the selfish choice of defection always yields a higher payoff than cooperation — no matter what the other prisoner does — but if both defect, both do worse than if both had cooperated. The prisoner's dilemma is to decide whether to defect or cooperate with the other prisoner.

The prisoner's dilemma can be played as a game between two players, where at each turn, each player either defects or cooperates with the other prisoner. The players then score according to the payoffs listed in the Table 1.2.

Player 1	Player 2	P_1	P_2	Comment
Defect	Defect	1	1	Punishment for mutual defection
Defect	Cooperate	5	0	Temptation to defect and sucker's payoff
Cooperate	Defect	0	5	Sucker's payoff, and temptation to defect
Cooperate	Cooperate	3	3	Reward for mutual cooperation

Table 1.2. Payoff table for prisoner's dilemma game: P_i is the payoff for Player i

We will now consider how a genetic algorithm might be used to learn a strategy for the prisoner's dilemma. A GA approach is to maintain a population of "players", each of which has a particular strategy. Initially, each player's strategy is chosen at random. Thereafter, at each step, players play games and their scores are noted. Some of the players are then selected for the next generation, and some of those are chosen to mate. When two players mate, the new player created has a strategy constructed from the strategies of its parents (crossover). A mutation, as usual, introduces some variability into players' strategies by random changes on representations of these strategies.

1.2.1 Representing a strategy

First of all, we need some way to represent a strategy (i.e., a possible solution). For simplicity, we will consider strategies that are deterministic and use the outcomes of the three previous moves to make a choice in the current move. Since there are four possible outcomes for each move, there are $4 \times 4 \times 4 = 64$ different histories of the three previous moves.

A strategy of this type can be specified by indicating what move is to be made for each of these possible histories. Thus, a strategy can be represented by a string of 64 bits (or Ds and Cs), indicating what move is to be made for each of the 64 possible histories. To get the strategy started at the beginning of the game, we also need to specify its initial premises about the three hypothetical moves which preceded the start of the game. This requires six more genes, making a total of seventy loci on the chromosome.

This string of seventy bits specifies what the player would do in every possible circumstance and thus completely defines a particular strategy. The string of 70 genes also serves as the player's chromosome for use in the evolution process.

1.2.2 Outline of the genetic algorithm

Axelrod's genetic algorithm to learn a strategy for the prisoner's dilemma works in four stages, as follows:

1. Choose an initial population. Each player is assigned a random string of seventy bits, representing a strategy as discussed above.
2. Test each player to determine its effectiveness. Each player uses the strategy defined by its chromosome to play the game with other players. The player's score is its average over all the games it plays.
3. Select players to breed. A player with an average score is given one mating; a player scoring one standard deviation above the average is given two matings; and a player scoring one standard deviation below the average is given no matings.
4. The successful players are randomly paired off to produce two offspring per mating. The strategy of each offspring is determined from the strategies of its parents. This is done by using two genetics operators: crossover and mutation.

After these four stages we get a new population. The new population will display patterns of behavior that are more like those of the successful individuals of the previous generation, and less like those of the unsuccessful ones. With each new generation, the individuals with relatively high scores will be more likely to pass on parts of their strategies, while the relatively unsuccessful individuals will be less likely to have any parts of their strategies passed on.

1.2.3 Experimental results

Running this program, Axelrod obtained quite remarkable results. From a strictly random start, the genetic algorithm evolved populations whose median member was just as successful as the best known heuristic algorithm. Some behavioral patterns evolved in the vast majority of the individuals; these are:

1. Don't rock the boat: continue to cooperate after three mutual cooperations (i.e., C after (CC)(CC)(CC)[2]).
2. Be provokable: defect when the other player defects out of the blue (i.e., D after receiving (CC)(CC)(CD)).
3. Accept an apology: continue to cooperate after cooperation has been restored (i.e., C after (CD)(DC)(CC)).
4. Forget: cooperate when mutual cooperation has been restored after an exploitation (i.e., C after (DC)(CC)(CC)).
5. Accept a rut: defect after three mutual defections (i.e., D after (DD)(DD)(DD)).

For more details, see [6].

[2]The last three moves are described by three pairs $(a_1b_1)(a_2b_2)(a_3b_3)$, where the a's are this player's moves (C for cooperate, D for defect) and the b's are the other player's moves.

1.3 Traveling salesman problem

In this section, we explain how a genetic algorithm can be used to approach the Traveling Salesman Problem (TSP). Note that we shall discuss only one possible approach. In Chapter 10 we discuss other approaches to the TSP as well.

Simply stated, the traveling salesman must visit every city in his territory exactly once and then return to the starting point; given the cost of travel between all cities, how should he plan his itinerary for minimum total cost of the entire tour?

The TSP is a problem in combinatorial optimization and arises in numerous applications. There are several branch-and-bound algorithms, approximate algorithms, and heuristic search algorithms which approach this problem. During the last few years there have been several attempts to approximate the TSP by genetic algorithms [72, pages 166–179]; here we present one of them.

First, we should address an important question connected with the chromosome representation: should we leave a chromosome to be an integer vector, or rather we should transform it into a binary string? In the previous two examples (optimization of a function and the prisoner's dilemma) we represented a chromosome (in a more or less natural way) as a binary vector. This allowed us to use binary mutation and crossover; applying these operators we got legal offspring, i.e., offspring within the search space. This is not the case for the traveling salesman problem. In a binary representation of a n cities TSP problem, each city should be coded as a string of $\lceil \log_2 n \rceil$ bits; a chromosome is a string of $n \cdot \lceil \log_2 n \rceil$ bits. A mutation can result in a sequence of cities, which is not a tour: we can get the same city twice in a sequence. Moreover, for a TSP with 20 cities (where we need 5 bits to represent a city), some 5-bit sequences (for example, 10101) do not correspond to any city. Similar problems are present when applying crossover operator. Clearly, if we use mutation and crossover operators as defined earlier, we would need some sort of a "repair algorithm"; such an algorithm would "repair" a chromosome, moving it back into the search space.

It seems that the integer vector representation is better: instead of using repair algorithms, we can incorporate the knowledge of the problem into operators: in that way they would "intelligently" avoid building an illegal individual. In this particular approach we accept integer representation: a vector $\mathbf{v} = \langle i_1 i_2 \ldots i_n \rangle$ represents a tour: from i_1 to i_2, etc., from i_{n-1} to i_n and back to i_1 ($\mathbf{v}$ is a permutation of $\langle\ 1\ 2\ \ldots\ n\ \rangle$).

For the initialization process we can either use some heuristics (for example, we can accept a few outputs from a greedy algorithm for the TSP, starting from different cities), or we can initialize the population by a random sample of permutations of $\langle\ 1\ 2\ \ldots\ n\ \rangle$.

The evaluation of a chromosome is straightforward: given the cost of travel between all cities, we can easily calculate the total cost of the entire tour.

In the TSP we search for the best ordering of cities in a tour. It is relatively easy to come up with some unary operators (unary type operators) which would search for better string orderings. However, using only unary operators, there is a little hope of finding even good orderings (not to mention the best one) [70]. Moreover, the strength of genetic algorithms arises from the structured information exchange of crossover combinations of highly fit individuals. So what we need is a crossover-like operator that would exploit important similarities between chromosomes. For that purpose we use a variant of a OX operator [31], which, given two parents, builds offspring by choosing a subsequence of a tour from one parent and preserving the relative order of cities from the other parent. For example, if the parents are

$$\langle\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10\ 11\ 12\ \rangle \text{ and}$$
$$\langle\ 7\ 3\ 1\ 11\ 4\ 12\ 5\ 2\ 10\ 9\ 6\ 8\ \rangle$$

and the chosen part is

$$(4\ 5\ 6\ 7),$$

the resulting offspring is

$$\langle\ 1\ 11\ 12\ 4\ 5\ 6\ 7\ 2\ 10\ 9\ 8\ 3\ \rangle.$$

As required, the offspring bears a structural relationship to both parents. The roles of the parents can then be reversed in constructing a second offspring.

A genetic algorithm based on the above operator outperforms random search, but leaves much room for improvements. Typical (average over 20 random runs) results from the algorithm, as applied to 100 randomly generated cities, gave (after 20000 generations) a value of the whole tour 9.4% above optimum.

For full discussion on the TSP, the representation issues and genetic operators used, the reader is referred to Chapter 10.

1.4 Hillclimbing, simulated annealing, and genetic algorithms

In this section we discuss three algorithms, i.e., hillclimbing, simulated annealing, and the genetic algorithm, applied to a simple optimization problem. This example underlines the uniqueness of the GA approach.

The search space is a set of binary strings $\mathbf{v}$ of the length 30. The objective function f to be maximized is given as

$$f(\mathbf{v}) = |11 \cdot one(\mathbf{v}) - 150|,$$

where the function $one(\mathbf{v})$ returns the number of 1s in the string $\mathbf{v}$.

For example, the following three strings

$$\mathbf{v}_1 = (11011010111010111111011011),$$
$$\mathbf{v}_2 = (11100010010011011001010100011),$$
$$\mathbf{v}_3 = (00001000001100100000010001000),$$

would evaluate to

$$f(\mathbf{v}_1) = |11 \cdot 22 - 150| = 92,$$
$$f(\mathbf{v}_2) = |11 \cdot 15 - 150| = 15,$$
$$f(\mathbf{v}_3) = |11 \cdot 6 - 150| = 84,$$

($one(\mathbf{v}_1) = 22$, $one(\mathbf{v}_2) = 15$, and $one(\mathbf{v}_3) = 6$).

The function f is linear and does not provide any challenge as an optimization task. We use it only to illustrate the ideas behind these three algorithms. However, the interesting characteristic of the function f is that it has one global maximum for

$$\mathbf{v}_g = (11111111111111111111111111111),$$

$f(\mathbf{v}_g) = |11 \cdot 30 - 150| = 180$, and one local maximum for

$$\mathbf{v}_l = (00000000000000000000000000000),$$

$f(\mathbf{v}_l) = |11 \cdot 0 - 150| = 150$.

There are a few versions of hillclimbing algorithms. They differ in the way a new string is selected for comparison with the current string. One version of a simple (iterated) hillclimbing algorithm (MAX iterations) is given in Figure 1.2 (steepest ascent hillclimbing). Initially, all 30 neighbors are considered, and the one $\mathbf{v}_n$ which returns the largest value $f(\mathbf{v}_n)$ is selected to compete with the current string $\mathbf{v}_c$. If $f(\mathbf{v}_c) < f(\mathbf{v}_n)$, then the new string becomes the current string. Otherwise, no local improvement is possible: the algorithm has reached (local or global) optimum ($local$ = TRUE). In a such case, the next iteration ($t \leftarrow t + 1$) of the algorithm is executed with a new current string selected at random.

It is interesting to note that the success or failure of the single iteration of the above hillclimber algorithm (i.e., return of the global or local optimum) is determined by the starting string (randomly selected). It is clear that if the starting string has thirteen 1s or less, the algorithm will always terminate in the local optimum (failure). The reason is that a string with thirteen 1s returns a value 7 of the objective function, and any single-step improvement towards the global optimum, i.e., increase the number of 1s to fourteen, decreases the value of the objective function to 4. On the other hand, any decrease of the number of 1s would increase the value of the function: a string with twelve 1s yields a value of 18, a string with eleven 1s yields a value of 29, etc. This would push the search in the "wrong" direction, towards the local maximum.

For problems with many local optima, the chances of hitting the global optimum (in a single iteration) are slim.

The structure of the simulated annealing procedure is given in Figure 1.3.

```
procedure iterated hillclimber
begin
  t ← 0
  repeat
    local ← FALSE
    select a current string v_c at random
    evaluate v_c
    repeat
      select 30 new strings in the neighborhood of v_c
        by flipping single bits of v_c
      select the string v_n from the set of new strings
        with the largest value of objective function f
      if f(v_c) < f(v_n)
        then v_c ← v_n
        else local ← TRUE
    until local
    t ← t + 1
  until t = MAX
end
```

Fig. 1.2. A simple (iterated) hillclimber

The function $random[0,1)$ returns a random number from the range $[0,1)$. The (termination-condition) checks whether 'thermal equilibrium' is reached, i.e., whether the probability distribution of the selected new strings approaches the Boltzmann distribution [1]. However, in some implementations [2], this repeat loop is executed just k times (k is an additional parameter of the method).

The temperature T is lowered in steps ($g(T,t) < T$ for all t). The algorithm terminates for some small value of T: the (stop-criterion) checks whether the system is 'frozen', i.e., virtually no changes are accepted anymore.

As mentioned earlier, the simulated annealing algorithm can escape local optima. Let us consider a string

$$\mathbf{v}_4 = (11100000010011011100101010000),$$

with twelve 1s, which evaluates to $f(\mathbf{v}_4) = |11 \cdot 12 - 150| = 18$. For $\mathbf{v}_4$ as the starting string, the hillclimbing algorithm (as discussed earlier) would approach the local maximum

$$\mathbf{v}_l = (00000000000000000000000000000),$$

since any string with thirteen 1s (i.e., a step 'towards' the global optimum) evaluates to 7 (less than 18). On the other hand, the simulated annealing algorithm would accept a string with thirteen 1s as a new current string with probability

$$p = \exp\{(f(\mathbf{v}_n) - f(\mathbf{v}_c))/T\} = \exp\{(7-18)/T\},$$

```
procedure simulated annealing
begin
  t ← 0
  initialize temperature T
  select a current string v_c at random
  evaluate v_c
  repeat
    repeat
      select a new string v_n
        in the neighborhood of v_c
        by flipping a single bit of v_c
      if f(v_c) < f(v_n)
        then v_c ← v_n
        else if random[0,1) < exp{(f(v_n) − f(v_c))/T}
            then v_c ← v_n
    until (termination-condition)
    T ← g(T, t)
    t ← t + 1
  until (stop-criterion)
end
```

Fig. 1.3. Simulated annealing

which, for some temperature, say, $T = 20$, gives

$$p = e^{-\frac{11}{20}} = 0.57695,$$

i.e., the chances for acceptance are better than 50%.

Genetic algorithms, as discussed in Section 1.1, maintain a population of strings. Two relatively poor strings

$$\mathbf{v}_5 = (11111000000011011100111010000) \text{ and}$$
$$\mathbf{v}_6 = (00000000000110111001010111111)$$

each of which evaluate to 16, can produce much better offspring (if the crossover point falls anywhere between the 5th and the 12th position):

$$\mathbf{v}_7 = (11111000000110111001010111111).$$

The new offspring $\mathbf{v}_7$ evaluates to

$$f(\mathbf{v}_7) = |11 \cdot 19 - 150| = 59.$$

For a detailed discussion on these and other algorithms (various variants of hillclimbers, genetic search, and simulated annealing) tested on several functions with different characteristics, the reader is referred to [2].

1.5 Conclusions

The three examples of genetic algorithms for function optimization, the prisoner's dilemma, and the traveling salesman problem, show a wide applicability of genetic algorithms. However, at the same time we should observe first signs of potential difficulties. The representation issues for the traveling salesman problem were not obvious. The new operator used (OX crossover) was far from trivial. What kind of further difficulties may we have for some other (hard) problems? In the first and third examples (optimization of a function and the traveling salesman problem) the evaluation function was clearly defined; in the second example (the prisoner's dilemma) a simple simulation process would give us an evaluation of a chromosome (we test each player to determine its effectiveness: each player uses the strategy defined by its chromosome to play the game with other players and the player's score is its average over all the games it plays). How should we proceed in a case where the evaluation function is not clearly defined? For example, the Boolean Satisfiability Problem (SAT) seems to have a natural string representation (the i-th bit represents the truth value of the i-th Boolean variable), however, the process of choosing an evaluation function is far from obvious [46].

The first example of optimization of an unconstrained function allows us to use a convenient representation, where any binary string would correspond to a value from the domain of the problem (i.e., $[-1..2]$). This means that any mutation and any crossover would produce a legal offspring. The same was true in the second example: any combination of bits represents a legal strategy. The third problem has a single constraint: each city should appear precisely once in a legal tour. This caused some problems: we used vectors of integers (instead of binary representation) and we modified the crossover operator. But how should we approach a constrained problem in general? What possibilities do we have?

The answers are not easy; we explore these issues later in the book.

2. GAs: How Do They Work?

To every thing there is a season,
and a time to every purpose under the heaven:
A time to be born and a time to die;
a time to plant, and a time to pluck up
that which is planted;
A time to kill, and a time to heal;
a time to break down, and a time to build up.

The Bible, Ecclesiastes, 3

In this chapter we discuss the actions of a genetic algorithm for a simple parameter optimization problem. We start with a few general comments; a detailed example follows.

Let us note first that, without any loss of generality, we can assume maximization problems only. If the optimization problem is to minimize a function f, this is equivalent to maximizing a function g, where $g = -f$, i.e.,

$$\min f(x) = \max g(x) = \max\{-f(x)\}.$$

Moreover, we may assume that the objective function f takes positive values on its domain; otherwise we can add some positive constant C, i.e.,

$$\max g(x) = \max\{g(x) + C\}.$$

Now suppose we wish to maximize a function of k variables, $f(x_1, \ldots, x_k) : R^k \rightarrow R$. Suppose further that each variable x_i can take values from a domain $D_i = [a_i, b_i] \subseteq R$ and $f(x_1, \ldots, x_k) > 0$ for all $x_i \in D_i$. We wish to optimize the function f with some required precision: suppose six decimal places for the variables' values is desirable.

It is clear that to achieve such precision each domain D_i should be cut into $(b_i - a_i) \cdot 10^6$ equal size ranges. Let us denote by m_i the smallest integer such that $(b_i - a_i) \cdot 10^6 \leq 2^{m_i} - 1$. Then, a representation having each variable x_i coded as a binary string of length m_i clearly satisfies the precision requirement. Additionally, the following formula interprets each such string:

$$x_i = a_i + decimal(1001...001_2) \cdot \frac{b_i - a_i}{2^{m_i} - 1},$$

where $decimal(string_2)$ represents the decimal value of that binary string.

Now, each chromosome (as a potential solution) is represented by a binary string of length $m = \sum_{i=1}^{k} m_i$; the first m_1 bits map into a value from the range $[a_1, b_1]$, the next group of m_2 bits map into a value from the range $[a_2, b_2]$, and so on; the last group of m_k bits map into a value from the range $[a_k, b_k]$.

To initialize a population, we can simply set some *pop_size* number of chromosomes randomly in a bitwise fashion. However, if we do have some knowledge about the distribution of potential optima, we may use such information in arranging the set of initial (potential) solutions.

The rest of the algorithm is straightforward: in each generation we evaluate each chromosome (using the function f on the decoded sequences of variables), select new population with respect to the probability distribution based on fitness values, and recombine the chromosomes in the new population by mutation and crossover operators. After some number of generations, when no further improvement is observed, the best chromosome represents an (possibly the global) optimal solution. Often we stop the algorithm after a fixed number of iterations depending on speed and resource criteria.

For the selection process (selection of a new population with respect to the probability distribution based on fitness values), a roulette wheel with slots sized according to fitness is used. We construct such a roulette wheel as follows (we assume here that the fitness values are positive, otherwise, we can use some scaling mechanism — this is discussed in Chapter 4):

- Calculate the fitness value $eval(\mathbf{v}_i)$ for each chromosome $\mathbf{v}_i$ ($i = 1, \ldots,$ *pop_size*).
- Find the total fitness of the population

$$F = \sum_{i=1}^{pop\_size} eval(\mathbf{v}_i).$$

- Calculate the probability of a selection p_i for each chromosome $\mathbf{v}_i$ ($i = 1, \ldots, pop\_size$):

$$p_i = eval(\mathbf{v}_i)/F.$$

- Calculate a cumulative probability q_i for each chromosome $\mathbf{v}_i$ ($i = 1, \ldots, pop\_size$):

$$q_i = \sum_{j=1}^{i} p_j.$$

The selection process is based on spinning the roulette wheel *pop_size* times; each time we select a single chromosome for a new population in the following way:

- Generate a random (float) number r from the range [0..1].
- If $r < q_1$ then select the first chromosome ($\mathbf{v}_1$); otherwise select the i-th chromosome $\mathbf{v}_i$ ($2 \leq i \leq pop\_size$) such that $q_{i-1} < r \leq q_i$.

Obviously, some chromosomes would be selected more than once. This is in accordance with the Schema Theorem (see next chapter): the best chromosomes get more copies, the average stay even, and the worst die off.

Now we are ready to apply the first recombination operator, crossover, to the individuals in the new population. As mentioned earlier, one of the parameters of a genetic system is probability of crossover p_c. This probability gives us the expected number $p_c \cdot pop\_size$ of chromosomes which undergo the crossover operation. We proceed in the following way:

For each chromosome in the (new) population:

- Generate a random (float) number r from the range [0..1];
- If $r < p_c$, select given chromosome for crossover.

Now we mate selected chromosomes randomly: for each pair of coupled chromosomes we generate a random integer number pos from the range $[1..m-1]$ (m is the total length — number of bits — in a chromosome). The number pos indicates the position of the crossing point. Two chromosomes

$$(b_1 b_2 \ldots b_{pos} b_{pos+1} \ldots b_m) \text{ and}$$
$$(c_1 c_2 \ldots c_{pos} c_{pos+1} \ldots c_m)$$

are replaced by a pair of their offspring:

$$(b_1 b_2 \ldots b_{pos} c_{pos+1} \ldots c_m) \text{ and}$$
$$(c_1 c_2 \ldots c_{pos} b_{pos+1} \ldots b_m).$$

The next recombination operator, mutation, is performed on a bit-by-bit basis. Another parameter of the genetic system, probability of mutation p_m, gives us the expected number of mutated bits $p_m \cdot m \cdot pop\_size$. Every bit (in all chromosomes in the whole population) has an equal chance to undergo mutation, i.e., change from 0 to 1 or vice versa. So we proceed in the following way.

For each chromosome in the current (i.e., after crossover) population and for each bit within the chromosome:

- Generate a random (float) number r from the range [0..1];
- If $r < p_m$, mutate the bit.

Following selection, crossover, and mutation, the new population is ready for its next evaluation. This evaluation is used to build the probability distribution (for the next selection process), i.e., for a construction of a roulette wheel with slots sized according to current fitness values. The rest of the evolution is just cyclic repetition of the above steps (see Figure 0.1 in the introduction).

The whole process is illustrated by an example. We run a simulation of a genetic algorithm for function optimization. We assume that the population size $pop\_size = 20$, and the probabilities of genetic operators are $p_c = 0.25$ and $p_m = 0.01$.

Let us assume also that we maximize the following function:

$$f(x_1, x_2) = 21.5 + x_1 \cdot \sin(4\pi x_1) + x_2 \cdot \sin(20\pi x_2),$$

where $-3.0 \leq x_1 \leq 12.1$ and $4.1 \leq x_2 \leq 5.8$. The graph of the function f is given in Figure 2.1.

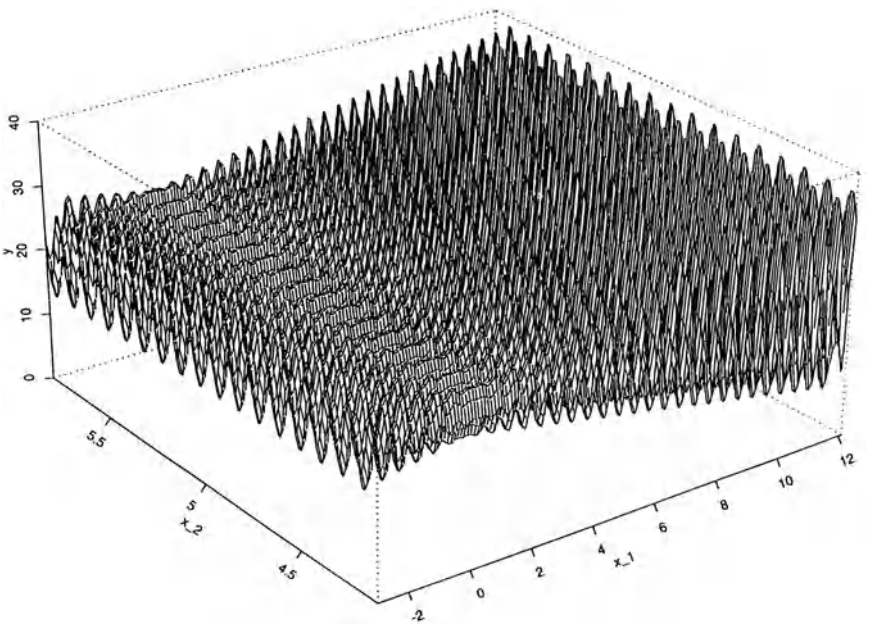

Fig. 2.1. Graph of the function $f(x_1, x_2) = 21.5 + x_1 \cdot \sin(4\pi x_1) + x_2 \cdot \sin(20\pi x_2)$

Let assume further that the required precision is four decimal places for each variable. The domain of variable x_1 has length 15.1; the precision requirement implies that the range $[-3.0, 12.1]$ should be divided into at least $15.1 \cdot 10000$ equal size ranges. This means that 18 bits are required as the first part of the chromosome:

$$2^{17} < 151000 \leq 2^{18}.$$

The domain of variable x_2 has length 1.7; the precision requirement implies that the range $[4.1, 5.8]$ should be divided into at least $1.7 \cdot 10000$ equal size ranges. This means that 15 bits are required as the second part of the chromosome:

$$2^{14} < 17000 \leq 2^{15}.$$

The total length of a chromosome (solution vector) is then $m = 18+15 = 33$ bits; the first 18 bits code x_1 and remaining 15 bits (19–33) code x_2.
Let us consider an example chromosome:

(010001001011010000111110010100010).

The first 18 bits,

010001001011010000,

represent $x_1 = -3.0 + decimal(010001001011010000_2) \cdot \frac{12.1-(-3.0)}{2^{18}-1} = -3.0 + 70352 \cdot \frac{15.1}{262143} = -3.0 + 4.052426 = 1.052426$.
The next 15 bits,

111110010100010,

represent $x_2 = 4.1 + decimal(111110010100010_2) \cdot \frac{5.8-4.1}{2^{15}-1} = 4.1 + 31906 \cdot \frac{1.7}{32767} = 4.1 + 1.655330 = 5.755330$.
So the chromosome

(010001001011010000111110010100010)

corresponds to $\langle x_1, x_2 \rangle = \langle 1.052426, 5.755330 \rangle$. The fitness value for this chromosome is

$f(1.052426, 5.755330) = 20.252640$.

To optimize the function f using a genetic algorithm, we create a population of $pop\_size = 20$ chromosomes. All 33 bits in all chromosomes are initialized randomly.

Assume that after the initialization process we get the following population:

$\mathbf{v}_1 = (100110100000001111111010011011111)$
$\mathbf{v}_2 = (111000100100110111001010100011010)$
$\mathbf{v}_3 = (000010000011001000001010111011101)$
$\mathbf{v}_4 = (100011000101101001111000001110010)$
$\mathbf{v}_5 = (000111011001010011010111111000101)$
$\mathbf{v}_6 = (000101000010010101001010111111011)$
$\mathbf{v}_7 = (001000100000110101111011011111011)$
$\mathbf{v}_8 = (100001100001110100010110101100111)$
$\mathbf{v}_9 = (010000000101100010110000001111100)$
$\mathbf{v}_{10} = (000001111000110000011010000111011)$
$\mathbf{v}_{11} = (011001111110110101100001101111000)$
$\mathbf{v}_{12} = (110100010111101101000101010000000)$
$\mathbf{v}_{13} = (111011111010001000110000001000110)$
$\mathbf{v}_{14} = (010010011000001010100111100101001)$
$\mathbf{v}_{15} = (111011101101110000100011111011110)$
$\mathbf{v}_{16} = (110011110000011111100001101001011)$
$\mathbf{v}_{17} = (011010111111001111010001101111101)$
$\mathbf{v}_{18} = (011101000000001110100111110101101)$
$\mathbf{v}_{19} = (000101010011111111110000110001100)$
$\mathbf{v}_{20} = (101110010110011110011000101111110)$

During the evaluation phase we decode each chromosome and calculate the fitness function values from (x_1, x_2) values just decoded. We get:

$$
\begin{aligned}
eval(\mathbf{v}_1) &= f(6.084492, 5.652242) = 26.019600\\
eval(\mathbf{v}_2) &= f(10.348434, 4.380264) = 7.580015\\
eval(\mathbf{v}_3) &= f(-2.516603, 4.390381) = 19.526329\\
eval(\mathbf{v}_4) &= f(5.278638, 5.593460) = 17.406725\\
eval(\mathbf{v}_5) &= f(-1.255173, 4.734458) = 25.341160\\
eval(\mathbf{v}_6) &= f(-1.811725, 4.391937) = 18.100417\\
eval(\mathbf{v}_7) &= f(-0.991471, 5.680258) = 16.020812\\
eval(\mathbf{v}_8) &= f(4.910618, 4.703018) = 17.959701\\
eval(\mathbf{v}_9) &= f(0.795406, 5.381472) = 16.127799\\
eval(\mathbf{v}_{10}) &= f(-2.554851, 4.793707) = 21.278435\\
eval(\mathbf{v}_{11}) &= f(3.130078, 4.996097) = 23.410669\\
eval(\mathbf{v}_{12}) &= f(9.356179, 4.239457) = 15.011619\\
eval(\mathbf{v}_{13}) &= f(11.134646, 5.378671) = 27.316702\\
eval(\mathbf{v}_{14}) &= f(1.335944, 5.151378) = 19.876294\\
eval(\mathbf{v}_{15}) &= f(11.089025, 5.054515) = 30.060205\\
eval(\mathbf{v}_{16}) &= f(9.211598, 4.993762) = 23.867227\\
eval(\mathbf{v}_{17}) &= f(3.367514, 4.571343) = 13.696165\\
eval(\mathbf{v}_{18}) &= f(3.843020, 5.158226) = 15.414128\\
eval(\mathbf{v}_{19}) &= f(-1.746635, 5.395584) = 20.095903\\
eval(\mathbf{v}_{20}) &= f(7.935998, 4.757338) = 13.666916
\end{aligned}
$$

It is clear, that the chromosome $\mathbf{v}_{15}$ is the strongest one, and the chromosome $\mathbf{v}_2$ the weakest.

Now the system constructs a roulette wheel for the selection process. The total fitness of the population is

$$F = \sum_{i=1}^{20} eval(\mathbf{v}_i) = 387.776822.$$

The probability of a selection p_i for each chromosome $\mathbf{v}_i$ $(i = 1, \ldots, 20)$ is:

$$
\begin{aligned}
p_1 &= eval(\mathbf{v}_1)/F = 0.067099 & p_2 &= eval(\mathbf{v}_2)/F = 0.019547\\
p_3 &= eval(\mathbf{v}_3)/F = 0.050355 & p_4 &= eval(\mathbf{v}_4)/F = 0.044889\\
p_5 &= eval(\mathbf{v}_5)/F = 0.065350 & p_6 &= eval(\mathbf{v}_6)/F = 0.046677\\
p_7 &= eval(\mathbf{v}_7)/F = 0.041315 & p_8 &= eval(\mathbf{v}_8)/F = 0.046315\\
p_9 &= eval(\mathbf{v}_9)/F = 0.041590 & p_{10} &= eval(\mathbf{v}_{10})/F = 0.054873\\
p_{11} &= eval(\mathbf{v}_{11})/F = 0.060372 & p_{12} &= eval(\mathbf{v}_{12})/F = 0.038712\\
p_{13} &= eval(\mathbf{v}_{13})/F = 0.070444 & p_{14} &= eval(\mathbf{v}_{14})/F = 0.051257\\
p_{15} &= eval(\mathbf{v}_{15})/F = 0.077519 & p_{16} &= eval(\mathbf{v}_{16})/F = 0.061549\\
p_{17} &= eval(\mathbf{v}_{17})/F = 0.035320 & p_{18} &= eval(\mathbf{v}_{18})/F = 0.039750\\
p_{19} &= eval(\mathbf{v}_{19})/F = 0.051823 & p_{20} &= eval(\mathbf{v}_{20})/F = 0.035244
\end{aligned}
$$

The cumulative probabilities q_i for each chromosome $\mathbf{v}_i$ $(i = 1, \ldots, 20)$ are:

$$
\begin{array}{llll}
q_1 = 0.067099 & q_2 = 0.086647 & q_3 = 0.137001 & q_4 = 0.181890 \\
q_5 = 0.247240 & q_6 = 0.293917 & q_7 = 0.335232 & q_8 = 0.381546 \\
q_9 = 0.423137 & q_{10} = 0.478009 & q_{11} = 0.538381 & q_{12} = 0.577093 \\
q_{13} = 0.647537 & q_{14} = 0.698794 & q_{15} = 0.776314 & q_{16} = 0.837863 \\
q_{17} = 0.873182 & q_{18} = 0.912932 & q_{19} = 0.964756 & q_{20} = 1.000000
\end{array}
$$

Now we are ready to spin the roulette wheel 20 times; each time we select a single chromosome for a new population. Let us assume that a (random) sequence of 20 numbers from the range [0..1] is:

0.513870	0.175741	0.308652	0.534534	0.947628
0.171736	0.702231	0.226431	0.494773	0.424720
0.703899	0.389647	0.277226	0.368071	0.983437
0.005398	0.765682	0.646473	0.767139	0.780237

The first number $r = 0.513870$ is greater than q_{10} and smaller than q_{11}, meaning the chromosome $\mathbf{v}_{11}$ is selected for the new population; the second number $r = 0.175741$ is greater than q_3 and smaller than q_4, meaning the chromosome $\mathbf{v}_4$ is selected for the new population, etc.

Finally, the new population consists of the following chromosomes:

$$
\begin{aligned}
\mathbf{v}'_1 &= (01100111111011010110000110111 1000)\ (\mathbf{v}_{11}) \\
\mathbf{v}'_2 &= (10001100010110100111100000111 0010)\ (\mathbf{v}_4) \\
\mathbf{v}'_3 &= (00100010000011010111101101111 1011)\ (\mathbf{v}_7) \\
\mathbf{v}'_4 &= (01100111111011010110000110111 1000)\ (\mathbf{v}_{11}) \\
\mathbf{v}'_5 &= (00010101001111111111000011000 1100)\ (\mathbf{v}_{19}) \\
\mathbf{v}'_6 &= (10001100010110100111100000111 0010)\ (\mathbf{v}_4) \\
\mathbf{v}'_7 &= (11101110110111000010001111101 1110)\ (\mathbf{v}_{15}) \\
\mathbf{v}'_8 &= (00011101100101001101011111100 0101)\ (\mathbf{v}_5) \\
\mathbf{v}'_9 &= (01100111111011010110000110111 1000)\ (\mathbf{v}_{11}) \\
\mathbf{v}'_{10} &= (00001000001100100000101011101 1101)\ (\mathbf{v}_3) \\
\mathbf{v}'_{11} &= (11101110110111000010001111101 1110)\ (\mathbf{v}_{15}) \\
\mathbf{v}'_{12} &= (01000000010110001011000000111 1100)\ (\mathbf{v}_9) \\
\mathbf{v}'_{13} &= (00010100001001010100101011111 1011)\ (\mathbf{v}_6) \\
\mathbf{v}'_{14} &= (10000110000111010001011010110 0111)\ (\mathbf{v}_8) \\
\mathbf{v}'_{15} &= (10111001011001111001100010111 1110)\ (\mathbf{v}_{20}) \\
\mathbf{v}'_{16} &= (10011010000000111111101001101 1111)\ (\mathbf{v}_1) \\
\mathbf{v}'_{17} &= (00000111100011000001101000011 1011)\ (\mathbf{v}_{10}) \\
\mathbf{v}'_{18} &= (11101111101000100011000000100 0110)\ (\mathbf{v}_{13}) \\
\mathbf{v}'_{19} &= (11101110110111000010001111101 1110)\ (\mathbf{v}_{15}) \\
\mathbf{v}'_{20} &= (11001111000001111110000110100 1011)\ (\mathbf{v}_{16})
\end{aligned}
$$

Now we are ready to apply the first recombination operator, crossover, to the individuals in the new population (vectors $\mathbf{v}'_i$). The probability of crossover $p_c = 0.25$, so we expect that (on average) 25% of chromosomes (i.e., 5 out of 20) undergo crossover. We proceed in the following way: for each chromosome

in the (new) population we generate a random number r from the range [0..1]; if $r < 0.25$, we select a given chromosome for crossover.

Let us assume that the sequence of random numbers is:

0.822951	0.151932	0.625477	0.314685	0.346901
0.917204	0.519760	0.401154	0.606758	0.785402
0.031523	0.869921	0.166525	0.674520	0.758400
0.581893	0.389248	0.200232	0.355635	0.826927

This means that the chromosomes $\mathbf{v}'_2$, $\mathbf{v}'_{11}$, $\mathbf{v}'_{13}$, and $\mathbf{v}'_{18}$ were selected for crossover. (We were lucky: the number of selected chromosomes is even, so we can pair them easily. If the number of selected chromosomes were odd, we would either add one extra chromosome or remove one selected chromosome — this choice is made randomly as well.) Now we mate selected chromosomes randomly: say, the first two (i.e., $\mathbf{v}'_2$ and $\mathbf{v}'_{11}$) and the next two (i.e., $\mathbf{v}'_{13}$ and $\mathbf{v}'_{18}$) are coupled together. For each of these two pairs, we generate a random integer number *pos* from the range [1..32] (33 is the total length — number of bits — in a chromosome). The number *pos* indicates the position of the crossing point. The first pair of chromosomes is

$$\mathbf{v}'_2 = (100011000|101101001111000001110010)$$
$$\mathbf{v}'_{11} = (111011101|101110000100011111011110)$$

and the generated number $pos = 9$. These chromosomes are cut after the 9th bit and replaced by a pair of their offspring:

$$\mathbf{v}''_2 = (100011000|101110000100011111011110)$$
$$\mathbf{v}''_{11} = (111011101|101101001111000001110010).$$

The second pair of chromosomes is

$$\mathbf{v}'_{13} = (000101000010010101 00|1010111111011)$$
$$\mathbf{v}'_{18} = (111011111010001000 11|0000001000110)$$

and the generated number $pos = 20$. These chromosomes are replaced by a pair of their offspring:

$$\mathbf{v}''_{13} = (000101000010010101 00|0000001000110)$$
$$\mathbf{v}''_{18} = (111011111010001000 11|1010111111011).$$

The current version of the population is:

$$\mathbf{v}'_1 = (011001111110110101100001101111000)$$
$$\mathbf{v}''_2 = (100011000101110000100011111011110)$$
$$\mathbf{v}'_3 = (001000100000110101111011011111011)$$
$$\mathbf{v}'_4 = (011001111110110101100001101111000)$$
$$\mathbf{v}'_5 = (000101010011111111110000110001100)$$
$$\mathbf{v}'_6 = (100011000101101001111000001110010)$$
$$\mathbf{v}'_7 = (111011101101110000100011111011110)$$

$\mathbf{v}'_8 = (00011101100101001101011111000101)$
$\mathbf{v}'_9 = (01100111111011010110000110111000)$
$\mathbf{v}'_{10} = (00001000001100100000010101110111011101)$
$\mathbf{v}''_{11} = (11101110110110100111100000111001 0)$
$\mathbf{v}'_{12} = (01000000010110001011000000011111 00)$
$\mathbf{v}''_{13} = (00010100001001010100000000010001 10)$
$\mathbf{v}'_{14} = (10000110000111010001011010110011 1)$
$\mathbf{v}'_{15} = (10111001011001111001100010111111 0)$
$\mathbf{v}'_{16} = (10011010000000111111101001101111 1)$
$\mathbf{v}'_{17} = (00000111100011000001101000011101 1)$
$\mathbf{v}''_{18} = (11101111101000100011101011111101 1)$
$\mathbf{v}'_{19} = (11101110110111000010001111101111 0)$
$\mathbf{v}'_{20} = (11001111000001111110000110100101 1)$

The next recombination operator, mutation, is performed on a bit-by-bit basis. The probability of mutation $p_m = 0.01$, so we expect that (on average) 1% of bits would undergo mutation. There are $m \times pop\_size = 33 \times 20 = 660$ bits in the whole population; we expect (on average) 6.6 mutations per generation. Every bit has an equal chance to be mutated, so, for every bit in the population, we generate a random number r from the range [0..1]; if $r < 0.01$, we mutate the bit.

This means that we have to generate 660 random numbers. In a sample run, 5 of these numbers were smaller than 0.01; the bit number and the random number are listed below:

Bit position	Random number
112	0.000213
349	0.009945
418	0.008809
429	0.005425
602	0.002836

The following table translates the bit position into chromosome number and the bit number within the chromosome:

Bit position	Chromosome number	Bit number within chromosome
112	4	13
349	11	19
418	13	22
429	13	33
602	19	8

This means that four chromosomes are affected by the mutation operator; one of the chromosomes (the 13th) has two bits changed.

The final population is listed below; the mutated bits are typed in boldface. We drop *primes* for modified chromosomes: the population is listed as new vectors $\mathbf{v}_i$:

$\mathbf{v}_1 = (01100111111011010110000110111 1000)$
$\mathbf{v}_2 = (10001100010111000010001111011110)$
$\mathbf{v}_3 = (00100010000011010111011011111011)$
$\mathbf{v}_4 = (01100111111**0**0101011000011011111000)$
$\mathbf{v}_5 = (00010101001111111111000011000 1100)$
$\mathbf{v}_6 = (10001100010110100111100000111 0010)$
$\mathbf{v}_7 = (11101110110111000010001111011110)$
$\mathbf{v}_8 = (00011101100101001101011111 1000101)$
$\mathbf{v}_9 = (01100111111011010110000110111 1000)$
$\mathbf{v}_{10} = (00001000001100100000101011101 1101)$
$\mathbf{v}_{11} = (11101110110110100**1**011000001110010)$
$\mathbf{v}_{12} = (01000000010110001011000000011111 00)$
$\mathbf{v}_{13} = (00010100001001010100**1**00001000111)$
$\mathbf{v}_{14} = (10000110000111010001011010110 0111)$
$\mathbf{v}_{15} = (10111001011001111001100010111 1110)$
$\mathbf{v}_{16} = (10011010000000111111101001101 1111)$
$\mathbf{v}_{17} = (00000111100011000001101000011 1011)$
$\mathbf{v}_{18} = (11101111101000100011101011111 1011)$
$\mathbf{v}_{19} = (11101110**0**10111000010001111011110)$
$\mathbf{v}_{20} = (11001111000001111110000110100 1011)$

We have just completed one iteration (i.e., one generation) of the **while** loop in the genetic procedure (Figure 0.1 from the introduction). It is interesting to examine the results of the evaluation process of the new population. During the evaluation phase we decode each chromosome and calculate the fitness function values from (x_1, x_2) values just decoded. We get:

$eval(\mathbf{v}_1) = f(3.130078, 4.996097) = 23.410669$
$eval(\mathbf{v}_2) = f(5.279042, 5.054515) = 18.201083$
$eval(\mathbf{v}_3) = f(-0.991471, 5.680258) = 16.020812$
$eval(\mathbf{v}_4) = f(3.128235, 4.996097) = 23.412613$
$eval(\mathbf{v}_5) = f(-1.746635, 5.395584) = 20.095903$
$eval(\mathbf{v}_6) = f(5.278638, 5.593460) = 17.406725$
$eval(\mathbf{v}_7) = f(11.089025, 5.054515) = 30.060205$
$eval(\mathbf{v}_8) = f(-1.255173, 4.734458) = 25.341160$
$eval(\mathbf{v}_9) = f(3.130078, 4.996097) = 23.410669$
$eval(\mathbf{v}_{10}) = f(-2.516603, 4.390381) = 19.526329$
$eval(\mathbf{v}_{11}) = f(11.088621, 4.743434) = 33.351874$
$eval(\mathbf{v}_{12}) = f(0.795406, 5.381472) = 16.127799$
$eval(\mathbf{v}_{13}) = f(-1.811725, 4.209937) = 22.692462$
$eval(\mathbf{v}_{14}) = f(4.910618, 4.703018) = 17.959701$
$eval(\mathbf{v}_{15}) = f(7.935998, 4.757338) = 13.666916$
$eval(\mathbf{v}_{16}) = f(6.084492, 5.652242) = 26.019600$

$$eval(\mathbf{v}_{17}) = f(-2.554851, 4.793707) = 21.278435$$
$$eval(\mathbf{v}_{18}) = f(11.134646, 5.666976) = 27.591064$$
$$eval(\mathbf{v}_{19}) = f(11.059532, 5.054515) = 27.608441$$
$$eval(\mathbf{v}_{20}) = f(9.211598, 4.993762) = 23.867227$$

Note that the total fitness of the new population F is 447.049688, much higher than total fitness of the previous population, 387.776822. Also, the best chromosome now ($\mathbf{v}_{11}$) has a better evaluation (33.351874) than the best chromosome ($\mathbf{v}_{15}$) from the previous population (30.060205).

Now we are ready to run the selection process again and apply the genetic operators, evaluate the next generation, etc. After 1000 generations the population is:

$$\mathbf{v}_1 = (11101111011001101110010101011011)$$
$$\mathbf{v}_2 = (11100110011000010001010101011000)$$
$$\mathbf{v}_3 = (11101111011101101110010101011011)$$
$$\mathbf{v}_4 = (11100110001000011000010101011001)$$
$$\mathbf{v}_5 = (11101111011101101110010101011011)$$
$$\mathbf{v}_6 = (11100110011000010000010001010001)$$
$$\mathbf{v}_7 = (11010110001001001000110001011000)$$
$$\mathbf{v}_8 = (11110110001000101000110101001001)$$
$$\mathbf{v}_9 = (11100110001001001000110001011001)$$
$$\mathbf{v}_{10} = (11101111011101101110010101011011)$$
$$\mathbf{v}_{11} = (11010110000001001000110001011000)$$
$$\mathbf{v}_{12} = (11010110001001001000110001011001)$$
$$\mathbf{v}_{13} = (11101111011101101110010101011011)$$
$$\mathbf{v}_{14} = (11100110011000010000010101011011)$$
$$\mathbf{v}_{15} = (11100110101011100101010011011001)$$
$$\mathbf{v}_{16} = (11100110011000010100010001010001)$$
$$\mathbf{v}_{17} = (11100110011000010000010101011011)$$
$$\mathbf{v}_{18} = (11100110011000010000010101011001)$$
$$\mathbf{v}_{19} = (11110110001000101000111000001001)$$
$$\mathbf{v}_{20} = (11100110011000010000010101011001)$$

The fitness values are:

$$eval(\mathbf{v}_1) = f(11.120940, 5.092514) = 30.298543$$
$$eval(\mathbf{v}_2) = f(10.588756, 4.667358) = 26.869724$$
$$eval(\mathbf{v}_3) = f(11.124627, 5.092514) = 30.316575$$
$$eval(\mathbf{v}_4) = f(10.574125, 4.242410) = 31.933120$$
$$eval(\mathbf{v}_5) = f(11.124627, 5.092514) = 30.316575$$
$$eval(\mathbf{v}_6) = f(10.588756, 4.214603) = 34.356125$$
$$eval(\mathbf{v}_7) = f(9.631066, 4.427881) = 35.458636$$
$$eval(\mathbf{v}_8) = f(11.518106, 4.452835) = 23.309078$$
$$eval(\mathbf{v}_9) = f(10.574816, 4.427933) = 34.393820$$
$$eval(\mathbf{v}_{10}) = f(11.124627, 5.092514) = 30.316575$$
$$eval(\mathbf{v}_{11}) = f(9.623693, 4.427881) = 35.477938$$

$$
\begin{aligned}
eval(\mathbf{v}_{12}) &= f(9.631066, 4.427933) = 35.456066 \\
eval(\mathbf{v}_{13}) &= f(11.124627, 5.092514) = 30.316575 \\
eval(\mathbf{v}_{14}) &= f(10.588756, 4.242514) = 32.932098 \\
eval(\mathbf{v}_{15}) &= f(10.606555, 4.653714) = 30.746768 \\
eval(\mathbf{v}_{16}) &= f(10.588814, 4.214603) = 34.359545 \\
eval(\mathbf{v}_{17}) &= f(10.588756, 4.242514) = 32.932098 \\
eval(\mathbf{v}_{18}) &= f(10.588756, 4.242410) = 32.956664 \\
eval(\mathbf{v}_{19}) &= f(11.518106, 4.472757) = 19.669670 \\
eval(\mathbf{v}_{20}) &= f(10.588756, 4.242410) = 32.956664
\end{aligned}
$$

However, if we look carefully at the progress during the run, we may discover that in earlier generations the fitness values of some chromosomes were better than the value 35.477938 of the best chromosome after 1000 generations. For example, the best chromosome in generation 396 had value of 38.827553. This is due to the stochastic errors of sampling — we discuss this issue in Chapter 4.

It is relatively easy to keep track of the best individual in the evolution process. It is customary (in genetic algorithm implementations) to store "the best ever" individual at a separate location; in that way, the algorithm would report the best value found during the whole process (as opposed to the best value in the final population).

3. GAs: Why Do They Work?

> Species do not evolve to perfection, but quite the contrary. The weak, in fact, always prevail over the strong, not only because they are in the majority, but also because they are the more crafty.
>
> Friedrich Nietzsche, *The Twilight of the Idols*

The theoretical foundations of genetic algorithms rely on a binary string representation of solutions, and on the notion of a schema (see e.g., [89]) — a template allowing exploration of similarities among chromosomes. A schema is built by introducing a *don't care* symbol ($\star$) into the alphabet of genes. A schema represents all strings (a hyperplane, or subset of the search space), which match it on all positions other than '$\star$'.

For example, let us consider the strings and schemata of the length 10. The schema ($\star$ 1 1 1 1 0 0 1 0 0) matches two strings

$$\{(0111100100), (1111100100)\},$$

and the schema ($\star$ 1 $\star$ 1 1 0 0 1 0 0) matches four strings:

$$\{(0101100100), (0111100100), (1101100100), (1111100100)\}.$$

Of course, the schema (1 0 0 1 1 1 0 0 0 1) represents one string only: (1001110001), and the schema ($\star\star\star\star\star\star\star\star\star\star$) represents all strings of length 10. It is clear that every schema matches exactly 2^r strings, where r is the number of *don't care* symbols '$\star$' in a schema template. On the other hand, each string of the length m is matched by 2^m schemata. For example, let us consider a string (1001110001). This string is matched by the following 2^{10} schemata:

(1 0 0 1 1 1 0 0 0 1)
($\star$ 0 0 1 1 1 0 0 0 1)
(1 $\star$ 0 1 1 1 0 0 0 1)
(1 0 $\star$ 1 1 1 0 0 0 1)
$\vdots$
(1 0 0 1 1 1 0 0 0 $\star$)

$$
\begin{array}{l}
(\star\star 0\,1\,1\,1\,0\,0\,0\,1) \\
(\star\,0\star 1\,1\,1\,0\,0\,0\,1) \\
\qquad\vdots \\
(1\,0\,0\,1\,1\,1\,0\,0\star\star) \\
(\star\star\star 1\,1\,1\,0\,0\,0\,1) \\
\qquad\vdots \\
(\star\star\star\star\star\star\star\star\star\star),
\end{array}
$$

Considering strings of the length m, there are in total 3^m possible schemata. In a population of size n, between 2^m and $n \cdot 2^m$ different schemata may be represented.

Different schemata have different characteristics. We have already noticed that the number of *don't care* conditions $\star$ in a schema determines the number of strings matched by the schema. There are two important schema properties, *order* and *defining length*; the Schema Theorem will be formulated on the basis of these properties.

The *order* of the schema S (denoted by $o(S)$) is the number of 0 and 1 positions, i.e., *fixed* positions (non-*don't care* positions), present in the schema. In other words, it is the length of the template minus the number of *don't care* ($\star$) symbols. The order defines the speciality of a schema. For example, the following three schemata, each of length 10,

$$
\begin{array}{l}
S_1 = (\star\star\star 0\,0\,1\star 1\,1\,0), \\
S_2 = (\star\star\star\star 0\,0\star\star 0\star), \\
S_3 = (1\,1\,1\,0\,1\star\star 0\,0\,1),
\end{array}
$$

have the following orders:

$$o(S_1) = 6,\ o(S_2) = 3, \text{ and } o(S_3) = 8,$$

and the schema S_3 is the most specific one.

The notion of the order of a schema is useful in calculating survival probabilities of the schema for mutations; we discuss it later in the chapter.

The *defining length* of the schema S (denoted by $\delta(S)$) is the distance between the first and the last fixed string positions. It defines the compactness of information contained in a schema. For example,

$$\delta(S_1) = 10 - 4 = 6,\ \delta(S_2) = 9 - 5 = 4, \text{ and } \delta(S_3) = 10 - 1 = 9.$$

Note that the schema with a single fixed position has a defining length of zero.

The notion of the defining length of a schema is useful in calculating survival probabilities of the schema for crossovers; we discuss it later in the chapter.

As discussed in the introduction (Figure 0.1), the simulated evolution process consists of four consecutively repeated steps:

$t \leftarrow t + 1$
select $P(t)$ from $P(t-1)$
recombine $P(t)$
evaluate $P(t)$

The first step ($t \leftarrow t + 1$) simply moves the evolution clock one unit further; during the last step (evaluate $P(t)$) we just evaluate the current population. The main phenomenon of the evolution process occurs in two remaining steps of the evolution cycle: selection and recombination. We discuss the effect of these two steps on the expected number of schemata represented in the population. We start with the selection step; we illustrate all formulae by a running example.

Let us assume, the population size $pop\_size = 20$, the length of a string (and, consequently, the length of a schema template) is $m = 33$ (as in the running example discussed in the previous chapter). Assume further that (at the time t) the population consists of the following strings:

$\mathbf{v}_1 = (100110100000001111111010011011111)$
$\mathbf{v}_2 = (111000100100110111001010100011010)$
$\mathbf{v}_3 = (000010000011001000001010111011101)$
$\mathbf{v}_4 = (100011000101101001111000001110010)$
$\mathbf{v}_5 = (000111011001010011010111111000101)$
$\mathbf{v}_6 = (000101000010010101001010111111011)$
$\mathbf{v}_7 = (001000100000110101111011011111011)$
$\mathbf{v}_8 = (100001100001110100010110101100111)$
$\mathbf{v}_9 = (010000000101100010110000001111100)$
$\mathbf{v}_{10} = (000001111000110000011010000111011)$
$\mathbf{v}_{11} = (011001111110110101100001101111000)$
$\mathbf{v}_{12} = (110100010111101101000101010000000)$
$\mathbf{v}_{13} = (111011111010001000110000001000110)$
$\mathbf{v}_{14} = (010010011000001010100111100101001)$
$\mathbf{v}_{15} = (111011101101110000100011111011110)$
$\mathbf{v}_{16} = (110011110000011111100001101001011)$
$\mathbf{v}_{17} = (011010111111001111010001101111101)$
$\mathbf{v}_{18} = (011101000000001110100111110101101)$
$\mathbf{v}_{19} = (000101010011111111110000110001100)$
$\mathbf{v}_{20} = (101110010110011110011000101111110)$

Let us denote by $\xi(S, t)$ the number of strings in a population at the time t, matched by schema S. For example, for a given schema

$$S_0 = (****1\ 1\ 1**********************),$$

$\xi(S_0, t) = 3$, since there are 3 strings, namely $\mathbf{v}_{13}$, $\mathbf{v}_{15}$, and $\mathbf{v}_{16}$, matched by the schema S_0. Note that the order of the schema S_0, $o(S_0) = 3$, and its defining length $\delta(S_0) = 7 - 5 = 2$.

Another property of a schema is its *fitness* at time t, $eval(S, t)$. It is defined as the average fitness of all strings in the population matched by the schema S. Assume there are p strings $\{\mathbf{v}_{i_1}, \ldots, \mathbf{v}_{i_p}\}$ in the population matched by a schema S at the time t. Then

$$eval(S, t) = \sum_{j=1}^{p} eval(\mathbf{v}_{i_j})/p.$$

During the selection step, an intermediate population is created: $pop\_size = 20$ single string selections are made. Each string is copied zero, one, or more times, according to its fitness. As we have seen in the previous chapter, in a single string selection, the string $\mathbf{v}_i$ has probability $p_i = eval(\mathbf{v}_i)/F(t)$ to be selected ($F(t)$ is the total fitness of the whole population at time t, $F(t) = \sum_{i=1}^{20} eval(\mathbf{v}_i)$).

After the selection step, we expect to have $\xi(S, t+1)$ strings matched by schema S. Since (1) for an average string matched by a schema S, the probability of its selection (in a single string selection) is equal to $eval(S,t)/F(t)$, (2) the number of strings matched by a schema S is $\xi(S,t)$, and (3) the number of single string selections is pop_size, it is clear that

$$\xi(S, t+1) = \xi(S,t) \cdot pop\_size \cdot eval(S,t)/F(t),$$

We can rewrite the above formula: taking into account that the average fitness of the population $\overline{F(t)} = F(t)/pop\_size$, we can write:

$$\xi(S, t+1) = \xi(S,t) \cdot eval(S,t)/\overline{F(t)}. \tag{3.1}$$

In other words, the number of strings in the population grows as the ratio of the fitness of the schema to the average fitness of the population. This means that an "above average" schema receives an increasing number of strings in the next generation, a "below average" scheme receives decreasing number of strings, and an average schema stays on the same level.

The long-term effect of the above rule is also clear. If we assume that a schema S remains above average by ϵ% (i.e., $eval(S,t) = \overline{F(t)} + \epsilon \cdot \overline{F(t)}$), then

$$\xi(S,t) = \xi(S,0)(1+\epsilon)^t,$$

and $\epsilon = (eval(S,t) - \overline{F(t)})/\overline{F(t)}$ ($\epsilon > 0$ for above average schemata and $\epsilon < 0$ for below average schemata).

This is a geometric progression equation: now we can say not only that an "above average" schema receives an increasing number of strings in the next generation, but that such a schema receives an *exponentially* increasing number of strings in the next generations.

We call the equation (3.1) the reproductive schema growth equation.

Let us return to the example schema, S_0. Since there are 3 strings, namely $\mathbf{v}_{13}$, $\mathbf{v}_{15}$, and $\mathbf{v}_{16}$ (at the time t) matched by the schema S_0, the fitness $eval(S_0)$ of the schema is

$$eval(S_0, t) = (27.316702 + 30.060205 + 23.867227)/3 = 27.081378.$$

At the same time, the average fitness of the whole population is

$$\overline{F(t)} = \sum_{i=1}^{20} eval(\mathbf{v_i})/pop\_size = 387.776822/20 = 19.388841,$$

and the ratio of the fitness of the schema S_0 to the average fitness of the population is

$$eval(S_0, t)/\overline{F(t)} = 1.396751.$$

This means that if the schema S_0 stays above average, then it receives an exponentially increasing number of strings in the next generations. In particular, if the schema S_0 stays above average by the constant factor of 1.396751, then, at time $t+1$, we expect to have $3 \times 1.396751 = 4.19$ strings matched by S_0 (i.e., most likely 4 or 5), at time $t+2$: $3 \times 1.396751^2 = 5.85$ such strings (i.e., very likely, 6 strings), etc.

The intuition is that such a schema S_0 defines a promising part of the search space and is being sampled in an exponentially increased manner.

Let us check these predictions on our running example for the schema S_0. In the population at the time t, the schema S_0 matched 3 strings, $\mathbf{v}_{13}$, $\mathbf{v}_{15}$, and $\mathbf{v}_{16}$. In the previous chapter we simulated the selection process using the same population. The new population consists of the following chromosomes:

$\mathbf{v}'_1$ = (01100111111011010110000110111000) ($\mathbf{v}_{11}$)
$\mathbf{v}'_2$ = (10001100010110100111100000111010) ($\mathbf{v}_4$)
$\mathbf{v}'_3$ = (00100010000011010111101101111011) ($\mathbf{v}_7$)
$\mathbf{v}'_4$ = (01100111111011010110000110111000) ($\mathbf{v}_{11}$)
$\mathbf{v}'_5$ = (00010101001111111111000011000100) ($\mathbf{v}_{19}$)
$\mathbf{v}'_6$ = (10001100010110100111100000111010) ($\mathbf{v}_4$)
$\mathbf{v}'_7$ = (11101110110111000010001111101110) ($\mathbf{v}_{15}$)
$\mathbf{v}'_8$ = (00011101100101001101011111000101) ($\mathbf{v}_5$)
$\mathbf{v}'_9$ = (01100111111011010110000110111000) ($\mathbf{v}_{11}$)
$\mathbf{v}'_{10}$ = (00001000001100100000101011011101) ($\mathbf{v}_3$)
$\mathbf{v}'_{11}$ = (11101110110111000010001111101110) ($\mathbf{v}_{15}$)
$\mathbf{v}'_{12}$ = (01000000010110001011000000011111100) ($\mathbf{v}_9$)
$\mathbf{v}'_{13}$ = (00010100001001010100101011111011) ($\mathbf{v}_6$)
$\mathbf{v}'_{14}$ = (10000110000111010001011010110011) ($\mathbf{v}_8$)
$\mathbf{v}'_{15}$ = (10111001011001111001100010111110) ($\mathbf{v}_{20}$)
$\mathbf{v}'_{16}$ = (11100110011000010100010001010001) ($\mathbf{v}_1$)
$\mathbf{v}'_{17}$ = (11100110011000010000010101011011) ($\mathbf{v}_{10}$)
$\mathbf{v}'_{18}$ = (11101111101000100011000000100011) ($\mathbf{v}_{13}$)
$\mathbf{v}'_{19}$ = (11101110110111000010001111101110) ($\mathbf{v}_{15}$)
$\mathbf{v}'_{20}$ = (11001111000001111110000101001011) ($\mathbf{v}_{16}$)

Indeed, the schema S_0 now (time $t+1$) matches 5 strings: $\mathbf{v}'_7$, $\mathbf{v}'_{11}$, $\mathbf{v}'_{18}$, $\mathbf{v}'_{19}$, and $\mathbf{v}'_{20}$.

However, selection alone does not introduce any new points (potential solutions) for consideration from the search space; selection just copies some strings to form an intermediate population. So the second step of the evolution cycle, recombination, takes the responsibility of introducing new individuals in the population. This is done by two genetic operators: crossover and mutation. We discuss the effect of these two operators on the expected number of schemata in the population in turn.

Let us start with crossover and consider the following example. As discussed earlier in the chapter, a single string from the population, say, $\mathbf{v}'_{18}$

(11101111101000100011000000100011 0),

is matched by 2^{30} schemata; in particular, the string is matched by these two schemata:

$S_0 = (****1\ 1\ 1 **********\ **********)$ and
$S_1 = (1\ 1\ 1 **********\ **********\ 1\ 0)$.

Let us assume further that the above string was selected for crossover (as happened in Chapter 2). Assume further (according to experiments from Chapter 2, where $\mathbf{v}'_{18}$ was crossed with $\mathbf{v}'_{13}$) that the generated crossing site $pos = 20$. It is clear that the schema S_0 survives such a crossover, i.e., one of the offspring still matches S_0. The reason is that the crossing site preserves the sequence '111' on the fifth, sixth, and seventh positions in the string in one of the offsprings: a pair

$\mathbf{v}'_{18} = (11101111101000100011|0000001000110)$,
$\mathbf{v}'_{13} = (00010100001001010100|1010111111011)$,

would produce

$\mathbf{v}''_{18} = (11101111101000100011|1010111111011)$,
$\mathbf{v}''_{13} = (00010100001001010100|0000001000110)$.

On the other hand, the schema S_1 would be destroyed: none of the offspring would match it. The reason is that the fixed positions '111' at the beginning of the template and the fixed positions '10' at the end are placed in different offspring.

It should be clear that the defining length of a schema plays a significant role in the probability of its destruction and survival. Note, that the defining length of the schema S_0 was $\delta(S_0) = 2$, and the defining length of the schema S_1 was $\delta(S_1) = 32$.

In general, a crossover site is selected uniformly among $m-1$ possible sites. This implies that the probability of destruction of a schema S is

$$p_d(S) = \frac{\delta(S)}{m-1},$$

and consequently, the probability of schema survival is

$$p_s(S) = 1 - \frac{\delta(S)}{m-1}.$$

Indeed, the probabilities of survival and destruction of our example schemata S_0 and S_1 are:

$$p_d(S_0) = 2/32,\ p_s(S_0) = 30/32,\ p_d(S_1) = 32/32 = 1,\ p_d(S_1) = 0,$$

so the outcome was predictable.

It is important to note that only some chromosomes undergo crossover and the selective probability of crossover is p_c. This means that the probability of a schema survival is in fact:

$$p_s(S) = 1 - p_c \cdot \frac{\delta(S)}{m-1}.$$

Again, referring to our example schema S_0 and the running example ($p_c = 0.25$):

$$p_s(S_0) = 1 - 0.25 \cdot \frac{2}{32} = 63/64 = 0.984375.$$

Note also that even if a crossover site is selected between fixed positions in a schema, there is still a chance for the schema to survive. For example, if *both* strings $\mathbf{v}'_{18}$ and $\mathbf{v}'_{13}$ started with '111' and ended with '10', the schema S_1 would survive crossover (however, the probability of such event is quite small). Because of that, we should modify the formula for the probability of schema survival:

$$p_s(S) \geq 1 - p_c \cdot \frac{\delta(S)}{m-1}.$$

So the combined effect of selection and crossover gives us a new form of the reproductive schema growth equation:

$$\xi(S, t+1) \geq \xi(S,t) \cdot eval(S,t)/\overline{F(t)} \left[1 - p_c \cdot \frac{\delta(S)}{m-1}\right]. \tag{3.2}$$

The equation (3.2) tells us about the expected number of strings matching a schema S in the next generation as a function of the actual number of strings matching the schema, relative fitness of the schema, and its defining length. It is clear that above-average schemata with short defining length would still be sampled at exponentially increased rates. For the schema S_0:

$$eval(S_0,t)/\overline{F(t)} \left[1 - p_c \cdot \frac{\delta(S)}{m-1}\right] = 1.396751 \cdot 0.984375 = 1.374927.$$

This means that the short, above-average schema S_0 would still receive an exponentially increasing number of strings in the next generations: at time $(t+1)$ we expect to have $3 \times 1.374927 = 4.12$ strings matched by S_0 (only slightly less than 4.19 — a value we got considering selection only), at time $(t+2)$: $3 \times 1.374927^2 = 5.67$ such strings (again, slightly less than 5.85).

The next operator to be considered is mutation. The mutation operator randomly changes a single position within a chromosome with probability p_m. The change is from zero to one or vice versa. It is clear that all of the fixed positions of a schema must remain unchanged if the schema survives mutation. For example, consider again a single string from the population, say, $\mathbf{v}'_{19}$:

(11101110110111000010001111011110)

and schema S_0:

$S_0 = (\star\star\star\star 1\ 1\ 1 \star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star)$.

Assume further that the string $\mathbf{v}'_{19}$ undergoes mutation, i.e., at least one bit is flipped, as happened in the previous chapter. (Recall also that four strings underwent mutation there: one of these strings, $\mathbf{v}'_{13}$, was mutated at two positions, three other strings — including $\mathbf{v}'_{19}$ — at one.) Since $\mathbf{v}'_{19}$ was mutated at the 8th position, its offspring,

$$\mathbf{v}''_{19} = (11101110\mathbf{0}10111000010001111011110)$$

still is matched by the schema S_0. If the selected mutation positions were from 1 to 4, or from 8 to 33, the resulting offspring would still be matched by S_0. Only 3 bits (fifth, sixth, and seventh — the fixed bit positions in the schema S_0) are "important": mutation of at least one of these bits would destroy the schema S_0. Clearly, the number of such "important" bits is equal to the order of the schema, i.e., the number of fixed positions.

Since the probability of the alteration of a single bit is p_m, the probability of a single bit survival is $1 - p_m$. A single mutation is independent from other mutations, so the probability of a schema S surviving a mutation (i.e., sequence of one-bit mutations) is

$$p_s(S) = (1 - p_m)^{o(S)}.$$

Since $p_m \ll 1$, this probability can be approximated by:

$$p_s(S) \approx 1 - o(S) \cdot p_m.$$

Again, referring to our example schema S_0 and the running example ($p_m = 0.01$):

$$p_s(S_0) \approx 1 - 3 \cdot 0.01 = 0.97.$$

The combined effect of selection, crossover, and mutation gives us a new form of the reproductive schema growth equation:

$$\xi(S, t+1) \geq \xi(S,t) \cdot eval(S,t)/\overline{F(t)} \left[1 - p_c \cdot \frac{\delta(S)}{m-1} - o(S) \cdot p_m\right]. \qquad (3.3)$$

As in the simpler forms (equations (3.1) and (3.2)), equation (3.3) tells us about the expected number of strings matching a schema S in the next generation as a function of the actual number of strings matching the schema, the relative fitness of the schema, and its defining length and order. Again, it is clear that above-average schemata with short defining length and low-order would still be sampled at exponentially increased rates.

For the schema S_0:

$eval(S_0, t)/\overline{F(t)} \left[1 - p_c \cdot \frac{\delta(S)}{m-1} - o(S_0) \cdot p_m\right] = 1.396751 \cdot 0.954375 = 1.333024.$

This means that the short, low-order, above-average schema S_0 would still receive an exponentially increasing number of strings in the next generations: at time $(t+1)$ we expect to have $3 \times 1.333024 = 4.00$ strings matched by S_0 (not much less than 4.19 — a value we got considering selection only, or than 4.12 — a value we got considering selections and crossovers), at time $(t+2)$: $3 \times 1.333024^2 = 5.33$ such strings (again, not much less than 5.85 or 5.67).

Note that equation (3.3) is based on the assumption that the fitness function f returns only positive values; when applying GAs to optimization problems

where the optimization function may return negative values, some additional mapping between optimization and fitness functions is required. We discuss these issues in the next chapter.

In summary, the growth equation (3.1) shows that selection increases the sampling rates of the above-average schemata, and that this change is exponential. The sampling itself does not introduce any new schemata (not represented in the initial $t = 0$ sampling). This is exactly why the crossover operator is introduced — to enable structured, yet random information exchange. Additionally, the mutation operator introduces greater variability into the population. The combined (disruptive) effect of these operators on a schema is not significant if the schema is short and low-order. The final result of the growth equation (3.3) can be stated as:

Theorem 1 (Schema Theorem.) *Short, low-order, above-average schemata receive exponentially increasing trials in subsequent generations of a genetic algorithm.*

An immediate result of this theorem is that GAs explore the search space by short, low-order schemata which, subsequently, are used for information exchange during crossover:

Hypothesis 1 (Building Block Hypothesis.) *A genetic algorithm seeks near-optimal performance through the juxtaposition of short, low-order, high-performance schemata, called the building blocks.*

As stated in [72]:

> "Just as a child creates magnificent fortresses through the arrangement of simple blocks of wood, so does a genetic algorithm seek near optimal performance through the juxtaposition of short, low-order, high performance schemata."

We have seen a perfect example of a building block through this chapter:

$$S_0 = (\star\star\star\star 1\ 1\ 1 \star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star).$$

S_0 is a short, low-order schema, which (at least in early populations) was also above average. This schema contributed towards finding the optimum.

Although some research has been done to prove this hypothesis [18], for most nontrivial applications we rely mostly on empirical results. During the last fifteen years many GAs applications were developed which supported the building block hypothesis in many different problem domains. Nevertheless, this hypothesis suggests that the problem of coding for a genetic algorithm is critical for its performance, and that such a coding should satisfy the idea of short building blocks.

Earlier in the chapter we stated that a population of *pop_size* individuals of length m processes at least 2^m and at most 2^{pop_size} schemata. Some of them are processed in a useful manner: these are sampled at the (desirable) exponentially

increasing rate, and are not disrupted by crossover and mutation (which may happen for long defining length and high-order schemata).

Holland [89] showed, that at least pop_size^3 of them are processed usefully — he has called this property an *implicit parallelism*, as it is obtained without any extra memory/processing requirements. It is interesting to note that in a population of *pop_size* strings there are many more than *pop_size* schemata represented. This constitutes possibly the only known example of a combinatorial explosion working to our advantage instead of our disadvantage.

In this chapter we have provided some standard explanations for why genetic algorithms work. Note, however, that the building block hypothesis is just an article of faith, which for some problems is easily violated. For example, assume that the two short, low-order schemata (this time, let us consider schemata of the total length of 11 positions):

$$S_1 = (1\ 1\ 1 \star\star\star\star\star\star\star\star) \text{ and}$$
$$S_2 = (\star\star\star\star\star\star\star\star\star 1\ 1)$$

are above average, but their combination

$$S_3 = (1\ 1\ 1 \star\star\star\star\star\star 1\ 1),$$

is much less fit than

$$S_4 = (0\ 0\ 0 \star\star\star\star\star\star 0\ 0).$$

Assume further that the optimal string is $s_0 = (1111111111111)$ (S_3 matches it). A genetic algorithm may have some difficulties in converging to s_0, since it may tend to converge to points like (00011111100). This phenomenon is called deception [18], [72]: some building blocks (short, low-order schemata) can mislead genetic algorithm and cause its convergence to suboptimal points.

Three approaches were proposed to deal with deception (see [73]). The first one assumes prior knowledge of the objective function to code it in an appropriate way (to get 'tight' building blocks). For example, prior knowledge about the objective function, and consequently about the deception, might result in a different coding, where the five bits required to optimize the function are adjacent, instead of being six positions apart.

The second approach uses the third genetic operator, *inversion*. Simple inversion is (like mutation) a unary operator: it selects two points within a string and inverts the order of bits between selected points, but remembering the bit's 'meaning'. This means that we have to identify bits in the strings: we do so by keeping bits together with a record of their original positions. For example, a string

$$s = ((1,0)(2,0)(3,0)|(4,1)(5,1)(6,0)(7,1)|(8,0)(9,0)(10,0)(11,1))$$

with two marked points, after inversion becomes

$$s' = ((1,0)(2,0)(3,0)|(7,1)(6,0)(5,1)(4,1)|(8,0)(9,0)(10,0)(11,1)).$$

A genetic algorithm with inversion as one of the operators searches for the best arrangements of bits for forming building blocks. For example, the desirable schema considered earlier

$$S_3 = (1\ 1\ 1 \star\star\star\star\star\star 1\ 1),$$

rewritten as

$$S_3 = ((1,1)(2,1)(3,1)(4,\star)(5,\star)(6,\star)(7,\star)(8,\star)(9,\star)(10,1)(11,1)),$$

might be regrouped (after successful inversion) into

$$S_3 = ((1,1)(2,1)(3,1)(11,1)(10,1)(9,\star)(8,\star)(7,\star)(6,\star)(5,\star)(4,\star)),$$

making an important building block. However, as stated in [73]:

> "An earlier study [70] argued that inversion — a unary operator — was incapable of searching efficiently for tight building blocks because it lacked the power of juxtaposition inherent in binary operators. Put another way, inversion is to orderings what mutation is to alleles: both fight the good fight against search-stopping lack of diversity, but neither is sufficiently powerful to search for good structures, allelic or permutational, on its own when good structures require epistatic interaction of the individual parts."

The third approach to fight the deception was proposed recently [73], [77]: a messy genetic algorithm (mGA). Since mGAs have other interesting properties as well, we discuss them briefly in the next chapter (Section 3).

4. GAs: Selected Topics

A man once saw a butterfly
struggling to emerge from
its cocoon, too slowly
for his taste, so he began
to blow on it gently. The
warmth of his breath speeded
up the process all right. But
what emerged was not a butterfly
but a creature with mangled
wings.

Anthony de Mello, *One Minute Wisdom*

GA theory provides some explanation why, for a given problem formulation, we may obtain convergence to the sought optimal point. Unfortunately, practical applications do not always follow the theory, with the main reasons being:

- the coding of the problem often moves the GA to operate in a different space than that of the problem itself,
- there is a limit on the hypothetically unlimited number of iterations, and
- there is a limit on the hypothetically unlimited population size.

One of the implications of these observations is the inability of GAs, under certain conditions, to find the optimal solutions; such failures are caused by a premature convergence to a local optimum. The premature convergence is a common problem of genetic algorithms and other optimization algorithms. If convergence occurs too rapidly, then the valuable information developed in part of the population is often lost. Implementations of genetic algorithms are prone to converge prematurely before the optimal solution has been found, as stated in [21]:

> "...While the performance of most implementations is comparable to or better than the performance of many other search techniques, it [GA] still fails to live up to the high expectations engendered by the theory. The problem is that, while the theory points to sampling

> rates and search behavior in the limit, any implementation uses a finite population or set of sample points. Estimates based on finite samples inevitably have a sampling error and lead to search trajectories much different from those theoretically predicted. This problem is manifested in practice as a premature loss of diversity in the population with the search converging to a sub-optimal solution."

Most of known research in this area relates to:

- the magnitude and kind of errors introduced by the sampling mechanism, and
- the characteristics of the function itself.

These two issues are closely related; however, we discuss them in turn (Sections 4.1 and 4.2). The last section presents some additional ideas for enhancing the genetic search.

4.1 Sampling mechanism

The first, and possibly the most recognized work, was due to DeJong [41] in 1975. He considered several variations of the simple selection presented in the previous chapter. The first variation, named the *elitist model*, enforces preserving the best chromosome. The second variation, the *expected value model*, reduces the stochastic errors of the selection routine. This is done by introducing a count for each chromosome $\mathbf{v}$, which is set initially to the $f(\mathbf{v})/\overline{f}$ value and decreased by 0.5 or 1 when the chromosome is selected for reproduction with crossover or mutation, respectively. When the chromosome count falls below zero, the chromosome is not available for selection any longer. In the third variation, the *elitist expected value model*, the first two variations are combined together. In the fourth model, the *crowding factor model*, a newly generated chromosome replaces an "old" one and the doomed chromosome is selected from those which resemble the new one.

In 1981 Brindle [23] considered some further modifications: *deterministic sampling*, *remainder stochastic sampling without replacement*, *stochastic tournament*, *remainder stochastic sampling with replacement*, and *stochastic tournament*. This study confirmed the superiority of some of these modifications over simple selection. In particular, the *remainder stochastic sampling with replacement* method, which allocates samples according to the integer part of the expected value of occurrences of each chromosome in a new population and where the chromosomes compete according to the fractional parts for the remaining places in the population, was the most successful one and adopted by many researchers as standard. In 1987 Baker [11] provided a comprehensive theoretical study of these modifications using some well defined measures, and also presented a new improved version called *stochastic universal sampling*. This

method uses a single wheel spin. This wheel, which is constructed in the standard way (Chapter 2), is spun with a number of equally spaced markers equal to the population size as opposed to a single one.

Other methods to sample a population are based on introducing artificial weights: chromosomes are selected proportionally to their rank rather than actual evaluation values (see e.g., [10], [192]). These methods are based on a belief that the common cause of rapid (premature) convergence is the presence of *super individuals*, which are much better than the average fitness of the population. Such super individuals have a large number of offspring and (due to the constant size of the population) prevent other individuals from contributing any offspring in next generations. In a few generations a super individual can eliminate desirable chromosomal material and cause a rapid convergence to (possibly local) optimum.

There are many methods to assign a number of offspring based on ranking. For example, Baker [10] took a user defined value, MAX, as the upper bound for the expected number of offspring, and a linear curve through MAX was taken such that the area under the curve equaled the population size. In that way we can easily determine the difference between expected numbers of offspring between "adjacent" individuals. For example, for MAX = 2.5 and pop_size = 100, the difference between expected numbers of offspring between "adjacent" individuals would be 0.025.

Another possibility is to take a user defined parameter q and define the nonlinear function:

$$prob(rank) = q(1-q)^{rank-1}.$$

The function returns the probability of an individual ranked in position $rank$ ($rank = 1$ means the best individual, $rank = pop\_size$ the worst one) to be selected in a single selection. For example, if $q = 0.04$ and $pop\_size = 50$, $prob(1) = 0.04$, $prob(2) = 0.04 \cdot 0.96 = 0.0384$, $prob(3) = 0.04 \cdot 0.96 \cdot 0.96 = .036864$, etc. Note that

$$\sum_{i=1}^{pop\_size} prob(i) = \sum_{i=1}^{pop\_size} q(1-q)^i \approx 1.$$

Such approaches, though shown to improve genetic algorithm behavior in some cases, has some apparent drawbacks. First, they put the responsibility on the user to decide when to use these mechanisms. Second, they ignore the information about the relative evaluations of different chromosomes. Third, they treat all cases uniformly, regardless of the magnitude of the problem. Finally, selection procedures based on ranking violate the Schema Theorem. On the other hand, as shown in some research studies [11], [192], they prevent scaling problems (discussed in the next section), they control better the selective pressure, and (coupled with one-at-a-time reproduction) they give the search a greater focus.

In [8] the authors discuss different categories of selection procedures. They divide selection procedures into *dynamic* and *static* methods — a static selection requires that selection probabilities remain constant between generations

(for example, ranking selection), whereas a dynamic selection does not have such a requirement (e.g., proportional selection). Another division of selection procedures is into *extinctive* and *preservative* methods — preservative selection requires non-zero selection probability for each individual, whereas extinctive selection does not. Extinctive selections are further divided into *left* and *right* selections: in left extinctive selection the best individuals are prevented from reproduction in order to avoid premature convergence due to super individuals (right selection does not). Additionally, some selection procedures are *pure* in the sense that parents are allowed to reproduce in one generation only (i.e., the life time of each individual is limited to one generation only regardless of its fitness). We shell return to extinctive, pure selections in Chapter 8, when we discuss evolution strategies and compare them with genetic algorithms. Some selections are *generational* in the sense that the set of parents is fixed until all offspring for the next generation are completely produced; in selections *on-the-fly* an offspring replaces its parent immediately. Some selections are *elitist* in the sense that some (or all) of the parents are allowed to undergo selection with their offspring — we have already seen such selection in the elitist model [41].

It seems that there are two important issues in the evolution process of the genetic search: population diversity and selective pressure. These factors are strongly related: an increase in the selective pressure decreases the diversity of the population, and vice versa. In other words, strong selective pressure "supports" the premature convergence of the GA search; a weak selective pressure can make the search ineffective. Thus it is important to strike a balance between these two factors. As observed by Whitley [192]:

> "It can be argued that there are only two primary factors (and perhaps only two factors) in genetic search: population diversity and selective pressure [...] In some sense this is just another variation on the idea of exploration versus exploitation that has been discussed by Holland and others. Many of the various parameters that are used to 'tune' genetic search are really indirect means of affecting selective pressure and population diversity. As selective pressure is increased, the search focuses on the top individuals in the population, but because of this 'exploitation' genetic diversity is lost. Reducing the selective pressure (or using larger population) increases 'exploration' because more genotypes and thus more schemata are involved in the search."

In most of the experiments discussed in this volume, we used a new, two–step variation of the basic selection algorithm. However, this modification is not just a new selection mechanism; it can use any of the sampling methods devised so far and is itself designed to decrease the (possible) undesirable influence of some functions' characteristics. It falls into the category of dynamic, preservative, generational, and elitist selection.

The structure of the modified genetic algorithm (modGA) is shown in Figure 4.1. The modification with respect to the classical genetic algorithm is that

in the modGA we do not perform the selection step "select $P(t)$ from $P(t-1)$", but rather we select independently r (not necessarily distinct) chromosomes for reproduction and r (distinct) chromosomes to die. These selections are performed with respect to the relative fitness of the strings: a string with a better than average performance has a higher chance to be selected for reproduction; strings with a worse than average performance have higher chances to be selected to die. After the "select-parents" and "select-dead" steps of the modGA are performed, there are three (not necessarily disjoint) groups of strings in the population:

- r (not necessarily distinct) strings to reproduce (parents),
- precisely r strings to die (dead), and
- the remaining strings, called neutral strings.

The number of neutral strings in a generation (at least $pop\_size - 2r$ and at most $pop\_size - r$) depends on the number of selected distinct parents and on the number of overlapping strings in categories "parents" and "dead". Then a new population $P(t+1)$ is formed, consisting of the $pop\_size - r$ strings (all strings except these selected to die) and r offspring of the r parents.

```
procedure modGA
begin
  t ← 0
  initialize P(t)
  evaluate P(t)
  while (not termination-condition) do
  begin
    t ← t + 1
    select-parents from P(t − 1)
    select-dead from P(t − 1)
    form P(t): reproduce the parents
    evaluate P(t)
  end
end
```

Fig. 4.1. The algorithm modGA

As presented, the algorithm has a potentially problematic step: how to select the r chromosomes to die. Obviously, we wish to perform this selection in such a way that stronger chromosomes have smaller chances of dying. We achieved this by changing the method of forming the new population $P(t+1)$ to the following one:

step 1: Select r parents from $P(t)$. Each selected chromosome (or rather each of selected copies of some chromosomes) is marked as applicable to exactly one fixed genetic operation.

step 2: Select $pop\_size - r$ distinct chromosomes from $P(t)$ and copy them to $P(t+1)$.

step 3: Let r parent chromosomes breed to produce exactly r offspring.

step 4: Insert these r new offspring into population $P(t+1)$.

The above selections (steps 1 and 2) are done according to the chromosomes' fitness (stochastic universal sampling method).

There are a few important differences between different selection routines discussed earlier and the one described above. Firstly, both parent and offspring have a very good chance to be present in a new generation: an above average individual has a good chances to be selected as a parent (step 1) and, in the same time, to be selected in a new population of $pop\_size - r$ elements (step 2). If so, one (or more) of its offspring would take some of the remaining r positions. Secondly, we apply genetic operators on whole individuals as opposed to individual bits (classical mutation). This would provide an uniform treatment of all operators used in evolution program (an evolution program, GENOCOP, uses six "genetic" operators; see Chapter 7). So, if three operators are used (e.g., mutation, crossover, inversion), some of the parents would undergo mutation, some others crossover, and the rest inversion.

The modified approach (modGA) enjoys similar theoretical properties as the classical genetic algorithm. We can rewrite the growth equation (3.3) from Chapter 3 as:

$$\xi(S, t+1) \geq \xi(S, t) \cdot p_s(S) \cdot p_g(S), \tag{4.1}$$

where $p_s(S)$ represents the probability of the survival of the schema S and $p_g(S)$ represents the probability of the growth of the schema S. The growth of the schema S happens during the selection stage (growing phase) where several copies of above-average schemata are copied into a new population. The probability $p_g(S)$ of the growth of the schema S, $p_g(S) = eval(S,t)/\overline{F(t)}$, and $p_g(S) > 1$ for better-than-average schemata. Then the selected chomosomes must survive the genetic operators crossover and mutation (shrinking phase). As discussed in Chapter 3, the probability $p_s(S)$ of survival of the schema S,

$$p_s(S) = 1 - p_c\frac{\delta(S)}{m-1} - p_m \cdot o(S) < 1.$$

Formula (4.1) implies that for short, low-order schemata, $p_g(S) \cdot p_s(S) > 1$; because of this, such schemata receive an exponentially increasing number of trials in subsequent generations. The same holds for the modified (modGA) version of the algorithm. The expected number of chromosomes of the schema S in the modGA algorithm is also a product of the number of chromosomes in the old population $\xi(S,t)$, the probability of survival ($p_s(S) < 1$), and the probability of the growth $p_g(S)$ — the only difference is in interpretation of growing and shrinking phases and their relative order. In the modGA version, the shrinking phase is the first one: $n - r$ chromosomes are selected for the new population. The probability of survival is defined as a fraction of chromosomes of the schema S which were not selected to die. The growing phase is next and

is manifested in the arrival of r new offspring. The probability of the growth $p_g(S)$ of the schema S is a probability that the schema S expands by a new offspring generated from the r parents. Again, for short, low-order schemata, $p_s(S) \cdot p_g(S) > 1$ holds and such schemata receive exponentially increasing trials in subsequent generations.

One of the ideas of the modGA algorithm is a better utilization of the available storage resource: population size. The new algorithm avoids leaving exact multiple copies of the same chromosome in the new populations (which may still happen accidently by other means but is very unlike). On the other hand, the classical algorithm is quite vulnerable to creation of such multiple copies. Moreover, such multi-occurrences of super individuals create a possibility for a chain reaction: there is an chance for an even larger number of such exact copies in the next population, etc. This way the already limited population size can actually represent only a decreasing number of unique chromosomes. Lower space utilization decreases the performance of the algorithm; note that the theoretical foundations of genetic algorithms assume infinite population size. In the modGA algorithm we may have a number of family members for a chromosome, but all such members are different (by a family we mean offspring of the same parent).

As an example consider a chromosome with an expected value of appearances in $P(t+1)$ equal $p = 3$. Also assume that the classical genetic algorithm has probability of crossover and mutation $p_c = 0.3$ and $p_m = 0.003$, a rather usual scenario. Following the selection, there will be exactly $p = 3$ copies of this chromosome in $P(t+1)$ before reproduction. After reproduction, assuming chromosome length $m = 20$, the expected number of exact copies of this chromosome remaining in $P(t+1)$ will be $p \cdot (1 - p_c - p_m \cdot m) = 1.92$. Therefore, it is safe to say that the next population will have two exact copies of such a chromosome, reducing the number of different chromosomes.

Additional feature of modGA is a better time complexity of the whole algorithm in comparison with the classical genetic algorithm. In the latter, mutation works at bit level, so we have to generate a random number for every bit in a population. Sometimes the length of a chromosome is quite significant (for example, in experiments reported in [123], a chromosome consisted of 100,000 bits!) and consequently the performance of a genetic algorithm is quite poor.

The modification used in the modGA is based on the idea of the crowding factor model [41], where a newly generated chromosome replaces some old one. But the difference is that in the crowding factor model the dying chromosome is selected from those which resemble the new one, whereas in the modGA the dying chromosomes are those with lower fitness.

The modGAs, for small values of the parameter r, belong to a class of Steady State GAs (SSGA) [191], [182]; the main difference between GAs and SSGAs is that in the latter only few members of the population are changed (within each generation). There is also some similarity between the modGA and classifier systems (Chapter 12): a genetic component of a classifier system changes the population as little as possible. In the modGA we can regulate such

a change using the parameter r, which determines the number of chromosomes to reproduce and the number of chromosomes to die. In the modGA, $pop\_size - r$ chromosomes are placed in a new population without any change. In particular, for $r = 1$, only one chromosome is replaced in each generation.

4.2 Characteristics of the function

The modGA algorithm provides a new mechanism for forming a new population from the old one. However, it seems that some additional measures might be helpful in fighting problems related to the characteristic of the function being optimized. Over the years we have seen three basic directions. One of them borrows the simulated annealing technique of varying the system's entropy, (see e.g., [170], where the authors control the rate of population convergence by thermodynamic operators, which use a global temperature parameter).

Another direction is based on allocation of reproductive trials according to rank rather than actual evaluation values (as discussed in the previous section), since ranking automatically introduces a uniform scaling across the population.

The last direction concentrates on trying to fix the function itself by introducing a scaling mechanism. Following Goldberg [72, pp. 122–124] we divide such mechanisms into three categories:

1. *Linear Scaling.* In this method the actual chromosomes' fitness is scaled as

   $$f_i' = a * f_i + b.$$

   The parameters a, b are normally selected so that the average fitness is mapped to itself and the best fitness is increased by a desired multiple of the average fitness. This mechanism, though quite powerful, can introduce negative evaluation values that must be dealt with. In addition, the parameters a, b are normally fixed for the population life and are not problem dependent.

2. *Sigma Truncation.* This method was designed as an improvement of linear scaling both to deal with negative evaluation values and to incorporate problem dependent information into the mapping itself. Here the new fitness is calculated according to:

   $$f_i' = f_i + (\overline{f} - c * \sigma),$$

   where c is chosen as a small integer and σ is the population's standard deviation; possible negative evaluations f' are set to zero.

3. *Power Law Scaling.* In this method the initial fitness is taken to some specific power:

   $$f_i' = f_i^k,$$

with some k close to one. The parameter k scales the function f; however, in some studies [62] it was concluded that the choice of k should be problem dependent. In the same study the author used $k = 1.005$ to obtain some experimental improvements.

In most of our experiments we used a method based on *power scaling*. However, we not only define k as a dynamic problem dependent parameter, but we also incorporate another desirable parameter, the *age of population*.

The most noticeable problem associated with the characteristic of the function under consideration involves differences in relative fitness. As an example consider two functions: $f_1(x)$ and $f_2(x) = f_1(x) + const$. Since they are both basically the same (i.e., they share the same optima), one would expect that both can be optimized with similar degree of difficulty. However, if $const \gg \overline{f_1}(x)$, then the function $f_2(x)$ will suffer from (or enjoy) much slower convergence than the function $f_1(x)$. In fact, in the extreme case, the second function will be optimized using a totally random search; such a behavior may be tolerable during the very early life of the population but would be devastating later on. Conversely, $f_1(x)$ might be converging too fast, pushing the algorithm into a local optimum.

In addition, due to the fixed size of the population, the behavior of a GA may be different from run to run — this is caused by errors of finite sampling. Consider a function $f_3(x)$ with a sample $x_i^t \in P(t)$ close to some local optimum and $f(x_i^t)$ much greater than the average fitness $\overline{f}(x^t)$ (i.e., x_i^t is a super individual). Furthermore, assume that there is no x_j^t close to the sought global maximum. This might be the case for a highly non–smooth function. In such a case, there is a fast convergence towards that local optimum. Because of that, the population $P(t+1)$ becomes over–saturated with elements close to that solution, decreasing the chance of a global exploration needed to search for other optima. While such a behavior is permissible at the later evolutionary stages, and even desired at the very final stages, it is quite disturbing at the early ones. Moreover, normally late populations (during late stages of the algorithm) are saturated with chromosomes of similar fitness as all of those are closely related (by the mating processes). Therefore, using the traditional selective techniques the sampling actually becomes random. Such a behavior is exactly the opposite of the most desirable one, where there is a decreased influence of relative chromosomes fitness on the selection process during the initial stages of population life and increased influence at late stages.

To deal with such problems it is helpful to provide for two factors:

- an a priori measure of possible problematic characteristics of the function, and
- the incorporation of such a measure into the sampling mechanism to accommodate for the anticipated problems.

We are mostly interested in some average function behavior with respect to its mean. However, we know such behavior only through finite sampling represented in the population. Therefore, we define such a measure using statistical

random sample variance and mean (we call such a measure a *span*):

$$s_t = \frac{\sqrt{\frac{1}{n-1}\sum_{i=1}^{n}\left(f(x_i^t) - \overline{f}(x^t)\right)^2}}{\frac{1}{n}\sum_{i=1}^{n} f(x_i^t)}$$

This measure can be rewritten as:

$$s_t = \sqrt{\frac{n^2 \sum_{i=1}^{n} f(x_i^t)^2}{(n-1)\cdot(\sum_{i=1}^{n} f(x_i^t))^2} - \frac{2n-1}{n-1}}$$

Genetic algorithm applications normally require that all chromosomes evaluate to nonnegative numbers; such a requirement relates to the sampling mechanism and can be easily enforced. In such a case $max(s_t) = \sqrt{n-1}$ since $\sum_i^n f_i^2 \leq (\sum_i^n f_i)^2$. However, such a maximal value is obtained only when all but one evaluations are zero — a very unlikely case. Experiments show that most *spans* are much less than one. Moreover, we have determined experimentally that $s^* = 0.1$ gives a good trade-off between the space exploration and the speed of convergence; therefore, one should treat all cases with respect to the relationship between s_t and s^*. To improve the efficiency we do not want to calculate s_t at each iteration. This is actually not necessary as this measure finds the function's characteristics determinable from a random sampling. We have determined experimentally that very often the initial population provides a very good approximation. However, if desired, such sampling may be performed for a desired time before the algorithm actually starts iterating (we call such *span* s_0).

For experimenting with our GA modifications we decided to use a single family of functions:

$$f(x) = a + \sum_{i=1}^{\#elems} \sin(x_i)\cdot \sin^{2m}\left(\frac{i\cdot x_i^2}{\pi}\right),$$

which is a sine modulation of a (*#elems*)–dimensional function of components of nonlinearly increasing frequency. All $x_i \in (0,\pi)$ for $i = 1,\ldots,\#elem$. This function, given appropriate parameters a and m, simulates functions of totally different characteristics:

A: $a = 0.1$, $m = 1$. This function, even though nontrivial, exhibits rather nice characteristics: its average *span* $s_0 \simeq 0.1$ matches the most desired s^*.

B: $a = 0.1$, $m = 250$. This function is difficult to analyze numerically due to its high non–smoothness. This characteristic is captured in its $s_0 \simeq 1.0$. Because of this, any numerical method (including GA) will tend to fall to false optima.

C: $a = 1000$, $m = 1$. This function is a constant transformation of the **A** function. Such a shift causes the average *span* to be decreased dramatically to $s_0 \simeq 0.001$.

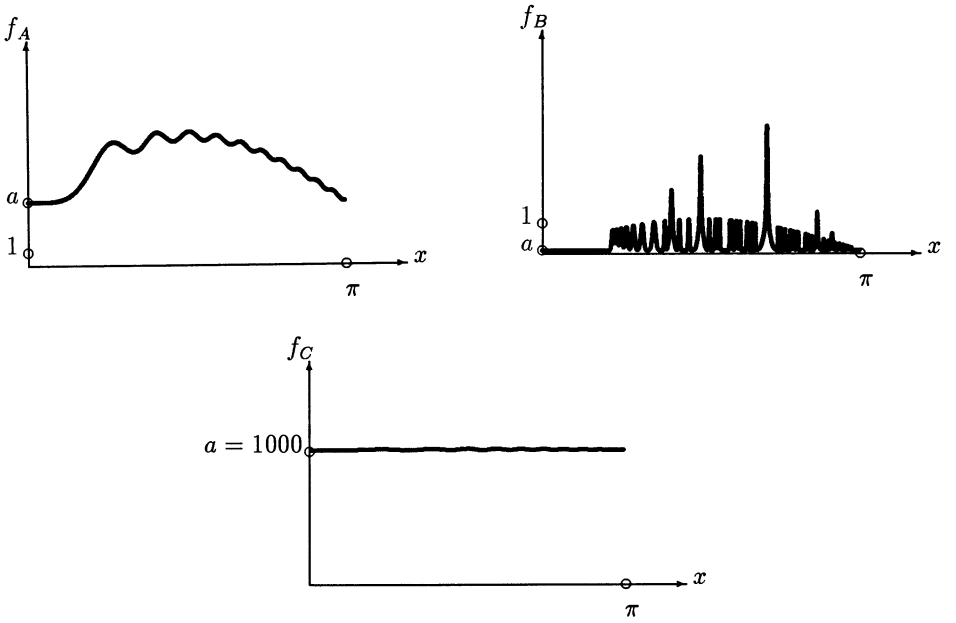

Fig. 4.2. Cross–sections of functions **A**, **B**, and **C**

For these particular experiments we used $\#elems = 10$. To visualize better the characteristics of these functions, their cross–sections are given in Figure 4.2.

For the construction of our scaling mechanism, which we call a *non–uniform scaling*, we use two dynamic parameters: *span* s and *population age* t (this parameter is taken to be the iteration number of the algorithm). What we seek is a mechanism which produces more random search for small *ages* and increasingly selects only the best chromosomes at late stages of the population life. Moreover, the net effect of such a mechanism should depend on the *span*; higher *span* should cause the whole mechanism to put less emphasis on chromosomes' fitness, somehow randomizing the sampling. The scaling itself uses the *power law scaling*:

$$f_i' = (f_i)^k$$

where $k \sim 0$ forces a random search, while $k > 1$ forces sampling allocated to only the fittest chromosomes. We construct k so that it changes from small to large values over the *population age* (with largest change very early and very late), and k's magnitude is larger and the speed of change smaller for problems with lower *span*. The following equation satisfies these goals:

$$k = \left(\frac{s^*}{s_0}\right)^{p_1} \cdot \tan^{p_2 \cdot (s_0/s^*)^\alpha}\left(\frac{t}{T+1} \cdot \frac{\pi}{2}\right),$$

where p_1 is a system parameter determining the influence of *span* s on the magnitude of k; p_2 and α are system parameters determining the speed of

changes of k. For all experiments we used the following parameter setting: $\alpha = 0.1$ $p1 = 0.05$ $p2 = 0.1$ $s^* = 0.1$. The values of the parameter k as a function of t are displayed in Figure 4.3.

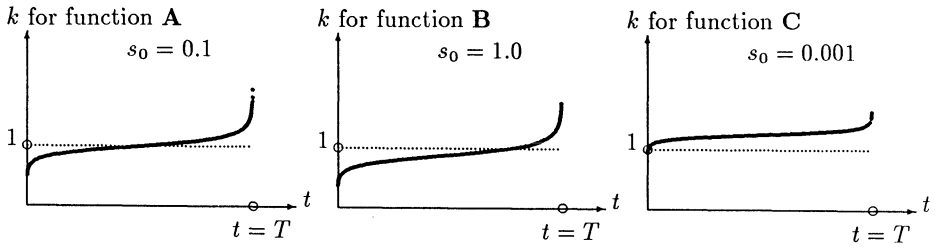

Fig. 4.3. k values for functions **A**, **B**, and **C**

To judge the quality of our two implementations (modGA and modGA-S with the *non-uniform scaling* mechanism turned on) we used a classical GA implementation as a control algorithm. All three algorithms used the same sampling mechanism (*remainder stochastic with replacement*) and the same binary representation, 30 bits to code one variable; therefore, for the used $\#elems = 10$ the length of the chromosomes was 300. Furthermore, for the best objective comparison, we used exactly the same operators (mutation and single–point crossover only) applied with the same frequency, the same population size ($pop\_size = 50$), and the same number of iterations (5000). Finally, we decided on the following two measures of the algorithm's performance:

- *Accuracy* — defined as the value of the best found chromosome relative to the value of the global optimum found in a separate GA run with extended resources:

$$\frac{f^{best} - a}{f^{global} - a}$$

  We subtract the a parameter so that we can easily compare the results of different functions.

- *Imprecision* — defined as the standard deviation of the optimal vector found. This measure evaluates the closeness of individual components selected in the found optimal chromosome as follows:

$$\sqrt{\frac{\sum_{i=1}^{\#elems}(index_i)^2}{\#elems - 1}},$$

  where $index_i$ is the index of the $i-th$ component of the found optimal chromosome (each dimension has its peaks ordered from 0 to the number

of dimensions minus one according to the modulated values of its peaks). For example, *imprecision*=0 means that the generated chromosome correctly selected the best modulated values along all dimensions; however, it does not have to generate the best function value due to local imperfectness associated with finite iteration time.

The results are summarized in Table 4.1; they represent the average of twenty-five independent runs. As expected, all systems performed quite worse (*accuracy*) while optimizing function **B**. Moreover, high *imprecision* indicates that the search was progressing in the wrong directions. Among the other two functions, function **A** was optimized slighly better on the average, but with a higher *imprecision*. This result indicates that function **C** was being searched in better directions, but with insufficient time available; given a longer time better *precision* should generate better *accuracy*. The reason this function was explored in better directions was that the high bias value (a) was prohibiting any premature convergence.

Across different systems, all results indicate better performance of the new GA algorithm modGA than the classical one. While such improvements were anticipated for difficult functions **A** and **B**, modGA's superior behavior on function **C** came as a warm surprise. In addition, the *non-uniform power scaling* (system modGA-S) improved all results of the new GA in terms of both *accuracy* and *imprecision*. These results indicate that such a mechanism improves the GA performance for all function characteristics: higher *accuracy* means that better local tuning was performed (due to increased k at late search stages) while lower *imprecision* means that fewer faulty convergences were obtained.

As to the increased computational complexity of calculating the k parameter, it was rather an insignificant change for the following reasons:

- The span s capturing the characteristics of the optimized function was calculated only once at iteration $t = 0$. Moreover, to increase the quality of this measure approximated by the final sampling, it was performed without the population size restriction; the number of such samples was set to 200.

- The formula can be partially rewritten to account for the constant and incremental parts of it.

Actually, the average CPU usage of the algorithm modGA-S was lower than that of the classical one due to the fact that the algorithm modGA itself can be implemented more efficiently than the traditional one.

4.3 Other ideas

In the previous sections of this chapter we have discussed some issues connected with removing possible errors in sampling mechanisms. The basic aim of that research was to enhance genetic search; in particular, to fight the premature

	Accuracy [%]			*Imprecision*		
function	Classical	modGA	modGA-S	Classical	modGA	modGA-S
A	96.05	96.61	97.18	1.049	0.974	0.847
B	74.83	83.17	83.64	2.243	2.108	2.035
C	95.71	96.47	97.58	0.873	0.876	0.870

Table 4.1. Output summary for the three test functions

convergence of GAs. During the last few years there have been some other research efforts in the quest for better schema processing, but using different approaches. In this section we discuss some of them.

The first direction is related to the genetic operator, crossover. This operator was inspired by a biological process; however, it has some drawbacks. For example, assume there are two high performance schemata:

$S_1 = (0\ 0\ 1 \star\star\star\star\star\star\star\star\ 0\ 1)$ and
$S_2 = (\star\star\star\star 1\ 1 \star\star\star\star\star\star\star)$.

There are also two strings in a population, $\mathbf{v}_1$ and $\mathbf{v}_2$, matched by S_1 and S_2, respectively:

$\mathbf{v}_1 = (\mathbf{001}0001101\mathbf{001})$ and
$\mathbf{v}_2 = (1110\mathbf{11}0001000)$.

Clearly, the crossover cannot combine certain combinations of features encoded on chromosomes; it is impossible to get a string to be matched by a schema

$S_3 = (0\ 0\ 1 \star 1\ 1 \star\star\star\star\star 0\ 1)$,

since the first schema will be destroyed.

Additional argument against the classical, one-point crossover is in some asymmetry between mutation and crossover: a mutation depends on the length of the chromosome and crossover does not. For example, if the probability of mutation is $p_m = 0.01$, and the length of a chromosome is 100, the expected number of mutated bits within the chromosome is one. If the length of a chromosome is 1000, the expected number of mutated bits within the chromosome is ten. On the other hand, in both cases, one-point crossover combines two strings on the basis of one cross site, regardless the length of the strings.

Some researchers (see, e.g., [52] or [182]) have experimented with other crossovers. For example, two-point crossover selects two crossing sites and chromosomal material is swapped between them. Clearly, strings $\mathbf{v}_1$ and $\mathbf{v}_2$ may produce a pair of offspring

$\mathbf{v}_1' = (\mathbf{001}|\mathbf{11}100|01\mathbf{001})$ and
$\mathbf{v}_2' = (111|00011|01000)$,

where $\mathbf{v}_1'$ is matched by

$$S_3 = (0\ 0\ 1 \star 1\ 1 \star\star\star\star\star 0\ 1),$$

which was not possible with one-point crossover.

Similarly, there are schemata that two-point crossover cannot combine. We can experiment then with multi-point crossover [52], a natural extension of two-point crossover. Note, however, that since multi-point crossover must alternate segments (obtained after cutting a chromosome into s pieces) between the two parents, the number of segments must be even, i.e., a multi-point crossover is not a natural extension of single-point crossover.

Schaffer and Morishima [157] experimented with a crossover which adapts the distribution of its crossover points by the same processes already in place (survival of the fittest and recombination). This was done by introducing special marks into string representation. These marks keep track of the sites in the string where crossover occurred. The hope was that the crossover sites would undergo an evolution process: if a particular site produces poor offspring, the site dies off (and vice versa). Experiments indicated [157] that adaptive crossover performed as well or better than a classical GA for a set of test problems.

Some researchers [52] experimented with other crossovers: segmented crossover and shuffle crossover. Segmented crossover is a variant of the multi-point crossover, which allows the number of crossover points to vary. The (fixed) number of crossover points (or segments) is replaced by a segment switch rate. This rate specifies the probability that a segment will end at any point in the string. For example, if the segment switch rate is $s = 0.2$, then, starting from the beginning of a segment, the chances of terminating this segment at each bit are 0.2. In other words, with the segment switch rate $s = 0.2$, the expected number of segments will be $m/5$, however, unlike multi-point crossover, the number of crossover points will vary. Shuffle crossover should be perceived as an additional mechanism which can be applied with other crossovers. It is independent of the number of crossover points. The shuffle crossover (1) randomly shuffles the bit positions of the two strings in tandem, (2) crosses the strings, i.e., exchanges segments between crossover points, and (3) unshuffles the strings.

A further generalization of one-point, two-point, and multi-point crossover is the uniform crossover [174], [182]: for each bit in the first offspring it decides (with some probability p) which parent will contribute its value in that position. The second offspring would receive the bit from the other parent.
For example, for $p = 0.5$ (0.5-uniform crossover), the strings

$$\mathbf{v}_1 = (0010001101001) \text{ and}$$
$$\mathbf{v}_2 = (1110110001000)$$

may produce the following pair of offspring:

$$\mathbf{v}_1' = (0_1 0_1 1_2 0_1 1_2 1_2 0_2 1_1 0_1 1_2 0_1 0_1 0_2) \text{ and}$$
$$\mathbf{v}_2' = (1_2 1_2 1_1 0_2 0_1 0_1 1_1 0_2 0_2 1_1 0_2 0_2 1_1),$$

where subscripts 1 and 2 indicate the parent (vectors $\mathbf{v}_1$ and $\mathbf{v}_2$, respectively) for a given bit. If $p = 0.1$ (0.1-uniform crossover), two strings $\mathbf{v}_1$ and $\mathbf{v}_2$ may produce

$$\mathbf{v}'_1 = (0_1 0_1 1_1 0_1 1_2 0_1 1_1 1_1 0_1 1_2 0_1 0_1 1_1) \text{ and}$$
$$\mathbf{v}'_2 = (1_2 1_2 1_2 0_2 0_1 1_2 0_2 0_2 0_2 1_1 0_2 0_2 0_2).$$

Since the uniform crossover exchanges bits rather than segments, it can combine features regardless their relative location. For some problems [182] this ability outweighs the disadvantage of destroying building blocks. However, for other problems, the uniform crossover was inferior to two-point crossover. Syswerda [182] compared theoretically the 0.5-uniform crossover with one-point and two-point crossovers. Spears and De Jong [174] provided an analysis of p-uniform crossovers, i.e., crossovers which involve on average $m \cdot p$ crossover points.

In [52] several experiments are reported for various crossover operators. The results indicate that the 'loser' is one-point crossover; however, there is no clear winner. A general comment on the above experiments is that each of these crossovers is particularly useful for some classes of problems and quite poor for other problems. This strengthens our idea of problem dependent operators leading us towards evolution programs.

Several researchers looked also at the effect of control parameters of a GA (population size, probabilities of operators) on the performance of the system. Grefenstette [80] applied a meta-GA to control parameters of another GA. Goldberg [71] provides a theoretical analysis of the optimal population size. A complete study on influence of the control parameters on the genetic search (online performance for function optimization) is presented in [159]. The results suggest that (1) mutation plays a stronger role than has previously been admitted (mutation is regarded as a background operator), (2) the importance of the crossover rate is smaller than expected, (3) the search strategy based on selection and mutation only might be a powerful search procedure even without the assistance of crossover (like evolution strategies, presented in Chapter 8). However, GAs still lack good heuristics to determine good values for their parameters: there is no single setting which would be universal for all considered problems. It seems that finding good values for the GA parameters is still more an art than a science.

Until this chapter we have discussed two basic genetic operators: crossover (one-point, two-point, uniform, etc.) and mutation, which were applied to individuals or single bits at some fixed rates (probabilities of crossover p_c and mutation p_m). As we have seen in the running example of Chapter 2, it was possible to apply crossover and mutation to the same individual (e.g., $\mathbf{v}_{13}$). In fact, these two basic operators can be viewed as one recombination operator, the "crossover and mutation" operator, since both operations can be applied to individuals at the same time. One possibility in experimenting with genetic operators is to make them independent: one or the other of these operators will

be applied during the reproduction event, but not both [38]. There are a few advantages of such separation. Firstly, a mutation will not be applied to the result of the crossover operator any longer, making the whole process conceptually simpler. Secondly, it is easier to extend a list of genetic operators by adding new ones: such a list may consist of several, problem dependent operators. This is precisely the idea behind evolution programming: there are many problem dependent "genetic operators", which are applied to individual data structures. Recall also, that for evolution programs presented in this book, we have developed a special selection routine modGA (previous section) which facilitates the above idea. Moreover, we can go further. Each operator may have its own fitness which would undergo some evolution process as well. Operators are selected and applied randomly, however, according to their fitness. This idea is not new and has been around for some time [37], [54], [38], but it is getting a new meaning and significance in the evolution programming technique.

Another interesting direction in search for a better schema processing, mentioned already in the previous chapter in connection with a deception problem, was proposed recently [73], [77]: a messy genetic algorithm (mGA).

The mGAs differ from classical GAs in many ways: representation, operators, size of population, selection, and phases of the evolution process. We discuss them briefly in turn.

First of all, each bit of a chromosome is tagged with its name (number) — the same trick we used discussing inversion operator in the previous chapter. Additionally, strings are of variable length and there is no requirement for a string to have full gene complements. A string may have redundant and even contradictory genes. For example, the following strings are legitimate chromosomes in mGA:

$$\begin{aligned}\mathbf{v_1} &= ((7,1)(1,0)),\\ \mathbf{v_2} &= ((3,1)(9,0)(3,1)(3,1),(3,1)),\\ \mathbf{v_3} &= ((2,1)(2,0)(4,1)(5,0)(6,0)(7,1)(8,1)).\end{aligned}$$

The first number in each parenthesis indicate a position, the second number the value of the bit. Thus the first string, $\mathbf{v_1}$, specifies two bits: bit 1 on the 7th position and bit 0 on the 1st position.

To evaluate such strings, we have to deal with overspecification (string $\mathbf{v_3}$, where two bits are specified on the 2nd position) and underspecification (all three vectors are underspecified, assuming 9 bit positions) problems. Overspecification can be handled in many ways; for example, some voting procedure (deterministic or probabilistic) can be used, or a positional precedence. Underspecification is harder to deal with, and we refer the interested reader to the source information [73], [76], [77].

Clearly, variable-length, overspecified or underspecified strings would influence the operators used. Simple crossover is replaced by two (even simpler) operators: *splice* and *cut*. The splice operator concatenates two selected strings (with the specified splice probability). For example, splicing strings $\mathbf{v_1}$ with $\mathbf{v_2}$ we get

$$\mathbf{v_4} = ((7,1)(1,0)(3,1)(9,0)(3,1)(3,1),(3,1)).$$

The cut operator cuts (with some cut probability) the selected string at a position determined randomly along its length. For example, cutting string $\mathbf{v_3}$ at position 4, we get

$$\mathbf{v_5} = ((2,1)(2,0)(4,1)(5,0)) \text{ and}$$
$$\mathbf{v_6} = ((6,0)(7,1)(8,1)).$$

In addition, there is an unchanged mutation operator, which changes 0 to 1 (or vice versa) with some specified probability.

There are some other differences between GA and mGA. Messy genetic algorithms (for reliable selection regardless of function scaling) use a form of tournament selection [73]: they divide the evolution process into two phases (the first phase selects building blocks, and only in the second phase are genetic operators invoked), and the population size changes in the process.

Messy genetic algorithms were tested on several deceptive functions with very good results [73], [76]. As stated by Goldberg [73]:

> "A difficult test function has been designed, and in two sets of experiments the mGA is able to find its global optimum. [...] In all runs on both sets of experiments, the mGA converges to the test function global optimum. By contrast, a simple GA using a random ordering of the string is able to get only 25% of the subfunctions correct."

and in [76]:

> "Because mGAs can converge in these worst-case problems, it is believed that they will find global optima in all other problems with bounded deception. Moreover, mGAs are structured to converge in computational time that grows only as a polynomial function of the number of decision variables on a serial machine and as a logarithmic function of the number of decision variables on a parallel machine. Finally, mGAs are a practical tool that can be used to climb a function's ladder of deception, providing useful and relatively inexpensive intermediate results along the way."

There were also some other attempts to enhance genetic search. A modification of GAs, called Delta Coding, was proposed recently by Whitley et al. [195]. Schraudoph and Belew [161] proposed a Dynamic Parameter Encoding (DPE) strategy, where the precision of the encoded individual is dynamically adjusted. These algorithms are discussed later in the book (Chapter 8).

Part II

Numerical Optimization

5. Binary or Float?

There were rules in the
monastery, but the Master
always warned against
the tyranny of the law.

Anthony de Mello, *One Minute Wisdom*

As discussed in the previous chapter, there are some problems that GA applications encounter that sometimes delay, if not prohibit, finding the optimal solutions with the desired precision. One of the implications of these problems was premature convergence of the entire population to a non–global optimum (Chapter 4); other consequences include inability to perform fine local tuning and inability to operate in the presence of nontrivial constraints (Chapters 6 and 7).

The binary representation traditionally used in genetic algorithms has some drawbacks when applied to multidimensional, high-precision numerical problems. For example, for 100 variables with domains in the range $[-500, 500]$ where a precision of six digits after the decimal point is required, the length of the binary solution vector is 3000. This, in turn, generates a search space of about 10^{1000}. For such problems genetic algorithms perform poorly.

The binary alphabet offers the maximum number of schemata per bit of information of any coding [72] and consequently the bit string representation of solutions has dominated genetic algorithm research. This coding also facilitates theoretical analysis and allows elegant genetic operators. But the 'implicit parallelism' result does not depend on using bit strings [4] and it may be worthwhile to experiment with large alphabets and (possibly) new genetic operators. In particular, for parameter optimization problems with variables over continuous domains, we may experiment with real-coded genes together with special "genetic" operators developed for them.

In [75] Goldberg wrote:

> "The use of real-coded or floating-point genes has a long, if controversial, history in artificial genetic and evolutionary search schemes, and their use as of late seems to be on the rise. This rising usage has been somewhat surprising to researchers familiar with fundamental

genetic algorithm (GA) theory ([72], [89]), because simple analyses seem to suggest that enhanced schema processing is obtained by using alphabets of low cardinality, a seemingly direct contradiction of empirical findings that real codings have worked well in a number of practical problems."

In this chapter we describe the results of experiments with various modifications of genetic operators on floating point representation. The main objective behind such implementations is (in line with the principle of evolution programming) to move the genetic algorithm closer to the problem space. Such a move forces, but also allows, the operators to be more problem specific — by utilizing some specific characteristics of real space. For example, this representation has the property that two points close to each other in the representation space must also be close in the problem space, and vice versa. This is not generally true in the binary approach, where the distance in a representation is normally defined by the number of different bit positions, even though some approaches (e.g., Gray coding, see [159]) reduces such discrepancy. We use a floating point representation as it is conceptually closest to the problem space and also allows for an easy and efficient implementation of closed and dynamic operators (see also Chapters 6 and 7).

Subsequently, we empirically compared a binary and floating point implementations using various new operators on many test cases. In this chapter, we illustrate the differences between binary and float representations on one typical test case of a dynamic control problem. This is a linear-quadratic problem, which is a particular case of a problem we use in the next chapter (together with two other dynamic control problems) to illustrate the progress of our evolution program in connection with premature convergence and local fine tuning. As expected, the results are better than those from binary representation. The same conclusion was also reached by other researchers e.g., [38].

5.1 The test case

For experiments we have selected the following dynamic control problem:

$$\min\left(x_N^2 + \sum_{k=0}^{N-1}(x_k^2 + u_k^2)\right),$$

subject to

$$x_{k+1} = x_k + u_k,\ k = 0, 1, \ldots, N-1,$$

where x_0 is a given initial state, $x_k \in R$ is a state, and $\mathbf{u} \in R^N$ is the sought control vector. The optimal value can be analytically expressed as

$$J^* = K_0 x_0^2,$$

where K_k is the solution of the Riccati equation:

$$K_k = 1 + K_{k+1}/(1 + K_{k+1}) \text{ and } K_N = 1.$$

During the experiments a chromosome represented a vector of the control states $\mathbf{u}$. We have also assumed a fixed domain $\langle -200, 200 \rangle$ for each u_i (actual solutions fall within this range for the class of tests performed). For all subsequent experiments we used $x_0 = 100$ and $N = 45$, i.e., a chromosome $\mathbf{u} = \langle u_0, \ldots, u_{44} \rangle$, having the optimal value $J^* = 16180.4$.

5.2 The two implementations

For the study we have selected two genetic algorithm implementations differing only by representation and applicable genetic operators, and equivalent otherwise. Such an approach gave us a better basis for a more direct comparison. Both implementations used the same selective mechanism: stochastic universal sampling [11].

5.2.1 The binary implementation

In the binary implementation each element of a chromosome vector was coded using the same number of bits. To facilitate fast run-time decoding, each element occupied its own word (in general it occupied more than one if the number of bits per element exceeded the word size, but this case is an easy extension) of memory: this way elements could be accessed as integers, which removed the need for binary to decimal decoding. Then, each chromosome was a vector of N words, which equals the number of elements per chromosome (or a multiple of such for cases where multiple words were required to represent a desired number of bits).

The precision of such an approach depends (for a fixed domain size) on the number of bits actually used and equals $(UB - LB)/(2^n - 1)$, where UB and LB are domain bounds and n is the number of bits per one element of a chromosome.

5.2.2 The floating point implementation

In the floating point (FP) implementation each chromosome vector was coded as a vector of floating point numbers, of the same length as the solution vector. Each element was forced to be within the desired range, and the operators were carefully designed to preserve this requirement.

The precision of such an approach depends on the underlying machine, but is generally much better than that of the binary representation. Of course, we can always extend the precision of the binary representation by introducing more bits, but this considerably slows down the algorithm (see Section 5.4).

In addition, the FP representation is capable of representing quite large domains (or cases of unknown domains). On the other hand, the binary representation must sacrifice precision with an increase in domain size, given fixed binary length. Also, in the FP representation it is much easier to design special tools for handling nontrivial constraints: this is discussed fully in Chapter 7.

5.3 The experiments

The experiments were conducted on a DEC3100 workstation. All results presented here represent the average values obtained from 10 independent runs. During all experiments the population size was kept fixed at 60, and the number of iterations was set at 20,000. Unless otherwise stated, the binary representation used $n = 30$ bits to code one variable (one element of the solution vector), making $30 \cdot 45 = 1350$ bits for the whole vector.

Because of possible differences in interpretation of the mutation operator, we accepted the probability of chromosomes' update as a fair measure for comparing the floating point and binary representations. All experimental values were obtained from runs with the operators set to achieve the same such rate; therefore, some number of iterations can be approximately treated interchangeably with the same number of function evaluations.

5.3.1 Random mutation and crossover

In this part of the experiment we ran both implementations with operators which are equivalent (at least for the binary representation) to the traditional ones.

5.3.1.1 Binary. The binary implementation used traditional operators of mutation and crossover. However, to make them more similar to those of the FP implementation, we allowed crossover only between elements. The probability of crossover was fixed at 0.25, while the probability of mutation varied to achieve the desired rate of chromosome update (shown in Table 5.1).

	Probability of chromosome's update				
implementation	0.6	0.7	0.8	0.9	0.95
Binary, p_m	0.00047	0.00068	0.00098	0.0015	0.0021
FP, p_m	0.014	0.02	0.03	0.045	0.061

Table 5.1. Probabilities of chromosome's update versus mutation rates

5.3.1.2 FP. The crossover operator was quite analogous to that of the binary implementation (split points between float numbers) and applied with the same probability (0.25). The mutation, which we call random, applies to a floating point number rather that to a bit; the result of such mutation is a random value from the domain $\langle LB, UB \rangle$.

	Probability of chromosome's update					standard
implementation	0.6	0.7	0.8	0.9	0.95	deviation
Binary	42179	46102	29290	52769	30573	31212
FP	46594	41806	47454	69624	82371	11275

Table 5.2. Average results as a function of probability of chromosome's update

5.3.1.3 Results. The results (Table 5.2) are slightly better for the binary case; however, it is rather difficult to judge them better as all fell well away from the optimal solution (16180.4). Moreover, an interesting pattern emerged that showed the FP implementation to be more stable, with much lower standard deviation.

In addition, it is interesting to note that the above experiment was not quite fair for the FP representation; its random mutation behaves "more" randomly than that of the binary implementation, where changing a random bit doesn't imply producing a totally random value from the domain. As an illustration let us consider the following question: what is the probability that after mutation an element will fall within $\delta\%$ of the domain range (400, since the domain is $\langle -200, 200 \rangle$) from its old value? The answer is:

FP: Such probability clearly falls in the range $\langle \delta, 2 \cdot \delta \rangle$. For example, for $\delta = 0.05$ it is in $\langle 0.05, 0.1 \rangle$.

Binary: Here we need to consider the number of low-order bits that can be safely changed. Assuming $n = 30$ as an element length and m as the length of permissible change, m must satisfy $m \leq n + \log_2 \delta$. Since m is an integer, then $m = \lfloor n + \log_2 \delta \rfloor = 25$ and the sought probability is $m/n = 25/30 = 0.833$, a quite different number.

Therefore, we will try to design a method of compensating for this drawback in the following subsection.

5.3.2 Non-uniform mutation

In this part of the experiments we ran, in addition to the operators discussed in Section 5.3.1, a special dynamic mutation operator aimed at both improving single-element tuning and reducing the disadvantage of random mutation in the FP implementation. We call it a *non-uniform mutation*; a full discussion of this operator is presented in the next chapter.

5.3.2.1 FP. The new operator is defined as follows: if $s_v^t = \langle v_1, \ldots, v_m \rangle$ is a chromosome (t is the generation number) and the element v_k was selected for this mutation, the result is a vector $s_v^{t+1} = \langle v_1, \ldots, v'_k, \ldots, v_m \rangle$, where

$$v'_k = \begin{cases} v_k + \triangle(t, UB - v_k) & \text{if a random digit is 0,} \\ v_k - \triangle(t, v_k - LB) & \text{if a random digit is 1,} \end{cases}$$

and LB and UB are lower and upper domain bounds of the variable v_k. The function $\triangle(t, y)$ returns a value in the range $[0, y]$ such that the probability of $\triangle(t, y)$ being close to 0 increases as t increases. This property causes this operator to search the space uniformly initially (when t is small), and very locally at later stages; thus increasing the probability of generating the new number closer to its successor than a random choice. We have used the following function:

$$\triangle(t, y) = y \cdot \left(1 - r^{(1 - \frac{t}{T})^b}\right),$$

where r is a random number from [0..1], T is the maximal generation number, and b is a system parameter determining the degree of dependency on iteration number (we used $b = 5$).

5.3.2.2 Binary. To be more than fair to the binary implementation, we modeled the dynamic operator into its space, even though it was introduced mainly to improve the FP mutation. Here, it is analogous to that of the FP, but with a differently defined v'_k:

$$v'_k = mutate(v_k, \nabla(t, n)),$$

where $n = 30$ is the number of bits per one element of a chromosome; $mutate(v_k, pos)$ means: mutate value of the k-th element on pos bit (0 bit is the least significant), and

$$\nabla(t, n) = \begin{cases} \lfloor \Delta(t, n) \rfloor & \text{if a random digit is 0,} \\ \lceil \Delta(t, n) \rceil & \text{if a random digit is 1,} \end{cases}$$

with the b parameter of Δ adjusted appropriately if similar behavior is desired (we used $b = 1.5$).

5.3.2.3 Results. We repeated similar experiments to those of Section 5.3.1.3 using also the non–uniform mutations applied at the same rate as the previously defined mutations.

Now the FP implementation shows a better average performance (Table 5.3). In addition, again the binary's results were more unstable. However, it is interesting to note here that despite its high average, the binary implementation produced the two single best results for this round (16205 and 16189).

5.3.3 Other operators

In this part of the experiment we decided to implement and use as many additional operators as could be easily defined in both representation spaces.

implementation	Probability of chromosome's update		standard deviation
	0.8	0.9	
Binary	35265	30373	40256
FP	20561	26164	2133

Table 5.3. Average results as a function of probability of chromosome's update

5.3.3.1 Binary. In addition to those previously described we implemented a multi–point crossover, and also allowed for crossovers within bits of an element. The multi–point operator had the probability of application to a single element controlled by a system parameter (set at 0.3).

5.3.3.2 FP. Here we also implemented a similar multi-point crossover. In addition, we implemented single and multi-point arithmetical crossovers; they average values of two elements rather that exchange them, at selected points. Such operators have the property that each element of the new chromosomes is still within the original domain. More details of these operators are provided in the next two chapters.

implementation	Probability of chromosome's update			standard deviation	Best
	0.7	0.8	0.9		
Binary	23814	19234	27456	6078	16188.2
FP	16248	16798	16198	54	16182.1

Table 5.4. Average results as a function of probability of chromosome's update

5.3.3.3 Results. Here the FP implementation shows an outstanding superiority (Table 5.4); even though the best results are not so very different, only the FP was consistent in achieving them.

5.4 Time performance

Many complain about the high time complexity of GAs on nontrivial problems. In this section we compare the time performance of both implementations. The results presented in Table 5.5 are those for runs of Section 5.3.3.

Table 5.5 compares CPU time for both implementations on varying number of elements in the chromosome. The FP version is much faster, even for the

implementation	Number of elements (N)				
	5	15	25	35	45
Binary	1080	3123	5137	7177	9221
FP	184	398	611	823	1072

Table 5.5. CPU time (sec) as a function of number of elements

moderate 30 bits per variable in the binary implementation. For large domains and high precision the total length of the chromosome grows, and the relative difference would expand as further indicated in Table 5.6.

implementation	Number of bits per binary element					
	5	10	20	30	40	50
Binary	4426	5355	7438	9219	10981	12734
FP	1072 (constant)					

Table 5.6. CPU time (sec) as a function of number of bits per element; $N = 45$

5.5 Conclusions

The conducted experiments indicate that the floating point representation is faster, more consistent from run to run, and provides a higher precision (especially with large domains where binary coding would require prohibitively long representation). At the same time its performance can be enhanced by special operators to achieve high (even higher than that of the binary representation) accuracy. In addition, the floating point representation, as intuitively closer to the problem space, is easier for designing other operators incorporating problem specific knowledge. This is especially essential in handling nontrivial, problem-specific constraints (Chapter 7).

These conclusions are in accordance with the reasons of the users of genetic-evolutionary techniques who prefer floating point representation given in [75]: (1) comfort with one-gene-one-variable correspondence, (2) avoidance of Hamming cliffs and other artifacts of mutation operating on bit strings treated as unsigned binary integers, (3) fewer generations to population conformity.

6. Fine Local Tuning

Weeks later, when a visitor asked him
what he taught his disciples, he said,
'To get their priorities right: Better
have the money than calculate it; better
have the experience than define it.'

Anthony de Mello, *One Minute Wisdom*

Genetic algorithms display inherent difficulties in performing local search for numerical applications. Holland suggested [89] that the genetic algorithm should be used as a preprocessor to perform the initial search, before turning the search process over to a system that can employ domain knowledge to guide the local search. As observed in [81]:

> "Like natural genetic systems, GAs progress by virtue of changing the distribution of high performance substructures in the overall population; individual structures are not the focus of attention. Once the high performance regions of the search space are identified by a GA, it may be useful to invoke a local search routine to optimize the members of the final population."

Local search requires the utilization of schemata of higher order and longer defining length than those suggested by the Schema Theorem. Additionally, there are problems where the domains of parameters are unlimited, the number of parameters is quite large, and high precision is required. These requirements imply that the length of the (binary) solution vector is quite significant (for 100 variables with domains in the range $[-500, 500]$, where the precision of six digits after the decimal point is required, the length of the binary solution vector is 3000). As mentioned in the previous chapter, for such problems the performance of genetic algorithms is quite poor.

To improve the fine local tuning capabilities of a genetic algorithm, which is a must for high precision problems, we designed a special mutation operator whose performance is quite different from the traditional one. Recall that a traditional mutation changes one bit of a chromosome at a time; therefore, such a change uses only local knowledge — only the bit undergoing mutation is known. Such a bit, if located in the left portion of a sequence coding a variable, is

very significant to the absolute magnitude of the mutation effect on the variable. On the other hand, bits far to the right of such a sequence have quite a smaller influence while mutated. We decided to use such positional global knowledge in the following way: as the population ages, bits located further to the right of each sequence coding one variable get higher probability of being mutated, while those on the left have such a probability decreasing. In other words, such a mutation causes global search of the search space at the beginning of the iterative process, but an increasingly local exploitation later on. We call this a non-uniform mutation and discuss it later in the chapter. First, we discuss the problems used for a test bed for this new operator.

6.1 The test cases

In general, the task of designing and implementing algorithms for the solution of optimal control problems is a difficult one. The highly touted dynamic programming is a mathematical technique that can be used in variety of contexts, particularly in optimal control [17]. However, this algorithm breaks down on problems of moderate size and complexity, suffering from what is called "the curse of dimensionality" [14].

Optimal control problems are quite difficult to deal with numerically. Some numerical dynamic optimization programs available for general users are typically offspring of the static packages [24] and they do not use dynamic-optimization specific methods. Thus the available programs do not make an explicit use of the Hamiltonian, transversality conditions, etc. On the other hand, if they did use the dynamic-optimization specific methods, they would be even more difficult for a layman to handle.

On the other hand, to the best of the author's knowledge, it is only recently that GAs have been applied to optimal control problems [132], [130]. We believe that previous GA implementations were too weak to deal with problems where high precision was required. In this chapter we present our modification of a GA designed to enhance its performance. We show the quality and applicability of the developed system by a comparative study of some dynamic optimization problems. Later, the system evolved to include typical constraints for such optimization problems — this is discussed in the next chapter (Section 7.2.). As a reference for these test cases (as well as many other experiments discussed later in the book), we use a standard computational package used for solving such problems: the Student Version of General Algebraic Modeling System with MINOS optimizer [24]. We will refer to this package in the rest of the book as GAMS.

Three simple discrete-time optimal control models (frequently used in applications of optimal control) have been chosen as test problems for the evolution program: the linear-quadratic problem, the harvest problem, and the (discretized) push-cart problem. We discuss them in turn.

6.1.1 The linear-quadratic problem

The first test problem is a one-dimensional linear-quadratic model:

$$\min q \cdot x_N^2 + \sum_{k=0}^{N-1} (s \cdot x_k^2 + r \cdot u_k^2), \tag{6.1}$$

subject to

$$x_{k+1} = a \cdot x_k + b \cdot u_k, \quad k = 0, 1, \ldots, N-1, \tag{6.2}$$

where x_0 is given, a, b, q, s, r are given constants, $x_k \in R$, is the state and $u_k \in R$ is the control of the system.

The value for the optimal performance of (6.1) subject to (6.2) is

$$J^* = K_0 x_0^2, \tag{6.3}$$

where K_k is the solution of the Riccati equation:

$$K_k = s + r a^2 K_{k+1}/(r + b^2 K_{k+1}), \qquad K_N = q. \tag{6.4}$$

In the sequel, the problem (6.1) subject to (6.2) will be solved for the sets of the parameters displayed in Table 6.1.

Case	N	x_0	s	r	q	a	b
I	45	100	1	1	1	1	1
II	45	100	10	1	1	1	1
III	45	100	1000	1	1	1	1
IV	45	100	1	10	1	1	1
V	45	100	1	1000	1	1	1
VI	45	100	1	1	0	1	1
VII	45	100	1	1	1000	1	1
VIII	45	100	1	1	1	0.01	1
IX	45	100	1	1	1	1	0.01
X	45	100	1	1	1	1	100

Table 6.1. Ten test cases

In the experiments the value of N was set at 45 as this was the largest horizon for which a comparative numerical solution from GAMS was still achievable.

6.1.2 The harvest problem

The harvest problem is defined as:

$$\max \sum_{k=0}^{N-1} \sqrt{u_k}, \tag{6.5}$$

subject to the equation of growth,

$$x_{k+1} = a \cdot x_k - u_k, \tag{6.6}$$

and one equality constraint,

$$x_0 = x_N, \tag{6.7}$$

where initial state x_0 is given, a is a constant, and $x_k \in R$ and $u_k \in R^+$ are the state and the (nonnegative) control, respectively.

The optimal value J^* of (6.5) subject to (6.6) and (6.7) is:

$$J^* = \sqrt{\frac{x_0 \cdot (a^N - 1)^2}{a^{N-1} \cdot (a-1)}}. \tag{6.8}$$

Problem (6.5) subject to (6.6) and (6.7) will be solved for $a = 1.1$, $x_0 = 100$, and the following values of $N = 2, 4, 10, 20, 45$.

6.1.3 The push-cart problem

The push-cart problem is to maximize the total distance $x_1(N)$ traveled in a given time (a unit, say), minus the total effort. The system is second order:

$$x_1(k+1) = x_2(k) \tag{6.9}$$

$$x_2(k+1) = 2x_2(k) - x_1(k) + \frac{1}{N^2}u(k), \tag{6.10}$$

and the performance index to be maximized is:

$$x_1(N) - \frac{1}{2N}\sum_{k=0}^{N-1} u^2(k). \tag{6.11}$$

For this problem the optimal value of index (6.11) is:

$$J^* = \frac{1}{3} - \frac{3N-1}{6N^2} - \frac{1}{2N^3}\sum_{k=0}^{N-1} k^2. \tag{6.12}$$

The push-cart problem will be solved for different values $N = 5, 10, 15, 20, 25, 30, 35, 40, 45$. Note that different N correspond to the number of discretization periods (of an equivalent continuous problem) rather than to the actual length of the optimization horizon which will be assumed as one.

6.2 The evolution program for numerical optimization

The evolution program we have built for numerical optimization problems is based on the floating point representation, and some new (specialized) genetic operators; we discuss them in turn.

6.2.1 The representation

In floating point representation each chromosome vector is coded as a vector of floating point numbers of the same length as the solution vector. Each element is initially selected as to be within the desired domain, and the operators are carefully designed to preserve this constraint (there is no such problem in the binary representation, but the design of the operators is rather simple; we do not see that as a disadvantage; on the other hand, it provides for other advantages mentioned below).

The precision of such an approach depends on the underlying machine, but is generally much better than that of the binary representation. Of course, we can always extend the precision of the binary representation by introducing more bits, but this considerably slows down the algorithm, as discussed in the previous chapter.

In addition, the floating point representation is capable of representing quite large domains (or cases of unknown domains). On the other hand, the binary representation must sacrifice precision with an increase in domain size, given fixed binary length. Also, in the floating point representation it is much easier to design special tools for handling nontrivial constraints: this is discussed fully in next chapter.

6.2.2 The specialized operators

The operators we use are quite different from the classical ones, as they work in a different space (real valued). However, because of intuitive similarities, we will divide them into the standard classes, mutation and crossover. In addition, some operators are non-uniform, i.e., their action depends on the age of the population.

Mutation group:

- **uniform mutation**, defined similarly to that of the classical version: if $x_i^t = \langle v_1, \ldots, v_n \rangle$ is a chromosome, then each element v_k has exactly equal chance of undergoing the mutative process. The result of a single application of this operator is a vector $\langle v_1, \ldots, v'_k, \ldots, v_n \rangle$, with $1 \leq k \leq n$, and v'_k a random value from the domain of the corresponding parameter $domain_k$.

- **non-uniform mutation** is one of the operators responsible for the fine tuning capabilities of the system. It is defined as follow: The non–uniform mutation operator was defined as follows: if $s_v^t = \langle v_1, \ldots, v_m \rangle$ is a chromosome and the element v_k was selected for this mutation (domain of v_k is $[l_k, u_k]$), the result is a vector $s_v^{t+1} = \langle v_1, \ldots, v'_k, \ldots, v_m \rangle$, with $k \in \{1, \ldots, n\}$, and

$$v'_k = \begin{cases} v_k + \triangle(t, u_k - v_k) & \text{if a random digit is 0,} \\ v_k - \triangle(t, v_k - l_k) & \text{if a random digit is 1,} \end{cases}$$

where the function $\Delta(t, y)$ returns a value in the range $[0, y]$ such that the probability of $\Delta(t, y)$ being close to 0 increases as t increases. This property causes this operator to search the space uniformly initially (when t is small), and very locally at later stages. We have used the following function:

$$\Delta(t, y) = y \cdot \left(1 - r^{(1-\frac{t}{T})^b}\right),$$

where r is a random number from [0..1], T is the maximal generation number, and b is a system parameter determining the degree of non-uniformity. Figure 6.1 displays the value of Δ for two selected times; this picture clearly indicates the behavior of the operator.

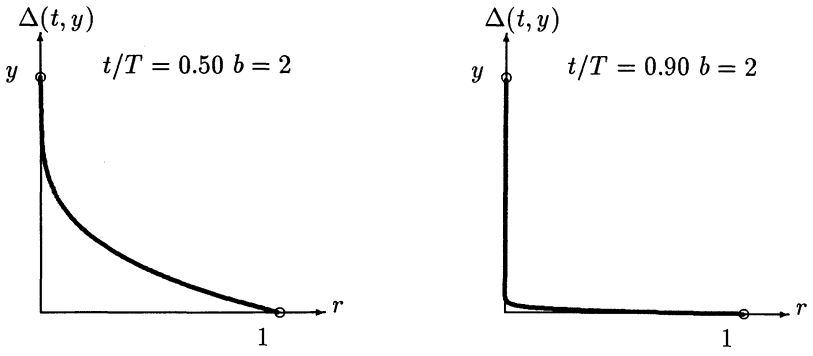

Fig. 6.1. $\Delta(t, y)$ for two selected times

Moreover, in addition to the standard way of applying mutation we have some new mechanisms: e.g., non-uniform mutation is also applied to a whole solution vector rather than a single element of it, causing the whole vector to be slightly slipped in the space.

Crossover group:

- **simple crossover**, defined in the usual way, but with the only permissible split points between v's, for a given chromosome x.
- **arithmetical crossover** is defined as a linear combination of two vectors: if s_v^t and s_w^t are to be crossed, the resulting offspring are $s_v^{t+1} = a \cdot s_w^t + (1-a) \cdot s_v^t$ and $s_w^{t+1} = a \cdot s_v^t + (1-a) \cdot s_w^t$ This operator can use a parameter a which is either a constant (uniform arithmetical crossover), or a variable whose value depends on the age of population (non-uniform arithmetical crossover).

Here again we have some new mechanisms to apply these operators; e.g., the arithmetical crossover may be applied either to selected elements of two vectors or to the whole vectors.

6.3 Experiments and results

In this section we present the results of the evolution program for the optimal control problems. For all test problems, the population size was fixed at 70, and the runs were made for 40,000 generations. For each test case we have made three random runs and reported the best results; it is important to note, however, that the standard deviations of such runs were almost negligibly small. The vectors $\langle u_0, \ldots, u_{N-1} \rangle$ were initialized randomly (but within a desired domains). Tables 6.2, 6.3, and 6.4 report the values found along with intermediate results at some generation intervals. For example, the values in column "10,000" indicate the partial result after 10,000 generations, while running 40,000. It is important to note that such values are worse than those obtained while running only 10,000 generation, due to the nature of some genetic operators. In the next section we compare these results with the exact solutions and solutions obtained from the computational package GAMS.

	Generations							Factor
Case	1	100	1,000	10,000	20,000	30,000	40,000	
I	17904.4	3.87385	1.73682	1.61859	1.61817	1.61804	1.61804	10^4
II	13572.3	5.56187	1.35678	1.11451	1.09201	1.09162	1.09161	10^5
III	17024.8	2.89355	1.06954	1.00952	1.00124	1.00102	1.00100	10^7
IV	15082.1	8.74213	4.05532	3.71745	3.70811	3.70162	3.70160	10^4
V	5968.42	12.2782	2.69862	2.85524	2.87645	2.87571	2.87569	10^5
VI	17897.7	5.27447	2.09334	1.61863	1.61837	1.61805	1.61804	10^4
VII	2690258	18.6685	7.23567	1.73564	1.65413	1.61842	1.61804	10^4
VIII	123.942	72.1958	1.95783	1.00009	1.00005	1.00005	1.00005	10^4
IX	7.28165	4.32740	4.39091	4.42524	4.31021	4.31004	4.31004	10^5
X	9971341	148233	16081.0	1.48445	1.00040	1.00010	1.00010	10^4

Table 6.2. Evolution program for the linear–quadratic problem (6.1)–(6.2)

Note, that the problem (6.5)–(6.7) has the final state constrained. It differs from the problem (6.1)–(6.2) in the sense that not every randomly initialized vector $\langle u_0, \ldots, u_{N-1} \rangle$ of positive real numbers generates an admissible sequence x_k (see condition (6.6)) such that $x_0 = x_N$, for given a and x_0. In our evolution program, we have generated a random sequence of $u_0, \ldots, u_{N-2}$, and have set $u_{N-1} = a \cdot x_{N-1} - x_N$. For negative u_{N-1}, we have discarded the sequence and repeated the initialization process (we discuss this process in detail in the next chapter, Section 7.2). The same difficulty occurred during the reproduction process. An offspring (after some genetic operations) need not satisfy the constraint $x_0 = x_N$. In such a case we replaced the last component of the offspring vector u using the formula $u_{N-1} = a \cdot x_{N-1} - x_N$. Again, if u_{N-1} turns out to be negative, we do not introduce such offspring into the new population.

It is the only test problem considered in this chapter which includes a non-trivial constraint. We discuss the general issue of constrained optimal control problems in the next chapter.

	Generations						
N	1	100	1,000	10,000	20,000	30,000	40,000
2	6.3310	6.3317	6.3317	6.3317	6.3317	6.3317	6.331738
4	12.6848	12.7127	12.7206	12.7210	12.7210	12.7210	12.721038
8	25.4601	25.6772	25.9024	25.9057	25.9057	25.9057	25.905710
10	32.1981	32.5010	32.8152	32.8209	32.8209	32.8209	32.820943
20	65.3884	68.6257	73.1167	73.2372	73.2376	73.2376	73.237668
45	167.1348	251.3241	277.3990	279.0657	279.2612	279.2676	279.271421

Table 6.3. Evolution program for the harvest problem (6.5)–(6.7)

	Generations						
N	1	100	1,000	10,000	20,000	30,000	40,000
5	-3.008351	0.081197	0.119979	0.120000	0.120000	0.120000	0.120000
10	-5.668287	-0.011064	0.140195	0.142496	0.142500	0.142500	0.142500
15	-6.885241	-0.012345	0.142546	0.150338	0.150370	0.150370	0.150371
20	-7.477872	-0.126734	0.149953	0.154343	0.154375	0.154375	0.154377
25	-8.668933	-0.015673	0.143030	0.156775	0.156800	0.156800	0.156800
30	-12.257346	-0.194342	0.123045	0.158241	0.158421	0.158426	0.158426
35	-11.789546	-0.236753	0.110964	0.159307	0.159586	0.159592	0.159592
40	-10.985642	-0.235642	0.072378	0.160250	0.160466	0.160469	0.160469
45	-12.789345	-0.342671	0.072364	0.160913	0.161127	0.161152	0.161152

Table 6.4. Evolution program for the push-cart problem (6.9)–(6.11)

6.4 Evolution program versus other methods

In this section we compare the above results with the exact solutions as well as those obtained from the computational package GAMS.

6.4.1 The linear-quadratic problem

Exact solutions of the problems for the values of the parameters specified in Table 6.1 have been obtained using formulae (6.3) and (6.4).

To highlight the performance and competitiveness of the evolution program, the same test problems were solved using GAMS. The comparison may be regarded as not totally fair for the evolution program since GAMS is based on search methods particularly appropriate for linear-quadratic problems. Thus the problem (6.1)–(6.2) must be an easy case for this package. On the other hand, if for these test problems the evolution program proved to be competitive, or close to, there would be an indication that it should behave satisfactorily in general. Table 6.5 summarizes the results, where columns D refer to the percentage of the relative error.

	Exact solution	Evolution Program		GAMS	
Case	value	value	D	value	D
I	16180.3399	16180.3928	0.000%	16180.3399	0.000%
II	109160.7978	109161.0138	0.000%	109160.7978	0.000%
III	10009990.0200	10010041.3789	0.000%	10009990.0200	0.000%
IV	37015.6212	37016.0426	0.000%	37015.6212	0.000%
V	287569.3725	287569.4357	0.000%	287569.3725	0.000%
VI	16180.3399	16180.4065	0.000%	16180.3399	0.000%
VII	16180.3399	16180.3784	0.000%	16180.3399	0.000%
VIII	10000.5000	10000.5000	0.000%	10000.5000	0.000%
IX	431004.0987	431004.4182	0.000%	431004.0987	0.000%
X	10000.9999	10001.0038	0.000%	10000.9999	0.000%

Table 6.5. Comparison of solutions for the linear–quadratic problem

As shown above, the performance of GAMS for the linear-quadratic problem is perfect. However, this was not at all the case for the second test problem.

6.4.2 The harvest problem

To begin with, none of the GAMS solutions was identical with the analytical one. The difference between the solutions increased with the optimization horizon as shown in Table 6.6, and for $N > 4$ the system failed to find any value.

It appears that GAMS is sensitive to non-convexity of the optimizing problem and to the number of variables. Even adding an additional constraint to the problem ($u_{k+1} > 0.1 \cdot u_k$) to to restrict the feasibility set so that the GAMS algorithm does not "lose itself"[1] has not helped much (see column "GAMS+"). As this column shows, for optimization horizons sufficiently long there is no chance to obtain a satisfactory solution from GAMS.

[1]This is "unfair" from the point of view of the genetic algorithm which works without such help.

N	Exact solution	GAMS		GAMS+		Genetic Alg	
		value	D	value	D	value	D
2	6.331738	4.3693	30.99%	6.3316	0.00%	6.3317	0.000%
4	12.721038	5.9050	53.58%	12.7210	0.00%	12.7210	0.000%
8	25.905710	*		18.8604	27.20%	25.9057	0.000%
10	32.820943	*		22.9416	30.10%	32.8209	0.000%
20	73.237681	*		*		73.2376	0.000%
45	279.275275	*		*		279.2714	0.001%

Table 6.6. Comparison of solutions for the harvest problem. The symbol * means that the GAMS failed to report a reasonable value

6.4.3 The push-cart problem

	Exact solution	GAMS		GA	
N	value	value	D	value	D
5	0.120000	0.120000	0.000%	0.120000	0.000 %
10	0.142500	0.142500	0.000%	0.142500	0.000 %
15	0.150370	0.150370	0.000%	0.150370	0.000 %
20	0.154375	0.154375	0.000%	0.154375	0.000 %
25	0.156800	0.156800	0.000%	0.156800	0.000 %
30	0.158426	0.158426	0.000%	0.158426	0.000 %
35	0.159592	0.159592	0.000%	0.159592	0.000 %
40	0.160469	0.160469	0.000%	0.160469	0.000 %
45	0.161152	0.161152	0.000%	0.161152	0.000 %

Table 6.7. Comparison of solutions for the push-cart problem

For the push-cart problem both GAMS and the evolution program produce very good results (Table 6.7). However, it is interesting to note the relationship between the times different search algorithms need to complete the task.

For most optimization programs, the time necessary for an algorithm to converge to the optimum depends on the number of decision variables. This relationship for dynamic programming is exponential ("curse of dimensionality"). For the search methods (like GAMS) it is usually "worse than linear".

Table 6.8 reports the number of iterations the evolution program needed to obtain an exact solution (with six decimal place rounding), the time needed for that, and the total time for all 40,000 iterations (for unknown exact solution we cannot determine the precision of the current solution). Also, the time for GAMS is given. Note that GAMS was run on PC Zenith z-386/20 and the evolution program on a DEC–3100 station.

N	No. of iterations needed	Time needed (CPU sec)	Time for 40,000 iterations (CPU sec)	Time for GAMS (CPU sec)
5	6234	65.4	328.9	31.5
10	10231	109.7	400.9	33.1
15	19256	230.8	459.8	36.6
20	19993	257.8	590.8	41.1
25	18804	301.3	640.4	47.7
30	22976	389.5	701.9	58.2
35	23768	413.6	779.5	68.0
40	25634	467.8	850.7	81.3
45	28756	615.9	936.3	95.9

Table 6.8. Time performance of evolution program and GAMS for the push-cart problem (6.9)–(6.11): number of iterations needed to obtain the result with precision of six decimal places, time needed for that number of iterations, time needed for all 40,000 iterations

It is clear that the evolution program is much slower than GAMS: there is a difference in absolute values of CPU time as well as computers used. However, let us compare not the times needed for both systems to complete their calculations, but rather their growth rates of the time as a function of the size of the problem. Figure 6.2 show the growth rate of the time needed to obtain the result for the evolution program and GAMS.

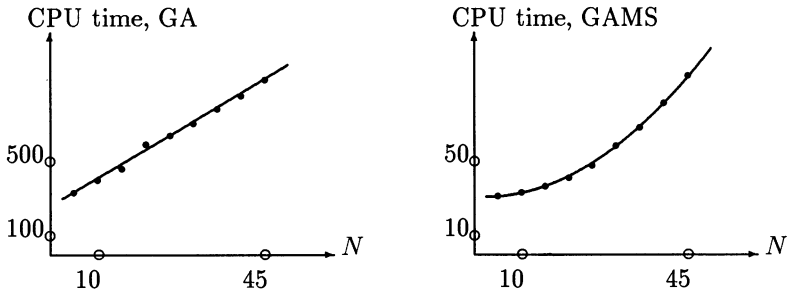

Fig. 6.2. Time as a function of problem size (N)

These graphs are self-explanatory: although the evolution program is generally slower, its linear growth rate is much better than that of GAMS (which is at least quadratic). Similar results hold for the linear-quadratic problem and the harvest problem.

6.4.4 The significance of non–uniform mutation

It is interesting to compare these results with the exact solutions as well as those obtained from another GA, exactly the same but without the *non–uniform mutation* on. Table 6.9 summarizes the results; columns labeled D indicate the relative errors in percents.

	Exact solution	GA w/ non-uniform mutation		GA w/o non-uniform mutation	
Case	value	value	D	value	D
I	16180.3399	16180.3939	0.000%	16234.3233	0.334%
II	109160.7978	109163.0278	0.000%	113807.2444	4.257%
III	10009990.0200	10010391.3989	0.004%	10128951.4515	1.188%
IV	37015.6212	37016.0806	0.001%	37035.5652	0.054%
V	287569.3725	287569.7389	0.000%	298214.4587	3.702%
VI	16180.3399	16180.6166	0.002%	16238.2135	0.358%
VII	16180.3399	16188.2394	0.048%	17278.8502	6.786%
VIII	10000.5000	10000.5000	0.000%	10000.5000	0.000%
IX	431004.0987	431004.4092	0.000%	431610.9771	0.141%
X	10000.9999	10001.0045	0.000%	10439.2695	4.380%

Table 6.9. Comparison of solutions for the linear-quadratic dynamic control problem

The genetic algorithm using the *non-uniform mutation* clearly outperforms the other one with respect to the accuracy of the found optimal solution; while the enhanced GA rarely erred by more than a few thousandths of one percent, the other one hardly ever beat one percent. Moreover, it also converged much faster to that solution.

As an illustration of the *non-uniform mutation*'s effect on the evolutionary process check Figure 6.3; the new mutation causes quite an increase in the number of improvements observed in the population at the end of the population's life. Moreover, a smaller number of such improvements prior to that time, together with an actually faster convergence, clearly indicates a better overall search.

6.5 Conclusions

In this chapter we have studied a new operator, a non-uniform mutation, to improve the fine local tuning capabilities of a GA. The experiments were successful on the three discrete-time optimal control problems which were selected as test cases. In particular, the results were encouraging because the closeness of the numerical solutions to the analytical ones was satisfying. Additionally, the computation effort was reasonable (for the 40,000 generations, a few minutes of CPU time on a CRAY Y-MP and up to 15 minutes on a DEC–3100 station).

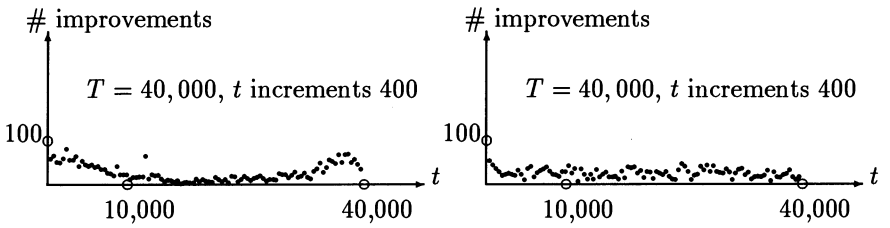

Fig. 6.3. Number of improvements on case I of the linear-quadratic dynamic control problem

The numerical results were compared with those obtained from a search-based computational package (GAMS). While the evolution program gave us results comparable with the analytic solutions for all test problems, GAMS failed for one of them. The developed evolution program displayed some qualities not always present in the other (gradient-based) systems:

- The optimization function for the evolution program need not be continuous. At the same time some optimization packages will not accept such functions at all.
- Some optimization packages are all-or-nothing propositions: the user has to wait until the program completes. Sometimes it is not possible to get partial (or approximate) results at some early stages. Evolution programs give the users additional flexibility, since the user can monitor the "state of the search" during the run time and make appropriate decisions. In particular, the user can specify the computation time (s)he is willing to pay for (longer time provides better precision in the answer).
- The computational complexity of evolution programs grows at a linear rate; most of other search methods are very sensitive to the length of the optimization horizon. As usual, we can easily improve the performance of the system using parallel implementations; often this is difficult for other optimization methods.

In general, an evolution program as an optimization system for optimal control problems, to be interesting for practitioners, has to be more control-problem specific than the system here introduced. In particular the system has to allow for at least:

- inequality constraints on the state variables,
- inequality constraints on the control variables,
- an equality constraint on the final state,
- inequality constraints between state and control variables.

In our experiments a model with one equality constraint on the final state was successfully handled (the harvest problem). In the next chapter (Section 7.2) we discuss a construction of the system GAFOC which satisfies the above requirements.

7. Handling Constraints

A traveler in quest of the divine
asked the Master how to distinguish
a true teacher from a false one when
he got back to his own land.

Said the Master, 'A good teacher
offers practice; a bad one offers
theories.'

'But how shall I know good practice
from bad?'

'In the same way that the farmer
knows good cultivation from bad.'

Anthony de Mello, *One Minute Wisdom*

The central problem in applications of genetic algorithms is that of constraints — few approaches to the constraint problem in genetic algorithms have previously been proposed. One of these uses penalty functions as an adjustment to the optimized objective function, other approaches use "decoders" or "repair" algorithms, which avoid building an illegal individual, or repair one, respectively. However, these approaches suffer from the disadvantage of being tailored to the specific problem and are not sufficiently general to handle a variety of problems.

Our approached is based on the evolution programming principle: appropriate data structures and specialized "genetic" operators should do the job of taking care of constraints.

In this chapter we discuss two evolution programs (GENOCOP and GAFOC); the former, presented in Section 7.1[1] optimizes a function with a set of linear constraints, the latter (Section 7.2) optimizes constrained control problems. Before we present them in detail, we discuss briefly other traditional constraint-handling methods for genetic algorithms.

One way of dealing with candidates that violate the constraints is to generate potential solutions without considering the constraints and then to penalize them by decreasing the "goodness" of the evaluation function. In other words,

[1]Portions reprinted, with permission, from the Communications of the ACM, 1992.

a constrained problem is transformed to an unconstrained problem by associating a penalty with all constraint violations and the penalties are included in the function evaluation. Thus, the original problem of optimizing a function $f(x_1, x_2, \ldots, x_q)$ is transformed into optimization of the function:

$$f(x_1, x_2, \ldots, x_q) + \epsilon \cdot \delta \sum_{i=1}^{p} \Phi_i$$

where p is the total number of constraints, δ is a penalty coefficient, ϵ is -1 for maximization and $+1$ minimization problems, and Φ_i is a penalty related to the i-th constraint ($i = 1, \ldots, p$).

However, though the evaluation function is usually well defined, there is no accepted methodology for combining it with the penalty. Let us repeat this important citation (already given in the Introduction) from [34]:

> "If one incorporates a high penalty into the evaluation routine and the domain is one in which production of an individual violating the constraint is likely, one runs the risk of creating a genetic algorithm that spends most of its time evaluating illegal individuals. Further, it can happen that when a legal individual is found, it drives the others out and the population converges on it without finding better individuals, since the likely paths to other legal individuals require the production of illegal individuals as intermediate structures, and the penalties for violating the constraint make it unlikely that such intermediate structures will reproduce. If one imposes moderate penalties, the system may evolve individuals that violate the constraint but are rated better than those that do not because the rest of the evaluation function can be satisfied better by accepting the moderate constraint penalty than by avoiding it".

In [169] and [153] the authors present the most recent approaches for using penalty functions in GAs for constrained optimization problems. However, the paper by Siedlecki and Sklansky [169] discusses a particular constrained optimization problem. This problem is frequently encountered in the design of statistical pattern classifiers; however, the proposed method is problem specific. The paper by Richardson, Palmer, Liepins, and Hillard [153] examines the penalty approach, discussing the strengths and weaknesses of various penalty function formulations, and illustrates a technique for solving three-dimensional constrained problems. However, in the last section of their article the authors say:

> "The technique used to solve the three-dimensional problem described above can't be generalized since quantities like the trend and maximum derivative are seldom available".

We do not believe this to be a promising direction. For a heavily constrained problem, such as the transportation problem, the probability of generating an infeasible candidate is too great to be ignored. At the best, the technique based

on penalty functions seems to work reasonably well for narrow classes of problems and for few constraints.

The difficulties arising in designing a genetic algorithm with penalty functions for a particular constrained problem are discussed later in Section 7.1.7.2.

Another approach concentrates on the use of special representation mappings (decoders) which guarantee (or at least increase the probability of) the generation of a feasible solution, and on the application of special repair algorithms to "correct" any infeasible solutions so generated. However, decoders are frequently computationally intensive to run [33], not all constraints can be easily implemented this way, and the resulting algorithm must be tailored to the particular application. The same is true for repair algorithms. Again, we do not believe this is a promising direction for incorporating constraints into genetic algorithms (see Section 7.1.7.3). Repair algorithms and decoders may work reasonably well but are highly problem specific, and the chances of building a general genetic algorithm to handle different types of constraints based on this principle seem to be slim.

7.1 An evolution program: the GENOCOP system

We consider a class of optimization problems that can be formulated as follows: Optimize a function $f(x_1, x_2, \ldots, x_q)$, subject to the following sets of linear constraints:

1. Domain constraints: $l_i \leq x_i \leq u_i$ for $i = 1, 2, \ldots, q$. We write $\mathbf{l} \leq \mathbf{x} \leq \mathbf{u}$, where $\mathbf{l} = \langle l_1, \ldots, l_q \rangle$, $\mathbf{u} = \langle u_1, \ldots, u_q \rangle$, $\mathbf{x} = \langle x_1, \ldots, x_q \rangle$.

2. Equalities: $A\mathbf{x} = \mathbf{b}$, where $\mathbf{x} = \langle x_1, \ldots, x_q \rangle$, $A = (a_{ij})$, $\mathbf{b} = \langle b_1, \ldots, b_p \rangle$, $1 \leq i \leq p$, and $1 \leq j \leq q$ (p is the number of equations).

3. Inequalities: $C\mathbf{x} \leq \mathbf{d}$, where $\mathbf{x} = \langle x_1, \ldots, x_q \rangle$, $C = (c_{ij})$, $\mathbf{d} = \langle d_1, \ldots, d_m \rangle$, $1 \leq i \leq m$, and $1 \leq j \leq q$ (m is the number of inequalities).

This formulation is general enough to handle a large class of standard Operations Research optimization problems with linear constraints and any objective function. The example considered later, the nonlinear transportation problem, is one of many problems in this class.

The proposed system (GENOCOP, for GEnetic algorithm for Numerical Optimization for COnstrained Problems) provides a way of handling constraints that is both general and problem independent. It combines some of the ideas seen in the previous approaches, but in a totally new context. The main idea behind this approach lies in (1) an elimination of the equalities present in the set of constraints, and (2) careful design of special "genetic" operators, which guarantee to keep all "chromosomes" within the constrained solution space. This can be done very efficiently for linear constraints and, while we do not claim these results extend easily to nonlinear constraints, the former class contains many interesting optimization problems.

7.1.1 The idea

Before we discuss the details of GENOCOP system, we present a small example which should provide some insight into the working of the system.
Let us assume we wish to optimize a function of six variables:

$$f(x_1, x_2, x_3, x_4, x_5, x_6),$$

subject to the following constraints:

$$\begin{aligned}
&x_1 + x_2 + x_3 = 5\\
&x_4 + x_5 + x_6 = 10\\
&x_1 + x_4 = 3\\
&x_2 + x_5 = 4\\
&x_1 \geq 0,\ x_2 \geq 0,\ x_3 \geq 0,\ x_4 \geq 0,\ x_5 \geq 0,\ x_6 \geq 0.
\end{aligned}$$

This problem is equivalent to the transportation problem (see Chapter 9) with two sources and three destinations.

We can take advantage of the presence of four independent equations and express four variables as functions of the remaining two:

$$\begin{aligned}
x_3 &= 5 - x_1 - x_2\\
x_4 &= 3 - x_1\\
x_5 &= 4 - x_2\\
x_6 &= 3 + x_1 + x_2.
\end{aligned}$$

We have reduced the original problem to the optimization problem of a function of two variables x_1 and x_2:

$$g(x_1, x_2) = f(x_1, x_2, (5 - x_1 - x_2), (3 - x_1), (4 - x_2), (3 + x_1 + x_2)),$$

subject to the following constraints (inequalities only):

$$\begin{aligned}
&x_1 \geq 0,\ x_2 \geq 0\\
&5 - x_1 - x_2 \geq 0\\
&3 - x_1 \geq 0\\
&4 - x_2 \geq 0\\
&3 + x_1 + x_2 \geq 0.
\end{aligned}$$

These inequalities can be further reduced to:

$$\begin{aligned}
&0 \leq x_1 \leq 3\\
&0 \leq x_2 \leq 4\\
&x_1 + x_2 \leq 5.
\end{aligned}$$

This would complete the first step of our algorithm: elimination of equalities.

Now let us consider a single point from the search space, $\mathbf{x} = \langle x_1, x_2 \rangle = \langle 1.8, 2.3 \rangle$. If we try to change the value of variable x_1 without changing the value of x_2 (uniform mutation, see Section 7.1.5), the variable x_1 can take any

value from the range $[0,\ 5 - x_2] = [0,\ 2.7]$. Additionally, if we have two points within search space, $\mathbf{x} = \langle x_1, x_2 \rangle = \langle 1.8, 2.3 \rangle$ and $\mathbf{x}' = \langle x'_1, x'_2 \rangle = \langle 0.9, 3.5 \rangle$, then any linear combination $a\mathbf{x} + (1 - a)\mathbf{x}'$, $0 \leq a \leq 1$, yields a point within search space, i.e., all constraints must be satisfied (whole arithmetical crossover, see Section 7.1.5). Therefore, both examples of operators would avoid moving a vector outside the constrained solution space.

The above example explains the main idea behind the GENOCOP system. Linear constraints were of two types: equalities and inequalities. We first eliminated all the equalities, reducing the number of variables and appropriately modifying the inequalities. Reducing the set of variables both decreased the length of the representation vector and reduced the search space. Since we were left with only linear inequalities, the search space was convex — which, in the presence of closed operators, could be searched efficiently. The problem then became one of designing such closed operators. We achieved this by defining them to be context-dependent, that is, dynamically adjusting to the current context.

Now we are ready to discuss details of the GENOCOP system.

7.1.2 Elimination of equalities

Suppose the equality constraint set is represented in matrix form:

$$A\mathbf{x} = \mathbf{b}.$$

We assume there are p independent equality equations (there are easy methods to verify this), i.e., there are p variables $x_{i_1}, x_{i_2}, \ldots, x_{i_p}$ ($\{i_1, \ldots, i_p\} \subseteq \{1, 2, \ldots, q\}$) which can be determined in terms of the other variables. These can, therefore, be eliminated from the problem statement, as follows.

We can split the array A vertically into two arrays A_1 and A_2, such that the j-th column of the matrix A belong to A_1 iff $j \in \{i_1, \ldots, i_p\}$. Thus A_1^{-1} exists. Similarly, we split matrix C and vectors $\mathbf{x}, \mathbf{l}, \mathbf{u}$ (*ie.* $\mathbf{x}^1 = \langle x_{i_1}, \ldots, x_{i_p} \rangle$, $\mathbf{l}_1 = \langle l_{i_1}, \ldots, l_{i_p} \rangle$, and $\mathbf{u}_1 = \langle u_{i_1}, \ldots, u_{i_p} \rangle$).

Then

$$A_1\mathbf{x}^1 + A_2\mathbf{x}^2 = \mathbf{b}$$

and it is easily seen that

$$\mathbf{x}^1 = A_1^{-1}\mathbf{b} - A_1^{-1}A_2\mathbf{x}^2.$$

Using the above rule, we can eliminate the variables $x_{i_1}, \ldots, x_{i_p}$ replacing them by a linear combination of remaining variables. However, each variable x_{i_j} ($j = 1, 2, \ldots, p$) is constrained additionally by a domain constraint: $l_{i_j} \leq x_{i_j} \leq u_{i_j}$. Eliminating all variables x_{i_j} leads us to introduce a new set of inequalities:

$$\mathbf{l}_1 \leq A_1^{-1}\mathbf{b} - A_1^{-1}A_2\mathbf{x}^2 \leq \mathbf{u}_1,$$

which is added to the original set of inequalities.

The original set of inequalities,

$$C\mathbf{x} \leq \mathbf{d},$$

can be represented as

$$C_1\mathbf{x}^1 + C_2\mathbf{x}^2 \leq \mathbf{d}.$$

This can be transformed into

$$C_1(A_1^{-1}\mathbf{b} - A_1^{-1}A_2\mathbf{x}^2) + C_2\mathbf{x}^2 \leq \mathbf{d}.$$

So, after eliminating the p variables $x_{i_1}, \ldots, x_{i_p}$, the final set of constraints consists of the following inequalities only:

1. original domain constraints: $\mathbf{l}_2 \leq \mathbf{x}^2 \leq \mathbf{u}_2$,
2. new inequalities: $A_1\mathbf{l}_1 \leq \mathbf{b} - A_2\mathbf{x}^2 \leq A_1\mathbf{u}_1$,
3. original inequalities (after removal of $\mathbf{x}^1$ variables): $(C_2 - C_1A_1^{-1}A_2)\mathbf{x}^2 \leq \mathbf{d} - C_1A_1^{-1}\mathbf{b}$.

7.1.3 Representation issues

As in the previous chapter, we have accepted the floating point representation for chromosomes. The floating point representation is, of course, not essential to the implementation of this constraints handling methodology. Rather, it was selected to simplify genetic operators' definitions and for obtaining a better performance of the genetic algorithm itself.

7.1.4 Initialization process

A genetic algorithm requires a population of potential solutions to be initialized and then maintained during the process. The GENOCOP system generates an initial set of potential solutions as follows:

- a subset of potential solutions is selected randomly from the space of the whole feasible region,
- the remaining subset of potential solutions consists entirely of points on the boundary of the solution space.

Their proportion *prop* is related to the relative frequences of different operators: some operators (like boundary mutation, Section 7.1.5) move the offspring towards the boundary of the solution space, whereas others (like arithmetical crossovers) spread the offspring over the whole feasible space.

7.1.5 Genetic operators

The operators used in the GENOCOP system are quite different from the classical ones for the following reasons:

1. We deal with a real valued space R^q, where a solution is coded as a vector with floating point type components,
2. The genetic operators are dynamic, i.e., a value of a vector component depends on the remaining values of the vector,
3. Some genetic operators are non-uniform, i.e., their action depends on the age of the population.

The value of the i-th component of a feasible solution $\mathbf{s} = \langle v_1, \ldots, v_m \rangle$ is always in some (dynamic) range $[l, u]$; the bounds l and u depend on the other vector's values $v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_m$, and the set of inequalities. We say that the i-th component (i-th gene) of the vector $\mathbf{s}$ is *movable* if $l < u$.

It is important to understand how these differences affect the genetic operators. The first one affects only the way a mutation is performed after an appropriate gene has been selected. In contrast, the next two affect both the selection of a gene for mutation and the selection of crossover points. They also dictate the space (domain) and probability distribution for the mutation and the behavior during crossover.

Before we describe the operators, we present two important characteristics of convex spaces (due to the linearity of the constraints, the solution space is always a convex space $\mathcal{S}$), which play an essential role in the definition of these operators:

1. For any two points s_1 and s_2 in the solution space $\mathcal{S}$, the linear combination $a \cdot s_1 + (1 - a) \cdot s_2$, where $a \in [0, 1]$, is a point in $\mathcal{S}$.
2. For every point $s_0 \in \mathcal{S}$ and any line p such that $s_0 \in p$, p intersects the boundaries of $\mathcal{S}$ at precisely two points, say $l_p^{s_0}$ and $u_p^{s_0}$.

Since we are only interested in lines parallel to each axis, to simplify the notation we denote by $l_{(i)}^s$ and $u_{(i)}^s$ the i-th components of the vectors l_p^s and u_p^s, respectively, where the line p is parallel to the axis i. We assume further that $l_{(i)}^s \leq u_{(i)}^s$.

Because of intuitive similarities, we cluster the operators into the standard two classes, mutation and crossover. The proposed crossover and mutation operators use the two properties to ensure that the offspring of a point in the convex solution space $\mathcal{S}$ belongs to $\mathcal{S}$. However, some operators (e.g., non-uniform mutation) have little to do with GENOCOP methodology; they have other "responsibilities" like fine tuning and prevention of premature convergence.

Mutation group: Mutations are quite different from the traditional ones with respect to both the actual mutation (a gene, being a floating point number,

is mutated in a dynamic range) and to the selection of an applicable gene. A traditional mutation is performed on static domains for all genes. In such a case the order of possible mutations on a chromosome does not influence the outcome. This is not true any more with dynamic domains. To solve the problem we proceed as follows: we randomly select $p_{um} \cdot pop\_size$ chromosomes for uniform mutation, $p_{bm} \cdot pop\_size$ chromosomes for boundary mutation , and $p_{nm} \cdot pop\_size$ chromosomes for non-uniform mutation (all with possible repetitions), where p_{um}, p_{bm}, and p_{nm} are probabilities of the three mutations defined below. Then, we perform these mutations in a random fashion on the selected chromosome.

- **uniform mutation** is defined similarly as in the previous chapter: we select a random gene k (from the set of movable genes of the given chromosome s determined by its current context). If $s_v^t = \langle v_1, \ldots, v_m \rangle$ is a chromosome and the k-th component is the selected gene, the result is a vector $s_v^{t+1} = \langle v_1, \ldots, v'_k, \ldots, v_m \rangle$, where v'_k is a random value (uniform probability distribution) from the range $[l_{(k)}^{s_v^t}, u_{(k)}^{s_v^t}]$. The dynamic values $l_{(k)}^{s_v^t}$ and $u_{(k)}^{s_v^t}$ are easily calculated from the set of constraints (inequalities).

- **boundary mutation** is a variation of uniform mutation with v'_k being either $l_{(k)}^{s_v^t}$ or $u_{(k)}^{s_v^t}$ with equal probability.

- **non-uniform mutation**, as discussed in the previous chapter, is one of the operators responsible for the fine tuning capabilities of the system. Let us recall its definition: if $s_v^t = \langle v_1, \ldots, v_m \rangle$ is a chromosome and the element v_k was selected for this mutation from the set of movable genes, the result is a vector $s_v^{t+1} = \langle v_1, \ldots, v'_k, \ldots, v_m \rangle$, with $k \in \{1, \ldots, n\}$, and

$$v'_k = \begin{cases} v_k + \triangle(t, u_{(k)}^{s_v^t} - v_k) & \text{if a random digit is 0,} \\ v_k - \triangle(t, v_k - l_{(k)}^{s_v^t}) & \text{if a random digit is 1.} \end{cases}$$

  The function $\triangle(t, y)$ returns a value in the range $[0, y]$ such that the probability of $\triangle(t, y)$ being close to 0 increases as t increases. This property causes this operator to search the space uniformly initially (when t is small), and very locally at later stages. We have used the following function:

$$\triangle(t, y) = y \cdot \left(1 - r^{(1 - \frac{t}{T})^b}\right),$$

  where r is a random number from [0..1], T is the maximal generation number, and b is a system parameter determining the degree of non-uniformity.

Crossover group: Chromosomes are randomly selected in pairs for application of the crossover operators according to appropriate probabilities.

- **simple crossover** is defined as follows: if $s_v^t = \langle v_1, \ldots, v_m \rangle$ and $s_w^t = \langle w_1, \ldots, w_m \rangle$ are crossed after the k-th position, the resulting offspring are:

$s_v^{t+1} = \langle v_1, \ldots, v_k, w_{k+1}, \ldots, w_m \rangle$ and $s_w^{t+1} = \langle w_1, \ldots, w_k, v_{k+1}, \ldots, v_m \rangle$. Note that the only permissible split points are between individual floating points (using float representation it is impossible to split anywhere else).

However, such an operator may produce offspring outside of the convex solution space $\mathcal{S}$. To avoid this problem, we use the property of the convex spaces that there exist $a \in [0, 1]$ such that

$$s_v^{t+1} = \langle v_1, \ldots, v_k, w_{k+1} \cdot a + v_{k+1} \cdot (1-a), \ldots, w_m \cdot a + v_m \cdot (1-a) \rangle \in \mathcal{S},$$
$$s_w^{t+1} = \langle w_1, \ldots, w_k, v_{k+1} \cdot a + w_{k+1} \cdot (1-a), \ldots, v_m \cdot a + w_m \cdot (1-a) \rangle \in \mathcal{S}.$$

The only question yet to be answered is how to find the largest a to obtain the greatest possible information exchange: due to the real interval, we cannot perform an extensive search. In GENOCOP we implemented a binary search (to some depth only for efficiency). Then, a takes the largest appropriate value found, or 0 if no value satisfied the constraints. The necessity for such actions is small in general and decreases rapidly over the life of the population. Note that the value of a is determined separately for each single arithmetical crossover and each gene.

- **single arithmetical crossover** is defined as follows: if $s_v^t = \langle v_1, \ldots, v_m \rangle$ and $s_w^t = \langle w_1, \ldots, w_m \rangle$ are to be crossed, the resulting offspring are $s_v^{t+1} = \langle v_1, \ldots, v'_k, \ldots, v_m \rangle$ and $s_w^{t+1} = \langle w_1, \ldots, w'_k, \ldots, w_m \rangle$, where $k \in [1, m]$, $v'_k = a \cdot w_k + (1-a) \cdot v_k$, and $w'_k = a \cdot v_k + (1-a) \cdot w_k$. Here, a is a dynamic parameter calculated in the given context (vectors s_v, s_w) so that the operator is closed (points x_v^{t+1} and x_w^{t+1} are in the convex constrained space $\mathcal{S}$). Actually, a is a random choice from the following range:

$$a \in \begin{cases} [\max(\alpha, \beta), \min(\gamma, \delta)], & \text{if } v_k > w_k, \\ [0, 0], & \text{if } v_k = w_k, \\ [\max(\gamma, \delta), \min(\alpha, \beta)], & \text{if } v_k < w_k, \end{cases}$$

where

$$\alpha = (l_{(k)}^{s_w} - w_k)/(v_k - w_k), \quad \beta = (u_{(k)}^{s_v} - v_k)/(w_k - v_k),$$
$$\gamma = (l_{(k)}^{s_v} - v_k)/(w_k - v_k), \quad \delta = (u_{(k)}^{s_w} - w_k)/(v_k - w_k).$$

To increase the applicability of this operator (to ensure that a will be non–zero, which actually always nullifies the results of the operator) it is wise to select the applicable gene as a random choice from the intersection of movable genes of both chromosomes. Again, note that the value of a is determined separately for each single arithmetical crossover and each gene.

- **whole arithmetical crossover** is defined as a linear combination of two vectors: if s_v^t and s_w^t are to be crossed, the resulting offspring are $s_v^{t+1} = a \cdot s_w^t + (1-a) \cdot s_v^t$ and $s_w^{t+1} = a \cdot s_v^t + (1-a) \cdot s_w^t$. This operator uses a simpler static system parameter $a \in [0..1]$, as it always guarantees closure (according to characteristic (1) of this section).

7.1.6 An example of the GENOCOP approach

In this section we present an example of the GENOCOP approach for solving a transportation problem (for a full discussion on the transportation problem, see Chapter 9).

We consider the following transportation problem:

$$\text{minimize } (10x_1 + 20x_3 + 11x_4 + 12x_5 + 7x_6 + 9x_7 + 20x_8 + 14x_{10} + 16x_{11} + 18x_{12}),$$

where (sources):

$$x_1 + x_2 + x_3 + x_4 = 15$$
$$x_5 + x_6 + x_7 + x_8 = 25$$
$$x_9 + x_{10} + x_{11} + x_{12} = 5 \ (\star)$$

and (destinations):

$$x_1 + x_5 + x_9 = 5$$
$$x_2 + x_6 + x_{10} = 15$$
$$x_3 + x_7 + x_{11} = 15$$
$$x_4 + x_8 + x_{12} = 10$$

Amount transported

	5	**15**	**15**	**10**
15	x_1	x_2	x_3	x_4
25	x_5	x_6	x_7	x_8
5	x_9	x_{10}	x_{11}	x_{12}

One of these seven equalities can be removed to obtain six independent equations — we eliminate the equality $(\star)$. We can therefore eliminate six variables forming vector $\mathbf{x}^1$. The problem specific domain constraints are:

$$0 \le x_1 \le 5 \qquad 0 \le x_7 \le 15$$
$$0 \le x_2 \le 15 \qquad 0 \le x_8 \le 10$$
$$0 \le x_3 \le 15 \qquad 0 \le x_9 \le 5$$
$$0 \le x_4 \le 10 \qquad 0 \le x_{10} \le 5$$
$$0 \le x_5 \le 5 \qquad 0 \le x_{11} \le 5$$
$$0 \le x_6 \le 15 \qquad 0 \le x_{12} \le 5$$

In our notation,

$$A = \begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$

(there are six equations, since one equation was eliminated),

$$A_1 = \begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}, \qquad A_2 = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \end{pmatrix},$$

$\mathbf{b} = \langle 15, 25, 5, 15, 15, 10 \rangle$,
$\mathbf{x} = \langle x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}, x_{11}, x_{12} \rangle$,
$\mathbf{x}^1 = \langle x_1, x_2, x_3, x_4, x_5, x_9 \rangle$,
$\mathbf{x}^2 = \langle x_6, x_7, x_8, x_{10}, x_{11}, x_{12} \rangle$,
$\mathbf{l} = \langle 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 \rangle$,
$\mathbf{l}_1 = \langle 0, 0, 0, 0, 0, 0 \rangle$,
$\mathbf{l}_2 = \langle 0, 0, 0, 0, 0, 0 \rangle$,
$\mathbf{u} = \langle 5, 15, 15, 10, 5, 15, 15, 10, 5, 5, 5, 5 \rangle$,
$\mathbf{u}_1 = \langle 5, 15, 15, 10, 5, 5 \rangle$,
$\mathbf{u}_2 = \langle 15, 15, 10, 5, 5, 5 \rangle$,

and $A\mathbf{x} = \mathbf{b}$, $\mathbf{l} \leq \mathbf{x} \leq \mathbf{u}$, $A_1\mathbf{x}^1 + A_2\mathbf{x}^2 = \mathbf{b}$, $\mathbf{l}_1 \leq \mathbf{x}^1 \leq \mathbf{u}_1$, $\mathbf{l}_2 \leq \mathbf{x}^2 \leq \mathbf{u}_2$.

The set of inequalities (aside from the domain constraints) is empty. It is an easy exercise to get:

$$\begin{aligned} x_1 &= x_6 + x_7 + x_8 + x_{10} + x_{11} + x_{12} - 25, \\ x_2 &= 15 - x_6 - x_{10} \\ x_3 &= 15 - x_7 - x_{11} \\ x_4 &= 10 - x_8 - x_{12} \\ x_5 &= 25 - x_6 - x_7 - x_8 \\ x_9 &= 5 - x_{10} - x_{11} - x_{12} \end{aligned}$$

i.e., $\mathbf{x}^1 = A_1^{-1}\mathbf{b} - A_1^{-1}A_2\mathbf{x}^2$.
The new inequalities

$$A_1\mathbf{l}_1 \leq \mathbf{b} - A_2\mathbf{x}^2 \leq A_1\mathbf{u}_1,$$

are

$$\begin{aligned} &0 \leq x_6 + x_7 + x_8 + x_{10} + x_{11} + x_{12} - 25 \leq 5, \\ &0 \leq 15 - x_6 - x_{10} \leq 15, \\ &0 \leq 15 - x_7 - x_{11} \leq 15, \\ &0 \leq 10 - x_8 - x_{12} \leq 10, \\ &0 \leq 25 - x_6 - x_7 - x_8 \leq 5, \\ &0 \leq 5 - x_{10} - x_{11} - x_{12} \leq 5, \end{aligned}$$

The above set together with the problem specific domain constraints

$$0 \leq x_6 \leq 15$$
$$0 \leq x_7 \leq 15$$
$$0 \leq x_8 \leq 10$$
$$0 \leq x_{10} \leq 5$$
$$0 \leq x_{11} \leq 5$$
$$0 \leq x_{12} \leq 5$$

constitutes the full set of constraints for the new six variable problem. The variables are independent and there are now no equations connecting them.

At this stage the structure of an individual solution is fixed (vector $\mathbf{x}^2$). Using the initialization routine (Section 7.1.4), a population of feasible solutions is created. These individuals undergo evolution; the genetic operators are closed in the space of feasible solutions and cause the system to converge. Having found the best chromosome $\mathbf{x}^2$, we restore the original twelve variables: these constitute the solution vector.

An example of a possible uniform mutation follows.

Assume the following vector undergoes uniform mutation:

$$\mathbf{g} = \langle 7.70, 14.39, 0.00, 0.79, 0.43, 3.03 \rangle$$

and that the second component (corresponding to variable x_7) is selected for mutation. From the set of inequalities it follows that

$$0 \leq x_6 + x_7 + x_8 + x_{10} + x_{11} + x_{12} - 25 \leq 5,$$

which (for fixed $x_6 = 7.70, x_8 = 0.00, x_{10} = 0.79, x_{11} = 0.43, x_{12} = 3.03$) gives

$$13.05 \leq x_7 \leq 18.05.$$

Other inequalities containing x_7 do not impose new limitations; the problem-specific domain constraints require that $x_7 \leq 15$. So, this mutation operation would change the value of variable x_7 from 14.39 into a random value within the range $[13.05, 15.0]$. □

7.1.7 Comparison of the GA approaches

In this section we compare the different genetic algorithm approaches for solving constrained optimization problems and our evolution program, the GENOCOP system.

7.1.7.1 The problem. To get some indication of the usefulness of the proposed approach, we have selected a single example of a 7×7 transportation problem (Table 7.1) and compared results from different approaches.

The problem is to minimize a function

$$f(\mathbf{x}) = f(x_1, \ldots, x_{49}),$$

subject to fourteen (thirteen independent) equations:

	20	**20**	**20**	**23**	**26**	**25**	**26**
27	x_1	x_2	x_3	x_4	x_5	x_6	x_7
28	x_8	x_9	x_{10}	x_{11}	x_{12}	x_{13}	x_{14}
25	x_{15}	x_{16}	x_{17}	x_{18}	x_{19}	x_{20}	x_{21}
20	x_{22}	x_{23}	x_{24}	x_{25}	x_{26}	x_{27}	x_{28}
20	x_{29}	x_{30}	x_{31}	x_{32}	x_{33}	x_{34}	x_{35}
20	x_{36}	x_{37}	x_{38}	x_{39}	x_{40}	x_{41}	x_{42}
20	x_{43}	x_{44}	x_{45}	x_{46}	x_{47}	x_{48}	x_{49}

Table 7.1. The 7×7 transportation problem

$$
\begin{aligned}
x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 &= 27 \\
x_8 + x_9 + x_{10} + x_{11} + x_{12} + x_{13} + x_{14} &= 28 \\
x_{15} + x_{16} + x_{17} + x_{18} + x_{19} + x_{20} + x_{21} &= 25 \\
x_{22} + x_{23} + x_{24} + x_{25} + x_{26} + x_{27} + x_{28} &= 20 \\
x_{29} + x_{30} + x_{31} + x_{32} + x_{33} + x_{34} + x_{35} &= 20 \\
x_{36} + x_{37} + x_{38} + x_{39} + x_{40} + x_{41} + x_{42} &= 20 \\
x_{43} + x_{44} + x_{45} + x_{46} + x_{47} + x_{48} + x_{49} &= 20
\end{aligned}
$$

$$
\begin{aligned}
x_1 + x_8 + x_{15} + x_{22} + x_{29} + x_{36} + x_{43} &= 20 \\
x_2 + x_9 + x_{16} + x_{23} + x_{30} + x_{37} + x_{44} &= 20 \\
x_3 + x_{10} + x_{17} + x_{24} + x_{31} + x_{38} + x_{45} &= 20 \\
x_4 + x_{11} + x_{18} + x_{25} + x_{32} + x_{39} + x_{46} &= 23 \\
x_5 + x_{12} + x_{19} + x_{26} + x_{33} + x_{40} + x_{47} &= 26 \\
x_6 + x_{13} + x_{20} + x_{27} + x_{34} + x_{41} + x_{48} &= 25 \\
x_7 + x_{14} + x_{21} + x_{28} + x_{35} + x_{42} + x_{49} &= 26
\end{aligned}
$$

The behavior of nonlinear optimization algorithms depends markedly on the form of the objective function. It is clear that different solution techniques may respond quite differently.

For purposes of testing, we have arbitrarily classified potential objective functions into those that might conceivably be seen in practical OR problems (practical), those that are mainly seen in textbooks on optimization (reasonable) and those that are more often seen as difficult test cases for optimization techniques (other). In brief, these may be described as follows:

- practical functions
  Typically piece-wise linear cost functions, these appear often in practice either because of data limitations or because the operation of the facility has domains where different costs apply. Often they are not smooth and certainly the derivatives can be discontinuous. They will often cause difficulties for gradient methods though approximations to turn them into differentiable functions are possible. Examples: $A(x)$ and $B(x)$.

- reasonable functions
  These functions are smooth and often simple powers of the flows. They can be further classified into convex and concave functions. Examples: C(x) and D(x).

- other functions
  These typically have multiple valleys (or peaks) with sub-optima that will cause difficulties for any gradient method. They are invented as severe tests of optimization algorithms and, we conjecture, infrequently appear in practice. Examples: E(x) and F(x).

Listed below are the two examples from each group of objective functions used in the tests. They are all separable functions of the components of the solution vector with no cross terms. The continuous versions of their graphs (already modified for the GAMS system) are presented in Figure 7.1.

- function A

$$A(x) = \begin{cases} 0, \text{ if } 0 < x \leq S \\ c_{ij}, \text{ if } S < x \leq 2S \\ 2c_{ij}, \text{ if } 2S < x \leq 3S \\ 3c_{ij}, \text{ if } 3S < x \leq 4S \\ 4c_{ij}, \text{ if } 4S < x \leq 5S \\ 5c_{ij}, \text{ if } 5S < x \end{cases}$$

  where S is less than a typical x value.

- function B

$$B(x) = \begin{cases} c_{ij}\frac{x}{S}, \text{ if } 0 \leq x \leq S \\ c_{ij}, \text{ if } S < x \leq 2S \\ c_{ij}(1 + \frac{x-2S}{S}), \text{ if } 2S < x \end{cases}$$

  where S is of the order of a typical x value.

- function C

$$C(x) = c_{ij}x^2$$

- function D

$$D(x) = c_{ij}\sqrt{x}$$

- function E

$$E(x) = c_{ij}\left(\frac{1}{1 + (x - 2S)^2} + \frac{1}{1 + (x - \frac{9}{4}S)^2} + \frac{1}{1 + (x - \frac{7}{4}S)^2}\right)$$

  where S is of the order of a typical x value.

- function F

$$F(x) = c_{ij}x(\sin(x\frac{5\pi}{4S}) + 1)$$

  where S is of the order of a typical x value.

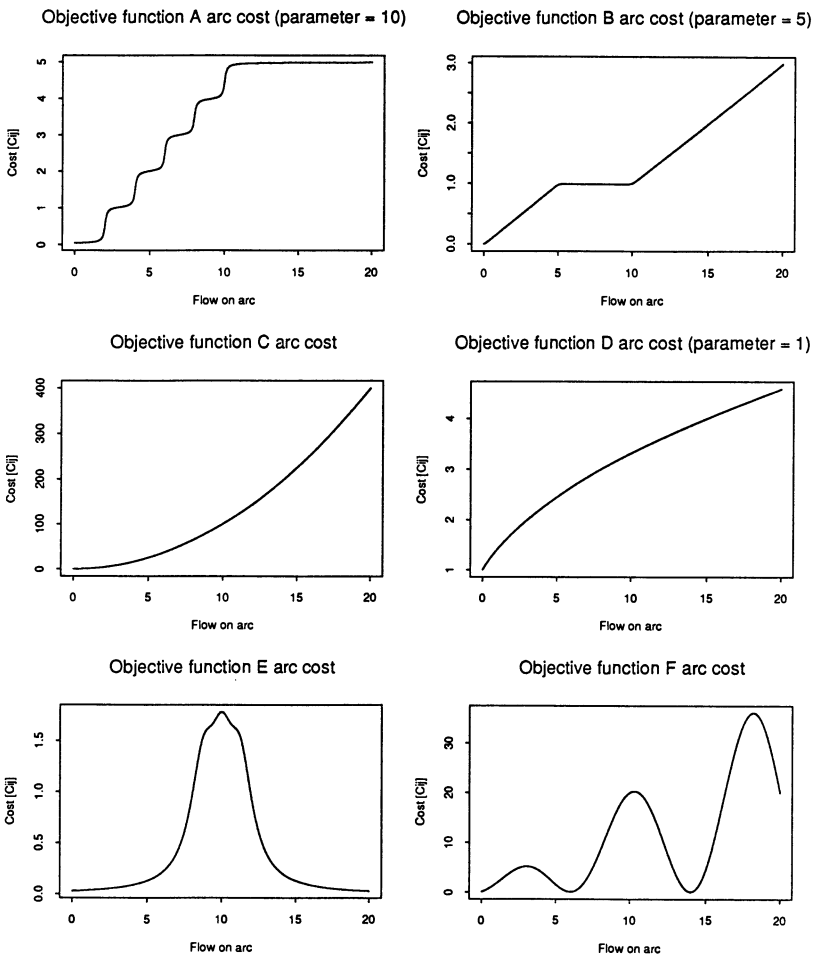

Fig. 7.1. Six test functions A – F

The objective function for the transportation problem is of the form

$$\sum_{ij} f(x_{ij})$$

where $f(x)$ is one of the functions above, the parameters c_{ij} are obtained from the parameter matrix (see Figure 7.2), and S is obtained from the attributes of the problem to be tested. To derive S, it is necessary to estimate the value of a typical x value; this was done by the way of preliminary runs to estimate the number and magnitudes of non-zero x_{ij}'s. In this way the average flow on each arc was estimated and a value for S found. For function A we used $S = 2$, while for B, E, and F, we used $S = 5$.

Note that the objective function is identical on each arc, so a cost-matrix was used to provide a variation between arcs. The matrix provides the c_{ij}'s which act to scale the basic function shape, thus providing 'one degree' of variability.

7.1.7.2 Penalty functions. The major issue in using the penalty function approach is assigning weights to the constraints: these weights play the role of penalties if a potential solution does not satisfy them. In experiments for the above transportation problem, the evaluation function, *eval*, is composed of the optimization function f and the penalty P:

$$eval(\mathbf{x}) = f(\mathbf{x}) + P,$$

For our experiments we have used a suggestion [153] to start with relaxed penalties and to tight them as the run progresses. We used

$$P = k \cdot (\tfrac{t}{T})^p \cdot \overline{f} \cdot \textstyle\sum_{i=1}^{14} d_i$$

where $\overline{f}$ is the average fitness of the population at the given generation t, k and p are parameters, T is the maximum number of generations, and d_i returns the "degree of constraint violation". For example, for a constraint

$$\textstyle\sum_{i \in W} x_i = val,\ W \subseteq \{1, \ldots, 49\},$$

and a chromosome $\langle v_1, \ldots, v_{49} \rangle$, the penalty (degree of constraint violation) d_i is

$$d_i = |\textstyle\sum_{i \in W} v_i - val|.$$

We experimented with various values of p (close to 1), k (close to $\frac{1}{14}$, where 14 is total number of constraints), and $T = 8000$.

This method did not lead to feasible solutions: in over 1200 runs (with different seeds for random number generator and various values for parameters k and p) the best chromosomes (after 8000 generations) violated at least 3 constraints in a significant way. For example, very often the algorithm converged to the solution where the numbers on one diagonal were equal to 20 and all others were zeros:

$x_1 = x_9 = x_{17} = x_{25} = x_{33} = x_{41} = x_{49} = 20$
and $x_i = 0$ for other i.

It is interesting to compare this "solution" with the arc parameter matrix given in Figure 7.2: this matrix has zeros on the diagonal, which was the reason for the faulty convergence.

It should be clear that the methods based on penalty functions must fail for such a heavily constrained problem. The presence of fourteen constraints (which are equations) reduces the probability of locating a feasible solution; when all constraints are essential and must be satisfied, the chances for finding a feasible solution are less than slim.

Number of Sources:	7						
Number of Destinations:	7						
Source Flows:	27	28	25	20	20	20	20
Destination Flows:	20	20	20	23	26	25	26

Arc Parameter Matrix (Source by Destination):

0	21	50	62	93	77	1000
21	0	17	54	67	1000	48
50	17	0	60	98	67	25
62	54	60	0	27	1000	38
93	67	98	27	0	47	42
77	1000	67	1000	47	0	35
1000	48	25	38	42	35	0

Fig. 7.2. Example problem description

7.1.7.3 Repair algorithms. Since the genetic operators often change a feasible solution of the transportation problem into a nonfeasible one, we may try to design appropriate repair algorithms for each operator to restore feasibility.

There are several problems to resolve. To simplify discussion, we will view a solution as a table. Assume that two random points (v_i and v_m, where $i < m$) are selected such that they do not belong to the same row or column. Let us assume that v_i, v_j, v_k, v_m $(i < j < k < m)$ are components of a solution-vector (selected for mutation) such that v_i and v_k, as well as v_j and v_m, belong to a single column, and v_i and v_j, as well as v_k and v_m, belong to a single row. That is, in matrix representation:

...	.	...	.	...
...	.	...	.	...
...	v_i	...	v_j	...
...	.	...	.	...
...	.	...	.	...
...	v_k	...	v_m	...
...	.	...	.	...
...	.	...	.	...

Assume that mutation changes the value of v_i. Further assume, that the new value for the v_i component of the solution vector is now $v_i + c$ $(c > 0)$. The easiest repair algorithm would update the bits for the three other components of solution vector v_j, v_k, v_m, in the following way:

- $v'_j = v_j - c$,
- $v'_k = v_k - c$,

- $v'_m = v_m + c$.

However, it may happen that $v_j < c$, $v_k < c$, or that $v'_m > u_m$ (u_m is the maximum value from the domain of variable x_m). We could set $c = min(v_j, v_k, u_m - v_m)$, and the repair algorithm would work.

The situation is more complex if we try to repair a chromosome after applying the crossover operator. Breaking a vector at a random point could result in a pair of chromosomes violating numerous constraints. If we try to modify these solutions to obey all constraints, they would lose most similarities with the parents. Moreover, the way to do this is far from obvious: if a vector $\mathbf{x}$ is outside the search space, "repairing" it might be as difficult as solving the original problem. Even if we succeeded in building a system based on repair algorithms, such a system would be highly problem specific with little chances for generalizations.

For these reasons we have not developed any system based on this approach. However, we developed another system using specialized data structures and operators: the system (GENETIC-2) is described in Chapter 9.

7.1.7.4 The GENOCOP system. In the selected problem, there are thirteen independent and one dependent equality constraint sets; therefore, we eliminate thirteen variables: $x_1, \ldots, x_8, x_{15}, x_{22}, x_{29}, x_{36}, x_{44}$. All remaining variables are renamed $y_1, \ldots, y_{36}$, preserving order, i.e., $y_1 = x_9$, $y_2 = x_{10}$, ..., $y_6 = x_{14}$, $y_7 = x_{16}$, ..., $y_{36} = x_{49}$. Each chromosome is a float vector $\langle y_1, \ldots, y_{36} \rangle$. Each of these variables has to satisfy four two-sided inequalities resulting from the original domains and our transformations. For example, for the first variable y_1 (which corresponds to the original x_9) we have:

$$0 \leq y_1 \leq 20$$
$$93 - \sum_{i=2}^{30} y_i + y_{31} \leq y_1 \leq 113 + y_{31} - \sum_{i=2}^{30} y_i$$
$$\sum_{i=31}^{36} y_i - 20 - y_7 - y_{13} - y_{19} - y_{25} \leq y_1 \leq \sum_{i=31}^{36} y_i - 20 - y_7 - y_{13} - y_{19} - y_{25}$$
$$8 - \sum_{i=2}^{6} y_i \leq y_1 \leq 28 - \sum_{i=2}^{6} y_i.$$

The GENOCOP finds an optimal chromosome after a number of generations, and the equalities allow us to restore the original forty-nine variables; these constitute the sought solution.

Again, for each function (A–F) we have repeated 3 separate runs of 8,000 generations; the best of those are reported in Table 7.2, along with intermediate results at some generation intervals.

The times of computations in the GENOCOP system are not meaningful; this is due to the pilot implementation of the system for the 7×7 problem, which was partially hard-coded. The parameters used by GENOCOP in all runs are displayed in Table 7.3.

	Generations						
Function	1	100	500	1,000	2,000	4,000	8,000
A	1085.8	856.5	273.4	230.5	153.0	94.5	24.15
B	932.4	595.8	410.6	258.4	250.3	209.2	205.60
C	83079.1	28947.2	14122.7	4139.0	2944.1	2772.7	2571.04
D	1733.6	1106.9	575.3	575.3	480.2	480.2	480.16
E	225.2	207.2	204.9	204.9	204.9	204.9	204.82
F	3799.3	1719.0	1190.1	550.8	320.9	166.4	119.61

Table 7.2. Progress of the GENOCOP for the 7 × 7 transportation problem

Parameter	Value
pop_size	40
prob_mut_{um}	.08
prob_mut_{bm}	.03
prob_mut_{nm}	.07
prob_cross_{sc}	.10
prob_cross_{sa}	.10
prob_cross_{wa}	.10
a	.25
b	2.0
prop	0.5

Table 7.3. Parameters used for the GENOCOP on the 7 × 7 transportation problem: population size (pop_size), probability of uniform mutation (prob_mut_{um}), probability of boundary mutation (prob_mut_{bm}), probability of non-uniform mutation (prob_mut_{nm}), probability of simple crossover (prob_cross_{sc}), probability of single arithmetical crossover (prob_cross_{sa}), probability of whole arithmetical crossover (prob_cross_{wa}), a coefficient (a) for whole arithmetical crossover, a coefficient (b) for function Δ of the non-uniform mutation, and *prop*, the proportion of initial potential solutions from the boundary of the solution space (Section 7.1.4)

7.1.7.5 Comparison. In testing an optimization algorithm on a linear problem it is possible to compare its solution with the known global optimum found using the standard linear programming technique — and, therefore, to determine how efficient the optimization algorithm is in absolute terms. For nonlinear objective functions the optimum may not be known. In such cases the results must be compared with those of other systems that can converge to a possibly good optimum.

We have chosen to compare the GENOCOP algorithm with GAMS [24]: we used the MINOS version of the optimizer.

For functions C, E, and F, the GAMS application was straightforward: using built–in nonlinear functions. Due to the requirement for gradient estimation of objective functions, GAMS could not handle functions A, B and D directly. In the case of A and B, the expression could not be formulated in GAMS while in the case of D (the square root function) difficulty was encountered in measuring gradients near zero. Therefore, we made the following modifications to the problem for the GAMS runs:

- function A
  Separate arc-tangent functions are used to approximate each of the five steps. A parameter, P_A, was used to control the 'tightness' of the fit. The cost on arc [i,j] is:

$$c_{ij} \cdot \begin{pmatrix} \arctan(P_A(x_{ij} - S))/\pi + \frac{1}{2} + \\ \arctan(P_A(x_{ij} - 2S))/\pi + \frac{1}{2} + \\ \arctan(P_A(x_{ij} - 3S))/\pi + \frac{1}{2} + \\ \arctan(P_A(x_{ij} - 4S))/\pi + \frac{1}{2} + \\ \arctan(P_A(x_{ij} - 5S))/\pi + \frac{1}{2} \end{pmatrix}$$

- function B
  The arc-tangent function was again used, this time to approximate each of the three gradients. A parameter, P_B, was used to control the tightness of the fit. The cost on arc [i,j] is:

$$c_{ij} \cdot \begin{pmatrix} (\frac{x_{ij}}{S}) \cdot (\arctan(P_B x_{ij})/\pi + \frac{1}{2}) + \\ (1 - \frac{x_{ij}}{S}) \cdot (\arctan(P_B(x_{ij} - S))/\pi + \frac{1}{2}) + \\ (\frac{x_{ij}}{S} - 2) \cdot (\arctan(P_B(x_{ij} - 2S))/\pi + \frac{1}{2}) \end{pmatrix}$$

- function D
  In order to avoid gradient problems at or near zero, the function D was changed to:

$$D'(x) = D(x + \epsilon) .$$

For each problem, multiple GAMS runs were made under different values of the parameter and the best result chosen. (The final objective function was calculated after the optimization based on the true function, rather than the modified function used within the optimization itself). GAMS was run on an Olivetti 386 with a math co-processor.

The comparative results of GAMS and GENOCOP are reported in Table 7.4. In addition, all the best transportation flows for both systems and all six functions are given in Figure 7.3.

It is clear that the proposed method performs much better than than the commercial package GAMS for most of the problems attempted. It is interesting to see that for practical cost functions (A and B) and for highly "irregular" cost function (F), the performance of GENOCOP is very good. For other functions (relatively smooth) the performance of GAMS and GENOCOP is quite similar.

Function	GAMS	GENOCOP	% difference
A	96.00	24.15	+297.52%
B	1141.60	205.60	+455.25%
C	2535.29	2571.04	−1.41%
D	565.15	480.16	+17.70%
E	208.25	204.82	+1.67%
F	43527.54	119.61	+36291.22%

Table 7.4. GENOCOP versus GAMS: results for 7×7 problem with cost matrix of Figure 7.2

Comparing GENOCOP and GAMS we should also stress the fact that the GENOCOP system allows the user to provide the desired number of iterations, which influences the precision of the result. Also, any time during the execution of the program the user can request a current solution which always obeys the constraints. On the other hand, the GAMS is "all or nothing" proposition. An additional difference is that the GENOCOP's user can declare any function for optimization, as a C procedure. An example declaration (for function A) may look like:

$$
\begin{aligned}
&sum = 0.0;\\
&\quad \text{for } (i = 0;\ i < k;\ ++i)\\
&\qquad \text{for } (j = 0;\ j < n;\ ++j)\{\\
&\qquad\quad \text{if } ((x[i][j] \geq 0.0)\ \&\&\ (x[i][j] \leq 2.0)\\
&\qquad\qquad sum = sum; \text{ else}\\
&\qquad\quad \text{if } ((x[i][j] > 2.0)\ \&\&\ (x[i][j] \leq 4.0))\\
&\qquad\qquad sum = sum + cost[i][j] * x[i][j]/5.0; \text{ else}\\
&\qquad\quad \text{if } ((x[i][j] > 4.0)\ \&\&\ (x[i][j] \leq 6.0))\\
&\qquad\qquad sum = sum + 2 * cost[i][j] * x[i][j]/5.0; \text{ else}\\
&\qquad\quad \text{if } ((x[i][j] > 6.0)\ \&\&\ (x[i][j] \leq 8.0))\\
&\qquad\qquad sum = sum + 3 * cost[i][j] * x[i][j]/5.0; \text{ else}\\
&\qquad\quad \text{if } ((x[i][j] > 8.0)\ \&\&\ (x[i][j] \leq 10.0))\\
&\qquad\qquad sum = sum + 4 * cost[i][j] * x[i][j]/5.0; \text{ else}\\
&\qquad\quad \text{if } (x[i][j] > 10.0)\\
&\qquad\qquad sum = sum + 5 * cost[i][j] * x[i][j]/5.0;\\
&\qquad \}
\end{aligned}
$$

Note that the implemented function is not continuous. On the other hand, in GAMS the user has to approximate such a noncontinuous function by a continuous one, as we did and discussed earlier in this section.

The much better results of GENOCOP against GAMS are mainly due to the fact that GENOCOP does not make any assumptions on the characteristic of the optimized function. Because of that, GENOCOP avoids some local optima which are reported as final solutions in the other system.

GENOCOP: function A Cost = 24.15

20.00	0.00	0.00	1.92	1.62	1.47	1.97
0.00	20.00	2.87	1.76	1.46	1.89	0.00
0.00	0.00	17.12	1.90	1.99	1.10	2.87
0.00	0.00	0.00	16.25	0.85	1.37	1.51
0.00	0.00	0.00	0.00	19.65	0.00	0.34
0.00	0.00	0.00	0.43	0.41	19.15	0.00
0.00	0.00	0.00	0.71	0.00	0.00	19.28

GAMS: function A Cost = 96.00

20.00	1.28	0.94	1.58	1.52	1.58	0.08
0.00	18.71	0.38	1.58	1.57	0.12	5.61
0.00	0.00	18.66	1.55	1.47	1.58	1.72
0.00	0.00	0.00	18.27	1.25	0.00	0.47
0.00	0.00	0.00	0.00	19.47	0.52	0.00
0.00	0.00	0.00	0.00	0.00	20.00	0.00
0.00	0.00	0.00	0.00	0.71	1.18	18.10

GENOCOP: function B Cost = 205.60

20.00	0.00	0.00	0.00	0.00	7.00	0.00
0.00	20.00	6.00	2.00	0.00	0.00	0.00
0.00	0.00	14.00	0.00	0.00	0.00	11.00
0.00	0.00	0.00	20.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	20.00	0.00	0.00
0.00	0.00	0.00	0.00	2.00	18.00	0.00
0.00	0.00	0.00	1.00	4.00	0.00	15.00

GAMS: function B Cost = 1141.60

20.00	0.00	0.00	0.00	0.00	0.00	7.00
0.00	18.99	0.00	0.00	9.01	0.00	0.00
0.00	1.00	20.00	0.00	0.00	0.00	3.99
0.00	0.00	0.00	20.00	0.00	0.00	0.00
0.00	0.00	0.00	3.00	16.99	0.00	0.00
0.00	0.00	0.00	0.00	0.00	20.00	0.00
0.00	0.00	0.00	0.00	0.00	4.99	15.00

GENOCOP: function C Cost = 2571.04

20.00	0.00	1.29	1.86	1.58	2.11	0.14
0.00	20.00	1.42	1.90	2.02	0.07	2.57
0.00	0.00	17.28	1.13	1.13	1.72	3.72
0.00	0.00	0.00	18.09	1.10	0.00	0.79
0.00	0.00	0.00	0.00	19.55	0.44	0.00
0.00	0.00	0.00	0.00	0.00	20.00	0.00
0.00	0.00	0.00	0.00	0.60	0.63	18.75

GAMS: function C Cost = 2535.29

20.00	0.52	0.85	1.82	1.58	2.07	0.13
0.00	19.47	1.85	1.89	2.03	0.14	2.58
0.00	0.00	17.29	1.17	1.07	1.75	3.70
0.00	0.00	0.00	18.10	1.27	0.04	0.57
0.00	0.00	0.00	0.00	19.73	0.26	0.00
0.00	0.00	0.00	0.00	0.00	0.00	20.00
0.00	0.00	0.00	0.00	0.29	0.70	19.99

GENOCOP: function D Cost = 480.16

20.00	7.00	0.00	0.00	0.00	0.00	0.00
0.00	13.00	15.00	0.00	0.00	0.00	0.00
0.00	0.00	5.00	0.00	0.00	0.00	20.00
0.00	0.00	0.00	20.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	20.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	20.00	0.00
0.00	0.00	0.00	3.00	6.00	5.00	6.00

GAMS: function D Cost = 565.15

20.00	2.00	0.00	0.00	0.00	5.00	0.00
0.00	18.00	4.00	0.00	6.00	0.00	0.00
0.00	0.00	16.00	3.00	0.00	0.00	6.00
0.00	0.00	0.00	20.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	20.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	20.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	20.00

GENOCOP: function E Cost = 204.82

0.56	0.00	0.00	0.43	0.00	0.00	26.00
0.00	0.25	0.00	2.00	0.75	25.00	0.00
0.00	0.00	0.00	0.00	25.00	0.00	0.00
19.43	0.00	0.00	0.56	0.00	0.00	0.00
0.00	19.75	0.00	0.00	0.25	0.00	0.00
0.00	0.00	20.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	20.00	0.00	0.00	0.00

GAMS: function E Cost = 208.25

0.00	0.00	0.00	0.00	1.00	0.00	26.00
0.00	0.00	0.45	0.00	2.54	25.00	0.00
0.00	0.00	0.77	23.00	1.22	0.00	0.00
0.00	0.00	0.00	0.00	20.00	0.00	0.00
0.00	0.00	18.77	0.00	1.22	0.00	0.00
0.00	20.00	0.00	0.00	0.00	0.00	0.00
20.00	0.00	0.00	0.00	0.00	0.00	0.00

GENOCOP: function F Cost = 119.61

8.30	6.30	6.38	0.00	6.00	0.00	0.00
0.00	13.69	0.30	14.00	0.00	0.00	0.00
6.00	0.00	13.30	0.00	0.00	0.00	5.69
5.69	0.00	0.00	9.00	0.00	0.00	5.30
0.00	0.00	0.00	0.00	20.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	20.00	0.00
0.00	0.00	0.00	0.00	0.00	5.00	15.00

GAMS: function F Cost = 43527.54

0.00	0.00	0.00	22.49	0.71	3.79	0.00
0.00	0.00	0.00	0.50	5.28	0.00	22.20
0.00	0.00	0.00	0.00	0.00	21.20	3.79
0.00	0.00	0.00	0.00	20.00	0.00	0.00
0.00	0.00	20.00	0.00	0.00	0.00	0.00
0.00	20.00	0.00	0.00	0.00	0.00	0.00
20.00	0.00	0.00	0.00	0.00	0.00	0.00

Fig. 7.3. Transportation flows

It also seems that the set of parameters we used for our experiments in GENOCOP should give satisfactory performance for many types of functions (in all experiments for all functions A–F the set of parameters was invariant; we were changing only the seed for the random number generator). However, some further experiments are required, and it might be desirable to add a parameter tuning module to the GENOCOP system.

7.2 An evolution program: the GAFOC system

Now we return to dynamic control problems discussed earlier in the previous chapter. Let us denote by $\mathbf{x} = (x_0, \ldots, x_N)$ a vector of states, and by $\mathbf{u} = (u_0, \ldots, u_{N-1})$ a vector of controls of a system. We consider a class $\mathcal{Z}$ of optimization problems with a linear state equation, that can be formulated as follows:

Optimize a function $J(\mathbf{x}, \mathbf{u})$, subject to

$$x_{k+1} = a \cdot x_k + b \cdot u_k, \quad k = 0, 1, \ldots, N-1,$$

and some (or all) constraints from the following categories:

1. Final state $x_N = C$, where C is some constant,
2. Boundaries for all states: $\alpha \leq x_k \leq \beta$, $k = 1, \ldots, N$,
3. Boundaries for all controls: $\gamma \leq u_k \leq \delta$, $k = 0, 1, \ldots, N-1$,
4. Relationship between states and controls, $\tau x_k \leq u_k$ or $\tau x_k \geq u_k$, $k = 0, 1, \ldots, N-1$.

For a given problem, parameters a and b are given together with the initial state x_0 of the system and the objective function J.

This formulation is general enough to handle a large class of standard discrete optimal control problems. In Chapter 6 we considered three optimal control problems as test cases for new "genetic" operators responsible for fine local tuning. The first two, the linear-quadratic problem and the harvest problem belong to the class $\mathcal{Z}$. The linear-quadratic problem was to minimize a function J:

$$J(\mathbf{x}, \mathbf{u}) = q \cdot x_N^2 + \sum_{k=0}^{N-1} (s \cdot x_k^2 + r \cdot u_k^2),$$

subject to

$$x_{k+1} = a \cdot x_k + b \cdot u_k, \quad k = 0, 1, \ldots, N-1.$$

Note that we do not have any additional constraints here ($\alpha, \gamma = -\infty$, $\beta, \delta = +\infty$).

The harvest problem was defined as maximization of the function J:

$$J(\mathbf{u}) = \sum_{k=0}^{N-1} \sqrt{u_k},$$

subject to the equation of growth

$$x_{k+1} = a \cdot x_k - u_k,$$

(i.e., $b = -1$). Additionally, there was one final state constraint $x_0 = x_N$, (i.e., $x_N = C$, where the constant C has the same value as initial state x_0), and $u_k \geq 0$ (i.e., $\gamma = 0$) for $k = 0, \ldots, N-1$.

In this section we present an evolution program (GAFOC, for Genetic Algorithm For Optimal Control problems) for the class of problems in $\mathcal{Z}$. This class includes typical (daily life) applications of optimal control problems.

7.2.1 GENOCOP and GAFOC

Note that since every state x_{k+1} is a (linear) function of the previous state x_k and a control u_k, the objective function $J(\mathbf{x}, \mathbf{u})$ depends only on a control vector $\mathbf{u}$. Also, every state x_k is a function of x_0, and $u_0, \ldots, u_{k-1}$, i.e.,

$$x_k = a^k x_0 + b \sum_{i=0}^{k-1} a^i u_{k-1-i},$$

$(k = 1, 2, \ldots, N)$. In all considered constraints (which, for problems in $\mathcal{Z}$, are linear) on components of the vector $\mathbf{u}$, we can replace x_k by the above (linear) expression. In that way we can transform any problem in $\mathcal{Z}$ into an optimization problem with linear constraints. In other words, the set of problems solvable by GAFOC is a subset of problems solvable by GENOCOP system.

For example, the linear-quadratic problem becomes:

$$\min J(\mathbf{u}) = q \cdot (a^N \cdot x_0 + b \sum_{i=0}^{N-1} a^i u_{N-1-i})^2 + s \cdot (x_0^2 + u_0^2) + \sum_{k=1}^{N-1} (s \cdot (a^k x_0 + b \sum_{i=0}^{k-1} a^i u_{k-1-i})^2 + r \cdot u_k^2).$$

The harvest problem is:

$$\max J(\mathbf{u}) = \sum_{k=0}^{N-1} \sqrt{u_k},$$

subject to

$$x_0 = x_N = a^N \cdot x_0 - \sum_{i=0}^{N-1} a^i u_{k-1-i} \text{ and}$$
$$u_k \geq 0, \text{ for } k = 0, 1, \ldots, N-1.$$

If we can use GENOCOP to optimize these problems, what would be the advantage of building a specific evolution program (GAFOC) for optimal control problems (from the class of problems $\mathcal{Z}$)?

Actually, there are some advantages. Throughout the book we see a tradeoff between weaker and stronger methods for problem solving. Genetic algorithms as such are too weak to be useful in many applications. This is why we incorporated problem-specific knowledge into our systems, getting evolution programs. GENOCOP is a stronger method than a GA and can be applied to a function with linear constraints only. The results of GENOCOP were much better than those of GAs. We hope for the same edge when we move closer to the specific class of optimal control problems. GAFOC should outperform GENOCOP for the problems from $\mathcal{Z}$.

We use similar reasoning in Chapter 9, where we discuss an evolution program (GENETIC-2) for the linear transportation problem. GENETIC-2 works only for transportation problems (which are a subset of problems handled by the GENOCOP system because of the linear constraints), and gives slightly better results than GENOCOP.

The second advantage of building a specific evolution program for optimal control problems is a possibility of generalizing the system into one which handles nonlinear state equations: we can have

$$x_{k+1} = g(x_k, u_k), \; k = 0, 1, \ldots, N-1,$$

where the function g need not be linear. In fact, if the above equation provides the value of u_k as a function of x_k and x_{k+1}, i.e.,

$$u_k = h(x_k, x_{k+1}),\ k = 0, 1, \ldots, N-1,$$

then after a few modifications, the system GAFOC would still function.

An additional reason for presenting the GAFOC system is to demonstrate the elegance and efficiency of an evolution program developed for a particular class of problems. Note that the classical GA, as a weak method, would maintain a population of potential solutions (vectors $\mathbf{u}$). During the evaluation part, the GA system would generate a sequence of states ($\mathbf{x}$) and impose some penalties for those states which are outside the bounds $[\alpha, \beta]$ or which do not satisfy other constraints ($u_k \leq x_k$, $u_k \geq x_k$, or $x_N = C$). This approach would have little chance of generating any legitimate solution. On the other hand, an evolution program would use problem-specific knowledge (i.e., for the class of problems $\mathcal{Z}$) to hide the constraints using appropriate data structures and "genetic" operators. We discuss these issues in the next section.

7.2.2 The GAFOC system

We discuss the major components of the GAFOC system in turn.

7.2.2.1 Representation. As in the GENOCOP system, GAFOC maintains a population of float number vectors; each vector $\mathbf{u} = (u_0, u_1, \ldots, u_{N-1})$ is a control vector for any problem in $\mathcal{Z}$.

7.2.2.2 Initial population. As discussed at the beginning of this section, there are a few constraints which may appear for a given problem. These are:

1. Final state $x_N = C$, where C is a constant,
2. Boundaries for states: $\alpha \leq x_k \leq \beta$, $k = 1, \ldots, N$,
3. Boundaries for controls: $\gamma \leq u_k \leq \delta$, $k = 0, 1, \ldots, N-1$,
4. Relationship between states and controls, $\tau x_k \leq u_k$ or $\tau x_k \geq u_k$, $k = 0, 1, \ldots, N-1$.

The GAFOC system initializes a population in a way that all of these constraints are satisfied. The procedure **initialization** used by the GAFOC system is described below:

procedure initialization;
begin
  success = FALSE
  status = TRUE
  if $(\tau x_k \leq u_k)$ **then** *left* = TRUE **else** *left* = FALSE
  if $(\tau x_k \geq u_k)$ **then** *right* = TRUE **else** *right* = FALSE
  if $(x_N = C)$ **then** *final* = TRUE **else** *final* = FALSE
  $k = 0$
  repeat
    if *left* **then** $lhs = \tau x_k$ **else** $lhs = -\infty$
    if *right* **then** $rhs = \tau x_k$ **else** $rhs = +\infty$
    if $(b > 0)$ **then**
      left-range $= \max\{(\alpha - a \cdot x_k)/b, \gamma, lhs\}$
      right-range $= \min\{(\beta - a \cdot x_k)/b, \delta, rhs\}$
    else *left-range* $= \max\{(\beta - a \cdot x_k)/b, \gamma, lhs\}$
      right-range $= \min\{(\alpha - a \cdot x_k)/b, \delta, rhs\}$
    if *right-range* $-$ *left-range* < 0 **then** *status* = *FALSE*
    if *status* **then** generate the k-th component of the vector $\mathbf{u}$, such that:
      left-range $\leq u_k \leq$ *right-range*
    calculate $x_{k+1} = a \cdot x_k + b \cdot u_k$
    $k = k + 1$
  until $k = N$
  if *final* **then** $u_{N-1} = (C - a \cdot x_{N-1})/b$
  if $(\max\{\gamma, lhs\} \leq u_{N-1} \leq \min\{\delta, rhs\}$ AND *status*$)$ **then** *success* = TRUE
end

The Boolean variables *left*, *right*, and *final* are initialized to TRUE if the problem requires relationship between states and controls ($\tau x_k \leq u_k$ or $\tau x_k \geq u_k$) or a constraint on its final state ($x_N = C$), respectively, otherwise they are initialized to FALSE. Next Boolean variable, *status*, is initialized to TRUE; its value is changed into FALSE if it is impossible to generate u_k, i.e., *right-range* is smaller than *left-range* in some iteration. The last Boolean variable, *success*, retains is initial value TRUE only if the initialization procedure generated a control vector (*status* = TRUE) and the vector satisfies all constraints.

The procedure **initialization** generates a control vector $\mathbf{u}$; for $b > 0$, its k-th component is randomly selected from the range

$$\max\{(\alpha - a \cdot x_k)/b, \gamma, lhs\} \leq u_k \leq \min\{(\beta - a \cdot x_k)/b, \delta, rhs\}$$

(the range is different for $b < 0$). The presence of γ and δ in the above formula is clear: all controls u_k must be taken from the range $[\gamma, \delta]$. Also, if $x_k \leq u_k$, then $lhs = x_k$, enforcing this relationship between state and control (similarly for $u_k \leq x_k$, $rhs = x_k$). Finally,

$$(\alpha - a \cdot x_k)/b \leq u_k \leq (\beta - a \cdot x_k)/b,$$

is equivalent to $(b > 0)$

$$\alpha - a \cdot x_k \leq b \cdot u_k \leq \beta - a \cdot x_k,$$

and to

$$\alpha \leq a \cdot x_k + b \cdot u_k \leq \beta,$$

which guarantee that the next state, x_{k+1} stays in boundaries $[\alpha, \beta]$. Similar reasoning applies to the case of $b < 0$.

If $x_N = C$ (i.e., *final* = TRUE), the procedure replaces the last component of the vector **u**:

$$u_{N-1} = (C - a \cdot x_{N-1})/b.$$

This implies

$$x_N = a \cdot x_{N-1} + b \cdot u_{N-1} = C,$$

however, the new component u_{N-1} need not belong to the range $[\gamma, \delta]$, nor satisfy relationship between state and control ($\tau x_{N-1} \leq u_{N-1}$ or $u_{N-1} \leq \tau x_{N-1}$). In such a case the initialization would fail (*success* = FALSE).

The success rate of such initialization procedure might be quite low — let us consider the following example.

Assume $\alpha = 0$, $\beta = 4$, $\gamma = -1$, $\delta = 1$, $x_0 = 1$, $a = 2$, $b = 1$. This means that

$$\begin{aligned} &0 \leq x_k \leq 4, \\ &-1 \leq u_k \leq 1, \\ &x_{k+1} = 2 \cdot x_k + u_k, \\ &x_0 = 1, \end{aligned}$$

for $k = 0, \ldots, N-1$.

The first component u_0 of the control vector is randomly selected from the range $[-1, +1]$: for any number ξ from this range, the next state $x_1 = 2 \cdot x_0 + \xi = 2 + \xi$ stays within its boundaries, $[0, 4]$. Assume, then, that the procedure selects $\xi = 0.6$ as the value of the u_0; consequently, $x_1 = 2.6$. However, during the next iteration, it is impossible to generate value for u_1: even for the smallest $u_1 = -1$, the value for the next state $x_2 = 2 \cdot x_1 + u_1 = 5.2 - 1 = 4.2$, which is outside the required range. Consequently, *status* = FALSE and *success* = FALSE.

It seems that for the above problem the generated values of control vector **u** should stay very low (close to -1) all the time. One "larger" value of u_k would push the state component outside the required range in the next few iterations. This is why the initialization procedure, for cases like the above example, may have a low success rate.

To increase the success rate of the initialization procedure, a minor modification was introduced: we modified the way the components of the vector **u** are generated. First we select a random fraction μ from the range $[0, 1]$ and the random bit b (0 or 1). If $b = 0$ we generate components of the vector **u** from the range [*left-range*, r], if $b = 1$ we generate components of the vector **u** from the range [r, *right-range*], where r =*left-range* + (*right-range* − *left-range*)·μ.

Let us refer to the last example to see the improvement. If $\mu = 0.02$ and $b = 0$, the first component of the vector $\mathbf{u}$ is generated from the range $[-1, -0.96]$. Consequently, the resulting state x_1 stays within the range $[1, 1.04]$. Note that $\mu = 0.02$ and $b = 0$ stay constant through all iterations during the initialization of a single vector $\mathbf{u}$, resulting in values u_k close to -1. This allows us to build much longer vectors than with random selection from the range $[-1, 1]$.

We performed many experiments with the initialization procedure on many optimal control problems. For all test cases of the linear–quadratic problem (Chapter 6, Figure 6.2) the success rate of the initialization procedure was 100%, for other problems (like the harvest problem), the success rate was lower. However, the performace of the initialization procedure is not very important: the procedure is run only once at the beginning of the evolution program until we complete the whole set of individuals of the population.

7.2.2.3 Evaluation. For any problem in $\mathcal{Z}$, the objective function J is given. As described earlier, any control vector $\mathbf{u}$ implies all states x_k, $(k = 1, \ldots, N)$, hence $eval(\mathbf{u}) = J(\mathbf{u}, \mathbf{x})$.

7.2.2.4 Genetic operators. The operators used in the GAFOC system were already described in Chapter 6.

The arithmetical crossover, defined as a linear combination of two (legitimate) control vectors, $\mathbf{u}^1$ and $\mathbf{u}^2$:

$$\mathbf{u} = a \cdot \mathbf{u}^1 + (1 - a) \cdot \mathbf{u}^2,$$

always produces a legitimate solution (for $a \in [0..1]$). This means that if a pair of control vectors $\mathbf{u}^1 = (u_0^1, \ldots, u_{N-1}^1)$ and $\mathbf{u}^2 = (u_0^2, \ldots, u_{N-1}^2)$ satisfy all constraints:

$\alpha \leq x_k^i \leq \beta$, for $k = 0, \ldots, N$, and $i = 1, 2$,
$\gamma \leq u_k^i \leq \delta$, for $k = 0, \ldots, N-1$, and $i = 1, 2$,
$u_k^i \leq \tau x_k^i$ (or $\tau x_k^i \leq u_k^i$), for $k = 0, \ldots, N-1$, and $i = 1, 2$, and
$x_N^i = C$, for $i = 1, 2$,

then their offspring $\mathbf{u} = (u_0, \ldots, u_{N-1})$ (after applying arithmetical crossover) would satisfy these constraints as well:

$\alpha \leq x_k \leq \beta$, for $k = 0, \ldots, N$,
$\gamma \leq u_k \leq \delta$, for $k = 0, \ldots, N-1$,
$u_k \leq \tau x_k$ (or $\tau x_k \leq u_k$), for $k = 0, \ldots, N-1$,
$x_N = C$.

The simple crossover, however, may produce non-legitimate offspring: the parents

$\mathbf{u}^1 = (u_0^1, \ldots, u_{N-1}^1)$ and
$\mathbf{u}^2 = (u_0^2, \ldots, u_{N-1}^2)$

would produce the following offspring

$$\mathbf{u}'^1 = (u'^1_0, \ldots, u'^1_{N-1}) = (u^1_0, \ldots, u^1_k, u^2_{k+1}, \ldots, u^2_{N-1}) \text{ and}$$
$$\mathbf{u}'^2 = (u'^2_0, \ldots, u'^2_{N-1}) = (u^2_0, \ldots, u^2_k, u^1_{k+1}, \ldots, u^1_{N-1})$$

(for some $0 \leq k \leq N-2$). The last components of these vectors are modified into

$$u'^1_{N-1} = (C - a \cdot x'^1_{N-1})/b \text{ and } u'^2_{N-1} = (C - a \cdot x'^2_{N-1})/b,$$

where $\mathbf{x}'^1$ and $\mathbf{x}'^2$ are new state vectors resulting from controls $\mathbf{u}'^1$ and $\mathbf{u}'^2$. The vectors $\mathbf{u}'^1$ and $\mathbf{u}'^2$ need not satisfy the constraints: for example $u'^1_p = u^2_p$ $(p > k)$ was selected from the range

$$[(\alpha - a \cdot x^1_p)/b,\ (\beta - a \cdot x^1_p)/b],$$

whereas now u'^1_p must satisfy

$$[(\alpha - a \cdot x'^1_p)/b,\ (\beta - a \cdot x'^1_p)/b].$$

The system would discard such offspring to maintain a population of legitimate individuals only. However, as the evolution process progresses, the success rate of simple crossover would increase: this is due to the convergence of the population.

The non-uniform mutation is responsible for the fine-tuning capabilities of the system. Again, its usefulness increases (together with its success rate) along with the evolution process.

7.2.2.5 Parameters. For all experiments, we have used $pop\_size = 70$. The runs were made for 40,000 generations. The parameter a (for arithmetical crossover) was 0.25; the probability of applying arithmetical and simple crossovers were $p_{ac} = 0.15$ and $p_{sc} = 0.05$, respectively. The probability of applying non-uniform mutation was $p_{nm} = 0.10$.

7.2.3 Experiments and results

We have performed a series of experiments with the GAFOC system. Some of the results were reported earlier in Chapter 6, where we tested specialized operators on three optimal control problems. Below we report the results on another test problem, which involves some yet untested constraints.

The selected test case was the problem of optimal allocation between consumption and investment:

$$\max \textstyle\sum_{k=0}^{N-1} q^k \cdot u_k^{1-v} + q^N \cdot s \cdot x_N^{1-v},$$

subject to

$$x_{k+1} = a(x_k - u_k), \text{ for } k = 0, 1, \ldots, N-1,$$

and additional constraints:

$x_k \geq 0$, for $k = 0, 1, \ldots, N$,
$u_k \geq 0$, for $k = 0, 1, \ldots, N-1$, and
$u_k \leq x_k$, for $k = 0, 1, \ldots, N-1$ ($right$ = TRUE, $left$ = FALSE, $\tau = 1$).

The optimal control vector for this problem is given by the formula

$$\hat{u}_k = x_k / A_k^{1/v},$$

where $A_k^{1/v}$ satisfies the difference equation

$$A_k^{1/v} = 1 + \gamma \cdot A_{k+1}^{1/v},$$

where A_N is given ($A_N = s$), and $\gamma = (q \cdot a^{1-v})^{1/v}$.
The optimal value of the objective function is

$$J^* = A_0 x_0^{1-v}.$$

It is possible to show that for $N \longrightarrow \infty$

$\hat{u}_k \longrightarrow (1-\gamma)x_k$, and
$J^* \longrightarrow x_0^{1-v}/(1-\gamma)^v$.

The problem was solved for the following parameter values: $a = 1.02$, $q = 0.95$, $s = 0.5$, $v = 0.5$, $x_0 = 1$, and $N = 5, 10, 20, 45$. The **initialization** procedure gave a success rate of 100% for the above problem. After 10,000 generations, the GAFOC system provided exact (i.e., six decimal places) solutions for $N = 5, 10, 20$, and 45 (see Table 7.5).

N	Exact solution	GAFOC	Error
5	2.105094	2.105094	0.000%
10	2.882455	2.882455	0.000%
20	3.198550	3.198550	0.000%
45	3.505587	3.505587	0.000%

Table 7.5. Comparison of solutions for the optimal allocation problem

We did not compare the results of GANOCOP and GAFOC, since the full version of the GENOCOP system is still under construction. However, we expect GAFOC, as a stronger method which incorporates problem-specific knowledge by means of representing state and control vectors separately, to perform better than GENOCOP on optimal control problems. A similar experiment is described in Chapter 9, where we compare GENETIC-2 (a system for optimizing transportation problems) with GENOCOP. As expected, GENETIC-2 perform better than GENOCOP.

8. Evolution Strategies and Other Methods

It were not best that we should think alike;
it is difference of opinion
that makes horse races.

Mark Twain, *Pudd'nhead Wilson*

Evolution strategies (ESs) are algorithms which imitate the principles of natural evolution as a method to solve parameter optimization problems [7], [162]. They were developed in Germany during the 1960s. As stated in [162]:

> "In 1963 two students at the Technical University of Berlin met and were soon collaborating on experiments which used the wind tunnel of the Institute of Flow Engineering. During the search for the optimal shapes of bodies in a flow, which was then a matter of laborious intuitive experimentation, the idea was conceived of proceeding strategically. However, attempts with the coordinate and simple gradient strategies were unsuccessful. Then one of the students, Ingo Rechenberg, now Professor of Bionics and Evolutionary Engineering, hit upon the idea of trying random changes in the parameters defining the shape, following the example of natural mutations. The evolution strategy was born."

(The second student was Hans-Paul Schwefel, now Professor of Computer Science and Chair of System Analysis).

Early evolution strategies may be perceived as evolution programs where a floating point number representation is used, with mutation being the only recombination operator. They have been applied to various optimization problems with continuously changeable parameters. Only recently they were extended for discrete problems [7], [84].

In this chapter we describe the early ESs based on a two-member population and the mutation operator, and various multimembered population ESs (Section 8.1). Section 8.2 compares ESs with GAs. Section 8.3 presents other strategies proposed recently by various researchers.

8.1 Evolution of evolution strategies

The earliest evolution strategies were based on a population consisting of one individual only. There was also only one genetic operator used in the evolution process: a mutation. However, the interesting idea (not present in GAs) was to represent an individual as a pair of float–valued vectors, i.e., $\mathbf{v} = (\mathbf{x}, \boldsymbol{\sigma})$. Here, the first vector $\mathbf{x}$ represents a point in the search space; the second vector $\boldsymbol{\sigma}$ is a vector of standard deviations: mutations are realized by replacing $\mathbf{x}$ by

$$\mathbf{x}^{t+1} = \mathbf{x}^t + N(0, \boldsymbol{\sigma}),$$

where $N(0, \boldsymbol{\sigma})$ is a vector of independent random Gaussian numbers with a mean of zero and standard deviations $\boldsymbol{\sigma}$. (This is in accordance with the biological observation that smaller changes occur more often than larger ones.) The offspring (the mutated individual) is accepted as a new member of the population (it replaces its parent) iff it has better fitness and all constraints (if any) are satisfied. For example, if f is the objective function without constraints to be maximized, an offspring $(\mathbf{x}^{t+1}, \boldsymbol{\sigma})$ replaces its parent $(\mathbf{x}^t, \boldsymbol{\sigma})$ iff $f(\mathbf{x}^{t+1}) > f(\mathbf{x}^t)$. Otherwise, the offspring is eliminated and the population remain unchanged.

Let us illustrate a single step of such an evolution strategy, considering a maximization problem we used as an example (for a simple genetic algorithm) in Chapter 2:

$$f(x_1, x_2) = 21.5 + x_1 \cdot \sin(4\pi x_1) + x_2 \cdot \sin(20\pi x_2),$$

where $-3.0 \leq x_1 \leq 12.1$ and $4.1 \leq x_2 \leq 5.8$.

As explained earlier, a population would consist of a single individual $(\mathbf{x}, \boldsymbol{\sigma})$, where $\mathbf{x} = (x_1, x_2)$ is a point within the search space ($-3.0 \leq x_1 \leq 12.1$ and $4.1 \leq x_2 \leq 5.8$) and $\boldsymbol{\sigma} = (\sigma_1, \sigma_2)$ represents two standard deviations to be used for the mutation operation. Let us assume that at some time t the single element population consists of the following individual:

$$(\mathbf{x}^t, \boldsymbol{\sigma}) = ((5.3, 4.9), (1.0, 1.0)),$$

and that the mutation results in the following change:

$$x_1^{t+1} = x_1^t + N(0, 1.0) = 5.3 + 0.4 = 5.7$$
$$x_2^{t+1} = x_2^t + N(0, 1.0) = 4.9 - 0.3 = 4.6.$$

Since

$$f(\mathbf{x}^t) = f(5.3, 4.9) = 18.383705 < 24.849532 = f(5.7, 4.6) = f(\mathbf{x}^{t+1}),$$

and both x_1^{t+1} and x_2^{t+1} stay within their ranges, the offspring will replace its parent in the single-element population.

Despite the fact that the population consists of a single individual which undergoes mutation, the evolution strategy discussed above is called a "two-membered evolution strategy". The reason is that the offspring competes with

its parent and at the competition stage there are (temporarily) two individuals in the population.

The vector of standard deviations $\boldsymbol{\sigma}$ remains unchanged during the evolution process. If all components of this vector are identical, i.e., $\boldsymbol{\sigma} = (\sigma, \ldots, \sigma)$, and the optimization problem is *regular*[1], it is possible to prove the convergence theorem [7]:

Theorem 1 (Convergence Theorem.) *For $\sigma > 0$ and a regular optimization problem with $f_{opt} > -\infty$ (minimalization) or $f_{opt} < \infty$ (maximization),*

$$p\{\lim_{t\to\infty} f(\mathbf{x}^t) = f_{opt}\} = 1$$

holds.

The Convergence Theorem states that the global optimum is found with probability one for sufficiently long search time; however, it does not provide any clues for the convergence rate (quotient of the distance covered towards the optimum and the number of elapsed generations needed to cover this distance). To optimize the convergence rate, Rechenberg proposed a "1/5 success rule":

> The ratio φ of successful mutations to all mutations should be 1/5. Increase the variance of the mutation operator, if φ is greater than 1/5; otherwise, decrease it.

The 1/5 success rule emerged as a conclusion of the process of optimizing convergence rates of two functions (the so-called corridor model and sphere model; see [7] for details). The rule was applied every k generations (k is another parameter of the method):

$$\boldsymbol{\sigma}^{t+1} = \begin{cases} c_d \cdot \boldsymbol{\sigma}^t, & \text{if } \varphi(k) < 1/5, \\ c_i \cdot \boldsymbol{\sigma}^t, & \text{if } \varphi(k) > 1/5, \\ \boldsymbol{\sigma}^t, & \text{if } p_s(k) = 1/5, \end{cases}$$

where $\varphi(k)$ is the success ratio of the mutation operator during the last k generations, and $c_i > 1$, $c_d < 1$ regulate the increase and decrease rates for the variance of the mutation. Schwefel in his experiments [162] used the following values: $c_d = 0.82$, $c_i = 1.22 = 1/0.82$.

The intuitive reason behind the 1/5 success rule is the increased efficiency of the search: if successful, the search would continue in "larger" steps; if not, the steps would be shorter. However, this search may lead to premature convergence for some classes of functions — this resulted in a refinement of the method: increased population size.

The multimembered evolution strategy differs from the previous two-membered strategy in the size of the population (*pop_size* > 1). Additional features of multimembered evolution strategies are:

[1] An optimization problem is regular if the objective function f is continuous, the domain of the function is a closed set, for all $\epsilon > 0$ the set of all internal points of the domain for which the function differs from the optimal value less than ϵ is non-empty, and for all $\mathbf{x}_0$ the set of all points for which the function has values less than or equal to $f(\mathbf{x}_0)$ (for minimalization problems; for maximization problems the relationship is opposite) is a closed set.

- all individuals in the populations have the same mating probabilities,
- possibility of introduction of a recombination operator (in the GA community called "uniform crossover"), where two (randomly selected) parents,

$$(\mathbf{x}^1, \boldsymbol{\sigma}^1) = ((x_1^1, \ldots, x_n^1), (\sigma_1^1, \ldots, \sigma_n^1)) \text{ and}$$
$$(\mathbf{x}^2, \boldsymbol{\sigma}^2) = ((x_1^2, \ldots, x_n^2), (\sigma_1^2, \ldots, \sigma_n^2)),$$

produce an offspring,

$$(\mathbf{x}, \boldsymbol{\sigma}) = ((x_1^{q_1}, \ldots, x_n^{q_n}), (\sigma_1^{q_1}, \ldots, \sigma_n^{q_n})),$$

where $q_i = 1$ or $q_i = 2$ with equal probability for all $i = 1, \ldots, n$.

The mutation operator and the adjustment of $\boldsymbol{\sigma}$ remain without changes.

There is still a similarity between two-membered and multimembered evolution strategies: both of them produce a single offspring. In the two-membered strategy, the offspring competes against its parent. In the multimembered strategy the weakest individual (among *pop_size* + 1 individuals; i.e., original *pop_size* individuals plus one offspring) is eliminated. A convenient notation, which explains also further refinement of evolution strategies, is:

$(1 + 1)$–ES, for a two membered evolution strategy, and
$(\mu + 1)$–ES, for a multimembered evolution strategy,

where $\mu = pop\_size$.

The multimembered evolution strategies evolved further [162] to mature as

$(\mu + \lambda)$–ESs and (μ, λ)–ESs;

the main idea behind these strategies was to allow control parameters (like mutation variance) to self-adapt rather than changing their values by some deterministic algorithm.

The $(\mu+\lambda)$–ES is a natural extension of a multimembered evolution strategy $(\mu + 1)$–ES, where μ individuals produce λ offspring. The new (temporary) population of $(\mu + \lambda)$ individuals is reduced by a selection process again to μ individuals. On the other hand, in the (μ, λ)–ES, the μ individuals produce λ offspring ($\lambda > \mu$) and the selection process selects a new population of μ individuals from the set of λ offspring only. By doing this, the life of each individual is limited to one generation. This allows the (μ, λ)–ES to perform better on problems with an optimum moving over time, or on problems where the objective function is noisy.

The operators used in the $(\mu+\lambda)$–ESs and (μ, λ)–ESs incorporate two-level learning: their control parameter $\boldsymbol{\sigma}$ is no longer constant, nor it is changed by some deterministic algorithm (like the 1/5 success rule), but it is incorporated in the structure of the individuals and undergoes the evolution process. To produce an offspring, the system acts in several stages:

- select two individuals,

$(\mathbf{x}^1, \boldsymbol{\sigma}^1) = ((x_1^1, \ldots, x_n^1), (\sigma_1^1, \ldots, \sigma_n^1))$ and
$(\mathbf{x}^2, \boldsymbol{\sigma}^2) = ((x_1^2, \ldots, x_n^2), (\sigma_1^2, \ldots, \sigma_n^2))$,

and apply a recombination (crossover) operator. There are two types of crossovers:

- discrete, where the new offspring is

  $$(\mathbf{x}, \boldsymbol{\sigma}) = ((x_1^{q_1}, \ldots, x_n^{q_n}), (\sigma_1^{q_1}, \ldots, \sigma_n^{q_n})),$$

  where $q_i = 1$ or $q_i = 2$ (so each component comes from the first or second preselected parent),
- intermediate, where the new offspring is

  $$(\mathbf{x}, \boldsymbol{\sigma}) = (((x_1^1+x_1^2)/2, \ldots, (x_n^1+x_n^2)/2), ((\sigma_1^1+\sigma_1^2)/2, \ldots, (\sigma_n^1+\sigma_n^2)/2)).$$

Each of these operators can be applied also in a global mode, where the new pair of parents is selected for *each* component of the offspring vector.

- apply mutation to the offspring $(\mathbf{x}, \boldsymbol{\sigma})$ obtained; the resulting new offspring is $(\mathbf{x}', \boldsymbol{\sigma}')$, where

  $$\boldsymbol{\sigma}' = \boldsymbol{\sigma} \cdot e^{N(0, \Delta\sigma)} \text{ and}$$
  $$\mathbf{x}' = \mathbf{x} + N(0, \boldsymbol{\sigma}'),$$

  where $\Delta\boldsymbol{\sigma}$ is a parameter of the method.

To improve the convergence rate of ESs, Schwefel [162] introduced an additional control parameter $\boldsymbol{\theta}$. This new control correlates mutations. For the ESs discussed so far (with the dedicated σ_i for each x_i), the preferred direction of the search can be established only along the axes of the coordinate system. Now, each individual in the population is represented as

$$(\mathbf{x}, \boldsymbol{\sigma}, \boldsymbol{\theta}).$$

The recombination operators are similar to those discussed in the previous paragraph, and the mutation creates offspring $(\mathbf{x}', \boldsymbol{\sigma}', \boldsymbol{\theta}')$ from the $(\mathbf{x}, \boldsymbol{\sigma}, \boldsymbol{\theta})$ in the following way:

$$\boldsymbol{\sigma}' = \boldsymbol{\sigma} \cdot e^{N(0, \Delta\sigma)},$$
$$\boldsymbol{\theta}' = \boldsymbol{\theta} + N(0, \Delta\boldsymbol{\theta}), \text{ and}$$
$$\mathbf{x}' = \mathbf{x} + C(0, \boldsymbol{\sigma}', \boldsymbol{\theta}'),$$

where $\Delta\boldsymbol{\theta}$ is an additional parameter of the method, and $C(0, \boldsymbol{\sigma}', \boldsymbol{\theta}')$ denotes a vector of independent random Gaussian numbers with mean zero and appropriate probability density (for details, see [162] or [7]).

Evolution strategies perform very well in numerical domains, since they were (at least, initially) dedicated to (real) function optimization problems. They are examples of evolution programs which use appropriate data structures (float

vectors extended by control strategy parameters) and "genetic" operators for the problem domain.

It is interesting to compare genetic algorithms and evolution strategies, their differences and similarities, their strengths and weaknesses. We discuss these issues in the following section.

8.2 Comparison of evolution strategies and genetic algorithms

The basic difference between evolution strategies and genetic algorithms lies in their domains. Evolution strategies were developed as methods for numerical optimization. They adopt a special hill-climbing procedure with self-adapting step sizes σ and inclination angles θ. Only recently have ESs been applied to discrete optimization problems [84]. On the other hand, genetic algorithms were formulated as (general purpose) adaptive search techniques, which allocate exponentially increasing number of trials for above-average schemata. GAs were applied in a variety of domains, and a (real) parameter optimization was just one field of their applications.

For that reason, it is unfair to compare the time and precision performance of ESs and GAs using a numerical function as the basis for the comparison. However, both ESs and GAs are examples of evolution programs and some general discussion of similarities and differences between them is quite natural.

The major similarity between ESs and GAs is that both systems maintain populations of potential solutions and make use of the selection principle of the survival of the fitter individuals. However, there are many differences between these approaches.

The first difference between ESs and classical GAs is in the way they represent the individuals. As mentioned on several occasions, ESs operate on floating point vectors, whereas classical GAs operate on binary vectors.

The second difference between GAs and ESs is hidden in the selection process itself. In a single generation of the ES, μ parents generate intermediate population which consists of λ offspring produced by means of the recombination and mutation operators (for $(\mu + \lambda)$–ES), plus (for (μ, λ)–ES) the original μ parents. Then the selection process reduces the size of this intermediate population back to μ individuals by removing the least fit individuals from the population. This population of μ individuals constitutes the next generation. In a single generation of the GA, a selection procedure selects *pop_size* individuals from the *pop_size*–sized population. The individuals are selected with repetition, i.e., a strong individual has a good chance to be selected several times to a new population. In the same time, even the weakest individual has a chance of being selected.

In ESs, the selection procedure is deterministic: it selects the best μ out of $\mu + \lambda$ ($(\mu + \lambda)$–ES) or λ ((μ, λ)–ES) individuals (no repetitions). On the other hand, in GAs, the selection procedure is random, selecting *pop_size* out of

pop_size individuals (with repetition), the chances of selection are proportional to the individual's fitness. Some GAs, in fact, use ranking selection; however, strong individuals can still be selected a few times. In other words, selection in ESs is static, extinctive, and (for (μ, λ)–ES) generational, whereas in GAs selection is dynamic, preservative, and on-the-fly (see Chapter 4).

The relative order of the procedures selection and recombination constitutes the third difference between GAs and ESs: in ESs, the selection process follows application of recombination operators, whereas in GAs these steps occur in the opposite order. In ESs, an offspring is a result of crossover of two parents and a further mutation. When the intermediate population of $\mu + \lambda$ (or λ) individuals is ready, the selection procedure reduces its size back to μ individuals. In GAs, we select an intermediate population first. Then we apply genetic operators (crossover and mutation) to some individuals (selected according to the probabilities of crossover) and some genes (selected according to the probability of mutation).

The next difference between ESs and GAs is that reproduction parameters for GAs (probability of crossover, probability of mutation) remain constant during the evolution process, whereas ESs change them ($\boldsymbol{\sigma}$ and $\boldsymbol{\theta}$) all the time: they undergo mutation and crossover together with the solution vector $\mathbf{x}$, since an individual is understood as a triplet $(\mathbf{x}, \boldsymbol{\sigma}, \boldsymbol{\theta})$. This is quite important — self-adaptation of control parameters in ESs is responsible for the fine local tuning of the system.

ESs and GAs also handle constraints in a different way. Evolution strategies assume a set of $q \geq 0$ inequalities,

$$g_1(\mathbf{x}) \geq 0, \ldots, g_q(\mathbf{x}) \geq 0,$$

as part of the optimization problem. If, during some iteration, an offspring does not satisfy all of these constraints, then the offspring is disqualified, i.e., it is not placed in a new population. If the rate of occurrence of such illegal offspring is high, the ESs adjust their control parameters, e.g., by decreasing the components of the vector $\boldsymbol{\sigma}$. The major strategy for genetic algorithms (already discussed in Chapter 7) for handling constraints is to impose penalties on individuals that violate them. That reason is that for heavily constrained problems we just cannot ignore illegal offspring (GAs do not adjust their control parameters) — otherwise the algorithm would stay in one place most of the time. At the same time, very often various decoders or repair algorithms are too costly to be considered (the effort to construct a good repair algorithm is similar to the effort to solve the problem). The penalty function technique has many disadvantages, one of which is problem dependence.

The above discussion implies that ESs and GAs are quite different with respect to many details. However, looking closer at the development of ESs and GAs during the last twenty years, one has to admit that the gap between these approaches is getting smaller and smaller.

Let us talk about some issues surrounding ESs and GAs again, this time from a historical perspective.

Quite early, there were signs that genetic algorithms display some difficulties in performing local search for the numerical applications (see Chapter 6). Many researchers experimented with different representations (Gray codes, floating point numbers) and different operators to improve the performance of the GA–based system. Today the first difference between GAs and ESs is not the issue any more: most GA applications for parameter optimization problems use floating point representation [38], adapting operators in appropriate way (see Chapter 5). It seems that the GA community borrowed the idea of vector representation from ESs.

Some papers, e.g., [56], [57], suggest that genetic algorithms are not fundamentally different from "evolutionary programming techniques", i.e., evolution strategies, which rely on random mutation and hill-climbing only (save and mutate the best). In particular, they suggest that the usage of operators more complex than mutation does not improve the algorithm's convergence. This contradicts Holland's work [89], where the relative unimportance of mutation is highlighted. A mutation is treated as a background (secondary) operator which guarantees only that every point in a search space can be reached.

The results of our experiments with our evolution programs provide an interesting observation: neither crossover nor mutation alone is satisfactory in the evolutionary process. Both operators (or rather both families of these operators) are necessary in providing a good performance of the system. The crossover operators are very important in exploring promising areas in the search space and are responsible for earlier (but not premature) convergence; in many systems (in particular those who work on richer data structures (Part III of the book) a decrease in the crossover rates deteriorates their performance. At the same time, the probability of applying mutation operators is quite high: the GENETIC-2 system (Chapter 9) uses a high mutation rate of 0.2.

A similar conclusion was reached by ES community: as a consequence, the crossover operator was introduced into ESs. Note that early ESs were based on a mutation operator only and the crossover operator was incorporated much later [162]. It seems that the score between GAs and ESs is even: the ES community borrowed the idea of crossover operators from GAs.

There are further interesting issues concerning relationships between ESs and GAs. Recently, some other crossover operators were introduced into GAs and ESs simultaneously [128], [129], [165]. Two vectors, $\mathbf{x}_1$ and $\mathbf{x}_2$ may produce two offspring, $\mathbf{y}_1$ and $\mathbf{y}_2$, which are a linear combination of their parents, i.e.,

$$\mathbf{y}_1 = a \cdot \mathbf{x}_1 + (1 - a) \cdot \mathbf{x}_2 \text{ and}$$
$$\mathbf{y}_2 = (1 - a) \cdot \mathbf{x}_1 + a \cdot \mathbf{x}_2.$$

Such a crossover was called

- in GAs: a *guaranteed average crossover* [37] (when $a = 1/2$), or an *arithmetical crossover* [128], [129], and
- in ESs: an *intermediate crossover* [165].

The self-adaptation of control parameters in ESs has its counterpart in GAs research. In general, the idea of adapting a genetic algorithm during its run was expressed some time ago; the ARGOT system [167] adapts the representation of individuals (see Section 8.3). The problem of adapting control parameters for genetic algorithms has also been recognized for some time [80], [37], [54]. It was obvious from the start that finding a good settings for GA parameters for a particular problem is not a trivial task. Several approaches were proposed. One approach [80] uses a supervisor genetic algorithm to optimize the parameters of the "proper" genetic algorithm for a class of problems. The parameters considered were population size, crossover rate, mutation rate, generation gap (percentage of the population to be replaced during each generation), and scaling window, and a selection strategy (pure or elitist). The other approach [37] involves adapting the probabilities of genetic operators: the idea is that the probability of applying an operator is altered in proportion to the observed performance of the individuals created by this operator. The intuition is that the operators currently doing "a good job" should be used more frequently. In [54] the author experimented with four strategies for allocating the probability of the mutation operator: (1) constant probability, (2) exponentially decreasing, (3) exponentially increasing, and (4) a combination of (2) and (3).

Also, if we recall non-uniform mutation (described in Chapter 6), we notice that the operator changes its action during the evolution process.

Let us compare briefly the genetic-based evolution program GENOCOP (Chapter 7) with an evolution strategy. Both systems maintain populations of potential solutions and use some selection routine to distinguish between 'good' and 'bad' individuals. Both systems use float number representation. They provide high precision (ES through adaptation of control parameters, GENOCOP through non-uniform mutation). Both systems handle constraints gracefully: GENOCOP takes advantage of the presence of linear constraints, ES works on sets of inequalities. Both systems can easily incorporate the 'constraint handling ideas' from each other. The operators are similar. One employs intermediate crossover, the other arithmetical crossover. Are they really different?

An interesting comparison between ESs and GAs from the perspective of evolution strategies is presented in [88].

8.3 Other evolution programs

As we have already discussed earlier, (classical) genetic algorithms are not appropriate tools for local fine tuning. For that reason, GAs give less precise solutions to numerical optimization problems than, for example, ESs, unless the representation of individuals in GAs is changed from binary into floating point and the (evolution) system provides specialized operators (like non–uniform mutation: Chapter 6). However, in the last decade there have been some other attempts to improve (directly or indirectly) this characteristic of GAs.

An interesting modification of GAs, called Delta Coding, was proposed recently by Whitley et al. [195]. The main idea behind this strategy is that it treats individuals in the population not as potential solutions to the problem, but rather as additional (small) values (called: *delta values*), which are added to the current potential solution. The (simplified) Delta Coding algorithm is listed in Figure 8.1.

```
procedure Delta Coding
begin
   apply GA on level x
   save the best solution (x)
   while (not termination-condition) do
   begin
      apply GA on level δ
      save the best solution (δ)
      modify the best solution (x level):
            x ← x + δ
   end
end
```

Fig. 8.1. A (simplified) Delta Coding algorithm

The Delta Coding algorithm applies genetic algorithm techniques on two levels: the level of potential solutions to the problem (level x) and (iterative phase) the level of delta changes (level δ). The best solution found on level x by a single application of a GA is saved ($\mathbf{x}$) and kept as a reference point. Then several iterations of the inner (level δ) GA are executed. A termination of a single execution of a GA on this level (i.e., when GA converges) results in the best modification vector δ, which updates the values of $\mathbf{x}$. After an update, the next iteration takes place. Each application of GA during the iteration phase reinitializes randomly the population of δ's. Of course, to evaluate individual δ, we evaluate $\mathbf{x} + \delta$.

The original Delta Coding algorithm is more complex, since it operates on bit strings. By doing this, Delta Coding preserves the theoretical foundations of genetic algorithms (since at each iteration there is a single run of GA). The termination conditions for GAs on both levels are expressed by means of the Hamming distance between the best and the worst element in the population (the algorithms terminate if the Hamming distance is not greater than one). Additionally, there is a variable *len* to denote the number of bits representing a single component of the vector δ (actually, only *len*-1 represent the absolute value of the component; the last bit is reserved for the sign of the value). If the best solution from the δ level yields a vector

$$\delta = (0, 0, \ldots, 0),$$

(i.e., no change for the best potential solution $\mathbf{x}$), the variable *len* is increased by one (to increase the precision of the solution), otherwise it is decreased by

one. Note also that Delta Coding makes mutations unnecessary, due to reinitialization of populations on level δ for each iteration.

We can simplify the original Delta Coding algorithm (Figure 8.1 provides such a simplified view) and improve its precision and time performance, if we represent both vectors $\mathbf{x}$ and $\boldsymbol{\delta}$ as sequences of floating point numbers.

Some of the ideas present in Delta Coding algorithm appeared earlier in the literature. For example, Schraudoph and Belew [161] proposed a Dynamic Parameter Encoding (DPE) strategy, where the precision of the encoded individual is dynamically adjusted. In this system, each component of the solution vector is represented by a fixed-length binary string; however, when (in some iteration) a genetic algorithm converges, the most significant bit of the solution is dropped (of course, after saving it!), the remaining bits are shifted one position left, and a new bit is introduced. This new bit in the least significant position increases precision by a finer partitioning of the search space. The process is repeated until some global termination condition is met.

The idea of reinitializing the population was discussed in [71], where Goldberg investigates the properties of systems which use small population size, but reinitialize it every time the genetic algorithm converges (and save the best individuals, of course!). The outline of such a strategy (called serial selection) is given in Figure 8.2.

```
procedure Serial Selection
begin
   generate a (small) population
   while (not termination-condition) do
   begin
      apply GA
      save the best solution (x)
      generate a new population by transferring
            the best individuals of the converged
            population and then generating the
            remaining individuals randomly
   end
end
```

Fig. 8.2. GA based on re-initialization population

Reinitializations of populations introduce diversity among individuals with a positive effect on system performance [71].

Some proposed strategies included a learning component, in a similar way as in evolution strategies. Grefenstette [80] proposed optimizing control parameters of a genetic algorithm (population size, crossover and mutation rates, etc.) by another, supervisor genetic algorithm. Shaefer [167] discussed the ARGOT strategy (Adaptive Representation Genetic Optimizer Technique), where the system learns the best internal representation for the individuals.

Another interesting evolution program, a selective evolutionary strategy (IRM, for immune recruitment mechanism), was proposed recently [16] as an optimization technique in real spaces (a similar system for optimizing functions in Hamming spaces was called GIRM). The strategy combines some previously seen ideas for directing the search in a desirable direction (e.g., tabu search). As in all evolution programming techniques, an offspring is generated from the current population. In classical genetic algorithms, such offspring replaces its parent. In evolution strategies, the offspring competes with its parent (early ESs), it competes with parents and other offspring ($(\mu + \lambda)$–ES), or it competes with other offspring ((μ, λ)–ES). In IRM systems, an offspring has to pass an additional test of affinity with its neighbors. The test checks whether it displays sufficient similarity with its close neighbors.

In general, a possible candidate k would pass the affinity test, if

$$\sum_i m(k,i) \cdot f_i > T,$$

where i indexes different species already present in the population, f_i is the concentration of the species i, $m(k,i)$ is an affinity function for species k and i, and T is the recruitment threshold.

The IRM strategy directs the search by accepting only individuals which satisfy the affinity test. Similar ideas were formulated by Glover [64], [67] in Scatter and Tabu Search. The Scatter Search techniques, like other evolution programs, maintain a population of potential solutions (vectors $\mathbf{x}^i$ are called reference points). This strategy unites preferred subsets of reference points to generate trial points (offspring) by weighted linear combinations, and selects the best members to become the source of new reference points (new population). A new twist here is the use of *multicrossover* (called weighted combination), where several (more than two) parents contribute in producing an offspring. In [67] Glover extended the idea of the Scatter Search by combining it with a Tabu Search — a technique which restricts the selection of new offspring (it requires memory where a historical set of individuals is kept) [65], [66].

It seems that the most promising direction in the search for the "optimal optimizer" lies somewhere among these ideas. Each strategy provides a new insight which might be useful in developing an evolution program for some class of problems. As stated in [67]:

> "The use of structural combinations makes it possible to combine component vectors in a way that is materially different from the results of [classical] crossover operations. Integrating such an approach with genetic algorithms may open the door to new types of search procedures. The fact that weighted and adaptive structured combinations can readily be created to exploit contexts where crossover has no evident meaning (or has difficulty insuring feasibility) suggests that such integrated search procedures may have benefits in settings where genetic algorithms presently have limited applications."

To see this clearly, let us move to the next chapter.

Part III

Evolution Programs

9. The Transportation Problem

Necessity knows no law.

Publilius Syrus, *Moral Sayings*

In Chapter 7 we compared different GA approaches for handling constraints using an example of the transportation problem. It seems that for this particular class of problems we can do better: we can use a more appropriate (natural) data structure (for a transportation problem, a matrix) and specialized "genetic" operators which operate on matrices. Such an evolution program would be much stronger method than GENOCOP: the GENOCOP optimizes any function with linear constraints, whereas the new evolution program optimizes only transportation problems (these problems have precisely $n + k - 1$ equalities, where n and k denote the number of sources and destinations, respectively; see the description of the transportation problem below). However, it would be very interesting to see what can we gain by introducing extra problem-specific knowledge into an evolution program.

Section 9.1 presents an evolution program for the linear transportation problem[1], and Section 9.2 presents one for the nonlinear transportation problem[2]

9.1 The linear transportation problem

The transportation problem (see, for example, [186]) is one of the simplest combinatorial problems involving constraints that has been studied. It seeks the determination of a minimum cost transportation plan for a single commodity from a number of sources to a number of destinations. It requires the specification of the level of supply at each source, the amount of demand at each destination, and the transportation cost from each source to each destination.

Since there is only one commodity, a destination can receive its demand from one or more sources. The objective is to find the amount to be shipped

[1]Portions reprinted, with permission, from IEEE Transactions on Systems, Man, and Cybernetics, Vol. 21, No. 2, pp. 445–452, 1991.

[2]Portions reprinted, with permission, from ORSA Journal on Computing, Vol. 3, No. 4, 1991, pp. 307–316, 1991.

from each source to each destination such that the total transportation cost is minimized.

The transportation problem is *linear* if the cost on a route is directly proportional to the amount transported; otherwise, it is *nonlinear*. While linear problems can be solved by OR methods, the nonlinear case lacks a general solving methodology.

Assume there are n sources and k destinations. The amount of supply at source i is $sour(i)$ and the demand at destination j is $dest(j)$. The unit transportation cost between source i and destination j is $cost(i,j)$. If x_{ij} is the amount transported from source i to destination j then the transportation problem is given as:

$$\text{minimize } \sum_{i=1}^{n} \sum_{j=1}^{k} f_{ij}(x_{ij})$$

subject to

$$\begin{gathered}\sum_{j=1}^{k} x_{ij} \leq sour(i), \text{ for } i = 1,2,\ldots,n,\\ \sum_{i=1}^{n} x_{ij} \geq dest(j), \text{ for } j = 1,2,\ldots,k,\\ x_{ij} \geq 0, \text{ for } i = 1,2,\ldots,n \text{ and } j = 1,2,\ldots,k.\end{gathered}$$

The first set of constraints stipulates that the sum of the shipments from a source cannot exceed its supply; the second set requires that the sum of the shipments to a destination must satisfy its demand. If $f_{ij}(x_{ij}) = cost_{ij} \cdot x_{ij}$ for all i and j, the problem is linear.

The above problem implies that the total supply $\sum_{i=1}^{k} sour(i)$ must at least equal total demand $\sum_{j=1}^{n} dest(j)$. When the total supply equals the total demand, the resulting formulation is called a *balanced transportation problem*. It differs from the above only in that all the corresponding constraints are equations; that is,

$$\begin{gathered}\sum_{j=1}^{k} x_{ij} = sour(i), \text{ for } i = 1,2,\ldots,n,\\ \sum_{i=1}^{n} x_{ij} = dest(j), \text{ for } j = 1,2,\ldots,k.\end{gathered}$$

If all $sour(i)$ and $dest(j)$ are integers, any optimal solution to a balanced linear transportation problem is an integer solution, i.e., all x_{ij} $(i = 1,2,\ldots,n,\ j = 1,2,\ldots,k)$ are integers. Moreover, the number of positive integers among the x_{ij} is at most $k + n - 1$. In this section we assume a balanced linear transportation problem. An example follows. For other information on the transportation problem and balancing, the reader is referred to any elementary text on operations research such as [186].

Example 9.1. Assume 3 sources and 4 destinations. The supply is:

$$sour(1) = 15,\ sour(2) = 25, \text{ and } sour(3) = 5.$$

The demand is:

$$dest(1) = 5,\ dest(2) = 15,\ dest(3) = 15, \text{ and } dest(4) = 10.$$

Note that the total supply and demand equal 45.

The unit transportation cost $cost(i,j)$ ($i = 1,2,3$, and $j = 1,2,3,4$) is given in the table below.

Cost

10	0	20	11
12	7	9	20
0	14	16	18

The optimal solution is shown below. The total cost is 315. The solution consists of integer values of x_{ij}.

Amount transported

	5	**15**	**15**	**10**
15	0	5	0	10
25	0	10	15	0
5	5	0	0	0

□

9.1.1 Classical genetic algorithms

By a "classical" genetic algorithm we mean, of course, one where the chromosomes (i.e., representations of solutions) are bit strings — lists of 0s and 1s. A straightforward approach in defining a bit vector for a solution in the transportation problem is to create a vector $\langle v_0, v_2, \ldots, v_p \rangle$ ($p = n \cdot k$), such that each component v_i ($i = 1,2,\ldots,p$), is a bit vector $\langle w_0^i, \ldots, w_s^i \rangle$ and represents an integer associated with row j and column m in the allocation matrix, where $j = \lfloor (i-1)/k+1 \rfloor$ and $m = (i-1)\ mod\ k+1$. The length of the vectors w (parameter s) determines the maximum integer $(2^{s+1}-1)$ that can be represented.

Let us discuss briefly the consequences of the above representation on constraint satisfaction, the evaluation function and genetic operators.

Constraint satisfaction: It is clear that every solution vector must satisfy the following:

- $v_q \geq 0$ for all $q = 1,2,\ldots,k \cdot n$,
- $\sum_{i=c\cdot k+1}^{c\cdot k+k} v_i = sour[c+1]$, for $c = 0,1,\ldots,n-1$,
- $\sum_{j=m,step\ k}^{k\cdot n} v_j = dest[m]$, for $m = 1,2\ldots,k$.

Note that the first constraint is always satisfied (we interpret a sequence of 0s and 1s as a positive integer). The other two constraints provide the totals for each source and each destination, though these formulas are not symmetrical.

Evaluation function: The natural evaluation function expresses the total cost of transporting items from sources to destinations and is given by the formula:

$$eval(\langle v_1, v_2, \ldots, v_p \rangle) = \sum_{i=1}^{p} v_i \cdot cost[j][m],$$

where $j = \lfloor (i-1)/k + 1 \rfloor$ and $m = (i-1)\ mod\ k + 1$.

Genetic operators: There is no natural definition of genetic operators for the transportation problem with the above representation. Mutation is usually defined as a change in a single bit in a solution vector. This would correspond to a change of one integer value, v_i. This, in turn, for our problems, would trigger a series of changes in different places (at least three other changes) in order to maintain the constraint equalities. Note also that we always have to remember in which column and row a change was made — despite a vector representation we think and operate in terms of rows and columns (sources and destinations). This is a reason for quite complex formulae; the first sign of this complexity is loss of symmetry in expressing the constraints.

There are some other open questions as well. Mutation is understood as a minimal change in a solution vector, but as we noted earlier, a single change in one integer would trigger at least three other changes in appropriate places. Assume that two random points (v_i and v_m, where $i < m$) are selected such that they do not belong to the same row or column. Let us assume that v_i, v_j, v_k, v_m $(i < j < k < m)$ are components of a solution vector (selected for mutation) such that v_i and v_k as well as v_j and v_m belong to a single column, and v_i and v_j as well as v_k and v_m belong to a single row. Now in trying to determine the smallest change in the solution vector we have a difficulty. Should we increase or decrease v_i? We can choose to change it by 1 (the smallest possible change) or by some random number in the range $\langle 0, 1, \ldots, v_i \rangle$. If we increase the value v_i by a constant C we have to decrease each of the values v_j and v_k by the same amount. What happens if $v_j < C$ or $v_k < C$? We could set $C = \min(v_i, v_j, v_k)$, but then most mutations would result in no change, since the probability of selecting three non-zero elements would be close to zero (less than $1/n$ for vectors of size n^2).

Thus methods involving single bit changes result in inefficient mutation operators with complex expressions for checking the corresponding row or column of the selected element.

As discussed in Chapter 7 (Section 7.1.7.3), the situation is even worse if we try to define a crossover operator.

We conclude that the above vector representation is not the most suitable for defining genetic operators in constrained problems of this type.

9.1.2 Incorporating problem-specific knowledge

Can we improve the representation of a solution while preserving the basic structure of this vector representation? We believe so, but we have to incorporate problem-specific knowledge into the representation.

First let us describe a way to create a solution which satisfies all constraints. We will call this procedure an **initialization** — it will be a fundamental component of the mutation operator when we discuss genetic operators for two-dimensional structures. It creates a matrix of at most $k + n - 1$ non-zero elements such that all constraints are satisfied. After sketching the algorithm we explain it using the matrix from Example 1.

input: arrays $dest[k]$, $sour[n]$;
output: an array $(v)_{ij}$ such that $v_{ij} \geq 0$ for all i and j,
$\sum_{j=1}^{k} v_{ij} = dest[i]$ for $i = 1, 2, \ldots, n$, and
$\sum_{i=1}^{n} v_{ij} = sour[j]$ for $j = 1, 2 \ldots, k$,
i.e., all constraints are satisfied.

procedure initialization;
begin
  set all numbers from 1 to $k \cdot n$ as unvisited
  repeat
    select an unvisited random number q
      from 1 to $k \cdot n$ and set it as visited
    set (row) $i = \lfloor (q-1)/k + 1 \rfloor$
    set (column) $j = (q-1) \; mod \; k + 1$
    set $val = \min(sour[i], \; dest[j])$
    set $v_{ij} = val$
    set $sour[i] = sour[i] - val$
    set $dest[j] = dest[j] - val$
  until all numbers are visited.
end

Example 9.2. With the matrix from Example 9.1, i.e.,

$$sour[1] = 15, \; sour[2] = 25, \text{ and } sour[3] = 5$$
$$dest[1] = 5, \; dest[2] = 15, \; dest[3] = 15, \text{ and } dest[4] = 10.$$

There are altogether $3 \cdot 4 = 12$ numbers, all of them are unvisited at the beginning. Select the first random number, say, 10. This translates into row number $i = 3$ and column number $j = 2$. The $val = \min(sour[3], \; dest[2]) = 5$, so $v_{32} = 5$. Note also that after the first iteration, $sour[3] = 0$ and $dest[2] = 10$.

We repeat these calculations with the next three random (unvisited) numbers, say 8, 5, and 3 (corresponding to row 2 and column 4, to row 2 and column 1, and to row 1 and column 3, respectively). The resulting matrix v_{ij} (so far) has the following contents:

	0	**10**	**0**	**0**
0			15	
10	5			10
0		5		

Note that the values of $sour[i]$ and $dest[j]$ are those given after 4 iterations.

If the further sequence of random numbers is 1, 11, 4, 12, 7, 6, 9, 2, the final matrix produced (with the assumed sequence of random numbers ⟨ 10, 8, 5, 3, 1, 11, 4, 12, 7, 6, 9, 2 ⟩) is:

	0	**0**	**0**	**0**
0	0	0	15	0
0	5	10	0	10
0	0	5	0	0

Obviously, after 12 iterations all (local copies of) $sour[i]$ and $dest[j] = 0$. Note also, that there are several sequences of numbers for which the procedure **initialization** would produce the optimal solution. For example, the optimal solution (given in Example 1) can be achieved for any of the following sequences: ⟨ 7, 9, 4, 2, 6, *, *, *, *, *, *, * ⟩ (where * denotes any unvisited number), as well as for many other sequences.

This technique can generate any feasible solution that contains at most $k + n - 1$ non-zero integer elements. It will not generate other solutions which, though feasible, do not share this characteristic. The initialization procedure would certainly have to be modified when we attempt to solve non-linear versions of the transportation problem.

This knowledge of the problem and its solution characteristics gives us another opportunity to represent a solution to the transportation problem as a vector. A solution vector will be a sequence of $k \cdot n$ distinct integers from the range $\langle 1, k \cdot n \rangle$, which (according to procedure **initialization**) would produce an acceptable solution. In other words, we would view a solution-vector as a permutation of numbers, and we would look for particular permutations which correspond to the optimal solution.

Let us discuss briefly the implications of this representation on constraint satisfaction, evaluation function and genetic operators.

Constraint satisfaction: Any permutation of $k \cdot n$ distinct numbers produces a unique solution which satisfies all constraints. This is guaranteed by procedure **initialization**.

Evaluation function: This is relatively easy: any permutation would correspond to a unique matrix, say, (v_{ij}). The evaluation function is $\sum_{i=1}^{k} \sum_{j=1}^{n} v_{ij} \cdot cost[i][j]$

Genetic operators: These are also straightforward:

- inversion: any solution vector $\langle x_1, x_2, \ldots, x_q \rangle$ $(q = k \cdot n)$ can be easily inverted into another solution vector $\langle x_q, x_{q-1}, \ldots, x_1 \rangle$

- mutation: any two elements of a solution vector $\langle x_1, x_2, \ldots, x_q \rangle$, say x_i and x_j can be swapped easily resulting in another solution vector.
- crossover: this is little more complex. Note that an arbitrary (blind) crossover operator would result in illegal solutions: applying such a crossover operator to sequences:

$$\langle\ 1, 2, 3, 4, 5, 6,|\ 7, 8, 9, 10, 11, 12\ \rangle \text{ and}$$
$$\langle\ 7, 3, 1, 11, 4, 12,|\ 5, 2, 10, 9, 6, 8\ \rangle$$

would result (where the crossover point is after the 6th position) in

$$\langle\ 1, 2, 3, 4, 5, 6, 5, 2, 10, 9, 6, 8\ \rangle \text{ and}$$
$$\langle\ 7, 3, 1, 11, 4, 12, 7, 8, 9, 10, 11, 12\ \rangle$$

neither of which is a legal solution.

Thus we have to use some form of heuristic crossover operator. There is some similarity between these sequences of solution vectors and those for the traveling salesman problem (see [81]). Here we use a heuristic crossover operator (of the PMX family of crossover operators, see [70] and Chapter 10), which, given two parents, creates an offspring by the following procedure:

1. make a copy of the second parent,
2. choose an arbitrary part from the first parent,
3. make minimal changes in the offspring necessary to achieve the chosen pattern.

For example, if the parents are as in the example above, and the chosen part is

$$(4, 5, 6, 7),$$

the resulting offspring is

$$\langle\ 3, 1, 11, 4, 5, 6, 7, 12, 2, 10, 9, 8\ \rangle.$$

As required, the offspring bears a structural relationship to both parents. The roles of the parents can then be reversed in constructing a second offspring.

A genetic system GENETIC-1 has been built on the above principles. The results of experiments with it are discussed in the next section.

9.1.3 A matrix as a representation structure

Perhaps the most natural representation of a solution for the transportation problem is a two-dimensional structure. After all, this is how the problem is presented and solved by hand. In other words, a matrix $V = (v_{ij})$ $(1 \leq i \leq k,\ 1 \leq j \leq n)$ may represent a solution.

Let us discuss the implications of the matrix representation on constraint satisfaction, evaluation function and genetics operators.

Constraint satisfaction: It is clear that every solution matrix $V = (v_{ij})$ should satisfy the following:

- $v_{ij} \geq 0$ for all $i = 1, \ldots, k$, and $j = 1, \ldots, n$,
- $\sum_{i=1}^{k} v_{ij} = dest[j]$ for $j = 1, \ldots, n$,
- $\sum_{j=1}^{n} v_{ij} = sour[i]$ for $i = 1, \ldots, k$.

This is similar to the set of constraints in the straightforward approach (Section 9.1.2), but the constraints are expressed in an easier and more natural way.

Evaluation function: The natural evaluation function expresses is the usual objective function:

$$\text{eval}(v_{ij}) = \sum_{i=1}^{k} \sum_{j=1}^{n} v_{ij} \cdot cost[j][m]$$

Again, the formula is much simpler than in the straightforward approach and faster than in the system GENETICS-1, where each sequence has to be converted (initialized) into a solution matrix before evaluation.

Genetic operators: We define here two genetic operators, mutation and crossover. It is difficult to define a meaningful inversion operator in this case.

- mutation:
  Assume that $\{i_1, i_2, \ldots, i_p\}$ is a subset of $\{1, 2, \ldots, k\}$, and $\{j_1, j_2, \ldots, j_q\}$ is a subset of $\{1, 2, \ldots, n\}$ such that $2 \leq p \leq k$, $2 \leq q \leq n$.
  Let us denote a parent for mutation by $(k \times n)$ matrix $V = (v_{ij})$. Then we can create a $(p \times q)$ submatrix $W = (w_{ij})$ from all elements of the matrix V in the following way: an element $v_{ij} \in V$ is in W if and only if $i \in \{i_1, i_2, \ldots, i_p\}$ and $j \in \{j_1, j_2, \ldots, j_q\}$ (if $i = i_r$ and $j = j_s$, then the element v_{ij} is placed in the r-th row and s-th column of the matrix W).
  Now we can assign new values $sour_W[i]$ and $dest_W[j]$ $(1 \leq i \leq p,\ 1 \leq j \leq q)$ for matrix W:

$$sour_W[i] = \sum_{j \in \{j_1, j_2, \ldots, j_q\}} v_{ij},\ 1 \leq i \leq p,$$
$$dest_W[j] = \sum_{i \in \{i_1, i_2, \ldots, i_p\}} v_{ij},\ 1 \leq j \leq q.$$

We can use the procedure **initialization** (Section 9.1.3) to assign new values to the matrix W such that all constraints $sour_W[i]$ and $dest_W[j]$ are satisfied. After that, we replace appropriate elements of matrix V by a new elements from the matrix W. In this way all global constraints ($sour[i]$ and $dest[j]$) are preserved.
The following example will illustrate the mutation operator.

Example 9.3. Given a problem with 4 sources and 5 destinations and the following constraints:

$$sour[1] = 8,\ sour[2] = 4,\ sour[3] = 12,\ sour[4] = 6,$$
$$dest[1] = 3,\ dest[2] = 5,\ dest[3] = 10,\ dest[14] = 7,\ dest[5] = 5.$$

Assume that the following matrix V is selected as a parent for mutation:

0	0	5	0	3
0	4	0	0	0
0	0	5	7	0
3	1	0	0	2

Select (at random) the two rows $\{2,4\}$ and three columns $\{2,3,5\}$. The corresponding submatrix W is:

4	0	0
1	0	2

Note, that $sour_W[1] = 4$, $sour_W[2] = 3$, $dest_W[1] = 5$, $dest_W[2] = 0$, $dest_W[3] = 2$. After the reinitialization of matrix W, the matrix may get the following values:

2	0	2
3	0	0

So, finally, the offspring of matrix V after mutation is:

0	0	5	0	3
0	2	0	0	2
0	0	5	7	0
3	3	0	0	0

□

- crossover:
  Assume that two matrices $V_1 = (v_{ij}^1)$ and $V_2 = (v_{ij}^2)$ are selected as parents for the crossover operation. Below we describe the skeleton of an algorithm we use to produce the pair of offspring V_3 and V_4.
  Create two temporary matrices: $DIV = (div_{ij})$ and $REM = (rem_{ij})$. These are defined as follows:

$$div_{ij} = \lfloor (v^1_{ij} + v^2_{ij})/2 \rfloor$$
$$rem_{ij} = (v^1_{ij} + v^2_{ij}) \bmod 2$$

Matrix DIV keeps rounded average values from both parents, the matrix REM keeps track of whether any rounding was necessary.

Matrix REM has some interesting properties: the number of 1s in each row and each column is even. In other words, the values of $sour_{REM}[i]$ and $dest_{REM}[j]$ (the marginal sums of rows and columns, respectively, of the matrix REM) are even integers. We use this property to transform the matrix REM into two matrices REM_1 and REM_2 such that

$$REM = REM_1 + REM_2,$$
$$sour_{REM_1}[i] = sour_{REM_2}[i] = sour_{REM}[i]/2, \text{ for } i = 1, \ldots, k,$$
$$dest_{REM_1}[j] = dest_{REM_2}[j] = dest_{REM}[j]/2, \text{ for } j = 1, \ldots, n.$$

Then we produce two offspring of V_1 and V_2:

$$V_3 = DIV + REM_1$$
$$V_4 = DIV + REM_2.$$

The following example will illustrate the case.

Example 9.4. Take the same problem as described in Example 9.1. Let us assume that the following matrices V_1 and V_2 were selected as parents for crossover:

V_1

1	0	0	7	0
0	4	0	0	0
2	1	4	0	5
0	0	6	0	0

V_2

0	0	5	0	3
0	4	0	0	0
0	0	5	7	0
3	1	0	0	2

The matrices DIV and REM are:

DIV

0	0	2	3	1
0	4	0	0	0
1	0	4	3	2
1	0	3	0	1

REM

1	0	1	1	1
0	0	0	0	0
0	1	1	1	1
1	1	0	0	0

The two matrices REM_1 and REM_2 are:

REM_1

0	0	1	0	1
0	0	0	0	0
0	1	0	1	0
1	0	0	0	0

REM_2

1	0	0	1	0
0	0	0	0	0
0	0	1	0	1
0	1	0	0	0

Finally, two offspring V_3 and V_4 are:

V_3

0	0	3	3	2
0	4	0	0	0
1	1	4	4	2
2	0	3	0	1

V_4

1	0	2	4	1
0	4	0	0	0
1	0	5	3	3
1	1	3	0	1

□

An evolution system GENETIC-2 has been built on the above principles. We have carried out experiments in first tuning and then comparing the the modified classical (vector based) GENETIC-1 and the alternative (matrix based) GENETIC-2 versions of the algorithm to solve the standard linear transportation problem.

Our purpose is not, of course, to compare the genetic methods with the standard optimization algorithm, for on every efficiency measure we choose, genetic methods will be unable to compete. This situation is different when we apply the methods to the nonlinear case. Rather, by using a range of problems of different sizes with known solutions, we aim to investigate the effects of problem representation (GENETIC-1 versus GENETIC-2).

Some randomly generated artificial problems and some published examples comprise the test problems. The artificial problems had randomly generated unit costs, supply and demand values, though the problems remained balanced. We felt that published examples would contain more typical cost structures than artificial problems. For example, a production-inventory problem has a recognizable pattern of costs when represented as a transportation problem.

In every case the problem was first solved using a standard transportation algorithm so that the optimum value was known for use as a stopping criterion and for later comparison of the techniques.

The problems were limited in size by the computers we were using (Macintosh SE/30, Macintosh II, AT&T 3B2 and Sun 3/60 machines, the latter two running under versions of the Unix operating system). The problems are referenced in Table 9.1.

Problem Name	size	Reference
prob01 to prob15	4 by 4 to 10 by 10	problems generated randomly
sas218	5 by 5	[155], p.218
taha170	3 by 4	[186], p.170
taha197	5 by 4	[186], p.197
win268	9 by 5	[196], p.268

Table 9.1. The problems used

In comparing optimization algorithms one first has to decide on the criteria to be used. One obvious criterion is the number of generations required to reach an optimum value, perhaps combined with the time or the number of operations required to complete each generation. A serious problem is that in some cases the genetic algorithms take many generations to reach an optimum. Moreover with some settings of the parameters no optimum is reached before the run has to be stopped. The number of generations needed also varies markedly with the random number starting seed used as well as the problem being solved. We felt that this measure, though natural, could not easily be used for these particular experiments.

Alternative, and more practical, criteria are based on the closeness to the optimum value reached in a fixed number of generations. We chose the percentage above the known optimum value reached in 100 generations. Observations were also made for 1000 generations but in most cases, for the problems studied here, the solution reached by then is at or very close to the optimum for both algorithms and comparisons are inconclusive.

Other workers have found (see, for example, [80]) that changing the parameters of the genetic algorithm can make a difference to its performance. We first deal with the results from experiments in tuning the parameters for the two methods. We kept the population size fixed at 40 and the number of the solutions chosen for reproduction in each generation fixed at 10 (25% of the population). This latter number is also the number of solutions removed each generation. We were then able to adjust the values of the mutation parameters, *cross*, *inv*, and *mut*, i.e., the number of parents chosen to reproduce by *crossover*, *inversion*, and *mutation*, respectively, keeping their sum at 10. The number subject to crossover, *cross*, has to be even since crossovers occur between pairs of parents. Inversion is only possible in the vector-based version, GENETIC-1. We also fixed the probability distribution parameter *sprob* that controls the geometric distribution used for choosing parents and those to be removed.

Over 1000 runs were carried out during the tuning process. Runs were made for five different random number seeds for each chosen combination of parameters. The average objective value of the five runs was converted into a percentage above the known optimum.

Figure 9.1 shows an example of the effect of varying the number of crossover pairs, *cross*, in the two programs for one particular published problem, sas218. Similar, though not identical, results were found for other published problems and for the artificial problems. Because we fixed the total number of parents, more crossovers implies fewer mutations and, for the vector model, fewer inversions. Each point in the figure is the average of five runs with different starting seeds. The percentage above the known optimum reached in 100 generations is graphed.

In general the matrix based GENETIC-2 version gave smoother curves as a function of the number of crossover pairs. Usually, the fewer the crossovers the better, with the best results occurring at zero crossover pairs. But results are

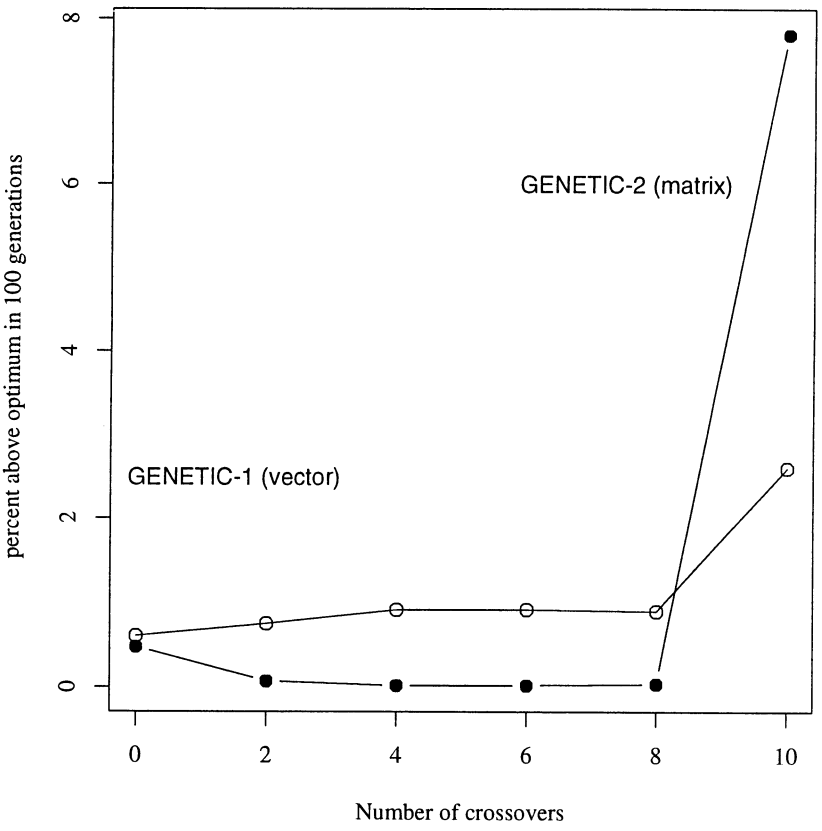

Fig. 9.1. Problem sas218 using the two algorithms

affected by the choice of problem. Problem sas218 is different from most others in that the best results are obtained with 2 to 4 crossover pairs. The GENETIC-1 results show generally increasing percentage values with crossovers, though, again, this is not universal. In both cases, as the figure demonstrates, the situation deteriorates when all pairs are used in a crossover mode, leaving none for random mutation. The GENETIC-2 model is particularly sensitive to this effect and in this case is much worse than GENETIC-1 for the runs demonstrated here.

For the class of problems studied we find that, though results differ over the problems, a small proportion of crossovers works best for both models, and, for the vector model, zero inversions are best.

Once optimum tuning parameters had been obtained, we carried out comparisons between the models running on the collection of problems. Once again

the results quoted are in every case the average of five runs with different starting seeds.

Figure 9.2 shows the results of runs on the whole set of problems for 2 crossover pairs graphed against problem size, $(n * k)$. In every case the matrix-based GENETIC-2 performs better (i.e., gets closer to the optimum in 100 generations) than the vector-based GENETIC-1. Never did the vector version outperform the matrix version on the same problem. GENETIC-1 was also much more unreliable than GENETIC-2. This effect is particularly striking with some problems, as can be seen by the outliers in the figure.

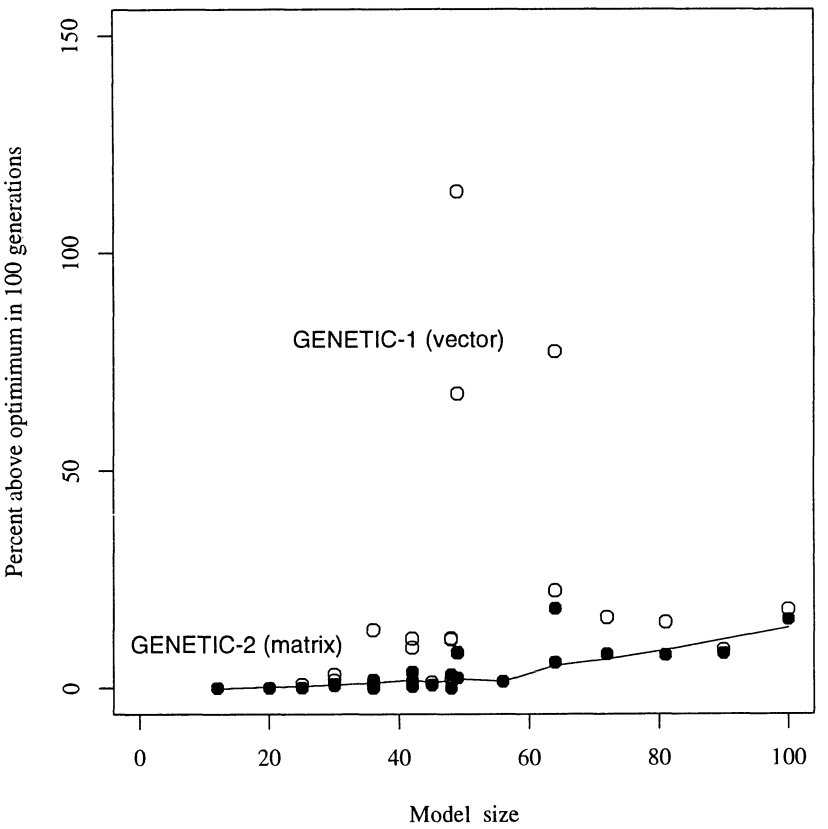

Fig. 9.2. GENETIC-1 versus GENETIC-2

9.1.4 Conclusions

We conclude that in these experiments the matrix based algorithm (GENETIC-2) performs better than the vector based version (GENETIC-1) using our cri-

terion. Note that this comparison is between a matrix based version and a *specially developed* vector based model. While the matrix based algorithm includes problem-specific knowledge in a natural manner, the vector based model also must rely on additional assumptions in order to proceed: a special **initialization** routine, based on a detailed analysis of the problem, had to be devised in order to make the vector based version work at all. For that reason the special vector based version (GENETIC-1) cannot easily be generalized but the matrix approach (GENETIC-2) is potentially very fruitful for generalization, including to the more complicated nonlinear versions of the transportation problem. Here, GENETIC-1 cannot work properly: procedure **initialization**, which serves as a basis of this system, depends heavily on knowledge of the solution form in the linear case. When the optimal solution need not be a matrix of integers and the number of non-zeros can be much larger than $k+n-1$, the GENETIC-1 system must fail. Even if we change **initialization** (for example, for each selected i-th row and j-th column, variable *val* is assigned a random number from the range $\langle 0, min(sour[i], dest[j])\rangle$), a vector representing the sequence of initialization points should be extended to record all selected random numbers as well. We conclude that all of these difficulties are due to the artificial representation of a solution as a vector.

The genetic algorithm is very slow and can in no way be compared with the special optimizing techniques based on the standard linear programming algorithm. The latter solves the problem in a number of iterations of the order of the problem size $(n * k)$, whereas each generation of the genetic method involves constructing a set of potential solutions to the problem. However it holds promise of being useful for non-linear and fixed-charge problems where the standard transportation methods cannot be used (next section).

9.2 The nonlinear transportation problem

We discuss our evolution program for the balanced nonlinear transportation problem in terms of the five components for genetic algorithms: representation, initialization, evaluation, operators, and parameters. The algorithm was named GENETIC-2 (as in the linear case).

9.2.1 Representation

As with the linear case, we have selected a two-dimensional structure for representing a solution (a chromosome) to the transportation problem: a matrix $V = (x_{ij})$ $(1 \leq i \leq k,\ 1 \leq j \leq n)$. This time each x_{ij} is a real number.

9.2.2 Initialization

The initialization procedure is identical to the one from the linear case (Section 9.1.3). As in the linear case, it creates a matrix of at most $k + n - 1$ non-zero

elements such that all constraints are satisfied. Although other initialization procedures are feasible, this method will generate a solution that is at a vertex of the simplex which describes the convex boundary of the constrained solution space.

9.2.3 Evaluation

In this case we have to minimize cost, a nonlinear function of the matrix entries. A number of functions were selected (these are described in Chapter 7) and the results of tests with them are presented in Section 9.2.6.

9.2.4 Operators

We define two genetic operators, *mutation* and *arithmetical crossover.*

- **mutation:**
  Two types of mutation operators are defined. The first, **mutation-1**, is identical to that used in the linear case and introduces as many zero entries into the matrix as possible. The second, **mutation-2**, is modified to avoid choosing zero entries by selecting values from a range. The **mutation-2** operator is identical to **mutation-1** except that in recalculating the contents of the chosen sub-matrix a modified version of the *initialization* routine is used.

  It is changed from that described in Section 9.1.3 as follows: The line

  set $val = \min(sour[i], dest[j])$

  is replaced by:

  set $val_1 = \min(sour[i], dest[j])$
  if (i is the last available row) **or**
  (j is the last available column)
  then $val = val_1$
  else set val = random (real) number from $\langle 0, val_1 \rangle$

  This change provides real numbers instead of integers and zeros but the procedure must be further modified as it currently produces a matrix which may violate the constraints.

  For example, using the matrix from Example 9.1, suppose that the sequence of selected numbers is $\langle 3, 6, 12, 8, 10, 1, 2, 4, 9, 11, 7, 5 \rangle$ and that the first real number generated for number 3 (first row, third column) is 7.3 (which is within the range $\langle 0.0, \min(sour[1], dest[3]) \rangle = \langle 0.0, 15.0 \rangle$). The second random real number for 6 (second row, second column) is 12.1, and the rest of the real numbers generated by the new initialization algorithm are: $3.3, 5.0, 1.0, 3.0, 1.9, 1.7, 0.4, 0.3, 7.4, 0.5$. The resulting matrix is:

	5.0	**15.0**	**15.0**	**10.0**
15.0	3.0	1.9	7.3	1.7
25.0	0.5	12.1	7.4	5.0
5.0	0.4	1.0	0.3	3.3

Only by adding 1.1 to the element x_{11} can we satisfy the constraints. So we need to add a final line to the **mutation-2** algorithm:

make necessary additions

This completes the modification of the it initialization procedure.

- **Crossover**

  Starting with two parents (matrices U and V) the crossover operator will produce two children X and Y, where

  $$X = c_1 \cdot U + c_2 \cdot V \text{ and } Y = c_1 \cdot V + c_2 \cdot U,$$

  (where $c_1, c_2 \geq 0$ and $c_1 + c_2 = 1$). Since the constraint set is convex this operation ensures that both children are feasible if both parents are. This is a significant simplification of the linear case where there was an additional requirement to maintain all components of the matrix as integers.

9.2.5 Parameters

In addition to the set of control parameters used for the linear case (population size, mutation and crossover rates, random number starting seed, etc.) a few more are needed. These are the crossover proportions, c_1 and c_2, and m_1, a parameter to determine the proportion of mutation-1 in the mutations applied.

9.2.6 Experiments and results

In testing the GENETIC-2 algorithm on the linear transportation problem (Section 9.1) we can compare its solution with the known optimum found using the standard algorithm. Hence we can determine how efficient the genetic algorithm is in absolute terms. Once we move to nonlinear objective functions, the optimum may not be known. Testing is reduced to comparing the results with those of other nonlinear solution methods that may themselves have converged to a local optimum.

As usual, we compare the GENETIC-2 algorithm method with the GAMS system as a typical example of an industry-standard efficient method of solution. This system, being essentially a gradient-controlled method, found some of the problems we set up difficult or impossible to solve. In these cases modifications to the objective functions could be made so that the method could at least find an approximate solution.

The objective for the transportation problem was then of the form

$$\sum_{ij} f(x_{ij})$$

where $f(x)$ is one of the six selected functions (discussed in the Chapter 7), the c_{ij} parameters are obtained from the parameter matrix and S from the attributes of the problem to be tested. S is approximated from the average non-zero arc flow determined from a number of preliminary runs to make sure the flows occurred in the interesting part of the objective function.

In some sense it is desirable to use completely randomly structured objective functions on each arc. Given that our objective is to demonstrate how the algorithm performs on a variety of problems the question reduces to asking how much variation between arcs is required for a particular function form. When the function is identical on each arc the problem may have many solutions with the same cost, reducing the information obtained when analyzing the algorithm.

In our experiments a cost-matrix was used to provide variation between arcs. The matrix provides the c_{ij}'s which act to scale the basic function shape, thus providing 'one degree' of variability. More matrices (providing more degrees of variability) were not required.

As discussed in Chapter 7, functions C, E, and F could be implemented directly in GAMS using the built-in nonlinear functions but it could not handle functions A, B and D directly. Expressions A and B cannot be formulated in GAMS while for D (the square-root function) difficulty is encountered in measuring gradients near zero. In the previous chapter we provided the description of the modifications made for the GAMS runs.

As for the linear case, for each problem, multiple GAMS runs were made under different values of the modification parameter and the best result chosen. The best values for the three parameters were found to be: P_A between 1 and 20, P_B very large (e.g., 1000), and ϵ (for function D) between 1 and 7. The final result values were always calculated after the optimization using the unmodified function, instead of the modified function.

For the main set of experiments, five 10×10 transportation matrices were used with each function. They were constructed from a set of independent uniformly distributed c_{ij} values and randomly chosen source and destination vectors with a total flow of 100 units. Each function–matrix combination was given 5 runs using different random number starting seeds for the genetic algorithm. Problems were run for 10,000 generations.

For function A, S was set to 2, while for functions B, E, and F a value of 5 was used.

The 10×10 node problems reach the limit of the student version of GAMS (where allowable problem size is restricted). From a listing of some example problems tested on the GAMS system, it appears that with the full version (where problem size is limited by available memory and internal limits) on a 640k memory AT computer, a 25×25 node problem should be possible. Note that an $N \times N$ node problem would be formulated by GAMS as having N^2 variables, 2N constraints and a nonlinear objective function. Clearly, larger

problems could be formulated on bigger systems (especially a mainframe) or with specialized solvers.

However, using much larger problems to compare the genetic system with nonlinear programming type solvers may be of limited value. Results of the 10 × 10 runs demonstrate the tendency for GAMS (and presumably, similar systems) to fall into local (non-global) optima. Ignoring the time spent evaluating the objective function and using the number of solutions tested as the measure of time it is clear that standard nonlinear programming techniques will always 'finish' faster than genetic systems. This is because they typically explore only a particular path within the current local optimum zone. They will do well only if the local optimum is a relatively good one.

A set of parameters were chosen for GENETIC-2 after experience with the linear problems and on the basis of tuning runs with the nonlinear problems. The population size was fixed at 40. The mutation rate was $p_m = 20\%$ with the proportion of mutation-1 being 50%, and the crossover rate was $p_c = 5\%$. The crossover proportions were $c_1 = .35$ and $c_2 = .65$.

It may appear that the chosen mutation rate is too high and the crossover rate too low in comparison with classical genetic algorithms. However, our operators are different from the classical ones, because (1) we select parents for mutations and crossovers, i.e., the *whole* structure (as opposed to single bits) undergoes mutation, and (2) mutation-1 creates an offspring 'pushing' the parent towards the surface of the solution space, whereas crossover and mutation-2 'push' the offspring towards the center of the solution space.

The use of high mutation rates may also suggest that the algorithm is nearly a random search. However, the random search algorithm (crossover rate 0%) performs quite poorly in comparison to the tuned algorithm used here. To demonstrate this, we tabulate below (Table 9.2) some typical results for different values of parameters p_m and p_c using functions A and F and for a particular 7 × 7 transportation problem. The values given are the average minimum cost achieved for 5 runs with different seeds in 10,000 generations.

Function	$p_c = 0\%$ $p_m = 25\%$	$p_c = 25\%$ $p_m = 0\%$	$p_c = 5\%$ $p_m = 20\%$
A	45.8	181.0	0.0
F	178.7	189.6	110.9

Table 9.2. Results for different values of p_c and p_m

The 7 × 7 transportation problem used was the same as the one given in Figure 7.3 in Chapter 7; the solutions found by the algorithm GENETIC-2 for different values of parameters p_c and p_m are given in Figure 9.3.

The GENETIC-2 system was run on SUN SPARCstation 1 computers while GAMS was run on an Olivetti 386. Although speed comparisons between the two machines are difficult it should be noted that in general GAMS finished each

p_c = 0%, p_m = 25%, function A, Cost = 45.8

20.00	0.00	0.00	1.00	2.00	2.00	2.00
0.00	20.00	1.00	2.00	2.00	2.00	1.00
0.00	0.00	19.00	0.00	2.00	1.00	3.00
0.00	0.00	0.00	20.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	20.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	20.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	20.00

p_c = 0%, p_m = 25%, function F, Cost = 178.7

15.00	6.00	0.00	6.00	0.00	0.00	0.00
0.00	14.00	0.00	14.00	0.00	0.00	0.00
0.00	0.00	20.00	0.00	0.00	0.00	5.00
5.00	0.00	0.00	3.00	6.00	0.00	6.00
0.00	0.00	0.00	0.00	20.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	20.00	0.00
0.00	0.00	0.00	0.00	0.00	5.00	15.00

p_c = 25%, p_m = 0%, function A, Cost = 181.2

18.25	0.00	0.11	1.81	3.30	3.53	0.00
0.00	18.21	3.94	3.05	0.00	1.97	0.82
1.75	1.79	13.24	1.46	1.46	1.48	3.83
0.00	0.00	1.91	14.87	1.18	0.35	1.69
0.00	0.00	0.72	0.97	18.10	0.21	0.00
0.00	0.00	0.02	0.71	1.96	17.30	0.00
0.00	0.00	0.05	0.14	0.00	0.16	19.65

p_c = 25%, p_m = 0%, function F, Cost = 189.6

20.00	0.25	0.00	0.75	0.00	6.00	0.00
0.00	5.75	0.00	22.25	0.00	0.00	0.00
0.00	0.00	20.00	0.00	0.00	0.00	5.00
0.00	14.00	0.00	0.00	6.00	0.00	0.00
0.00	0.00	0.00	0.00	20.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	14.00	6.00
0.00	0.00	0.00	0.00	0.00	5.00	15.00

p_c = 5%, p_m = 20%, function A, Cost = 0.0

19.87	0.00	0.68	1.80	1.33	1.80	1.51
0.08	20.00	1.00	1.90	1.48	1.61	1.92
0.00	0.00	18.32	1.89	1.08	1.78	1.93
0.05	0.00	0.00	17.09	1.91	0.96	0.00
0.00	0.00	0.00	0.00	19.92	0.00	0.08
0.00	0.00	0.00	0.00	0.00	18.60	1.40
0.00	0.00	0.00	0.31	0.28	0.25	19.16

p_c = 5%, p_m = 20%, function F, Cost = 110.9

14.31	6.31	6.39	0.00	0.00	0.00	0.00
0.00	13.69	0.31	14.00	0.00	0.00	0.00
0.00	0.00	13.31	6.00	0.00	0.00	5.69
5.69	0.00	0.00	3.00	6.00	0.00	5.31
0.00	0.00	0.00	0.00	20.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	19.31	0.69
0.00	0.00	0.00	0.00	0.00	5.69	14.31

Fig. 9.3. Solutions found by GENETIC-2 for different values of p_c and p_m

run well before the genetic system. An exception is case A (in which GAMS evaluates numerous arc-tangent functions) where the genetic algorithm took no more than 15 minutes to complete while GAMS averaged about twice that. For cases A,B, and D, where the extra GAMS modification parameter meant that multiple runs had to be performed to find its best solution, the genetic system overall was much faster.

A typical comparison of the optima between GENETIC-2 (averaged over 5 seeds) and GAMS is shown in the Table 9.3 for a single 10×10 problem; its description is given in Figure 9.4.

Function	GAMS	GENETIC-2	% difference
A	281.0	202.0	−28.1%
B	180.8	163.0	−9.8%
C	4402.0	4556.2	+3.5%
D	408.4	391.1	−4.2%
E	145.1	79.2	−45.4%
F	1200.8	201.9	−83.2%

Table 9.3. Comparison between GAMS and GENETIC-2

Number of Sources: 10
Number of Destinations: 10

Source Flows:	8	8	2	26	12	1	6	18	18	1
Destination Flows:	19	2	33	5	11	11	2	14	2	1

Arc Parameter Matrix (Source by Destination):

15	3	23	1	19	14	6	16	41	33
13	17	30	36	20	17	26	19	3	33
37	17	30	5	48	27	8	25	36	21
13	13	31	7	35	11	20	41	34	3
31	24	8	30	28	33	2	8	1	8
32	36	12	9	18	1	44	49	11	11
49	6	17	0	42	45	22	9	10	47
2	21	18	40	47	27	27	40	19	42
13	16	25	21	19	0	32	20	32	35
23	42	2	0	9	30	5	29	31	29

Fig. 9.4. Example problem description

Figure 9.5 displays the results for all five considered problems. For the class of 'practical' problems, A and B, GENETIC-2 is, on average, better than GAMS by 24.5% in case A and by 11.5% in case B. For the 'reasonable' functions the results were different. In case C (the square function), the genetic system performed worse by 7.5% while in case D (the square-root function), the genetic system was better by just 2.0%, on average. For the 'other' functions, E and F, the genetic system dominates; it resulted in improvements of 33.0% and 54.5% over GAMS, averaging over the five problems.

9.2.7 Conclusions

Our objective was to investigate the behavior of a type of genetic algorithm on problems with multiple constraints and nonlinear objective functions. The transportation problem was chosen for study as it provided a relatively simple convex feasible set. This was to make it is easier to maintain feasibility in the solutions. We were then able to examine the influence of the objective function alone on the algorithm's behavior.

The results demonstrate the efficiency of the genetic method in finding the global optimum in difficult problems, though GAMS did well on the smooth monotonic, 'reasonable', functions. The gradient controlled techniques are most suited to these situations. For function C, GAMS found better solutions much faster than GENETIC-2.

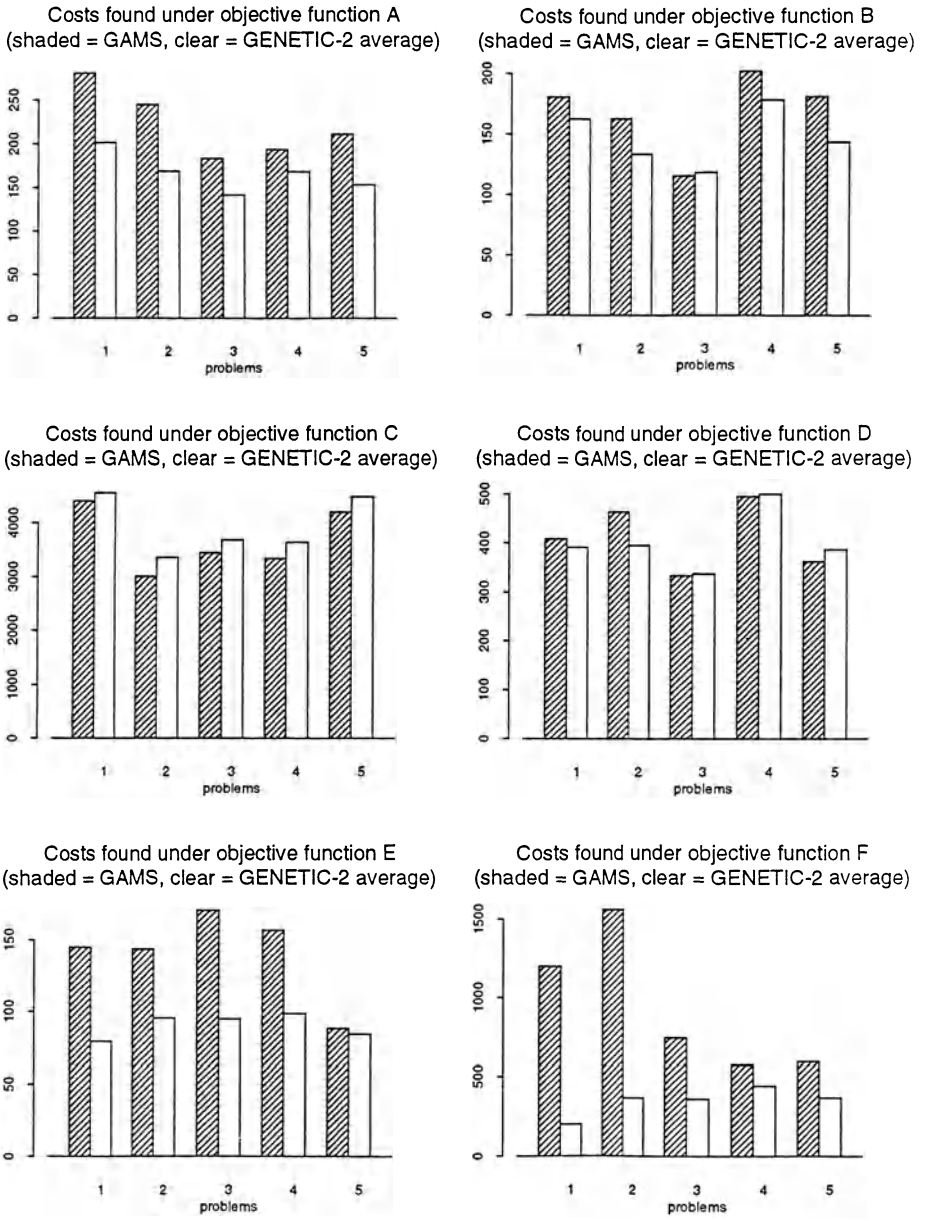

Fig. 9.5. Results

For the 'practical' problems, the gradient techniques have difficulty 'seeing around the corner' to new zones of better costs. The genetic type of algorithm,

taking a more global approach, is able to move to new zones readily, hence generating much better solutions.

The 'other' problems, although they are both smooth, have significant structural features that were admittedly designed to cause real difficulties for the gradient methods. GENETIC-2 excelled over GAMS here even more than in the 'practical' cases.

It is also interesting to compare GENOCOP (Chapter 7) with GENETIC-2 (see Table 9.4). In general, their results are very similar. However, note again that the matrix approach was tailored to the specific (transportation) problem, whereas GENOCOP is problem independent and works without any hard-coded domain knowledge. In other words, while one might expect the GENOCOP to perform similarly well for other constrained problems, GENETIC-2 cannot be used at all.

Function	GENETIC-2	GENOCOP	% difference
A	00.00	24.15	
B	203.81	205.60	0.87%
C	2564.23	2571.04	0.26%
D	480.16	480.16	0.00%
E	204.73	204.82	0.04%
F	110.94	119.61	7.24%

Table 9.4. GENETIC-2 versus GENOCOP: the results for the 7×7 problem, with transportation cost functions A–F (previous chapter) and cost matrix given in Figure 7.3

While comparing all three systems (GAMS, GENOCOP, GENETIC-2), it is important to underline that two of them, GAMS and GENOCOP, are problem independent: they are capable of optimizing any function subject to any set of linear constraints. The third system, GENETIC-2, was designed for transportation problems only: the particular constraints are incorporated into matrix data structures and special "genetic" operators.

GENETIC-2 was specifically tailored to transportation problems but an important characteristic is that it handles any type of cost function (which need not even be continuous). It is also possible to modify it to handle many similar operations research problems including allocation and some scheduling problems. This seems to be a promising research direction which may result in a generic technique for solving matrix based constrained optimization problems.

10. The Traveling Salesman Problem

There is nothing worse for mortals
than a wandering life.

Homer, *Odyssey*

In the next chapter, we present several examples of evolution programs tailored to specific applications (graph drawing, minimum spanning tree, scheduling). The traveling salesman problem (TSP) is just one of such applications; however, we treat it as a special problem — the mother of all problems — and discuss it in a separate chapter. What are the reasons?

Well, there are many. First of all, the TSP is conceptually very simple: the traveling salesman must visit every city in his territory exactly once and then return to the starting point. Given the cost of travel between all cities, how should he plan his itinerary for minimum total cost of the entire tour? The search space for the TSP is a set of permutations of n cities. Any single permutation of n cities yields a solution (which is a complete tour of n cities). The optimal solution is a permutation which yields the minimum cost of the tour. The size of the search space is $n!$.

The TSP is a relatively old problem: it was documented as early as 1759 by Euler (though not by that name), whose interest was in solving the knights' tour problem. A correct solution would have a knight visit each of the 64 squares of a chessboard exactly once in its tour.

The term 'traveling salesman' was first used in a 1932 a German book *The traveling salesman, how and what he should do to get commissions and be successful in his business*, written by a veteran traveling salesman (see [115]). Though not the main topic of the book, the TSP and scheduling are discussed in the last chapter.

The TSP was introduced by the RAND Corporation in 1948. The Corporation's reputation helped to make the TSP a well known and popular problem. The TSP also became popular at that time due to the new subject of linear programming and attempts to solve combinatorial problems.

The Traveling Salesman Problem was proved to be NP-hard [61]. It arises in numerous applications and the number of cities might be quite significant — as stated in [103]:

> "Circuit board drilling applications with up to 17,000 cities are mentioned in [120], X-ray crystallography instances with up to 14,000 cities are mentioned in [19], and instances arising in VLSI fabrication have been reported with as many as 1.2 million cities [107]. Moreover, 5 hours on a multi-million dollar computer for an optimal solution may not be cost-effective if one can get within a few percent in seconds on a PC. Thus there remains a need for heuristics."

During the last decades, several algorithms emerged to approximate the optimal solution: nearest neighbor, greedy algorithm, nearest insertion, farthest insertion, double minimum spanning tree, strip, spacefilling curve, algorithms by Karp, Litke, Christofides, etc. [103] (some of these algorithms assume that the cities correspond to points in the plane under some standard metric). Another group of algorithms (2-opt, 3-opt, Lin-Kernighan) aims at a local optimization: an improvement of a tour by local perturbations. The TSP also became a target for the GA community: several genetic–based algorithms were reported [59], [70], [79], [81], [102], [117], [142], [145], [166], [175], [179], [188], [193]. These algorithms aim at producing near-optimal solutions by maintaining a population of potential solutions which undergoes some unary and binary transformations ('mutations' and 'crossovers') under a selection scheme biased towards fit individuals. It is interesting to compare these approaches, paying particular attention to the representation and genetic operators used — this is what we intend to do in this chapter. In other words, we shall trace the evolution of evolution programs for TSP.

To underline some important characteristics of the TSP, let us consider the CNF-satisfiability problem first. A logical expression in conjunctive normal form (CNF) is a sequence of clauses separated by the Boolean operator $\wedge$; a clause is a sequence of literals separated by the Boolean operator $\vee$; a literal is a logical variable or its negation; a logical variable is a variable that may be assigned values TRUE or FALSE (1 or 0).

For example, the following logical expression is in CNF:

$$(a \vee \bar{b} \vee c) \wedge (b \vee c \vee d \vee \bar{e}) \wedge (\bar{a} \vee c) \wedge (a \vee \bar{c} \vee \bar{e}),$$

where a, b, c, d, and e are logical variables; $\bar{a}$ denotes the negation of variable a ($\bar{a}$ has the value TRUE if and only if a has the value FALSE).

The problem is to determine whether there exists a truth assignment for the variables in the expression, so that the whole expression evaluates to TRUE. For example, the above CNF logical expression has several truth assignments, for which the whole expression evaluates to TRUE, e.g., any assignment with a = TRUE and c = TRUE.

If we try to apply a genetic algorithm to the CNF-satisfiability problem, we notice that it is hard to imagine a problem with better suited representation: a binary vector of fixed length (the length of the vector corresponds to the number of variables) should do the job. Moreover, there are no dependencies between bits: any change would result in a legal (meaningful) vector. Thus we can apply mutations and crossovers without any need for decoders or repair

algorithms. However, the choice of the evaluation function is the hardest task. Note that all logical expressions evaluate to TRUE or FALSE, and if a specific truth assignment evaluates the whole expression to TRUE, then the solution to the problem is found. The point is that during the search for a solution, all chromosomes (vectors) in a population would evaluate to FALSE (unless a solution is found), so it is impossible to distinguish between 'good' and 'bad' chromosomes. In short, the CNF-satisfiability problem has natural representation and operators, without any natural evaluation function. For a further discussion on the problems related to a selection of an appropriate evaluation function, the reader is referred to [46].

On the other hand, the TSP has an extremely easy (and natural) evaluation function: for any potential solution (a permutation of cities), we can refer to the table with distances between all cities and (after $n-1$ addition operations) we get the total length of the tour. Thus, in a population of tours, we can easily compare any two of them. However, the choice of the representation of a tour and the choice of operators to be used are far from clear.

An additional reason for treating the TSP in a separate chapter is that similar techniques were used for variety of other sequencing problems, like scheduling and partitioning. Some of these problems are discussed in the next chapter.

There is an agreement in the GA community that the binary representation of tours is not well suited for the TSP. It is not hard to see why: after all, we are interested in the best permutation of cities, i.e.,

$$(i_1, i_2, \ldots, i_n),$$

where $(i_1, i_2, \ldots, i_n)$ is a permutation of $\{1, 2, \ldots, n\}$. The binary code of these cities will not provide any advantage. Just the opposite is true: the binary representation would require special repair algorithms, since a change of a single bit may result in a illegal tour. As observed in [193]:

> "Unfortunately, there is no practical way to encode a TSP as a binary string that does not have ordering dependencies or to which operators can be applied in a meaningful fashion. Simply crossing strings of cities produces duplicates and omissions. Thus, to solve this problem some variation on standard genetic crossover must be used. The ideal recombination operator should recombine critical information from the parent structures in a non-destructive, meaningful manner."

It is interesting to note that a recent paper by Lidd [117] describes a GA approach for the TSP with a binary representation and classical operators (crossover and mutation). The illegal tours are evaluated on the basis of complete (not necessarily legal) tours created by a greedy algorithm. The reported results are of surprisingly high quality, however, the largest considered test case consisted of 100 cities only.

During the last few years there have been three vector representations considered in connection with the TSP: *adjacency, ordinal*, and *path* representations. Each of these representations has its own "genetic" operators — we shall discuss them in turn. Since it is relatively easy to come up with some sort of mutation operator which would introduce a small change into a tour, we shall concentrate on crossover operators. In all three representations, a tour is described as a list of cities. In the following discussions we use a common example of 9 cities numbered from 1 to 9.

Adjacency Representation:

The adjacency representation represents a tour as a list of n cities. The city j is listed in the position i if and only if the tour leads from city i to city j. For example, the vector

(2 4 8 3 9 7 1 5 6)

represents the following tour:

1 – 2 – 4 – 3 – 8 – 5 – 9 – 6 – 7

Each tour has only one adjacency list representation; however, some adjacency lists can represent illegal tours, e.g.,

(2 4 8 1 9 3 5 7 6),

which leads to

1 – 2 – 4 – 1,

i.e., the (partial) tour with a (premature) cycle.

The adjacency representation does not support the classical crossover operator. A repair algorithm might be necessary. Three crossover operators were defined and investigated for the adjacency representation: *alternating edges, subtour chunks*, and *heuristic* crossovers [79].

- alternating-edges crossover builds an offspring by choosing (at random) an edge from the first parent, then selects an appropriate edge from the second parent, etc. — the operator extends the tour by choosing edges from alternating parents. If the new edge (from one of the parents) introduces a cycle into the current (still partial) tour, the operator selects instead a (random) edge from the remaining edges which does not introduce cycles. For example, the first offspring from the two parents

  p_1 = (2 3 8 7 9 1 4 5 6) and
  p_2 = (7 5 1 6 9 2 8 4 3)

  might be

  o_1 = (2 5 8 7 9 1 6 4 3),

where the process started from the edge (1,2) from the parent p_1, and the only random edge introduced during the process of alternating edges was (7,6) instead of (7,8), which would have introduced a premature cycle.

- subtour-chunks crossover constructs an offspring by choosing a (random length) subtour from one of the parents, then choosing a (random length) subtour from another parent, etc. — the operator extends the tour by choosing edges from alternating parents. Again, if some edge (from one of the parents) introduces a cycle into the current (still partial) tour, the operator selects instead a (random) edge from the remaining edges which does not introduce cycles.

- heuristic crossover builds an offspring by choosing a random city as the starting point for the offspring's tour. Then it compares the two edges (from both parents) leaving this city and selects the better (shorter) edge. The city on the other end of the selected edge serves as a starting point in selecting the shorter of the two edges leaving this city, etc. If, at some stage, a new edge would introduce a cycle into the partial tour, then the tour is extended by a random edge from the remaining edges which does not introduce cycles.

  In [102], the authors modified the above heuristic crossover by changing two rules: (1) if the shorter edge (from a parent) introduces a cycle in the offspring tour, check the other (longer) edge. If the longer edge does not introduce a cycle, accept it; otherwise (2) select the shortest edge from a pool of q randomly selected edges (q is a parameter of the method).

  The effect of this operator is to glue together short subpaths of the parent tours. However, it may leave undesirable crossings of edges — it is why the heuristic crossover is not appropriate for fine local tuning of the tours. Suh and Gucht [179] introduced an additional heuristic operator (based on 2-opt algorithm [119]) appropriate for local tuning. The operator randomly selects two edges, $(i\ j)$ and $(k\ m)$, and checks whether

$$dist(i,j) + dist(k,m) > dist(i,m) + dist(k,j),$$

  where $dist(a,b)$ is a given distance between cities a and b. If this is the case, the edges $(i\ j)$ and $(k\ m)$ in the tour are replaced by edges $(i\ m)$ and $(k\ j)$.

An advantage of adjacency representation is that it allows schemata analysis similar to one discussed in Chapter 4, where binary strings were considered. Schemata correspond to natural building blocks, i.e., edges; for example, the schema

$$(* * * 3 * 7 * * *)$$

denotes the set of all tours with edges (4 3) and (6 7). However, the main disadvantage of this representation is relatively poor results for all operators. The

alternating-edges crossover often disrupts good tours due to its own operation by alternating edges from two parents. The subtour-chunk crossover performs better than alternating-edges crossover, since the disruption rate is lower. However, the performance is still quite low. The heuristic crossover, of course, is the best operator here. The reason is that the first two crossovers are blind, i.e., they do not take into account the actual lengths of the edges. On the other hand, heuristic crossover selects the better edgne out of two possible edges — this is the reason it performs much better than the other two. However, the performance of the heuristic crossover is not outstanding: in three experiments reported [79] on 50, 100, and 200 cities, the system found tours within 25%, 16%, and 27% of the optimum, in approximately 15000, 20000, and 25000 generations, respectively.

Ordinal Representation:

The ordinal representation represents a tour as a list of n cities; the i-th element of the list is a number in the range from 1 to $n-i+1$. The idea behind the ordinal representation is as follows. There is some ordered list of cities C, which serves as a reference point for lists in ordinal representations. Assume, for example, that such an ordered list (reference point) is simply

$$C = (1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9).$$

A tour

$$1 - 2 - 4 - 3 - 8 - 5 - 9 - 6 - 7$$

is represented as a list l of references,

$$l = (1\ 1\ 2\ 1\ 4\ 1\ 3\ 1\ 1),$$

and should be interpreted as follows:

- the first number on the list l is 1, so take the first city from the list C as the first city of the tour (city number 1), and remove it from C. The partial tour is

  1

- the next number on the list l is also 1, so take the first city from the current list C as the next city of the tour (city number 2), and remove it from C. The partial tour is

  $1 - 2$

- the next number on the list l is 2, so take the second city from the current list C as the next city of the tour (city number 4), and remove it from C. The partial tour is

  $1 - 2 - 4$

- the next number on the list l is 1, so take the first city from the current list C as the next city of the tour (city number 3), and remove it from C. The partial tour is

  1 – 2 – 4 –3

- the next number on the list l is 4, so take the fourth city from the current list C as the next city of the tour (city number 8), and remove it from C. The partial tour is

  1 – 2 – 4 – 3 – 8

- the next number on the list l is again 1, so take the first city from the current list C as the next city of the tour (city number 5), and remove it from C. The partial tour is

  1 – 2 – 4 – 3 – 8 – 5

- the next number on the list l is 3, so take the third city from the current list C as the next city of the tour (city number 9), and remove it from C. The partial tour is

  1 – 2 – 4 – 3 – 8 – 5 – 9

- the next number on the list l is 1, so take the first city from the current list C as the next city of the tour (city number 6), and remove it from C. The partial tour is

  1 – 2 – 4 – 3 – 8 – 5 – 9 – 6

- the last number on the list l is 1, so take the first city from the current list C as the next city of the tour (city number 7, the last available city), and remove it from C. The final tour is

  1 – 2 – 4 – 3 – 8 – 5 – 9 – 6 – 7

The main advantage of the ordinal representation is that the classical crossover works! Any two tours in the ordinal representation, cut after some position and crossed together, would produce two offspring, each of them being a legal tour. For example, the two parents

$p_1 = (1\ 1\ 2\ 1 \mid 4\ 1\ 3\ 1\ 1)$ and
$p_2 = (5\ 1\ 5\ 5 \mid 5\ 3\ 3\ 2\ 1)$,

which correspond to the tours

1 – 2 – 4 – 3 – 8 – 5 – 9 – 6 – 7 and
5 – 1 – 7 – 8 – 9 – 4 – 6 – 3 – 2,

with the crossover point marked by '|', would produce the following offspring:

$o_1 = (1\ 1\ 2\ 1\ 5\ 3\ 3\ 2\ 1)$ and
$o_2 = (5\ 1\ 5\ 5\ 4\ 1\ 3\ 1\ 1)$;

these offspring correspond to

1 – 2 – 4 – 3 – 9 – 7 – 8 – 6 – 5 and
5 – 1 – 7 – 8 – 6 – 2 – 9 – 3 – 4.

It is easy to see that partial tours to the left of the crossover point do not change, whereas partial tours to the right of the crossover point are disrupted in a quite random way. Poor experimental results indicate [79] that this representation together with classical crossover is not appropriate for the TSP.

Path Representation:

The path representation is perhaps the most natural representation of a tour. For example, a tour

5 – 1 – 7 – 8 – 9 – 4 – 6 – 2 – 3

is represented simply as

(5 1 7 8 9 4 6 2 3).

Until recently, three crossovers were defined for the path representation: *partially -mapped* (PMX), *order* (OX), and *cycle* (CX) crossovers. We will now discuss them in turn.

- PMX — proposed by Goldberg and Lingle [70] — builds an offspring by choosing a subsequence of a tour from one parent and preserving the order and position of as many cities as possible from the other parent. A subsequence of a tour is selected by choosing two random cut points, which serve as boundaries for swapping operations. For example, the two parents (with two cut points marked by '|')

  $p_1 = (1\ 2\ 3 \mid 4\ 5\ 6\ 7 \mid 8\ 9)$ and
  $p_2 = (4\ 5\ 2 \mid 1\ 8\ 7\ 6 \mid 9\ 3)$

  would produce offspring in the following way. First, the segments between cut points are swapped (the symbol 'x' can be interpreted as 'at present unknown'):

  $o_1 = (x\ x\ x \mid 1\ 8\ 7\ 6 \mid x\ x)$ and
  $o_2 = (x\ x\ x \mid 4\ 5\ 6\ 7 \mid x\ x)$.

  This swap defines also a series of mappings:

  $1 \leftrightarrow 4$, $8 \leftrightarrow 5$, $7 \leftrightarrow 6$, and $6 \leftrightarrow 7$.

  Then we can fill further cities (from the original parents), for which there is no conflict:

o_1 = (x 2 3 | 1 8 7 6 | x 9) and
o_2 = (x x 2 | 4 5 6 7 | 9 3).

Finally, the first x in the offspring o_1 (which should be 1, but there was a conflict) is replaced by 4, because of the mapping $1 \leftrightarrow 4$. Similarly, the second x in the offspring o_1 is replaced by 5, and the x and x in the offspring o_2 are 1 and 8. The offspring are

o_1 = (4 2 3 | 1 8 7 6 | 5 9) and
o_2 = (1 8 2 | 4 5 6 7 | 9 3).

The PMX crossover exploits important similarities in the value and ordering simultaneously when used with an appropriate reproductive plan [70].

- OX — proposed by Davis [31] — builds offspring by choosing a subsequence of a tour from one parent and preserving the relative order of cities from the other parent. For example, two parents (with two cut points marked by '|')

p_1 = (1 2 3 | 4 5 6 7 | 8 9) and
p_2 = (4 5 2 | 1 8 7 6 | 9 3)

would produce the offspring in the following way. First, the segments between cut points are copied into offspring:

o_1 = (x x x | 4 5 6 7 | x x) and
o_2 = (x x x | 1 8 7 6 | x x).

Next, starting from the second cut point of one parent, the cities from the other parent are copied in the same order, omitting symbols already present. Reaching the end of the string, we continue from the first place of the string. The sequence of the cities in the second parent (from the second cut point) is

9 – 3 – 4 – 5 – 2 – 1 – 8 – 7 – 6;

after removal of cities 4, 5, 6, and 7, which are already in the first offspring, we get

9 – 3 – 2 – 1 – 8.

This sequence is placed in the first offspring (starting from the second cut point):

o_1 = (2 1 8 | 4 5 6 7 | 9 3).

Similarly we get the other offspring:

o_2 = (3 4 5 | 1 8 7 6 | 9 2).

The OX crossover exploits a property of the path representation, that the order of cities (not their positions) are important, i.e., the two tours

$$9 - 3 - 4 - 5 - 2 - 1 - 8 - 7 - 6 \text{ and}$$
$$4 - 5 - 2 - 1 - 8 - 7 - 6 - 9 - 3$$

are in fact identical.

- CX — proposed by Oliver [145] — builds offspring in such a way that each city (and its position) comes from one of the parents. We explain the mechanism of the cycle crossover using the following example. Two parents

$$p_1 = (1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9) \text{ and}$$
$$p_2 = (4\ 1\ 2\ 8\ 7\ 6\ 9\ 3\ 5)$$

would produce the first offspring by taking the first city from the first parent:

$$o_1 = (1\ x\ x\ x\ x\ x\ x\ x\ x).$$

Since every city in the offspring should be taken from one of its parents (from the same position), we do not have any choice now: the next city to be considered must be city 4, as the city from the parent p_2 just "below" the selected city 1. In p_1 this city is at position '4', thus

$$o_1 = (1\ x\ x\ 4\ x\ x\ x\ x\ x).$$

This, in turn, implies city 8, as the city from the parent p_2 just "below" the selected city 4. Thus

$$o_1 = (1\ x\ x\ 4\ x\ x\ x\ 8\ x);$$

Following this rule, the next cities to be included in the first offspring are 3 and 2. Note, however, that the selection of city 2 requires selection of city 1, which is already on the list — thus we have completed a cycle

$$o_1 = (1\ 2\ 3\ 4\ x\ x\ x\ 8\ x).$$

The remaining cities are filled from the other parent:

$$o_1 = (1\ 2\ 3\ 4\ 7\ 6\ 9\ 8\ 5).$$

Similarly,

$$o_2 = (4\ 1\ 2\ 8\ 5\ 6\ 7\ 3\ 9).$$

The CX preserves the absolute position of the elements in the parent sequence.

It is possible to define other operators for the path representation. For example, Syswerda [183] defined two modified versions of the order crossover operator. (However, this work was in connection with the scheduling problem and we will discuss this problem in the next chapter). The first modification (called order-based crossover) selects (randomly) several positions in a vector, and the order of cities in the selected positions in one parent is imposed on the corresponding cities in the other parent. For example, consider two parents

$p_1 = (1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9)$ and
$p_2 = (4\ 1\ 2\ 8\ 7\ 6\ 9\ 3\ 5)$.

Assume that the selected positions are 3rd, 4th, 6th, and 9th; the ordering of the cities in these positions from parent p_2 will be imposed on parent p_1. The cities at these positions (in the given order) in p_2 are 2, 8, 6, and 5. In parent p_1 these cities are present at positions 2, 5, 6, and 8. In the offspring the elements on these positions are reordered to match the order of the same elements from p_2 (the order is 2 – 8 – 6 – 5). The first offspring is a copy of p_1 on all positions except positions 2, 5, 6, and 8:

$o_1 = (1\ x\ 3\ 4\ x\ x\ 7\ x\ 9)$.

All other elements are filled in the order given in parent p_2, i.e., 2, 8, 6, 5, so finally,

$o_1 = (1\ 2\ 3\ 4\ 8\ 6\ 7\ 5\ 9)$.

Similary, we can construct the second offspring:

$o_2 = (3\ 1\ 2\ 8\ 7\ 4\ 6\ 9\ 5)$.

The second modification (called position-based crossover) is more similar to the original order crossover. The only difference is that in position-based crossover, instead of selecting one subsequence of cities to be copied, several cities are (randomly) selected for that purpose.

It is interesting to note that these two operators (order-based crossover and position based crossover) are, in some sense, equivalent to each other. An order-based crossover with some number of positions selected as crossover points, and a position-based crossover with compliment positions as its crossover points will always produce the same result. This means that if the average number of crossover points is $m/2$ (m is the total number of cities), these two operators should give the same performance. However, if the average number of crossover points is, say, $m/10$, then the two operators display different characteristics. For more information on these operators and some theoretical and empirical results comparing some of them, the reader is referred to [72], [59], [145], [175], and [182].

In surveying different reordering operators which have emerged during the last few years, we should mention the inversion operator as well. Simple inversion [89] selects two points along the length of the chromosome, which is cut at these points, and the substring between these points is reversed. For example, a chromosome:

(1 2 | 3 4 5 6 | 7 8 9)

with two cut points marked by '|', is changed into

(1 2 | 6 5 4 3 | 7 8 9).

Such simple inversion guarantees that the resulting offspring is a legal tour; some theoretical investigations [89] indicate that the operator should be useful in finding good string orderings. It is reported [193] that in a 50-city TSP, a system with inversion outperformed a system with a "cross and correct" operator. However, an increase in the number of cut points decreases the performance of the system. Also, inversion (like a mutation) is a unary operator, which can only supplement recombination operators — the operator is unable to recombine information by itself. Several versions of the inversion operator have been investigated [72]. Holland [89] provides a modification of a schema theorem to include its effect.

At this point we should also mention recent attempts to solve the TSP using evolution strategies [84], [165]. One of the attempts [84] experimented with four different mutations operators (mutation is still the basic operator in evolution strategies — see Chapter 8):

- *inversion* — as described above;
- *insertion* — selects a city and inserts it in a random place;
- *displacement* — selects a subtour and inserts it in a random place;
- *reciprocal exchange* — swaps two cities.

Also, a version of the heuristic crossover operator was used. In this modification, several parents contribute in producing offspring. After selecting the first city of the offspring tour (randomly), all left and right neighbors of that city (from all parents) are examined. The city which yields the shortest distance is selected. The process continues until the tour is completed.

Another application of evolution strategy [165] generates a (float) vector on n numbers (n corresponds to the number of cities). The evolution strategy is applied as for any continuous problem. The trick is in coding. Components of the vector are sorted and their order determines the tour. For example, the vector

$$\mathbf{v} = (2.34, -1.09, 1.91, 0.87, -0.12, 0.99, 2.13, 1.23, 0.55)$$

corresponds to the tour

2 - 5 — 9 - 4 - 6 - 8 - 3 - 7 - 1,

since the smallest number, -1.09 is the second component of the vector $\mathbf{v}$, the second smallest number, -0.12 is the fifth component of the vector $\mathbf{v}$, etc.

Most of the operators discussed so far take into account cities (i.e., their positions and order) as opposed to edges — links between cities. What might be important is not the particular position of a city in a tour, but rather the linkage of this city with other cities. As observed by Homaifar and Guan [94]:

> "Considering the problem carefully, we can argue that the basic building blocks for TSP are edges as opposed to the position representation of cities. A city or shorth path in a given position without adjacent or surrounding information has little meaning for constructing a good tour. However, it is hard to argue that injecting city a in position 2 is better than injecting it in position 5. Although this is the extreme case, the underlying assumption is that a good operator should extract edge information from parents as much as possible. This assumption can be partially explained from the experimental results in Oliver's paper [145] that OX does 11% better than PMX, and 15% better than the cycle crossover."

Grefenstette [81] developed a class of heuristic operators that emphasizes edges. They work along the following lines:

1. randomly select a city to be the current city c of the offspring,
2. select four edges (two from each parent) incident to the current city c,
3. define a probability distribution over selected edges based on their cost. The probability for the edge associated with a previously visited city is 0,
4. select an edge. If at least one edge has non-zero probability, selection is based on the above distribution; otherwise, selection is random (from unvisited cities),
5. the city on 'the other end' of the selected edge becomes the current city c,
6. if the tour is complete, stop; otherwise, go to step 2.

However, as reported in [81], such operators transfer around 60% of the edges from parents — which means that 40% of edges are selected randomly.

Whitley, Starweather, and Fuquay [193] have developed a new crossover operator: the *edge recombination* crossover (ER), which transfers more than 95% of the edges from the parents to the single offspring. The ER operator explores the information on edges in a tour, e.g., for the tour

(3 1 2 8 7 4 6 9 5),

the edges are (3 1), (1 2), (2 8), (8 7), (7 4), (4 6), (6 9), (9 5), and (5 3). After all, edges — not cities — carry values (distances) in the TSP. The objective function to be minimized is the total of edges which constitute a legal tour. The position of a city in a tour is not important: tours are circular. Also, the direction of an edge is not important: both edges (3 1) and (1 3) signal only that cities 1 and 3 are directly connected.

The general idea behind the ER crossover is that an offspring should be built exclusively from the edges present in both parents. This is done with help of the edge list created from both parent tours. The edge list provides, for each city

c, all other cities connected to city c in at least one of the parents. Obviously, for each city c there are at least two and at most four cities on the list. For example, for the two parents

$p_1 = (1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9)$ and
$p_2 = (4\ 1\ 2\ 8\ 7\ 6\ 9\ 3\ 5)$,

the edge list is

city 1: edges to other cities: 9 2 4
city 2: edges to other cities: 1 3 8
city 3: edges to other cities: 2 4 9 5
city 4: edges to other cities: 3 5 1
city 5: edges to other cities: 4 6 3
city 6: edges to other cities: 5 7 9
city 7: edges to other cities: 6 8
city 8: edges to other cities: 7 9 2
city 9: edges to other cities: 8 1 6 3.

The construction of the offspring starts with a selection of an initial city from one of the parents. In [193] the authors selected one of the initial cities (e.g., 1 or 4 in the example above). The city with the smallest number of edges in the edge list is selected. If these numbers are equal, a random choice is made. Such selection increases the chance that we complete a tour with all edges selected from the parents. With a random selection, the chance of having edge failure, i.e., being left with a city without a continuing edge, would be much higher. Assume we have selected city 1. This city is directly connected with three other cities: 9, 2, and 4. The next city is selected from these three. In our example, cities 4 and 2 have three edges, and city 9 has four. A random choice is made between cities 4 and 2; assume city 4 was selected. Again, the candidates for the next city in the constructed tour are 3 and 5, since they are directly connected to the last city, 4. Again, city 5 is selected, since it has only three edges as opposed to the four edges of city 3. So far, the offspring has the following shape:

(1 4 5 x x x x x x).

Continuing this procedure we finish with the offspring

(1 4 5 6 7 8 2 3 9),

which is composed entirely of edges taken from the two parents. From a series of experiments [193], edge failure occured at a very low rate (1% – 1.5%).

The ER operator was tested [193] on three TSPs with 30, 50, and 75 cities — in all cases it returned a solution better than the previously "best known" sequence.

Two years later, the edge recombination crossover was further enhanced [175]. The idea was that the 'common subsequences' were not preserved in the ER crossover. For example, if the edge list contains the row with three edges

city 4: edges to other cities: 3 5 1,

one of these edges repeats itself. Referring to the previous example, it is the edge (4 5). This edge is present in both parents. However, it is listed as other edges, e.g., (4 3) and (4 1), which are present in one parent only. The proposed solution [175] modifies the edge list by storing 'flagged' cities:

city 4: edges to other cities: 3 -5 1;

the character '-' means simply that the flagged city 5 should be listed twice. In the previous example of two parents

p_1 = (1 2 3 4 5 6 7 8 9) and
p_2 = (4 1 2 8 7 6 9 3 5),

the (enhanced) edge list is:

city 1: edges to other cities: 9 -2 4
city 2: edges to other cities: -1 3 8
city 3: edges to other cities: 2 4 9 5
city 4: edges to other cities: 3 -5 1
city 5: edges to other cities: -4 6 3
city 6: edges to other cities: 5 -7 9
city 7: edges to other cities: -6 -8
city 8: edges to other cities: -7 9 2
city 9: edges to other cities: 8 1 6 3.

The algorithm for constructing a new offspring gives priority to flagged entries: this is important only in the cases where three edges are listed — in two other cases either there are no flagged cities, or both cities are flagged. This enhancement (plus a modification for making better choices when random edge selection is necessary) further improved the performance of the system [175].

The edge recombination operators indicate clearly that the path representation might be too poor to represent important properties of a tour — this is why it was complemented by the edge list. Are there other representations more suitable for the traveling salesman problem? Well, we cannot give a positive 'yes' for the answer. However, it is worthwhile to experiment with other, possibly non-vector, representations.

During the last two years, there were at least three independent attempts to construct an evolution program using matrix representation for chromosomes. These were by Fox and McMahon [59], Seniw [166], and Homaifar and Guan [94], We discuss them briefly, in turn.

Fox and McMahon [59] represented a tour as a precedence binary matrix M. Matrix element m_{ij} in row i and column j contains a 1 if and only if the city i occurs before city j in the tour. For example, a tour

(3 1 2 8 7 4 6 9 5)

	1	2	3	4	5	6	7	8	9
1	0	1	0	1	1	1	1	1	1
2	0	0	0	1	1	1	1	1	1
3	1	1	0	1	1	1	1	1	1
4	0	0	0	0	1	1	0	0	1
5	0	0	0	0	0	0	0	0	0
6	0	0	0	0	1	0	0	0	1
7	0	0	0	1	1	1	0	0	1
8	0	0	0	1	1	1	1	0	1
9	0	0	0	0	1	0	0	0	0

Fig. 10.1. Matrix representation of a tour

is represented in matrix form in Figure 10.1.

In this representation, the $n \times n$ matrix M representing a tour (total order of cities) has the following properties:

1. the number of 1s is exactly $\frac{n(n-1)}{2}$,
2. $m_{ii} = 0$ for all $1 \leq i \leq n$, and
3. if $m_{ij} = 1$ and $m_{jk} = 1$ then $m_{ik} = 1$.

If the number of 1s in the matrix is less than $\frac{n(n-1)}{2}$, and the two other requirements are satisfied, then the cities are partially ordered. This means that we can complete such a matrix (in at least one way) to get a legal tour (total order of cities). As stated in [59]:

> "The Boolean matrix representation of a sequence encapsulates all of the information about the sequence, including both the micro-topology of individual city-to-city connections and the macro-topology of predecessors and successors. The Boolean matrix representation can be used to understand existing operators and to develop new operators that can be applied to sequences to produce desired effects while preserving the necessary properties of the sequence."

The two new operators developed in [59] were *intersection* and *union*. Both are binary operators (crossover-like operators). As for other evolution programs (e.g., GENETIC-2 for the transportation problem; Chapter 9), such operators should combine the features of both parents and preserve constraints (requirements) at the same time.

The intersection operator is based on the observation that the intersection of bits from both matrices results in a matrix where (1) the number of 1s is not greater than $\frac{n(n-1)}{2}$, and (2) the two other requirements are satisfied. Thus, we can complete such a matrix to get a legal tour (total order of cities).

For example, two parents

$p_1 = (1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9)$ and
$p_2 = (4\ 1\ 2\ 8\ 7\ 6\ 9\ 3\ 5)$

are represented by two matrices (Figure 10.2).

The intersection of these two matrices gives the matrix displayed in Figure 10.3.

	1	2	3	4	5	6	7	8	9
1	0	1	1	1	1	1	1	1	1
2	0	0	1	1	1	1	1	1	1
3	0	0	0	1	1	1	1	1	1
4	0	0	0	0	1	1	1	1	1
5	0	0	0	0	0	1	1	1	1
6	0	0	0	0	0	0	1	1	1
7	0	0	0	0	0	0	0	1	1
8	0	0	0	0	0	0	0	0	1
9	0	0	0	0	0	0	0	0	0

	1	2	3	4	5	6	7	8	9
1	0	1	1	0	1	1	1	1	1
2	0	0	1	0	1	1	1	1	1
3	0	0	0	0	1	0	0	0	0
4	1	1	1	0	1	1	1	1	1
5	0	0	0	0	0	0	0	0	0
6	0	0	1	0	1	0	0	0	1
7	0	0	1	0	1	1	0	0	1
8	0	0	1	0	1	1	1	0	1
9	0	0	1	0	1	0	0	0	0

Fig. 10.2. Two parents

	1	2	3	4	5	6	7	8	9
1	0	1	1	0	1	1	1	1	1
2	0	0	1	0	1	1	1	1	1
3	0	0	0	0	1	0	0	0	0
4	0	0	0	0	1	1	1	1	1
5	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	1
7	0	0	0	0	0	0	0	0	1
8	0	0	0	0	0	0	0	0	1
9	0	0	0	0	0	0	0	0	0

Fig. 10.3. First phase of the intersection operator

The partial order imposed by the result of intersection requires that city 1 precedes cities 2, 3, 5, 6, 7, 8, and 9; city 2 precedes cities 3, 5, 6, 7, 8, and 9; city 3 precedes city 5; city 4 precedes cities 5, 6, 7, 8, and 9; and cities 6, 7, and 8 precede city 9.

During the next stage of the intersection operator, one of the parents is selected; some 1s (that are unique to this parent) are 'added', and the matrix is completed into a sequence through an analysis of the sums of the rows and columns. For example, the matrix from Figure 10.4 is a possible result after completing the second stage; it represents a tour

(1 2 4 8 7 6 3 5 9).

	1	2	3	4	5	6	7	8	9
1	0	1	1	1	1	1	1	1	1
2	0	0	1	1	1	1	1	1	1
3	0	0	0	0	1	0	0	0	1
4	0	0	1	0	1	1	1	1	1
5	0	0	0	0	0	0	0	0	1
6	0	0	1	0	1	0	0	0	1
7	0	0	1	0	1	1	0	0	1
8	0	0	1	0	1	1	1	0	1
9	0	0	0	0	0	0	0	0	0

Fig. 10.4. Final result of the intersection

The union operator is based on the observation that the subset of bits from one matrix can be safely combined with a subset of bits from the other matrix, provided that these two subsets have empty intersection. The operator partitions the set of cities into two disjoint groups (in [59] a special method was used to make this partition). For the first group of cities, it copies the bits from the first matrix; and for the second group of cities, it copies the bits from the second matrix. Finally, it completes the matrix into a sequence through an analysis of the sums of the rows and columns (as for intersection operator).

For example, the two parents p_1 and p_2 and the partition of cities into $\{1, 2, 3, 4\}$ and $\{5, 6, 7, 8, 9\}$ produce the matrix (Figure 10.5), which is completed as for the intersection operator.

	1	2	3	4	5	6	7	8	9
1	0	1	1	1	x	x	x	x	x
2	0	0	1	1	x	x	x	x	x
3	0	0	0	1	x	x	x	x	x
4	0	0	0	0	x	x	x	x	x
5	x	x	x	x	0	0	0	0	0
6	x	x	x	x	1	0	0	0	1
7	x	x	x	x	1	1	0	0	1
8	x	x	x	x	1	1	1	0	1
9	x	x	x	x	1	0	0	0	0

Fig. 10.5. First phase of the union operator.

The experimental results on different topologies of the cities (random, clusters, concentric circles) reveal an interesting characteristic of the union and intersection operators, which makes progress even when the elitism (preserving the best) option was not used. This was not the case for either ER or PMX operators. A solid comparison of several binary and unary (swap, slice, and invert) operators in terms of performance, complexity, and execution time is provided in [59].

The second approach in using matrix representation was described by one of my Master students, David Seniw [166]. Matrix element m_{ij} in the row i and column j contains a 1 if and only if the tour goes from city i directly to city j. This means that there is only one nonzero entry for each row and each column in the matrix (for each city i there is exactly one city visited prior to i, and exactly one city visited next to i). For example, a chromosome in Figure 10.6(a) represents a tour that visits the cities (1, 2, 4, 3, 8, 6, 5, 7, 9) in this order. Note also that this representation avoids the problem of specifying the starting city, i.e., Figure 10.6(a) also represents the tours (2, 4, 3, 8, 6, 5, 7, 9, 1), (4, 3, 8, 6, 5, 7, 9, 1, 2), etc.

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
1	0	1	0	0	0	0	0	0	0
2	0	0	0	1	0	0	0	0	0
3	0	0	0	0	0	0	0	1	0
4	0	0	1	0	0	0	0	0	0
5	0	0	0	0	0	0	1	0	0
6	0	0	0	0	1	0	0	0	0
7	0	0	0	0	0	0	0	0	1
8	0	0	0	0	0	1	0	0	0
9	1	0	0	0	0	0	0	0	0

(a)

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
1	0	1	0	0	0	0	0	0	0
2	0	0	0	1	0	0	0	0	0
3	0	0	0	0	0	0	0	1	0
4	0	0	0	0	1	0	0	0	0
5	0	0	0	0	0	0	1	0	0
6	0	0	0	0	0	0	0	0	1
7	1	0	0	0	0	0	0	0	0
8	0	0	0	0	0	1	0	0	0
9	0	0	1	0	0	0	0	0	0

(b)

Fig. 10.6. Binary matrix chromosomes

It is interesting to note that each complete tour is represented as a binary matrix with only one bit in each row and one bit in each column set to one. However, not every matrix with these properties would represent a single tour. Binary matrix chromosomes may represent multiple subtours. Each subtour will eventually loop back onto itself, without connecting to any other subtour in the chromosome. For example, a chromosome from Figure 10.6(b) represents two subtours (1, 2, 4, 5, 7) and (3, 8, 6, 9).

The subtours were allowed in the hope that natural clustering would take place. After the evolution program terminated, the best chromosome is reduced to a single tour by successively combining pairs of subtours using a deterministic algorithm. Subtours of one city (a tour leaving a city to travel right back to itself), having a distance cost of zero, were not allowed. A lower limit of $q = 3$

cities in a subtour was set in an attempt to prevent the GA from reducing a TSP problem to a large number of subtours each with very few cities (q is a parameter of the method).

Figure 10.7(a) depicts the subtours resulting from a sample run of the algorithm on a number of cities intentionally placed in clusters. As expected, the algorithm developed isolated subtours. Figure 10.7(b) depicts the tour after the subtours have been combined.

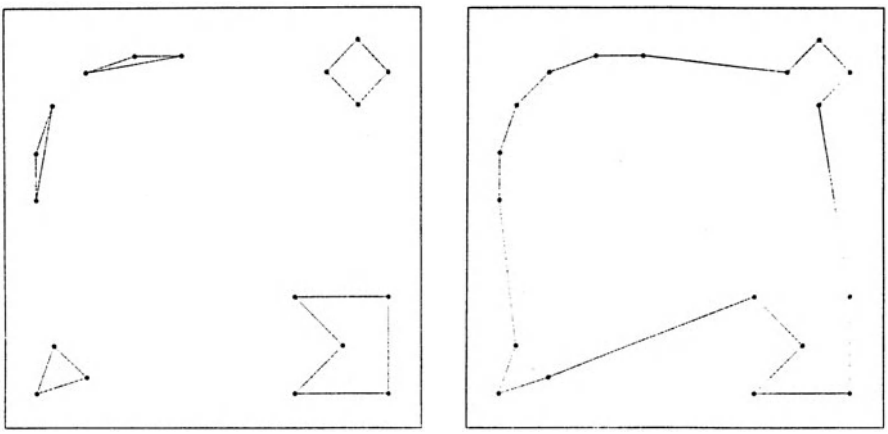

Fig. 10.7. Separate subtours and the final tour

Two genetics operators were defined: mutation and crossover. The mutation operator takes a chromosome, randomly selects several rows and columns in that chromosome, removes the set bits in the intersections of those rows and columns, and randomly replaces them in possibly a different configuration.

For example, let us consider the tour from Figure 10.6(a), representing the tour:

(1, 2, 4, 3, 8, 6, 5, 7, 9).

Assume that rows 4 and 6 and columns 2, 3, and 5 are randomly selected to participate in a mutation. The marginal sums for these rows and columns are calculated. The bits at the intersections of these rows and columns are removed and replaced randomly, though they must agree with the marginal sums. In other words, the submatrix corresponding to rows 4 and 6, and columns 2, 3, and 5 from the original matrix (Figure 10.8(a)), is replaced by another submatrix (Figure 10.8(b)).

The resulting chromosome represents a chromosome with two subtours:

(1, 2, 4, 5, 7) and (3, 8, 6, 9)

Fig. 10.8. Part of chromosome before (a) and after (b) mutation

and is represented in Figure 10.6(b).

The crossover operator begins with a child chromosome that has all bits reset to zero. The operator first examines the two parent chromosomes, and when it discovers the same bit (identical row and column) set (i.e., 1) in both parents, it sets a corresponding bit in the child (phase 1). The operator then alternately copies one set bit from each parent, until no bits exist in either parent which may be copied without violating the basic restrictions of chromosome construction (phase 2). Finally, if any rows in the child chromosome still do not contain a set bit, the chromosome will be filled in randomly (final phase). As the crossover traditionally produces two child chromosomes, the operator is executed a second time with the parent chromosomes transposed.

The crossover operator takes the common bits between the two parent chromosomes and places them in the initially empty child chromosome (phase 1). Then it alternately copies a bit from each parent until no more bits can be copied from either parent without violating the chromosome limitations (phase 2). If there are still empty places in the child chromosome, the chromosome will be filled in randomly (final phase). This produces the first child, after which the operator is run a second time with the parent chromosomes transposed. This will produce two offspring.

The following example of crossover starts with the first parent chromosome in Figure 10.9(a) representing two subtours:

(1, 5, 3, 7, 8) and (2, 4, 9, 6).

and the second parent chromosome (Figure 10.9(b)) representing a single tour:

(1, 5, 6, 2, 7, 8, 3, 4, 9).

The first two phases of building the first offspring are displayed in Figure 10.10.

The first offspring for crossover, after the final phase, is displayed in Figure 10.11. It represents a subtour:

(1, 5, 6, 2, 3, 4, 9, 7, 8).

The second offspring represents

(1, 5, 3, 4, 9) and (2, 7, 8, 6).

Note that there are common segments of the parent chromosomes in both offspring.

This evolution program gave a reasonable performance on several test cases from 30 cities to 512 cities. However, it is not clear what is the influence of the

	1	2	3	4	5	6	7	8	9
1	0	0	0	0	1	0	0	0	0
2	0	0	0	1	0	0	0	0	0
3	0	0	0	0	0	0	1	0	0
4	0	0	0	0	0	0	0	0	1
5	0	0	1	0	0	0	0	0	0
6	0	1	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	1	0
8	1	0	0	0	0	0	0	0	0
9	0	0	0	0	0	1	0	0	0

(a)

	1	2	3	4	5	6	7	8	9
1	0	0	0	0	1	0	0	0	0
2	0	0	0	0	0	0	1	0	0
3	0	0	0	1	0	0	0	0	0
4	0	0	0	0	0	0	0	0	1
5	0	0	0	0	0	1	0	0	0
6	0	1	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	1	0
8	0	0	1	0	0	0	0	0	0
9	1	0	0	0	0	0	0	0	0

(b)

Fig. 10.9. First (a) and second (b) parents

	1	2	3	4	5	6	7	8	9
1	0	0	0	0	1	0	0	0	0
2	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	1
5	0	0	0	0	0	0	0	0	0
6	0	1	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	1	0
8	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0

(a)

	1	2	3	4	5	6	7	8	9
1	0	0	0	0	1	0	0	0	0
2	0	0	1	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	1
5	0	0	0	0	0	1	0	0	0
6	0	1	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	1	0
8	1	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0

(b)

Fig. 10.10. Offspring for crossover, after (a) phase 1 and (b) phase 2

parameter q (minimum number of cities in a subtour) on the quality of the final solution. Also, the algorithms for combining several subtours into a single tour are far from obvious. On the other hand, the method has some similarities with Litke's recursive clustering algorithm [120], which recursively replaces clusters of size B by single representative cities until less than B cities remain. Then, the smaller problem is solved optimally. All clusters are expanded one by one and the algorithm sequences the expanded set between the two neighbors in the current tour.

The third approach based on matrix representation was recently proposed by Homaifar and Guan [94]. As in the previous approach, the m_{ij} element of the binary matrix M is set to 1 if and only if there is an edge from city i to city j. However, they used different crossover operators and heuristic inversion — we discuss them in turn.

	1	2	3	4	5	6	7	8	9
1	0	0	0	0	1	0	0	0	0
2	0	0	1	0	0	0	0	0	0
3	0	0	0	1	0	0	0	0	0
4	0	0	0	0	0	0	0	0	1
5	0	0	0	0	0	1	0	0	0
6	0	1	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	1	0
8	1	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	1	0	0

Fig. 10.11. Offspring for crossover, after the final phase

Two matrix crossover (MX) operators were defined [94]. These operators exchange all entries of the two parent matrices either after a single crossover point (1-point crossover) or between two crossover points (2-point crossover). An additional "repair algorithm" is run to (1) remove duplications, i.e., to ensure that each row and each column has precisely one 1, and (2) cut and connect cycles (if any) to produce a legal tour.

A 2-point crossover is illustrated by the following example. Two parent matrices are given in Figure 10.12; they represent two legal tours:

(1 2 4 3 8 6 5 7 9) and (1 4 3 6 5 7 2 8 9)

Two crossovers points were selected; these are points between columns 2 and 3 (first point), and between columns 6 and 7 (second point). The crossover points cut the matrices vertically: for each matrix, the first two columns constitute the first part of the division, columns 3, 4, 5, and 6 the middle part, and the last three columns the third part.

After the first step of the 2-point MX operator, entries of both matrices are exchanged between the crossover points (i.e., entries in columns 3, 4, 5, and 6). The intermediate result is given in Figure 10.13.

Both offspring, (a) and (b) are illegal; however, the total number of 1s in each intermediate matrix is correct (i.e., 9). The first step of the "repair algorithm" moves some 1s in matrices in such a way that each row and each column has precisely one 1. For example, in the offspring from Figure 10.13(a) the duplicate 1s occur in rows 1 and 3. The algorithm may move the entry $m_{14} = 1$ into m_{84}, and the entry $m_{38} = 1$ into m_{28}. Similarly, in the other offspring (Figure 10.13(b)) the duplicate 1s occur in rows 2 and 8. The algorithm may move the entry $m_{24} = 1$ into m_{34}, and the entry $m_{86} = 1$ into m_{16}. After the completion of the first step of the repair algorithm, the first offspring represents a (legal) tour,

(1 2 8 4 3 6 5 7 9),

	1	2	3	4	5	6	7	8	9
1	0	1	0	0	0	0	0	0	0
2	0	0	0	1	0	0	0	0	0
3	0	0	0	0	0	0	0	1	0
4	0	0	1	0	0	0	0	0	0
5	0	0	0	0	0	0	1	0	0
6	0	0	0	0	1	0	0	0	0
7	0	0	0	0	0	0	0	0	1
8	0	0	0	0	0	1	0	0	0
9	1	0	0	0	0	0	0	0	0

(a)

	1	2	3	4	5	6	7	8	9
1	0	0	0	1	0	0	0	0	0
2	0	0	0	0	0	0	0	1	0
3	0	0	0	0	0	1	0	0	0
4	0	0	1	0	0	0	0	0	0
5	0	0	0	0	0	0	1	0	0
6	0	0	0	0	1	0	0	0	0
7	0	1	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	1
9	1	0	0	0	0	0	0	0	0

(b)

Fig. 10.12. Binary matrix chromosomes with crossover points marked

	1	2	3	4	5	6	7	8	9
1	0	1	0	1	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	1	0	1	0
4	0	0	1	0	0	0	0	0	0
5	0	0	0	0	0	0	1	0	0
6	0	0	0	0	1	0	0	0	0
7	0	0	0	0	0	0	0	0	1
8	0	0	0	0	0	0	0	0	0
9	1	0	0	0	0	0	0	0	0

(a)

	1	2	3	4	5	6	7	8	9
1	0	0	0	0	0	0	0	0	0
2	0	0	0	1	0	0	0	1	0
3	0	0	0	0	0	0	0	0	0
4	0	0	1	0	0	0	0	0	0
5	0	0	0	0	0	0	1	0	0
6	0	0	0	0	1	0	0	0	0
7	0	1	0	0	0	0	0	0	0
8	0	0	0	0	0	1	0	0	1
9	1	0	0	0	0	0	0	0	0

(b)

Fig. 10.13. Two intermediate offspring after the first step of MX operator

and the second offpring represents a tour which consists of two subtours,

(1 6 5 7 2 8 9) and (3 4).

The second step of the repair algorithm should be applied to the second offspring only. During this stage, the algorithm cuts and connects subtours to produce a legal tour. The cut and connect phase takes into account the existing edges in the original parents. For example, the edge (2 4) is selected to connect these two subtours, since this edge is present in one of the parents. Thus the complete tour (a legal second offspring) is

(1 6 5 7 2 4 3 8 9).

The second operator used by Homaifar and Guan [94] to complement MX crossover was heuristic inversion. The operator reverses the order of cities be-

tween two cut points (just as simple inversion did — we discussed it earlier in this chapter). If the distance between the two cut points is large (high order inversion), the operator explores connections between 'good' paths, otherwise (low order inversion), the operator performs local search. However, there are two differences between classical and proposed inversion operators. The first is that the resulting offspring is accepted only if the new tour is better than the original. The second difference is that the inversion procedure selects a single city in a tour and checks for improvement for inversions of the (lowest possible) order 2. The first inversion which results in an improvement is accepted and the inversion procedure terminates. Otherwise, inversions of the order 3 are considered, and so on.

The reported results [94] indicate that evolution program with 2-point MX and inversion operators performed successfully on 30–100- city TSP problems. In the most recent experiment, the result of this algorithm for a 318-city problem was only 0.6% away from the optimal solution.

In this chapter we did not provide the exact results of various experiments for different data structures and 'genetic' operators. Rather, we made a general overview of numerous attempts in building a successful evolution program for the TSP. One of the reasons is that most of the quoted performance results heavily depend on many details (population size, number of generations, size of the problem, etc.). Moreover, many results were related to relatively small sizes of the TSP (up to 100 cities); as observed in [103]:

> "It does appear that instances as small as 100 cities must now be considered to be well within the state of the global optimization art, and instances must be considerably larger than this for us to be sure that heuristic approaches are really called for."

However, most of the papers cited in this chapter compare the proposed approach with other approaches. For these comparisons, two families of test cases were used:

- random collection of cities. Here, an empirical formula for the expected length of L^* of a minimal TSP tour is useful:

  $$L^* = k\sqrt{n \cdot R},$$

  where n is the number of cities, R is the area of the square box within which the cities were randomly placed, and k is an empirical constant of approximately 0.765 (see [177]).

- publicly available collection of cities (partly with optimal solutions), compiled by Gerhard Reinelt (Institut für Mathematik, Universität Augsburg). The collection is available via ftp as follows:

```
ftp titan.rice.edu (or, ftp 128.42.1.30)
Login Userid : anonymous
Password: anonymous
ftp> cd public
ftp> type binary
ftp> get tsplib.tar.Z
```

More information on a traveling salesman problem library is provided in [150].

To get a complete picture of the application of genetic algorithm techniques to the TSP, we should report the work of other researchers, e.g., [142] and [188], who used GAs for local optimization of the TSP. Local optimization algorithms (2-opt, 3-opt, Lin-Kernighan) are quite efficient. As stated in [103]:

> "Given that the problem is NP-hard, and hence polynomial-time algorithms for finding optimal tours are unlikely to exist, much attention has been addressed to the question of efficient approximation algorithms, fast algorithms that attempt only to find near-optimal tours. To date, the best such algorithms in practice have been based on (or derived from) a general technique known as local optimization, in which a given solution is iteratively improved by making local changes."

Local optimization algorithms, for a given (current) tour, specify a set of neighboring tours and replace the current tour by a (possibly) better neighbor. This step is applied until a local optimum is reached. For example, the 2-opt algorithm defines neighboring tours as tours where one can be obtained from the other by modifying two edges only.

Local search algorithms served as the basis for the development of the genetic local search algorithm [188], which

- uses a local search algorithm to replace each tour in the current population (of size μ) by a (local optimum) tour,
- extends the population by additional λ tours — offspring of the recombination operator applied to some tours in the current population,
- uses (again) a local search algorithm to replace each of the λ offspring in the extended population by a (local optimum) tour,
- reduces the extended population to its original size, μ, according to some selection rules (survival of the fittest),
- repeats the last three steps until some stopping condition is met (evolution process).

Note that there are some similarities between the genetic local search algorithm and the $(\mu + \lambda)$ evolution strategy (Chapter 8). As in the $(\mu + \lambda)$–ES, μ individuals produce λ offspring and the new (extended) population of $(\mu + \lambda)$ individuals is reduced by a selection process again to μ individuals.

The above genetic local search algorithm [188] is similar to a genetic algorithm for the TSP proposed earlier by Mühlenbein, Gorges-Schleuter, and Krämer [142], which encourages "intelligent evolution" of individuals. The algorithm

- uses a local search algorithm (opt-2) to replace each tour in the current population by a (local optimum) tour,
- selects partners for mating (above-average individuals get more offspring),
- reproduces (crossover and mutation),
- searches for the minimum by each individual (reduction, problem solver, expansion),
- repeats the last three steps until some stopping condition is met (evolution process).

The crossover used in this algorithm is a version of the order crossover (OX). Here, two parents (with two cut points marked by '|'),

p_1 = (1 2 3 | 4 5 6 7 | 8 9) and
p_2 = (4 5 2 | 1 8 7 6 | 9 3),

would produce the offspring in the following way. First, the segments between cut points are copied into offspring:

o_1 = (x x x | 4 5 6 7 | x x) and
o_2 = (x x x | 1 8 7 6 | x x).

Next, (instead of starting from the second cut point of one parent as was the case for OX), the cities from the other parent are copied in the same order from the beginning of the string, omitting symbols already present:

o_1 = (2 1 8 | 4 5 6 7 | 9 3) and
o_2 = (2 3 4 | 1 8 7 6 | 5 9).

The experimental results were at least encouraging. The algorithm found a tour for the 532 city problem and the length was found to be 27702, which is within 0.06% of the optimal solution (27686, found by Padberg and Rinaldi [146]).

It seems that a good evolution program for the TSP should incorporate local improvement operators (mutation group), based on algorithms for local optimization, together with carefully designed binary operator(s) (crossover group), which would incorporate heuristic information about the problem. We conclude this chapter by a simple observation: the quest for an evolution program for the TSP, which would include 'the best' representation and 'genetic' operators to be performed on them, is still going on!

11. Drawing Graphs, Scheduling, and Partitioning

Said a disappointed visitor,
'Why has my stay here yielded
no fruit?'
'Could it be because you lacked
the courage to shake the tree?'
said the Master benignly.

Anthony de Mello, *One Minute Wisdom*

As stated in the Introduction, it seems that most researchers "modified" their implementations of genetic algorithms either by using non-standard chromosome representation or by designing problem-specific genetic operators (e.g., [63], [185], [29], [36], etc.) to accommodate the problem to be solved, thus building efficient evolution programs. The first two modifications were discussed in detail in the previous two chapters (Chapters 9 and 10) for the transportation problem and the traveling salesman problem, respectively. In this chapter, we have made a somewhat arbitrary selection of a few other evolution programs developed by the author and other researchers, which are based on non-standard chromosome representation and/or problem-specific knowledge operators. We present systems for the graph drawing problem (Section 11.1), scheduling problems (Section 11.2), the timetable problem (Section 11.3), and partitioning problems (Section 11.4). The described systems and the results of their applications provide an additional argument to support the evolution programming approach, which promotes creation of data structures together with operators for a particular class of problems.

11.1 Drawing a directed graph

Drawing a directed graph can be considered as a problem of finding a layout of nodes and arcs on a page which is optimized according to certain aesthetic criteria chosen to characterize "good" drawings of graphs. For most reasonable aesthetic criteria, however, finding an exact solution for large graphs turns out

to be prohibitively expensive. The problem of drawing directed graphs has become of great practical importance with the recent development of interactive software tools. Such systems often use diagrams such as transition diagrams or structure diagrams, that are essentially some form of directed graph, to represent and illustrate various aspects of software systems. Since these diagrams are often generated or modified by the system, the system must be able to display the resulting diagrams in an intelligible fashion. In these systems, ensuring reasonable response time is important and a layout that is nearly optimal is generally quite acceptable. It is therefore desirable to investigate methods for finding approximate solutions.

A large number of algorithms have been proposed for drawing graphs. The kinds of algorithms used, and their costs, vary according to the class of graph for which they are intended (e.g., trees, planar graphs, hierarchic graphs or general undirected graphs), the aesthetic criteria they consider, and the methods they use for optimizing the layout. In most cases, finding optimal layouts for large graphs is prohibitively expensive, so a number of heuristic methods have been investigated that find approximate solutions in a reasonable amount of time. A good discussion of the problem of drawing graphs, aesthetic criteria that have been considered, and various methods that have been proposed is given in [187]. A more extensive bibliography is given in [51].

Eades and Lin [50] discuss an approach to drawing directed graphs that is based upon the following three aesthetic criteria[1]:

C_1 Arcs pointing upward should be avoided.
C_2 Nodes should be distributed evenly over the page.
C_3 There should be as few arcs crossing as possible.

Eades and Lin's approach works in three stages, addressing each of these criteria in turn. At each stage the problem of satisfying the corresponding criterion is formulated as a graph-theoretic problem.

In the first stage, the graph is turned into a directed acyclic graph (DAG) by reversing some of the arcs. The resulting DAG can be arranged so that all arcs point downwards by performing a topological sort. The DAG thus obtained is then used as the input to stages 2 and 3, and the arcs that were reversed are restored to their proper direction after all three stages have been completed. The major problem at stage 1 is to find a minimal set of arcs to reverse; this is called the "feedback arc set problem" and is known to be NP-hard [25].

In the second stage, nodes are distributed evenly on the page. This is done by arranging the graph in layers and allocating nodes to layers in such a way as to match the height and width of the graph to the size of the page. This is equivalent to the "multiprocessor scheduling problem" and is also known to be NP-hard [25].

In the third stage, nodes are ordered within layers so as to minimize the number of arc crossings. This problem is also NP-hard, even when there are only two layers in the graph.

[1]The first criterion assumes that the preferred direction for arcs is downwards, but this can easily be modified so that the preferred direction is left to right.

Since all three stages are NP-hard, Eades and Lin discuss a number of heuristics that can be used at each stage and discuss some of their properties. At stage 3 they add dummy nodes to arcs spanning more than one layer, so that stage 3 can be done a layer at a time. In the resulting graph, any bends in arcs occur at dummy nodes.

In order to investigate the applicability of genetic algorithms to the problem of drawing directed graphs, we have implemented two systems, called GRAPH-1 and GRAPH-2, which use genetic algorithms to find layouts for directed graphs. We have based our approach loosely on that described by Eades and Lin, but for our initial implementations we have only considered aesthetic criteria C_1 and C_3. The way in which we approach C_1 does not place all nodes on separate layers, so we do some of what their stage 2 does. At present, however, we do not attempt to spread nodes evenly on the page; we are currently developing an additional procedure to address this.

GRAPH-1 tries to optimize the layout of the graph according to both C_1 and C_3 at once, as in classical genetic algorithms. GRAPH-2 tackles the problem in two phases, first optimizing according to criterion C_1, then optimizing according to criterion C_3. This is closer to Eades and Lin's approach, and leads to a genetic algorithm for an optimization task with a decomposed evaluation function.

In formulating the graph drawing problem, we consider the page to be divided into a number of squares. The graph will be drawn with each node in one square and arcs drawn as straight lines between squares. The graph is assumed to have N nodes, numbered 1 through N, and the page is assumed to be H squares high and W squares wide. The size of square used is arbitrary, but it determines the "granularity" of layouts considered and hence the precision with which a near-optimal layout can be obtained.

In constructing evaluation functions for our systems, we need to consider, for a given candidate, the number of arcs that do not point downwards and the number of arc crossings. In order to allow more possible solutions to be explored, we consider arcs pointing upwards and arcs pointing horizontally separately. In GRAPH-2, we will also need to consider whether two nodes occupy the same square on the page. These are calculated, for a candidate M, by the following functions:

- $n_u(M)$ is the number of upward pointing arcs in M,
- $n_h(M)$ is the number of horizontal arcs in M,
- $n_c(M)$ is the number of crossings between arcs in M, and
- $n_o(M)$ is the number of nodes in M which occupy the same square.

Each evaluation function will be a linear combination of some of these functions, with weightings chosen according to the penalty to be associated with each feature.

11.1.1 Graph-1

In GRAPH-1 we use perhaps the most natural data structure for representing a solution for the problem of graph layout: an $H \times W$ matrix whose elements correspond to the squares on the page. Each matrix element is set to n if node n is located in the corresponding square on the page, and 0 if that square is empty. Figure 11.1 shows a graph, G_1, used in our experiments; Figure 11.2 shows the genetic representation used by GRAPH-1 for this graph when the page size is 6×10.

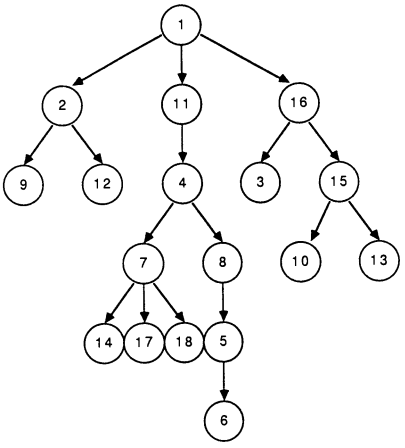

Fig. 11.1. Example graph G_1

0	0	0	0	1	0	0	0	0	0
0	2	0	0	11	0	0	16	0	0
9	0	12	0	4	0	3	0	15	0
0	0	0	7	0	8	0	10	0	13
0	0	14	17	18	5	0	0	0	0
0	0	0	0	0	6	0	0	0	0

Fig. 11.2. Genetic representation of G_1 for a 6×10 page

Every candidate in the initial population has each natural number between 1 and N, inclusive, assigned to a random element in the matrix, and 0s in the rest. In order to avoid invalid solutions being generated, every candidate generated must satisfy the constraint that each natural number between 1 and N, inclusive, occurs exactly once in the matrix.

The evaluation function used in GRAPH-1, for matrix M, is:

$$Eval(M) = a_u \cdot n_u(M) + a_h \cdot n_h(M) + a_c \cdot n_c(M) + 1$$

The coefficients a_u, a_h, and a_c determine the significance of upward pointing, horizontal and crossing arcs, respectively. The minimum value of this function is 1, which occurs when all arcs point downward and no arcs cross. In our experiments, we have used $a_u = 2$, $a_h = a_c = 1$.

The genetic operators used in GRAPH-1 were designed to preserve the constraint mentioned above.

Swap-rows: Select two arbitrary rows in the matrix and swap them.

Swap-columns: The same operator for columns.

Swap-in-row: Select a row; if the row contains at least two nodes, select two nodes and swap them.

Swap-in-col: The same for columns.

Single-mutate: Relocate a single node in a matrix. This preserves the intuition behind the classical mutation operator which is to perform the smallest possible change.

Small-mutate: Select two entries in a matrix and swap them.

Large-cont-mutate: Select two disjoint parts of a matrix, each consisting of contiguous rows and columns, and swap them.

Invert-a-row: Reverse the order of elements in a row.

Invert-a-column: Reverse the order of elements in a column.

Invert-part-row: Reverse the order of some elements in a row.

Invert-part-column: Reverse the order of some elements in a column.

Crossover: Select some rows and columns; for each row i and column j selected, swap the corresponding elements of the two parent matrices. Following this operation, some repairs may be necessary to ensure that the resulting matrices are valid solutions (i.e., that they satisfy the constraint mentioned above).

Cont-crossover: Same as the previous operator, but selected rows and columns constitute contiguous sets.

11.1.2 Graph-2

In GRAPH-2 we split the optimization task into separate phases. The first phase only considers the directions of arcs — whether they point upwards, downwards or horizontally. Once a satisfactory population is found, it is used to generate the initial population for the second phase, which only considers whether arcs or nodes cross. Once a satisfactory population is found here, an evaluation function

which is a combination of the two previous ones is used to determine the best candidate for the ultimate layout for the output graph.

The data structure used in GRAPH-2 is a $2 \times N$ matrix, in which the i-th column represents the (x, y) coordinates of node i. Phase 1 ignores the x coordinates, while phase 2 utilizes both coordinates. Figure 11.3 shows another graph, G_2, used in our experiments; Figure 11.4 shows the genetic representation of this graph used by GRAPH-2 when the page size is 5×9.

Every candidate in the initial population has each node allocated to a random square on the page. At no point in our algorithm do we ensure that every node is allocated to a different square. Later, we will see why we do not need to explicitly preserve this constraint.

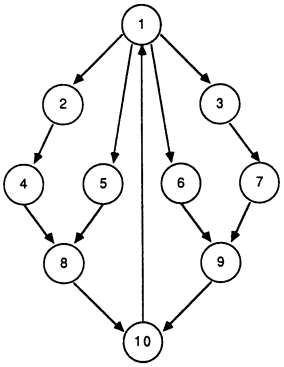

Fig. 11.3. Example graph G_2

N	1	2	3	4	5	6	7	8	9	10
x	4	2	6	0	3	5	8	2	6	4
y	4	3	3	2	2	2	2	1	1	0

Fig. 11.4. Genetic representation of G_2 for a 5×9 page

As stated earlier, phase 1 only considers the direction of arcs (whether they point upwards, downwards or horizontally). Consequently, the evaluation function used in phase 1 is:

$$Eval_1(M) = a_u \cdot n_u(M) + a_h \cdot n_h(M) + 1$$

Again, in our experiments, we have used $a_u = 2$ and $a_h = 1$.

The genetic operators used at this stage are variations of the three classical operators:

Y-swap: Swap the y coordinates of 2 random nodes within a structure.

Y-crossover: Crossover the y coordinates of a random node from one structure with the corresponding node of another structure.

Y-mutate: Create a new random y coordinate for any node of a structure.

Note that these operators do nothing to guarantee that two nodes do not occupy the same square on the page. They simply help us to search for a low-cost arrangement of nodes along the vertical dimension. The Y-swap operator alters two nodes which, if their y coordinates were swapped, would reduce the evaluation cost. The Y-crossover operator is like the classical crossover operator, in that it exchanges useful information between structures. Likewise, the Y-mutate operator allows new points in the search space to be explored.

Phase 1 is repeated a number of times with different initial populations. Each time the best solution found is saved and the set of vertical arrangements thus obtained is used to generate the initial population for phase 2.

The initial population for phase 2 is constructed by using the good set of y coordinates from phase 1 and random x coordinates. This phase considers whether arcs cross and whether nodes overlap, since our representation allows this to occur. Consequently, the evaluation function used in phase 2 is:

$$Eval_2(M) = a_c \cdot n_c(M) + a_o \cdot n_o(M) + 1$$

In our experiments, we have used $a_c = 1, a_o = 2$.

In designing the set of genetic operators to be used at this stage, we must take care not to undo the progress made in phase 1. Thus, these operators will not change y coordinates. The genetic operators used at this stage are, again, variations of the three classical operators:

X-swap: Swap the x coordinates of 2 random nodes within a structure.

X-crossover: Crossover the x coordinate of a random node from one structure with the corresponding node of another structure.

X-mutation: Randomly change the x coordinate of a given structure's node.

Note again that these operators do not guarantee that two nodes will not occupy the same square on the page. We simply let the evaluation function and the power of the genetic algorithm search for the best solution.

Once phase 2 is complete, we evaluate each structure using a combination of the evaluation functions used previously and take the best solution found.

11.1.3 Experiments and results

We have performed a number of experiments with the systems GRAPH-1 and GRAPH-2. In this section we present the results of applying these systems to the two graphs, G_1 and G_2, shown in Figures 11.1 and 11.3. In each case we used a page with 10 rows and 6 columns.

In the system GRAPH-1, we ran the genetic algorithm for 200 generations with each graph. The resulting layouts are shown in Figures 11.5 and 11.6.

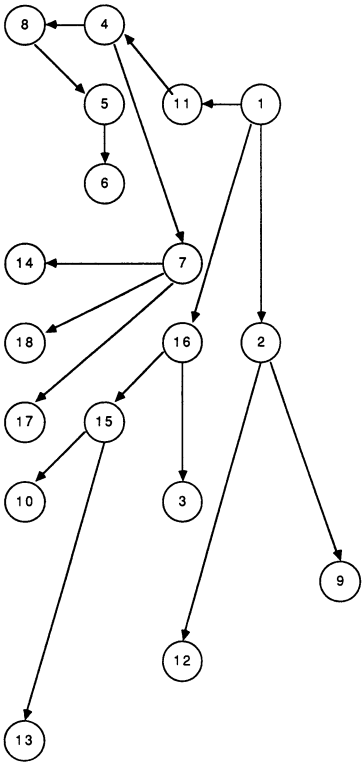

Fig. 11.5. The result of the GRAPH-1 system for G_1

These results are encouraging, given that the only factors taken into account in the evaluation function were the numbers of arcs that cross, point up or are horizontal. Both graphs display high regularity and constitute an excellent input for the algorithm to distribute nodes evenly on a page. Similar results were obtained in experiments with a larger number of nodes and a larger page size.

In the system GRAPH-2, for each graph, we ran the genetic algorithm for 200 generations for each of 10 passes through phase 1, and 200 generations for

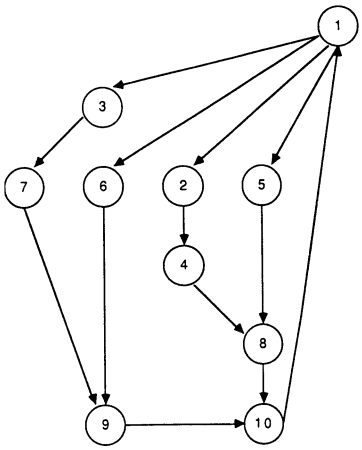

Fig. 11.6. The result of the GRAPH-1 system for G_2

phase 2. Using a population size of 50 meant that the initial population of phase 2 consisted of 5 structures for every y coordinate arrangement found by phase 1. The resulting layouts are shown in Figures 11.7 and 11.8.

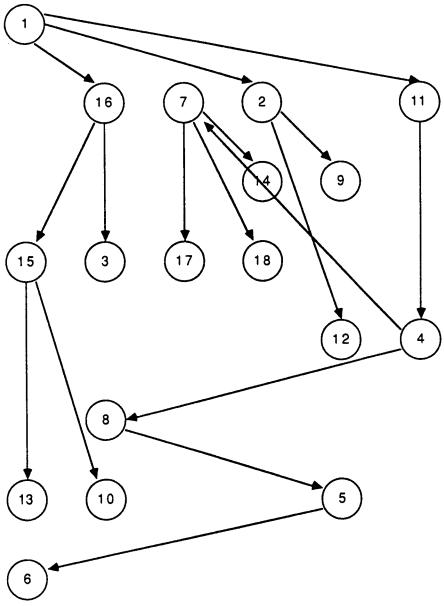

Fig. 11.7. The result of the GRAPH-2 system for G_1

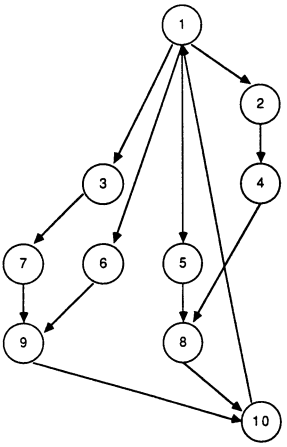

Fig. 11.8. The result of the GRAPH-2 system for G_2

Again, these results are encouraging, given the attributes taken into consideration by the evaluation functions.

For G_1, the general shape of the layout produced by GRAPH-2 is probably better than that produced by GRAPH-1. The odd positioning of nodes 1, 8, 5 and 6 would be rectified by the node distribution algorithm. However, it unnecessarily has one arc pointing upwards and one intersection of two arcs.

For G_2, the general shape of the layout produced by GRAPH-1 looks marginally better than that produced by GRAPH-2, though this difference would probably be negligible once node distribution has been performed. Again, GRAPH-2 unnecessarily has one arc pointing upwards and one intersection of two arcs.

These two graphs are not sufficient to make any conclusive comparisons between the two systems. However, they do suggest that GRAPH-1 is better at avoiding crossing arcs, while GRAPH-2 seems to be better at avoiding horizontal and upward arcs. Varying the number of generations, the probabilities associated with the various genetic operators and the weightings used for the various aesthetic features would clearly affect the results. Obviously, many more tests need to be performed before more definite conclusions can be drawn.

For several reasons, we are not yet able to evaluate the two presented approaches fully or compare them with other methods. Firstly, we only have results from few sample graphs; we need to test both systems on many other examples. Secondly, the algorithm to distribute nodes evenly on the page is still being implemented; we need to complete this before we can make meaningful comparisons with other methods. Thirdly, the two approaches presented here are, in some sense, too different. They use different genetic representations and dif-

ferent operators, and additionally, the system GRAPH-2 is based on the idea of a decomposed evaluation function.

To allow the various factors involved in these two systems to be compared and evaluated more precisely, we plan to implement a variation of GRAPH-1 which uses the same representation and genetic operators, but has a decomposed evaluation function, and a GRAPH-2 which uses the same representation and genetic operators, but has a single evaluation function. These should allow more meaningful comparisons to be made.

When we have these systems working well, we plan to incorporate the most successful system into an interactive programming tool. We could then allow the user to set various operational parameters, so the tradeoff between picture quality and speed is under the users' control. In particular, the user could ask for high quality drawings at the expense of extra time in order to generate publication quality diagrams from the system. We also plan to adding other aesthetic criteria (as discused in [187]) and allow user to control which ones are used.

11.2 Scheduling

A job shop is a process–organized manufacturing facility; its main characteristic is a great diversity of jobs to be performed [86]. A job shop produces goods (parts); these parts have one or more alternative process plans. Each process plan consists of a sequence of operations; these operations require resources and have certain (predefined) durations on machines. A job shop processes orders, where each order is for some number of the same part. The task of planning, scheduling, and controlling the work is very complex, and only limited analytical procedures are available to assist in these tasks [86], [60].

The job shop scheduling problem is to select a sequence of operations together with an assignment of start/end times and resources for each operation. The main considerations to be taken into account are the cost of having idle machine and labor capacity, the cost of carrying in-process inventory, and the need to meet certain order completion due dates. As stated in [86]:

> "Unfortunately, these considerations tend to conflict with each other. One can have a low cost of idle machine and labor capacity by providing only a minimum amount of machinery and manpower. However, this would result in considerable work waiting to be done and, therefore, large in-process inventories and difficulty in meeting completion due dates. On the other hand, one can essentially guarantee meeting completion due dates by providing so much machine and labor capacity that orders usually would not have to wait to be processed. However, this would result in excessive costs for idle machine and labor capacity. Therefore, it is necessary to strive for an economic compromise between these considerations."

There are various versions of the job shop scheduling problem, each characterized by some additional constraints (e.g., maintenance, machine down and setup times, etc.).

Let us consider a simple example of a job shop problem to illustrate the above description.

Example 11.1. Assume there are three orders, o_1, o_2, and o_3. For each order, the parts and the number of units to be produced are:

o_1 : 30× part a;
o_2 : 45× part b;
o_3 : 50× part a.

Each part has one or more alternative process plans:

a : plan # 1_a (opr_2, opr_7, opr_9);
a : plan # 2_a (opr_1, opr_3, opr_7, opr_8);
a : plan # 3_a (opr_5, opr_6);
b : plan # 1_b (opr_2, opr_6, opr_7);
b : plan # 2_b (opr_1, opr_9);

where terms opr_i denote the required operations to be performed. Each operation requires some times on one or more machines; these are:

opr_1 : (m_1 10) (m_3 20);
opr_2 : (m_2 20);
opr_3 : (m_2 20) (m_3 30);
opr_4 : (m_1 10) (m_2 30)(m_3 20);
opr_5 : (m_1 10) (m_3 30);
opr_6 : (m_1 40);
opr_7 : (m_3 20);
opr_8 : (m_1 50) (m_2 30) (m_3 10);
opr_9 : (m_2 20) (m_3 40).

Finally, each machine has its setup time necessary for changes in operation:

m_1 : 3;
m_2 : 5;
m_3 : 7.

□

The job shop problem enjoyed some interest in GA community. One of the first attempts to approach this problem was reported by Davis [32]. The main idea of his approach was to encode the representation of a schedule in such a way, that (1) the genetic operators would operate in a meaningful way, and (2) a decoder would always produce a legal solution to the problem. This strategy, to encode solutions for operations and to decode them for evaluation, is quite general and might be applied to a variety of constrained problems — the same

idea was used by Jones [104] to approach the partitioning problem (see Section 11.4).

In general, we would like to represent information on schedules, e.g., "machine m_2 performs operation o_1 on part a from time t_1 to time t_2". However, most operators (mutations, crossovers) applied to such a message would result in illegal schedules — this is why Davis [32] used an encoding/decoding strategy.

Let us see how the encoding strategy was applied to the job shop problem. The system developed by Davis [32] maintained a list of preferences for each machine; these preferences were linked to times. An initial member of a list is a time at which the list went into effect, the remaining part of the list is made up of some permutation of the orders, plus two additional elements: 'wait' and 'idle'. The decoding procedure simulated the job's operations in such way that whenever a machine was ready to make a choice, the first allowable operation from its preference list was taken. So if the preference list for the machine m_1 was

$$m_1 : (40\ o_3\ o_1\ o_2\ \text{'wait'}\ \text{'idle'}),$$

then the decoding procedure at time 40 would search for a part from the order o_3 for the machine m_1 to work on. If unsuccessful, the decoding procedure would search for a part from the orders o_1 and o_2 (i.e., first from o_1; in the case of failure, from o_2). This representation guarantees a legal schedule.

The operators were problem specific (they were derived from deterministic methods):

run-idle: this operator is applied only to preference lists of the machines that have been waiting for more than an hour. It inserts the 'idle' as the second member of the preference list and reset the first member (time) of the preference list to 60 (minutes);

scramble: the operator "scrambles" the members of a preference list;

crossover: the operator exchanges preference lists for selected machines.

The probabilities of these operators varied, from 5% and 40% for scramble and crossover, respectively, at the beginning of a run, down to 1% and 5%. The probability of run-idle was set to the percentage of the time the machine spent waiting, divided by the total time of the simulation.

However, the experiments were made on a small example of two orders, six machines, and three operations [32], thus it is difficult to evaluate the usefulness of this approach.

Another group of researchers approached the job shop problem from the TSP point of view [27], [183], [184], [193]. The motivation was that most operators developed for the TSP were 'blind', i.e., they did not use any information about the actual distances between cities (see Chapter 10). This means that these operators might be useful in other sequencing problems, where there is no distance between two points (cities, orders, jobs, etc.). However, it need not be

the case. Although both problems, the TSP and the scheduling problem, are sequencing problems, they display different (problem-specific) characteristics. For the TSP the important information is adjacency information on cities, whereas in the scheduling problem the relative order of items is of main concern. The adjacency information is useless for the scheduling problem, whereas the relative order is not important for the TSP due to cyclic nature of the tours: tours (1 2 3 4 5 6 7 8) and (4 5 6 7 8 1 2 3) are, in fact, identical. This is why we need different operators for different applications. As observed in [175]:

> "Gil Syswerda [183] conducted a study in which 'edge recombination' (a genetic operator specifically designed for the TSP) performed poorly relative to other operators on a job sequence scheduling task. While the population size used by Syswerda was small (30 strings) and good results were obtained on this problem using mutation alone (no recombination), Syswerda's discussion of the relative importance of position, order, and adjacency for different sequencing tasks raises an issue that has not been adequately addressed. Researchers, including ourselves [193], [194], seem to tacitly assume that all sequencing tasks are similar and that one genetic operator should suffice for all types of sequencing problems."

A similar observation was made one year earlier by Fox and McMahon [59]:[2]

> "An important concern is the applicability of each genetic operator to a variety of sequencing problems. For example, in the TSP, the value of a sequence is equivalent to the value of that sequence in reverse order. This trait is not true of all sequencing problems. In scheduling problems, this is a gross error."

In [175] six sequencing operators (order crossover, partially mapped crossover, cycle crossover, enhanced edge recombination, order-based crossover, and position-based crossover — all these operators were discussed in the previous chapter) were compared on two different sequencing tasks: a 30-city (blind) TSP and a 195-element sequencing task for a scheduling application. As expected, the results of schedule optimization (as far as the 'goodness' of the six operators is concern) were almost the opposite of the results from the TSP. In the case of schedule optimization, the enhanced edge recombination operator was the best, followed closely by order crossover, order-based crossover, and position-based crossover, with PMX and cycle crossovers being the worst. On the other hand, in the case of the TSP, the best were position-based and order-based crossovers, followed by the cycle crossover and PMX, with order crossover and enhanced edge recombination being the worst. These differences can be explained by examining how these operators preserve adjacency (for the TSP) and order (for the scheduling problem) information.

[2]1991, Reprinted with permission from Rawlins, G., *Foundations of Genetic Algorithms.*

Similar observations can be made for other sequencing (ordering) problems. In [38] Davis describes an order-based genetic algorithm for the following graph coloring problem:

> Given a graph with weighted nodes and n colors, achieve the highest score by assigning colors to nodes, such that no pair of connected (by a direct link) nodes can have the same color; the score is the total of weights of the colored nodes.

A simple greedy algorithm would sort the set of nodes in order of decreasing weights and process nodes (i.e., assigning the first legal color to the node from the list of colors) in this order. Clearly, this is a sequencing problem — at least one permutation of nodes would return the maximum profit, so we search for the optimal sequence of nodes. It is also clear that the simple greedy algorithm does not guarantee the optimum solution: some other techniques should be used. Again, on the surface, the problem is similar to the TSP, where we were after the best order of cities to be visited by a salesman. However, the 'nature' of the problem is very different: for example, in the graph coloring problem there are weights for nodes, whereas in the TSP the weights are distributed between nodes (as distances). In [38] Davis represented an ordering as a list of nodes (e.g., (2 4 7 1 4 8 3 5 9), as the path representation for the TSP) and used two operators: the order-based crossover (discussed in the previous chapter as an operator used for the TSP) and scramble sublist mutation. Even mutation, which should carry out a local modification of a chromosome, seems to be problem dependent [38]:

> "It is tempting to think of mutations as the swapping of the values of two fields on the chromosome. I have tried this on several different problems, however, and it doesn't work as well for me as an operator I call scramble sublist mutation."

The scramble sublist mutation selects a sublist of nodes from a parent and scrambles it in the offspring, i.e., the parent (with the beginning and the end of the selected sublist marked by |):

$$p = (2\ 4 \mid 7\ 1\ 4\ 8 \mid 3\ 5\ 9)$$

may produce the offspring

$$o = (2\ 4 \mid 4\ 8\ 1\ 7 \mid 3\ 5\ 9).$$

However, it remains to be seen how this operator performs for other ordering or scheduling problems. Again, let us cite from [38]:

> "Many other types of mutations can be employed on order-based problems. Scramble sublist mutation is the most general one I have used. To date nothing has been published on these types of operators, although this is a promising topic for future work."

Let us return to the scheduling problems. As mentioned earlier, Syswerda [183] developed an evolution program for scheduling problems. However, a simple chromosome representation was chosen:

> "In choosing a chromosome representation for the [...] scheduler, we have two basic elements to choose from. The first is the list of tasks to be scheduled. This list is very much like the list of cities to be visited in the TSP problem. [...] An alternative to using a sequence of tasks is directly to use a schedule as a chromosome. This may seem like an overly cumbersome representation, necessitating complicated operators to work on them, but is has a decided advantage when dealing with complicated real-world problems like scheduling. [...] In our case, the appeal of a clean and simple chromosome representation won over the more complicated one. The chromosome syntax we use for the scheduling problem is what was described above for the TSP, but instead of cities we will use orderings of tasks."

The chromosome was interpreted by a schedule builder — a piece of software which 'understands' the details of the scheduling task. This representation was supported with specialized operators. Three mutations were considered: position-based mutation (two tasks are selected at random, and the second task is placed before the first), order-based mutation (two tasks selected at random are swapped), and scramble mutation (same as Davis' scramble sublist mutation described in the previous paragraph). All three performed much better than random search, with order-based mutation being the clear winner. As mentioned earlier, the best crossover operators for the scheduling problem were order-based and position-based crossovers.

It seems, however, that the choice of a simple representation was not the best. Judging from other (unrelated) experiments, e.g., transportation problem (Chapter 9), we feel that the chromosome representation used should be much closer to the scheduling problem. It is true that in a such cases a significant effort must be placed in designing problem-specific 'genetic' operators; however, this effort would pay off in increased speed and improved performance of the system. Moreover, some operators might be quite so simple [183]:

> "A simple greedy algorithm running over the schedule could find a place for the high-priority task by removing a low-priority task or two and replacing them with the high-priority task."

We believe that in general, and for the scheduling problems in particular, this is the direction to follow: to incorporate the problem specific-knowledge not only in operators (as was done for a simple chromosome representation), but in the chromosome structures as well. The first attempts to apply such an approach have already emerged. In their study, Husbands, Mill, and Warrington [96] represented a chromosome as a sequence

$$(opr_1\ m_1\ s_1)\ (opr_2\ m_2\ s_2)\ (opr_3\ m_3\ s_3)\ \ldots\ ,$$

where opr_i, m_i, and s_i denote the i-th operation, machine, and setup, respectively.

In [9] the authors compared three representations, from the simplest (representation–1):

$$(o_1)\ (o_2)\ (o_3)\ \ldots\ ,$$

through intermediate (representation–2):

$$(o_1 \text{ plan } \#\ 1_a)\ (o_2 \text{ plan } \#\ 2_b)\ (o_3 \text{ plan } \#\ 2_a)\ \ldots\ ,$$

to the most complex (representation–3):

$$(o_1\ \langle opr_2 : m_2,\ opr_7 : m_3,\ opr_9 : m_2\rangle)\ (o_2\ \langle opr_1 : m_3,\ opr_9 : m_2\rangle)$$
$$(o_3\ \langle opr_1 : m_1,\ opr_3 : m_2,\ opr_7 : m_3,\ opr_8 : m_1\rangle)\ \ldots$$

The results for representation–3 were significantly better than for the two other representations. In the concluding discussion, the authors observed [9]:

> "The operators themselves must be adjusted to suit the domain requirements. The chromosome representation should contain all the information that pertain to the optimization problem."

We agree. This is, after all, what evolution programming is all about.

11.3 The timetable problem

One of the most interesting problems in Operations Research is the timetable problem. The timetable problem has important practical applications: it has been intensively studied and it is known to be NP-hard [53].

The timetable problem incorporates many nontrivial constraints of various kinds — this is probably why it was only recently that first (as far as the author is aware) attempt was made [28] to apply genetic algorithm techniques to approach this problem. There are many versions of the timetable problem; one of them can be described by

- a list of teachers $\{T_1, \ldots, T_m\}$,
- a list of time intervals (hours) $\{H_1, \ldots, H_n\}$,
- a list of classes $\{C_1, \ldots, C_k\}$.

The problem is to find the optimal timetable (teachers – times – classes); the objective function aims to satisfy some goals (soft constraints). These include didactic goals (e.g., spreading some classes over the whole week), personal goals (e.g., keeping afternoons free for some part-time teachers), and organizational goals (e.g., each hour has an additional teacher available for temporary teaching post).

The constraints include:

- there is a predefined number of hours for every teacher and every class; a legal timetable must "agree" with these numbers,
- there is only one teacher in a class at a time,
- a teacher cannot teach two classes at a time,
- for each class scheduled at some time slot, there is a teacher.

It seems that the most natural chromosome representation of a potential solution of the timetable problem is a matrix representation: a matrix $(R)_{ij}$ ($1 \leq i \leq m$, and $1 \leq j \leq n$), where each row corresponds to a teacher, and each column to an hour; the elements of the matrix R are classes ($r_{ij} \in \{C_1, \ldots, C_k\}$).[3]

In [28] the constraints were managed mainly by genetic operators (the authors used also a repair algorithm to eliminate cases where more than one teacher is present in the same class at the same time). The following genetic operators were used:

mutation of order k: the operator selects two contiguous sequences of k elements from the same row in the matrix R, and swaps them,

day mutation: this operator is a special case of the previous one: it selects two groups of columns (hours) of the matrix R which correspond to different days, and swaps them,

crossover: given two matrices R_1 and R_2, the operator sorts the rows of the first matrix in order of decreasing values of a so-called local fitness function (a part of the fitness function due only to characteristics specific to each teacher) and the best b rows (b is a parameter determined by the system on the basis of the local fitness function and both parents) are taken as a building block; the remaining $m - b$ rows are taken from the matrix R_2.

The resulting evolution program was successfully tested on data for a large school in Milan, Italy [28].

11.4 Partitioning objects and graphs

Several evolution programs were constructed by Jones and Beltramo [104] for a class of partitioning problems: partitioning n objects into k categories. These programs used different representations and several operators to manipulate them. It is interesting to observe the influence of the incorporation of the problem-specific knowledge on the performance of the developed evolution programs. Two test problems were selected:

[3] Actually, in [28] the elements of the matrix R were classes with three possible subscripts to include the concepts of sections, temporary teaching posts, etc.

- to divide n numbers into k groups to minimize the differences among the group sums, and
- to partition the 48 states of the continental U.S. into 4 color groups to minimize the number of bordering state pairs that are in the same group.

The first group of evolution programs encoded partitions as n-strings of integer numbers

$$(i_1, \ldots, i_n),$$

where the j-th integer $i_j \in \{1, \ldots, k\}$ indicates the group number assigned to object j; this is a *group-number* encoding.

The group-number encoding creates a possibility of applying standard operators. A mutation would replace a single (randomly selected) gene i_j by a (random) number from $\{1, \ldots, k\}$. Crossovers (single-point or uniform) would always produce a legitimate offspring. However, as pointed out in [104], an offspring (after mutation or crossover) may contain less than k groups; moreover, an offspring of two parents, both representing the same partition, may represent totally different partition, due to different numbering of groups. Special repair algorithms (rejection method, renumbering the parents) were used to eliminate these problems. Also, we can consider applying the edge-based crossover (defined in the previous chapter). Here we assume that two objects are connected by an edge, if and only if they are in the same group. The edge-based crossover constructs an offspring by combining edges from the parents.

It is interesting to note that several experiments on the two test problems indicate the superiority of the edge-based operator [104]; however, the representation used does not support this operator. As reported, it took 2–5 times more computation time per iteration than the other crossover methods. This is due to inappropriate representation: for example, two parents

$p_1 = (11222233)$ and
$p_2 = (12222333)$

represent the following edges:

edges for p_1: (12), (34), (35), (36), (45), (46), (56), (78),
edges for p_2: (23), (24), (25), (34), (35), (45), (67), (68), (78).

An offspring should contain edges present in at least one parent, e.g.,

(11222333)

represents the following edges:

(12), (34), (35), (45), (67), (68), (78).

However, the process of selection of edges is not straightforward: the selection of (56) and (67) — where both these edges are represented above — implies the presence of the edge (57), which is not there.

It seems that some other representations might be more suitable for the problem. The second group of evolution programs encoded partitions as $n+k-1$-strings of distinct integer numbers

$$(i_1, \ldots, i_{n+k-1});$$

integers from the range $\{1, \ldots, n\}$ represent the objects, and integers from the range $\{n+1, \ldots, n+k-1\}$ represent separators; this is a *permutation with separators* encoding. For example, the 7-string

$$(1122233)$$

is represented as a 9-string

$$(128345967),$$

where 8 and 9 are separators.

Of course, all $k-1$ separators must be used; also, they cannot appear at the first or last position, and they cannot appear together, one next to the other (otherwise, a string would decode into less than k groups).

As usual, care should be taken in designing operators. These would be similar to some operators used for solving the traveling salesman problem, where the TSP is represented as a permutation of cities. A mutation would swap two objects (separators are excluded). Two crossovers were considered: order crossover (OX) and partially matched crossover (PMX) — these were discussed in Chapter 10. A crossover would repeat its operation until the offspring[4] decodes into a partition with k groups.

Generally, the results of the evolution programs based on permutation with separators encoding were better than for the programs based on group-number encoding. However, neither coding method makes significant use of the problem-specific knowledge. One can build a third family of evolution programs, incorporating knowledge in the objective function. This very thing was done in [104] in the following way.

The representation used was the simplest one: each n-string represents n objects:

$$(i_1, \ldots, i_n),$$

where the $i_j \in \{1, \ldots, n\}$ denotes the object number — hence $i_j \neq i_p$ for $j \neq p$. The interpretation of this representation uses a greedy heuristic: the first k objects in the string are used to initialize k groups, i.e., each of the first k objects is placed in a separate group. The remaining objects are added on first-come first-go basis, i.e., they are added in the order they appear in the string; they are placed in a group which yields the best objective value.

This greedy heuristic also simplifies operators: every permutation encodes a valid partition, so we can use the same operators as for the traveling salesman problem. Needless to say, the "greedy decoding" approach significantly outperforms evolution programs based on other decodings: group-number and permutation with separators [104].

[4]Jones and Beltramo [104] produced only one offspring per crossover.

Another interesting approach to the partitioning problem[5] is presented in [114]. Von Laszewski encodes partitions using group-number encoding, i.e., partitions are represented as n-strings of integer numbers,

$$(i_1, \ldots, i_n),$$

where the j-th integer $i_j \in \{1, \ldots, k\}$ indicates the group number assigned to object j. However, this representation is supported by "intelligent structural operators": structural crossover and structural mutation. We discuss them in turn.

structural crossover: the mechanism of the structural crossover is explained by the following example. Assume, there are two selected parents (12-strings)

$$p_1 = (112311232233) \text{ and } p_2 = (112123122333).$$

These strings decode into the following partitions:

$$p_1 : \{1,2,5,6\}, \{3,7,9,10\}, \{4,8,11,12\}, \text{ and}$$
$$p_2 : \{1,2,4,7\}, \{3,5,8,9\}, \{6,10,11,12\}.$$

First, a random partition is chosen: say, partition #2. This partition is copied from p_1 into p_2:

$$p_2' = (112123222233).$$

The copying process (as seen in the above example) usually destroys the requirement of equal partition sizes, hence we apply a repair algorithm. Note that in the original p_2 there were elements assigned to partition #2, which were not elements of the copied partition: these were elements 5 and 8. These elements are erased,

$$p_2'' = (1121 * 32 * 2233),$$

and replaced (randomly) by numbers of other partitions, which were overwritten in the copying step. Thus the final offspring might be

$$p_2''' = (112133212233).$$

As mentioned earlier, in the group-number coding, two identical partitions may be represented by different strings due to different numbering of partitions. To take care of this, before the crossover is executed, the codings are adapted to minimize the difference between the two parents.

structural mutation: typically, a mutation would replace a single component of a string by some random number; however, this would destroy the requirement of the equal sizes of partitions. Structural mutation was defined as a swap of two numbers in the string. Thus a parent

[5]The additional requirement in this version of the partitioning problem was that the sizes of the partitions are equal or nearly equal.

$p = (112133212233)$

may produce the following offspring (the numbers on positions 4 and 6 are swapped):

$p' = (112331212233)$.

The algorithm was implemented as a parallel genetic algorithm enhanced with additional strategies (such as a parent replacement strategy); on random graphs of 900 nodes with maximum node degree of 4, this evolution program significantly outperformed other heuristic algorithms [114].

12. Machine Learning

The real problem is not
whether machines think,
but whether men do.

B.F. Skinner, *Contingencies of Reinforcement*

Machine learning is primarily devoted towards building computer programs able to construct new knowledge or to improve already possessed knowledge by using input information; much of this research employs heuristic approaches to learning rather than algorithmic ones. The most active research area in recent years [140] has continued to be symbolic empirical learning (SEL). This area is concerned with creating and/or modifying general symbolic descriptions, whose structure is unknown *a priori*. The most common topic in SEL is developing concept descriptions from concept examples [113], [140]. In particular, the problems in attribute-based spaces are of practical importance: in many such domains it is relatively easy to come up with a set of example events, on the other hand it is quite difficult to formulate hypotheses. The goal of a system implementing this kind of supervised learning is:

> *Given the initial set of example events and their membership in concepts, produce classification rules for the concepts present in the input set.*

Depending on the output language, we can divide all approaches to automatic knowledge acquisition into two categories: symbolic and non-symbolic. Non-symbolic systems do not represent knowledge explicitly. For example, in statistical models knowledge is represented as a set of examples together with some statistics on them; in a connectionist model, knowledge is distributed among network connections [154]. On the other hand, symbolic systems produce and maintain explicit knowledge in a high-level descriptive language. The best known examples of this category of system are AQ and ID families [137] and [147].

In this chapter we describe two genetic-based machine learning methodologies and discuss an evolution program (GIL, for Genetic Inductive Learning), proposed in [97]. The first two approaches fall somewhere between symbolic

and non-symbolic systems: they use (to some extent) a high-level descriptive language, on the other hand, the operators used are not defined in that language and operate at the non-symbolic level. Even in some recent problem-oriented representations ([68],[172],[156], [47]) the operators still operate at the conservative traditional subsymbolic level. The third system GIL is an evolution program tailored to "learning from examples"; the system incorporates the problem-specific knowledge in data structures and operators.

Let us consider an example, which will be used throughout the chapter.

Example 12.1. This is taken from the world of Emerald's robots (see [105] and [198]). Each robot is described by the values of six attributes; the attributes with their domains are:

Attributes:	**Values of Attributes:**
Head_**S**hape	**R**ound, **S**quare, **O**ctagon
Body_Shape	**R**ound, **S**quare, **O**ctagon
Is_Smiling	**Y**es, **N**o
Holding	**S**word, **B**alloon, **F**lag
Jacket_Color	**R**ed, **Y**ellow, **G**reen, **B**lue
Has_**T**ie	**Y**es, **N**o

The boldface letters are used to identify attributes and their values, e.g., (**J** = **Y**) means "Jacket_Color is Yellow". The examples of concepts descriptions (where each concept C_i is described in terms of these six attributes and their values) are:

C_1 Head is round and jacket is red, or head is square and is holding a balloon
C_2 Smiling and holding balloon, or head is round
C_3 Smiling and not holding sword
C_4 Jacket is red and is wearing no tie, or head is round and is smiling
C_5 Smiling and holding balloon or sword

□

Attributes are of three types: nominal (their domains are sets of values), linear (their domains are linearly ordered), and structured (their domains are partially ordered). Events represent different decision classes: events from a particular class constitute its positive examples, all other events its negative examples. Learning examples are given in the form of events, and each event is a vector of attribute values.

The concept descriptions are represented in VL_1 (simplified version of the Variable Valued Logic System) [137] — a widely accepted language to represent input events for any program operating in an attribute–based space.

A description of a concept C is a disjunction of complexes

$$c_1 \vee \ldots \vee c_k \Rightarrow C;$$

each complex (c_i) is expressed as a conjunction of selectors, which are (attribute relation set_of_values) triplets (e.g., $\langle J = R \rangle$ for "Jacket_Color is red").

The concepts C_1 – C_5 can be expressed as

$$\langle S = R \rangle \wedge \langle J = R \rangle \vee \langle S = S \rangle \wedge \langle H = B \rangle \Rightarrow C_1$$
$$\langle I = Y \rangle \wedge \langle H = B \rangle \vee \langle S = R \rangle \Rightarrow C_2$$
$$\langle I = Y \rangle \wedge \langle H \neq S \rangle \Rightarrow C_3$$
$$\langle J = R \rangle \wedge \langle T \neq Y \rangle \vee \langle S = R \rangle \wedge \langle I = Y \rangle \Rightarrow C_4$$
$$\langle I = Y \rangle \wedge \langle H = \{B, S\} \rangle \Rightarrow C_5.$$

Note that the selector $\langle T \neq Y \rangle$ can be interpreted as $\langle T = N \rangle$, since the attribute T is a Boolean (nominal) attribute, and the selector $\langle H = \{B, S\} \rangle$ is interpreted as "Holding Balloon or Sword" (internal disjunction).

The problem is to construct a system to learn the concepts, i.e., to determine decision rules that account for all positive examples and no negative ones. We can evaluate and compare systems on all robot descriptions in terms of error rates and complexities of generated rules. The system should be able to predict a classification of previously unseen examples, or suggest (possibly more than one) classifications of partially specified descriptions.

During the past two decades there has been a growing interest in applying evolution programming techniques to machine learning (GBML systems, for genetics based machine learning systems) [48]. This was due to the attractive idea that chromosomes, representing knowledge, are treated as data to be manipulated by genetic operators, and, at the same time, as executable code to be used in performing some task. However, early applications, although partially successful (e.g.,[152],[156],[157]), also encountered many problems [45]. In general, in the GA community there are two competing approaches to address the problem. As stated by De Jong [45]:

> "To anyone who has read Holland [92], a natural way to proceed is to represent an entire rule set as a string (an individual), maintain a population of candidate rule sets, and use selection and genetic operators to produce new generations of rule sets. Historically, this was the approach taken by De Jong and his students while at the University of Pittsburgh (e.g., see Smith [172], [173]), which gave rise to the phrase 'the Pitt approach'.
>
> However, during the same time period, Holland developed a model of cognition (classifier systems) in which the members of the population are individual rules and a rule set is represented by the entire population (e.g., see Holland and Reitman, [90]; Booker [20]). This quickly became known as 'the Michigan approach' and initiated a friendly but provocative series of discussions concerning the strengths and weaknesses of the two approaches."

We believe that a third approach based on evolution programming techniques should be the most fruitful one. The idea of incorporating problem-specific knowledge, as usual by (1) careful design of appropriate data structures,

and (2) problem-specific "genetic" operators, must pay off in system precision and performance. However, the Pitt approach is closer to our idea of evolution programming, since it maintains a population of complete solutions (set of rules) to the problem, whereas the Michigan approach (classifier systems), through bidding, the bucket brigade algorithm, and a genetic component in modifying rules, establishes a new methodology very different to evolution programming technique. On the other hand, implementations of the Pitt approach, even when they represent a chromosome in a high-level description language, do not use any learning methodology to modify their operators. This is the basic difference between the Pitt approach and the evolution program approach.

In the sequel, we discuss the basic principles behind the classifier systems (the Michigan approach, Section 12.1), the Pitt approach (Section 12.2), and Janikow's [97] evolution program (Section 12.3) for inductive learning of decision rules in attribute-based examples.

12.1 The Michigan approach

Classifier systems are a kind of rule-based system with general mechanisms for processing rules in parallel, for adaptive generation of new rules, and for testing the effectiveness of existing rules. Classifier systems provide a framework in which a population of rules encoded as bit strings evolves on the basis of intermittently given stimuli and reinforcement from its environment. The system "learns" which responses are appropriate when a stimulus is presented. The rules in a classifier system form a population of individuals evolving over time.

A classifier system (see Figure 12.1) consists of the following components:

- detector and effector,
- message system (input, output, and internal message lists),
- rule system (population of classifiers),
- apportionment of credit system (bucket brigade algorithm), and
- genetic procedure (reproduction of classifiers).

The environment sends a message (a move on a board, an example of a new event, etc.), which is accepted by the classifier system's detectors and placed on the input message list. The detectors decode the message into one or more (decoded) messages and place them on the (internal) message list. The messages activate classifiers; strong, activated classifiers place messages on the message list. These new messages may activate other classifiers or they can send some messages to the output message list. In the latter case, the classifier system's effectors code these messages into an output message (a move on a board, decision, etc.), which is returned to the environment. The environment evaluates the action of the system (environment feedback), and the bucket brigade algorithm updates the strengths of the classifiers.

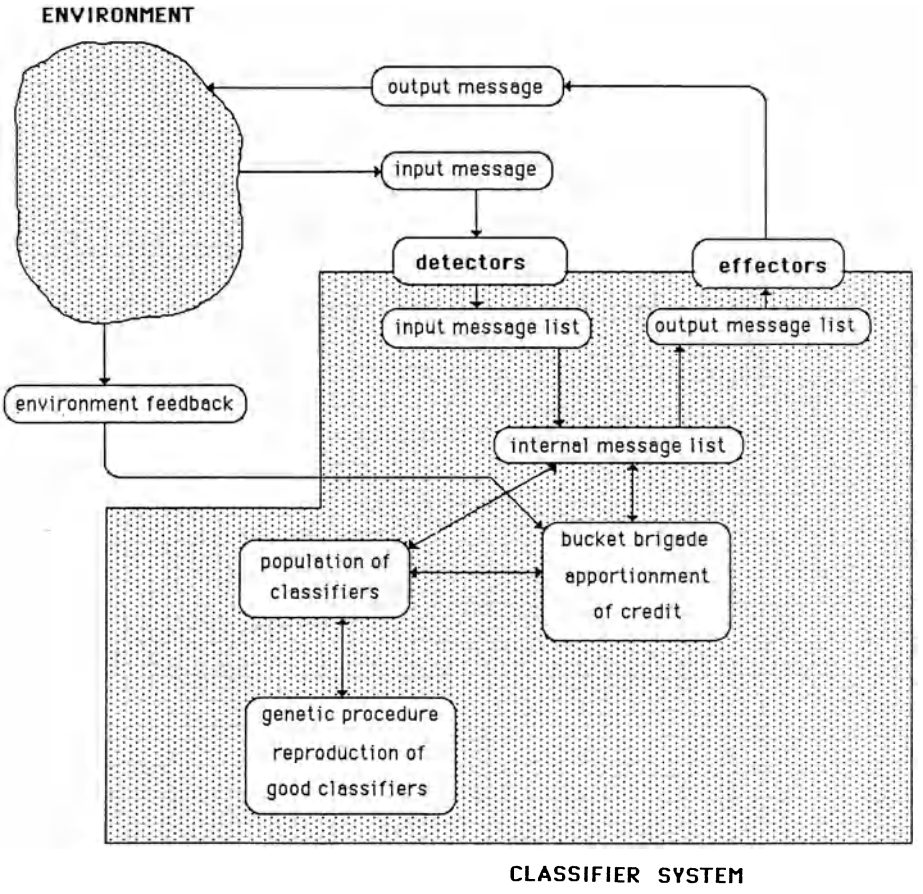

Fig. 12.1. A classifier system and its environment

We discuss some of these actions in more detail using the world of Emerald's robots as example. Let us provide some basic preliminaries first. A single classifier consists of two parts: (1) a condition part, and (2) a message. The condition part is a finite-length string over some alphabet; the alphabet includes the "don't care" symbol, "*". The message part is a finite length string over the same alphabet; however, it does not contain the "don't care" symbol. Each classifier has its own strength. The strength is important in the bidding process, where classifiers compete to post their messages — we discuss it later in this section. Now, we return to the world of Emerald's robots. We can express a decision rule as one or more classifiers. Each classifier has a form

$$(p_1, p_2, p_3, p_4, p_5, p_6) : d,$$

where p_i denotes the value of the i-th attribute ($1 \leq i \leq 6$) for the domains described above (e.g., $p_4 \in \{S, B, F, *\}$ (Holding)) and $d \in \{C_1, C_2, C_3, C_4, C_5\}$.

For example, the classifier

$$(R * * * R*) : C_1$$

represents the following: "If head is round and jacket is red then concept C_1".

The set of classifiers for five concepts given in this chapter is:

$$\begin{array}{lcl}
(R * * * R*) & : & C_1 \\
(S * *B * *) & : & C_1 \\
(* * YB * *) & : & C_2 \\
(R * * * **) & : & C_2 \\
(* * YB * *) & : & C_3 \\
(* * YF * *) & : & C_3 \\
(* * * * RN) & : & C_4 \\
(R * Y * **) & : & C_4 \\
(* * YB * *) & : & C_5 \\
(* * YS * *) & : & C_5.
\end{array}$$

To simplify the example, we assume the system learns a single concept C_1; any system can be easily generalized to handle multiple concepts. In that case each classifier has a form

$$(p_1, p_2, p_3, p_4, p_5, p_6) : d,$$

where $d = 1$ (membership to the concept C_1) or $d = 0$ (otherwise).

Let us assume that at some stage of the learning process there is a small (random) population of classifiers q in the system (each classifier is given with its strength s):

$$\begin{array}{lcll}
q_1 = (* * *SR *) & : & 1, & s_1 = 12.3, \\
q_2 = (* * Y * *N) & : & 0, & s_2 = 10.1, \\
q_3 = (SR * * * *) & : & 1, & s_3 = 8.7, \\
q_4 = (*O * * * *) & : & 0, & s_4 = 2.3.
\end{array}$$

Assume further that an input message arrives from the environment (a new example event):

$$(ROYSRN)$$

It is a description of a single robot with round (R) head, round (R) body, smiling (Y), holding sword (S), in red (R) jacket and without tie (N). Obviously, this robot constitutes the positive example of the concept C_1 due to its round head and red jacket.

Three classifiers are activated by this message: q_1, q_2, and q_4. These classifiers bid; each bid is proportional to the classifier's strength ($bid\_i = b \cdot s_i$). The strongest classifier q_1 wins and posts its message; since the message is decoded into the correct classification, the classifier gets a reward $r > 0$; the strength of the classifier becomes

$$s_1 := s_1 - bid_1 + r$$

(if the message was decoded into wrong answer, the "reward" r would be negative). For coefficients $b = 0.2$ and $r = 4.0$, the new strength of the classifier q_1 is $s_1 = 12.3 - 2.46 + 4.0 = 13.84$.

One of the parameters of the classifier system is the GA period, t_{ga}, which specifies the number of time steps (number of cycles just described above) between GA calls. Of course, t_{ga} can be a constant, it can be generated randomly (with the average equal to t_{ga}), or it need not be even specified and the decision of invocation of GA can be made of the basis of the performance of the system. Anyway, let us assume that the time has come to apply the genetic algorithm to the classifiers.

We consider the strengths of the classifiers as their fitness — a selection process can be performed using roulette wheel selection (Chapter 2). However, in the classifier systems we are no longer interested in the strongest (the most fit) classifier, but rather in a whole population of classifiers that perform the classification task. This implies that we should not generate the whole population and that we should be careful in selecting individuals for replacement. Usually, the *crowding factor model* (Chapter 4) is used, since it replaces similar population members.

The operators used are, again, mutation and crossover. However, some modifications are necessary. Let us consider the first attribute, Head_Shape — its domain is the set $\{R, S, O, *\}$. Thus, when mutation is called, we would change the mutated character to one of the other three characters (with equal probability):

$$\begin{aligned} R &\rightarrow \{S, O, *\}, \\ S &\rightarrow \{R, O, *\}, \\ O &\rightarrow \{R, S, *\}, \\ * &\rightarrow \{R, S, O\}. \end{aligned}$$

The strength of the offspring usually is the same as its parent.

The crossover does not require any modification. We take advantage of the fact that all classifiers are of equal length; to crossover two selected parents, say q_1 and q_2:

$$(***|SR*) \ : \ 1, \text{ and } (**Y|**N) \ : \ 0,$$

we generate a random crossover position point (say, we crossover after the third character, as marked), and the offspring are

$$(*****N) \ : \ 0, \text{ and } (**YSR*) \ : \ 1.$$

The strengths of the offspring are a (possibly weighted) average of those of the parents.

Now, the classifier system is ready to continue its learning process and start another cycle of t_{ga} steps, accepting further positive and negative events as examples, modifying the strengths of classifiers. We hope that finally the population of classifiers converges to some number of strong individuals, e.g.,

$$\begin{array}{rcl} (R***R*) & : & 1, \\ (S**B**) & : & 1, \\ (O*****) & : & 0, \\ (***SY*) & : & 0. \end{array}$$

The example discussed above was very simple: we aimed at explaining the basic components of the classifier system using a learning paradigm for the world of Emerald's robots. Note, however, that we used the simplest bidding system (e.g., we did not use *effective bid* variable $e\_bid = bid + N(0, \sigma_{bid})$, which introduces some random noise with standard deviation σ_{bid} and expected value zero), that we did not use any taxation system (usually each classifier is taxed to prevent biasing the population towards productive rules), and that we select only a single winner (the strongest classifier) at each step — in general, more then one classifier can be a winner, placing its message on the message list. Moreover, the bucket brigade algorithm was not used in the sense that the reward was available at every time step (for every provided example). There was no relationship between examples and there was no need to trace a chain of rewards to apportion credit to the classifiers whose messages activated the current (winner) classifier. For some problems, like planning problems, the length of the message part is the same as the length of the condition part. In such cases, an activated classifier (old winner) would place its message on the (internal) message list, which, in turn, may activate some other classifiers (new winners). Then the strength of the old winners is increased by a reward — a payment from the new winners.

For a full discussion on different versions of the classifier systems and the historical perspective, the reader is referred to [72].

12.2 The Pitt approach

The Michigan approach can be perceived as a computational model of cognition: the knowledge of a cognitive entity is expressed as a collection of rules which undergo modifications over a time. We can evaluate the whole cognitive unit in terms of its interaction with the environment; an evaluation of a single rule (i.e., a single individual) is meaningless.

On the other hand, the Pitt approach adopts the view that each individual in a population represents the whole set of rules (a separate cognitive entity).

These individuals compete among themselves, the weak individuals die, the strong survive and reproduce: this is done by means of natural selection proportional to the their fitness, crossover, and mutation operators. In short, the Pitt approach applies the genetic algorithm to the learning problem. By doing this, Pitt approach avoids the delicate credit assignment problem, for which a heuristic method (such as the bucket brigade algorithm) should distribute (positive or negative) credit among the rules which cooperated in produced (desirable or undesirable) behavior.

However, there are some interesting issues to be addressed. The first one is representation. Should we use fixed-length binary vectors (with a fixed field format) to represent set of rules? Such a representation would be ideal for generating new rules, since we can use classical crossover and mutation for that purpose. However, it seems to be too restrictive and appropriate only for systems which work at a lower sensory level. What about genes in a chromosome representing values of attributes (i.e., number of genes equal to number of attributes)? This decision (not supported by specialized operators) does not seem to work: if the cardinalities of some domains are large, the probability that the system would converge prematurely is very high [45], even with much higher mutation rates than usual. It seems that some internal representational structure is necessary to provide punctuation marks between units in a chromosome, together with operators which are aware of chromosomal representation. Such an approach was implemented and discussed in [173] and [156].

Smith [172] went even further and experimented with variable-length individuals. He generalized many results previously valid only for GAs with fixed length strings to apply to GAs with variable length strings.

However, it seems that some other bold decisions are necessary to address the complex issues of representation and operators. Only recently [12] was the need for such decisions recognized:

> "A solution to this problem is to select different genetic operators that are more appropriate to 'natural representation'. There is nothing sacred about the traditional string oriented genetic operators. The mathematical analysis of GAs shows that they work best when the internal representation encourages the emergence of useful building blocks that can be subsequently combined with other to produce improved performance. String representations are just one of many ways of achieving this."

In the next section, we present an evolution program (based on the Pitt approach), which does just this: the representation is rich and natural, with specialized, representation sensitive operators, taken directly from a learning methodology.

12.3 An evolution program: the GIL system

The implemented evolution program GIL [97] moves the genetic algorithm (the Pitt approach) closer to the symbolic level — mainly by defining specialized operators manipulating at the problem level. Again, the basic components of GIL, like any other evolution program, are data structures and genetic operators. The system was designed for learning single concepts only. However, it can be easily extended to learn in multi-concept environments by introducing multiple populations.

12.3.1 Data structures

One chromosome represents a solution to the concept description being learned; the assumption is that any event not covered by such a description belongs to the negation of the concept (does not belong to the concept). Each chromosome is a set (disjunction) of a number of complexes. The number of complexes in a chromosome can vary — the assumption that all chromosomes are of equal length (as GAs maintain populations of fixed-length strings) would be, to say the least, artificial, so it would be contrary to evolution programming technique. This decision, however, is not new in the GA community: as described in the previous section, Smith [172] has extended many formal results on genetic algorithms to variable-length strings and implemented a system that maintained a population of such (variable-length) strings.

Each complex, as defined in VL_1, is a conjunction of a number of selectors corresponding to different attributes. Each selector, in turn, is the internal disjunction from the domain of its attribute. For example, the following might be a chromosome (and therefore a description of the concept C_1):

$$(\langle S = R\rangle \wedge \langle J = R\rangle) \vee (\langle S = S\rangle \wedge \langle H = B\rangle)$$

Mostly for efficiency reasons, binary representation for selectors was used for internal representation of chromosomes. A binary 1 at position i implies the inclusion of the i-th domain value in this selector. This means that the size of the domain of an attribute is equal to the length of the binary substring corresponding to this attribute. Note that a collection of all 1s for some selector is equivalent to the *don't care* symbol for this attribute. Thus the chromosome describing concept C_1 in a notation similar to one used for classifier systems is represented as

$$\langle R***R* \vee S**B**\rangle,$$

whereas its internal representation in the GIL system is

$$\langle 100|111|11|111|1000|11 \vee 010|111|11|010|1111|11\rangle,$$

where bars separate selectors. Note that such a representation handles internal disjunction gracefully, e.g., the concept C_5

$$\langle I = Y \rangle \wedge \langle H = \{B, S\} \rangle,$$

can be represented as

$$\langle 111|111|10|110|1111|11 \rangle.$$

12.3.2 Genetic operators

The operators of the GIL system are modeled on the methodology of inductive learning provided by Michalski [136]; the methodology describes various inductive operators that constitute the process of inductive inference. These include: *condition dropping*, e.g., dropping a selector from the concept description, *adding alternative rule* and *dropping a rule*, *extending reference* — extending an internal disjunction, *closing interval* — for linear domains filling up missing values between two present values, and *climbing generalization* — for structured domains climbing the generalization tree.

The GIL system defines inductive operators separately on three abstract levels: chromosome level, complex level, and selector level. We discuss some of them in turn.

Chromosome level: the operators act on the whole chromosomes:

- *RuleExchange:* the operator is similar to a crossover of the classical GA, as it exchanges selected complexes between two parent chromosomes. For example, two parents

  $\langle 100|111|11|111|1000|11 \vee 010|111|11|010|1111|11 \rangle$ and
  $\langle 111|001|01|111|1111|01 \vee 110|100|10|111|0010|01 \rangle$

  may produce the following offspring:

  $\langle 100|111|11|111|1000|11 \vee 111|001|01|111|1111|01 \rangle$ and
  $\langle 010|111|11|010|1111|11 \vee 110|100|10|111|0010|01 \rangle$.

- *RuleCopy:* the operator is similar to *RuleExchange*; however, it copies random complexes from one parent to another. For example, two parents

  $\langle 100|111|11|111|1000|11 \vee 010|111|11|010|1111|11 \rangle$ and
  $\langle 111|001|01|111|1111|01 \vee 110|100|10|111|0010|01 \rangle$

  may produce the following offspring:

  $\langle 100|111|11|111|1000|11 \vee 111|001|01|111|1111|01 \vee 110|100|10|111|0010|01 \rangle$
  and
  $\langle 010|111|11|010|1111|11 \rangle$.

- *NewPEvent:* this unary operator incorporates a description of a positive event into the selected chromosome. For example, for a parent

  $\langle 100|111|11|111|1000|11 \vee 010|111|11|010|1111|11 \rangle$

  and an uncovered event

⟨100|010|10|010|0010|01⟩,

the following offspring is produced:

⟨100|111|11|111|1000|11∨010|111|11|010|1111|11∨100|010|10|010|0010|(

- *RuleGeneralization:* this unary operator generalizes a random subset of complexes. For example, for a parent

  ⟨100|111|11|111|1000|11∨010|111|11|010|1111|11∨100|010|10|010|0010|(

  and the second and third complexes selected for generalization, the following offspring is produced:

  ⟨100|111|11|111|1000|11 ∨ 110|111|11|010|1111|11⟩.

- *RuleDrop:* this unary operator drops a random subset of complexes. For example, for a parent

  ⟨100|111|11|111|1000|11∨010|111|11|010|1111|11∨100|010|10|010|0010|(

  the following offspring might be produced:

  ⟨100|111|11|111|1000|11⟩.

- *RuleSpecialization:* this unary operator specializes a random subset of complexes. For example, for a parent

  ⟨100|111|11|111|1000|11∨010|111|11|010|1111|10∨111|010|10|010|1111|

  and the second and third complexes selected for specialization, the following offspring is produced:

  ⟨100|111|11|111|1000|11 ∨ 010|010|10|010|1111|10⟩.

Complex level: the operators act on complexes of the chromosomes:

- *RuleSplit:* this operator acts on a single complex splitting it into a number of complexes. For example, a parent

  ⟨100|111|11|111|1000|11⟩

  may produce the following offspring (the operator splits the second selector):

  ⟨100|011|11|111|1000|11 ∨ 100|100|11|111|1000|11⟩.

- *SelectorDrop:* this operator acts on a single complex and "drops" a single selector, i.e., all values for selected selector are replaced by a string '11...1'. For example, a parent

  ⟨100|010|11|111|1000|11⟩

  may produce the following offspring (the operator works on the fifth selector):

  ⟨100|010|11|111|1111|11⟩.

- *IntroSelector:* this operator acts by "adding" a complex, i.e., it eliminates a selector with a string '11...1'. For example, a parent

  ⟨100|010|11|111|1111|11⟩

may produce the following offspring (the operator works on the fifth selector):

⟨100|010|11|111|0001|11⟩.

- *NewNEvent:* this unary operator incorporates a description of a negative event to the selected chromosome. For example, for a parent

  ⟨110|010|11|111|1111|11⟩

  and the covered negative event

  (100|010|10|010|0100|10),

  the following offspring is produced:

  ⟨010|010|11|111|1111|11∨110|010|01|111|1111|11∨110|010|11|101|1111|11∨
  110|010|11|111|1011|11∨110|010|11|111|1111|01⟩.

Selector level: the operators act on selectors:

- *ReferenceChange:* the operator adds or removes a single value (0 or 1) from the chromosome, i.e., from the domain of one of the selectors. For example, a parent

  ⟨100|010|11|111|0001|11⟩

  may produce the following offspring (note the difference in the fourth selector):

  ⟨100|010|11|110|0001|11⟩

- *ReferenceExtension:* the operator extends the domain of a selector by allowing a number of additional values. For different types of attributes (nominal, linear, structured) it uses different probabilities of selecting values. For example, a parent

  ⟨100|010|11|111|1010|11⟩

  may produce the following offspring (the operator "closes" the domain of the fifth selector):

  ⟨100|010|11|111|1110|11⟩.

- *ReferenceRestriction:* this operator removes some domain values from a selector. For example, a parent

  ⟨100|010|11|111|1011|11⟩

  may produce the following offspring:

  ⟨100|010|11|111|1000|11⟩.

The GIL system is quite complex and requires a number of parameters (e.g., the probabilities of applying these operators). For a discussion of these and other implementational issues, the reader is referred to [97]. It is interesting to note, however, that the operators are given a priori probabilities, but the actual probabilities are computed as a function of these probabilities and

two other parameters: an (a priori) desired balance between specialization and generalization, and a (dynamic) measure of the current coverage. This idea is similar to one discussed earlier (last section of Chapter 8).

At each iteration all chromosomes are evaluated with respect to their completeness and consistency (and possibly cost if desired), and a new population is formed with those better ones more likely to appear. Then, the operators are applied to the new population, and the cycle repeats.

12.4 Comparison

A recent publication [198] provides an evaluation of a number of learning strategies, including a classifier system (CFS), a neural net (BpNet), a decision tree learning program (C4.5), and a rule learning program (AQ15). The systems used examples from the world of Emerald robots; the systems were supposed to learn five concepts (C_1 - C_5) given at the beginning of this chapter, while seeing only a varying percentage of the positive and negative examples (there were a total of 432 different robots present, i.e., all possible combinations of the attribute values were there). The systems were compared by providing an average error in recognizing all of the 432 (seen and unseen) robots; the results (from [198]) are given in Table 12.1.

System	Learning Scenario (Positive % / Negative %)				
	6% / 3%	10% / 10%	15% / 10%	25% / 10%	100% / 10%
AQ15	22.8%	5.0%	4.8%	1.2%	0.0%
BpNet	9.7%	6.3%	4.7%	7.8%	4.8%
C4.5	9.7%	8.3%	11.3%	2.5%	1.6%
CFS	21.3%	20.3%	21.5%	19.7%	23.0%

Table 12.1. Summary of the error rates for different systems

Table 12.2 provides a recognition rate for the GIL system on the individual concepts basis (from [97]). As expected, the evolution program GIL performs much better than system CFS based on the classifier system approach; surprisingly, GIL outperformed other learning systems as well. Its superiority is most visible in the cases with a small percentage of seen and unseen examples.

Concept	Learning Scenario (Positive % / Negative %)				
	6% / 3%	10% / 10%	15% / 10%	25% / 10%	100% / 10%
C_1	11.1%	5.3%	0.0%	0.0%	0.0%
C_2	0.0%	0.0%	0.0%	0.0%	0.0%
C_3	0.0%	0.0%	0.0%	0.0%	0.0%
C_4	10.4%	0.0%	0.0%	0.0%	0.0%
C_5	0.0%	0.0%	0.0%	0.0%	0.0%

Table 12.2. Summary of the error rates for the evolution program GIL

For a further discussion on comparison of these systems (e.g., complexity of the generated rules), implementational issues of the GIL system, and results of the other experiments (multiplexers, breast cancer, etc.), the reader is referred to [97].

Conclusions

A disciple was one day recalling how
Buddha, Jesus, and Mohammed were branded
as rebels and heretics by their contemporaries.
Said the Master, 'Nobody can be said
to have attained the pinnacle of Truth
until a thousand sincere people have
denounced him for blasphemy.'

Anthony de Mello, *One Minute Wisdom*

In this book we discussed different strategies, called Evolution Programs, which might be applied to hard optimization problems and which were based on the principle of evolution. Evolution programs borrow heavily from genetic algorithms. However, they incorporate problem-specific knowledge by using "natural" data structures and problem-sensitive "genetic" operators. The basic difference between GAs and EPs is that the former are classified as weak, problem-independent methods, which is not the case for the latter.

The boundary between weak and strong methods is not well defined. Different evolution programs can be built which display a varying degree of problem dependence. For a particular problem P, in general, it is possible to construct a family of evolution programs EP_i, each of which would 'solve' the problem (Figure 13.1). The term 'solve' means 'provide a reasonable solution', i.e., a solution which need not, of course, be optimal, but is feasible (it satisfies problem constraints).

The evolution program EP_5 (Figure 13.1) is the most problem specific and it addresses the problem P only. The system EP_5 will not work for any modified version of the problem (e.g., after adding a new constraint or after changing the size of the problem). The next evolution program, EP_4, can be applied to some (relatively small) class of problems, which includes the problem P; other evolution programs EP_3 and EP_2 work on larger domains, whereas EP_1 is domain independent and can be applied to any optimization problem.

We have already seen a part of such a hierarchy in various places of the book. Let us consider a particular 20×20 nonlinear transportation problem, P. There are 400 variables with $20 + 20 = 40$ equations (of which 39 are independent). Additional constraints require that the variables must take nonnegative values. In principle, it is possible to construct an evolution program, say EP_5, which would solve this particular problem. It might be a genetic algorithm with 39 penalty functions tuned very carefully for these constraints or with decoders and/or repair algorithms. Any change in the size of the problem (moving from 20×20 to 20×21) or any change in a transportation cost from one source to one destination would result in a failure of the EP_5 system.

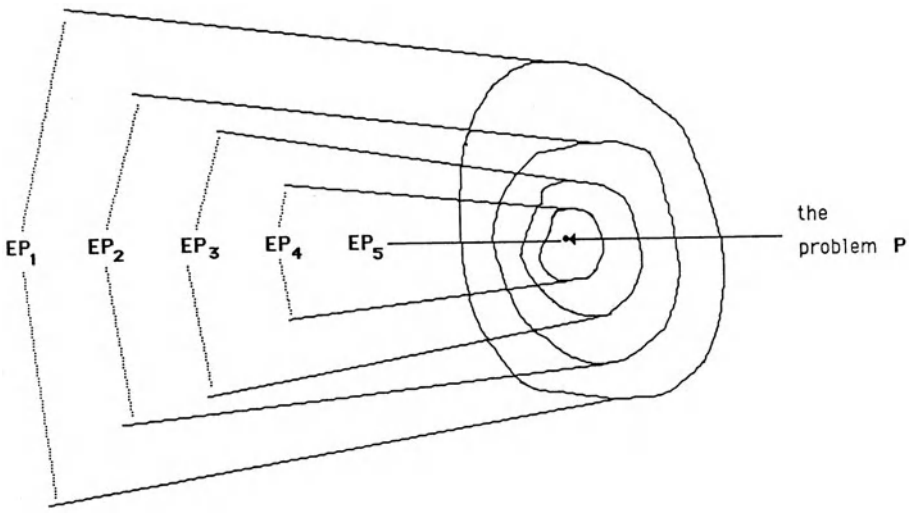

Fig. 13.1. A hierarchy of evolution programs

There is also an evolution program, GENETIC-2 (Chapter 9), which can be applied to any transportation problem. Let us call this system EP_4. This system still belongs to the class of strong methods, since it can be applied only for nonlinear transportation problems. However, it is much weaker than EP_5, since it can handle any transportation problem.

Another evolution program, say EP_3, applicable to the problem P, is GENOCOP (Chapter 7). It optimizes any function with the presence of any set of linear constraints, which is the case for the transportation problem, P. Obviously, EP_3 is a weaker method than EP_4. However, it can still be considered as a relatively strong method, since it can be applied only to numerical optimization problems with linear constraints.

Yet another evolution program (let us call it EP_2) can be applied to our 20×20 transportation problem, P: an evolution strategy (Chapter 8). Evolution strategies can be applied to any numerical optimization problem with (not necessarily linear) inequality constraints. Clearly, the problem P belongs to the domain of EP_2; also, EP_2 is a weaker method than EP_3, since it handles any type of inequalities (for the problem P, equalities can be easily replaced by inequalities using the method discussed in Chapter 7).

We can also construct a general purpose evolution program, EP_1, which might be just a classical genetic algorithm with a standard set of penalty functions; each penalty function would correspond to one of the problem's constraints. The system EP_1 is domain independent — it can approach any optimization problem with any set of constraints. For numerical optimization problems, the constraints may include nonlinear equalities, which makes this system

a weaker method than EP_2, which is restricted to inequalities only. Moreover, EP_1 can be applied also to other (non-numerical) problems as well. Here we assume that the program EP_1 always return a feasible solution. We can enforce it easily, if the initial population consists of feasible solutions and penalty functions, decoders, or if repair algorithms keep individuals within the search space.

Let us denote by $dom(EP_i)$ a set of all problems to which the evolution program EP_i can by applied, i.e., the program returns a feasible solution. Clearly,

$$dom(EP_5) \subseteq dom(EP_4) \subseteq dom(EP_3) \subseteq dom(EP_2) \subseteq dom(EP_1).$$

Obviously, the above example is by no means complete: it is possible to create other evolution programs which would fit between EP_i and EP_{i+1} for some $1 \leq i \leq 4$. Of course, there might be also other evolution programs which overlap with others in the above hierarchy. For example, we can build systems to optimize transportation problems with the cost functions restricted to polynomials, or to optimize problems restricted to a convex search space, or problems with constraints in the form of nonlinear equations. The GAFOC system for control problems (Chapter 7) can be considered as a stronger method than GENOCOP. However, it is not comparable with GENETIC-2. In other words, the set of evolution programs is partially ordered; we denote the ordering relation by $\prec$ with the following meaning: if $EP_p \prec EP_q$ then the evolution program EP_p is a weaker method than EP_q, i.e., $dom(EP_q) \subseteq dom(EP_p)$. Referring to our example of a transportation problem, P, and a hierarchy of evolution programs, EP_i,:

$$EP_1 \prec EP_2 \prec EP_3 \prec EP_4 \prec EP_5.$$

The hypothesis is that if $EP_p \prec EP_q$, then the stronger method, EP_q, should in general perform better than a weaker system, EP_p. We do not have any proof of this hypothesis, of course, since it is based solely on a number of experiments and the simple intuition that problem-specific knowledge enhances an algorithm in terms of performance (time and precision) and at the same time narrows its applicability. We have already seen the better performance of GENETIC-2 against GENOCOP, and we discussed how GENOCOP outperforms classical GA on a particular class of problems. If this hypothesis is true, GENOCOP should give better results than an evolution program based on evolution strategy for problems with linear constraints, since ES is a weaker method than GENOCOP (however, we did not complete such experiments due to the fact that the full vesion of the GENOCOP system is still under construction). Some other researchers support the above hypothesis; let us cite from Davis [37]:

> "It is a truism in the expert system field that domain knowledge leads to increased performance in optimization, and this truism has certainly been borne out of my experience applying genetic algorithms to industrial problems. Binary crossover and binary mutation

are knowledge-blind operators. Hence, if we resist adding knowledge to our genetic algorithms, they are likely to underperform nearly any reasonable optimization algorithm that does take into account of such domain knowledge."

Goldberg [74], [72], provides an additional perspective. Let us quote from [74]:

"Certainly humans have developed very efficient search procedures for narrow classes of problems — genetic algorithms are unlikely to beat conjugate direction or gradient methods on continuous, quadratic optimization problems — but this misses the point. [...] The breadth combined with relative — if not peak — efficiency defines the primary theme of genetic search: robustness."

We visualized this observation on Figure 13.2, where a (classical) method, Q, works well for a problem, P, and nowhere else, whereas GAs perform reasonably across the spectrum. (The Figure 13.2 is a simplification of similar figures given in [74] and [72].)

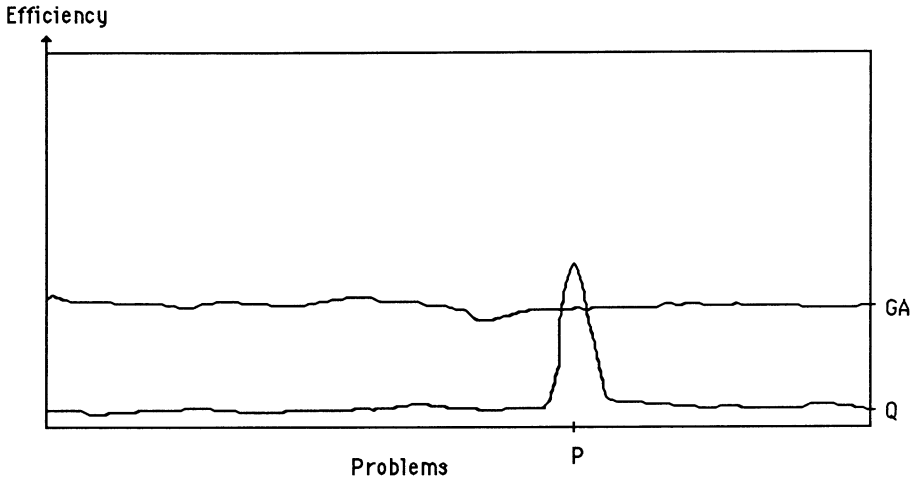

Fig. 13.2. Efficiency/problem spectrum and GAs

However, in the presence of nontrivial, hard constraints, the performance of GAs deteriorates quite often. On the other hand, evolution programs, by incorporating some problem-specific knowledge, may outperform even classical methods (Figure 13.3).

We should emphasise, again, that most evolution programs presented in the book do not have theoretical support. There is neither a Schema Theorem (as for classical genetic algorithms) nor convergence theorems (as for evolution strategies). It is also important to underline that evolution programs are generally much slower than other optimization techniques. On the other hand, their

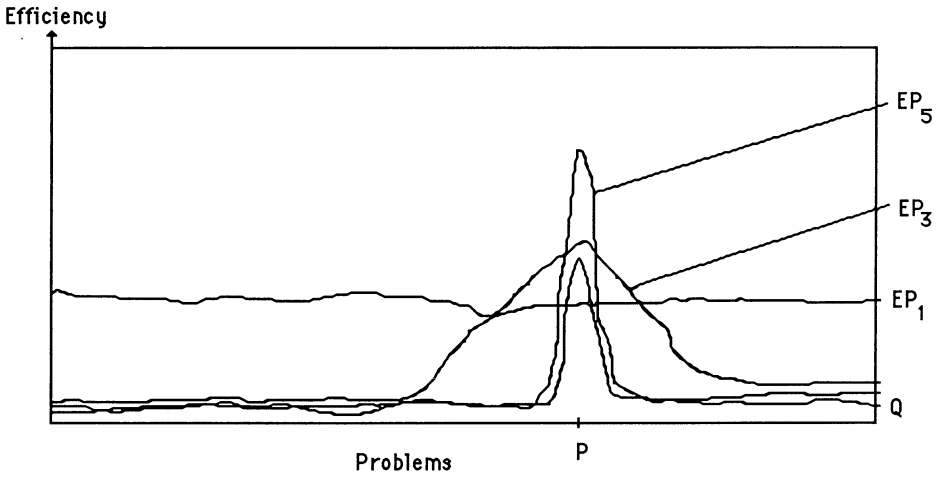

Fig. 13.3. Efficiency/problem spectrum and EPs

time complexity quite often grows in a linear manner together with the problem size — which is not the case for most of the other techniques.

After these introductory remarks, we are ready to address two practical issues connected with evolution programming: For a given problem, P,

(1) how weak (or strong) should an evolution program be?

(2) how should we proceed to construct an evolution program?

There are no easy answers for these questions. In the sequel, we provide some general comments and intuition developed on the basis of various experiments, mixed with a dose of wishful thinking.

The first question is on optimizing the selection of an evolution program to be constructed. For a given problem, P, should we construct EP_2, or rather EP_4? Our hypothesis suggests that incorporation of problem-specific knowledge gives better results in terms of computational time and precision. On the other hand, we may have already some standard packages, like Grefenstette's GENESIS, GENITOR [192], OOGA [38], or one of the evolution strategy systems [7]. So, is it worthwhile to build a new system from scratch?

Well, sometimes yes, sometimes no. If one is solving a transportation problem with hard constraints (i.e., constraints which must be satisfied), there is very little chance that some standard package would produce any feasible solution, or, if we start with a population of feasible solutions and force the system to maintain them, we may get no progress whatsoever — in such cases the system does not perform better than a random search routine. On the other hand, for some other problems such standard packages may produce quite satisfactory results. In short, the responsibility for making a decision on question (1) lies with

the user; the decision is a function of many factors, which include the demands of the precision of the required solution, time complexity of the algorithm, cost of developing a new system, feasibility of the solution found (i.e., importance of the problem constraints), frequency of using the developed system, and others.

Assume, then, that (for some reason) we have to (or like to) build a new system to solve a nontrivial optimization problem. This might be the case when standard GA packages do not provide acceptable feasible solutions and there are no computational packages available appropriate for the problem. Then we have to make a choice: either we can try to construct an evolution program or we can approach the problem using some traditional (heuristic) methods. It is interesting to note that in a traditional approach it usually takes three steps to solve an optimization problem:

1. understand the problem,
2. solve the problem,
3. implement the algorithm found in the previous step.

In the traditional approach, a programmer should **solve** the problem — only then may a correct program be produced. However, very often an algorithmic solution of a problem is not possible, or at least is very hard. On the top of that, for some applications, it is not important to find the optimal solution — any solution with a reasonable margin of error (relative distance from the optimum value) will do. For example, in some transportation problem one may look only for a *good* transportation plan — finding the optimal value is not required. In our experiments some evolution systems which were developed proposed (relatively quickly) a solution with a value, say, 1109, where the optimum value was 1102. In such a case (error less than 1%), the approximate solution might be more desirable.

An evolution programming approach usually eliminates the second, most difficult step. Just after we understand the problem, we can move to the implementational issues. The major task of a programmer in constructing an evolution program is a selection of appropriate data structures as well as "genetic" operators to operate on them (the rest is left for the evolution process). This task need not be trivial, since apart from the variety of data structures which can be used for chromosomal representation, each data structure may provide a wide selection of genetic operators. This would involve an understanding of the nature of the problem by a programmer; however, there is no need to solve the problem first. To construct an evolution program, a programmer would follow five basic steps:

1. First, (s)he selects a genetic representation of solutions to the problem. As we have already observed, this requires some understanding of the nature of the problem. However, there is no need to solve the problem. Selected representation of solutions to the problem should be "natural" and this decision is left to the programmer (note that in the current programming

environments, programmers select the appropriate data structures on their own).

2. The second task of a programmer is the creation of an initial population of solutions. This can be done in random way. However, some care should be taken in the cases when a set of constraints must be satisfied. Quite often a population of feasible solutions is accepted as a starting point of an evolution program.

3. Selection of an evaluation function (which rates solutions in terms of their fitness) should not pose a serious difficulty for optimization problems.

4. The genetic operators should be designed carefully — the design should be based on the problem itself and its constraints. These operators should match our intuition for transformation of a feasible solution (mutation group) into another feasible solution, or recombination of two (or more) feasible solutions (crossover group) to produce a legal offspring.

5. Values for various parameters that the program uses may be provided by a programmer. However, in the more advanced versions of evolution programming environments, these may be controlled by a supervisor genetic process (like one discussed in [80]), whose only task is to tune all parameters.

As seen in the above recipe for a construction of an evolution program, the general idea behind it lies within "natural" data structures and problem-sensitive "genetic operators". However, it still might be difficult to build such a system (for a particular problem) from scratch. An experienced programmer would manage this task; however, the resulting program might be quite inefficient. To assist the user in this task, we plan to create a new programming methodology supported by software (special programming languages) and hardware (parallel computers) — this is where our wishful thinking starts.

Currently, there are a number of different programming methodologies in computer science: structured programming, logic programming, object-oriented programming, functional programming. None of these fully support the construction of an evolution program. The goal of the new methodology would be the creation of appropriate tools for learning (here optimization is understood as a learning process) using a parallel processor architecture.

We hope to develop this idea to design a programming language PROBIOL (for PROgramming in BIOLogy) to support an EVA programming environment (EVA for EVolution progrAmming). An important issue in this methodology is an implementation of programs to control the evolution of the evolution process occurring in the "evolution engine" — the evolution engine is represented by a "society of microprocessors" [144]; some of these issues were briefly discussed in [101]. The key motivation of the new programming environment is to provide programming tools based on a parallel architecture. This is significant; as stated in [3]:

> "Parallelism is sure to change the way we think about and use computers. It promises to put within our reach solutions to problems and frontiers of knowledge never dreamed of before. The rich variety of architectures will lead to the discovery of novel and more efficient solutions to both old and new problems."

The goal of building such a programming environment to support evolution programming still seems to be quite distant. However, there are some other possibilities.

An interesting approach to evolution programming methodology was developed recently by Koza [108], [111]. Instead of building an evolution program, Koza suggests that the desired program should evolve itself during the evolution process. In other words, instead of solving a problem, and instead of building an evolution program to solve the problem, we should rather search the space of possible computer programs for the best one (the most fit). Koza developed a new methodology, named Genetic Programming, which provides a way to run such a search. A population of computer programs is created, individual programs compete against each other, weak programs die, and strong ones reproduce (crossover, mutation)...

There are five major steps in using genetic programming for a particular problem. These are

- selection of terminals,
- selection of a function,
- identification of the evaluation function,
- selection of parameters of the system,
- selection of the termination condition.

It is important to note that the structure which undergoes evolution is a hierarchically structured computer program.[1] The search space is a hyperspace of valid programs, which can be viewed as a space of rooted trees. Each tree is composed of functions and terminals appropriate to the particular problem domain; the set of all functions and terminals is selected *a priori* in such a way that some of the composed trees yield a solution. The initial population is composed of such trees; construction of a (random) tree is straightforward. The evaluation function assigns a fitness value which evaluates the performance of a tree (program). The evaluation is based on a preselected set of test cases; in general, the evaluation function returns the sum of distances between the correct and obtained results on all test cases. The selection is proportional; each tree has a probability of being selected to the next generation proportional to its fitness. The primary operator is a crossover that produces two offspring from two selected parents. The crossover creates offspring by exchanging subtrees

[1] Actually, Koza has chosen LISP's S-expressions for all his experiments.

between two parents. There are other operators as well: mutation, permutation, editing, and a define-building-block operation [108].

There is a significant difference between the EVA evolution programming environment and Koza's genetic programming paradigm; however, both are based on genetic algorithms and the principle of evolution, in general. It is worthwhile to note that there are many other approaches to learning, optimization, and problem solving, which are based on other natural metaphors from nature — the best known examples include neural networks and simulated annealing. There is a growing interest in all these areas; the most fruitful and challenging direction seems to be a "recombination" of some ideas at present scattered in different fields. As Schwefel and Männer wrote in the introduction to the proceedings of the First Workshop on Parallel Problem Solving from Nature [164]:

> "It is a matter of fact that in Europe evolution strategies and in the USA genetic algorithms have survived more than a decade of non-acceptance or neglect. It is also true, however, that so far both strata of ideas have evolved in geographical isolation and thus not led to recombined offspring. Now it is time for a new generation of algorithms which make use of the rich gene pool of ideas on both sides of Atlantic, and make use too of the favorable environment showing up in the form of massively parallel processor systems [...]. There are not many paradigms so far to help us make good use of the new situation. We have become too familiar with monocausal thinking and linear programming, and there is now a lack of good ides for using massively parallel computing facilities."

This book was an attempt to address some of these issues.

References

[1]Aarts, E.H.L. and Korst, J., *Simulated Annealing and Boltzmann Machines*, Wiley, Chichester, UK, 1989.

[2]Ackley, D.H., *An Empirical Study of Bit Vector Function Optimization*, in [33], pp.170–204.

[3]Akl, S.G., *The Design and Analysis of Parallel Algorithms*, Prentice-Hall, Englewood Cliffs, NJ, 1989.

[4]Antonisse, H.J., *A New Interpretation of Schema Notation that Overturns the Binary Encoding Constraint*, in [160], pp.86–91.

[5]Antonisse, H.J. and Keller, K.S., *Genetic Operators for High Level Knowledge Representation*, in [82], pp.69–76.

[6]Axelrod, R., *Genetic Algorithm for the Prisoner Dilemma Problem*, in [33], pp.32–41.

[7]Bäck, T., Hoffmeister, F., and Schwefel, H.-P., *A Survey of Evolution Strategies*, in [13], pp.2–9.

[8]Bäck, T., and Hoffmeister, F., *Extended Selection Mechanisms in Genetic Algorithms*, in [13], pp.92–99.

[9]Bagchi, S., Uckun, S., Miyabe, Y., and Kawamura, K., *Exploring Problem-Specific Recombination Operators for Job Shop Scheduling*, in [13], pp.10–17.

[10]Baker, J.E., *Adaptive Selection Methods for Genetic Algorithms*, in [78], pp.101–111.

[11]Baker, J.E., *Reducing Bias and Inefficiency in the Selection Algorithm*, in [82], pp.14–21.

[12]Bala, J., DeJong, K.A., Pachowicz, P., *Using Genetic Algorithms to Improve the Performance of Classification Rules Produced by Symbolic Inductive Method*, Proceedings of the 6th International Symposium on Methodologies of Intelligent Systems, Springer-Verlag, Lecture Notes in Artificial Intelligence, Vol.542, pp.286–295, 1991.

[13]Belew, R. and Booker, L. (Editors), Proceedings of the Fourth International Conference on Genetic Algorithms, Morgan Kaufmann Publishers, Los Altos, CA, 1991.

[14]Bellman, R., *Dynamic Programming*, Princeton University Press, Princeton, NJ, 1957.

[15]Bennett, K., Ferris, M.C., and Ioannidis, Y.E., *A Genetic Algorithm for Database Query Optimization*, in [13], pp.400–407.

[16]Bersini, H. and Varela, F.J., *The Immune Recruitment Mechanism: A Selective Evolutionary Strategy*, in [13], pp.520–526.

[17]Bertsekas, D.P., *Dynamic Programming. Deterministic and Stochastic Models*, Prentice Hall, Englewood Cliffs, NJ, 1987.

[18]Bethke, A.D., *Genetic Algorithms as Function Optimizers*, Doctoral Dissertation, University of Michigan, 1980.

[19]Bland, R.G. and Shallcross, D.F., *Large Traveling Salesman Problems Arising From Experiments in X-Ray Crystallography: A Preliminary Report on Computation*, Operations Research Letters, Vol.8, pp.125–128, 1989.

[20]Booker, L.B., *Intelligent Behavior as an Adaptation to the Task Environment*, Doctoral Dissertation, University of Michigan, 1982.

[21]Booker, L.B., *Improving Search in Genetic Algorithms*, in [33], pp.61–73.

[22]Bosworth, J., Foo, N., and Zeigler, B.P., *Comparison of Genetic Algorithms with Conjugate Gradient Methods*, Washington, DC, NASA (CR–2093), 1972.

[23]Brindle, A., *Genetic Algorithms for Function Optimization*, Doctoral Dissertation, University of Alberta, Edmonton, 1981.

[24]Brooke, A., Kendrick, D., and Meeraus, A., *GAMS: A User's Guide*, The Scientific Press, 1988.

[25]Carey, M.R. and Johnson, D.S., *Computers and Intractibility - A Guide to the Theory of NP-Completeness*, W.H. Freeman, San Francisco, 1979.

[26]Četverikov, S.S., *On Some Aspects of the Evolutionary Process From the Viewpoint of Modern Genetics*, (in Russian), Journal Exper. Biol., Vol.2, No.1, pp.3–54, 1926.

[27]Cleveland, G.A. and Smith, S.F., *Using Genetic Algorithms to Schedule Flow Shop Releases*, in [160], pp.160–169.

[28]Colorni, A., Dorigo, M., and Maniezzo, V., *Genetic Algorithms and Highly Constrained Problems: The Time-Table Case*, in [164], pp.55–59.

[29]Coombs, S., and Davis, L., *Genetic Algorithms and Communication Link Speed Design: Theoretical Considerations*, in [82], pp.252–256.

[30]Davidor, Y., *Genetic Algorithms and Robotics*, World Scientific, London, 1991.

[31]Davis, L., *Applying Adaptive Algorithms to Epistatic Domains*, Proceedings of the International Joint Conference on Artificial Intelligence, pp.162–164, 1985.

[32]Davis, L., *Job Shop Scheduling with Genetic Algorithms*, in [78], pp.136–140.

[33]Davis, L., (Editor), *Genetic Algorithms and Simulated Annealing*, Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1987.

[34]Davis, L. and Steenstrup, M., *Genetic Algorithms and Simulated Annealing: An Overview*, in [33], pp.1–11.

[35]Davis, L. and Ritter, F., *Schedule Optimization with Probabilistic Search*, Proceedings of the Third Conference on Artificial Intelligence Applications, Computer Society Press, pp.231–236, 1987.

[36]Davis, L., and Coombs, S., *Genetic Algorithms and Communication Link Speed Design: Constraints and Operators*, in [82], pp.257–260.

[37]Davis, L., *Adapting Operator Probabilities in Genetic Algorithms*, in [160], pp.61–69.

[38]Davis, L., (Editor), *Handbook of Genetic Algorithms*, Van Nostrand Reinhold, New York, 1991.

[39]Davis, L., *Bit-Climbing, Representational Bias, and Test Suite Design*, in [13], pp.18–23.

[40]Dawkins, R., *The Selfish Gene*, Oxford University Press, New York, 1976.

[41]DeJong, K.A., *An Analysis of the Behavior of a Class of Genetic Adaptive Systems*, Doctoral Dissertation, University of Michigan, 1975.

[42]DeJong, K.A., *Adaptive System Design: A Genetic Approach*, IEEE Transactions on Systems, Man, and Cybernetics, Vol.10, No.3, pp.556–574, 1980.

[43]DeJong, K.A., *Genetic Algorithms: A 10 Year Perspective*, in [78], pp.169–177.

[44]DeJong, K.A., *On Using Genetic Algorithms to Search Program Spaces*, in [82], pp.210–216.

[45]DeJong, K.A., *Learning with Genetic Algorithm: An Overview*, Machine Learning, Vol.3, pp.121–138, 1988.

[46]DeJong K.A., Spears, W.M., *Using Genetic Algorithms to Solve NP-Complete Problems*, in [160], pp.124–132.

[47]DeJong K.A., Spears, W.M., *Using Genetic Algorithms for Supervised Concept Learning*, Proceedings of the Second International Conference on Tools for AI, pp.335–341, 1990.

[48]DeJong K.A., *Genetic-Algorithm-Based Learning*, in [106], pp.611–638.

[49]Dhar, V. and Ranganathan, N., *Integer Programming vs. Expert Systems: An Experimental Comparison*, Communications of the ACM, Vol.33, No.3, pp.323–336, 1990.

[50]Eades, P. and Lin, X., *How to Draw a Directed Graph*, Technical Report, Department of Computer Science, University of Queensland, Australia, 1989.

[51]Eades, P. and Tamassia, R., *Algorithms for Drawing Graphs: An Annotated Bibliography*, Technical Report No. CS-89-09, Brown University, Department of Computer Science, October, 1989.

[52]Eshelman, L.J., Caruana, R.A., and Schaffer, J.D., *Biases in the Crossover Landscape*, in [160], pp.10–19.

[53]Even, S., Itai, A., and Shamir, A., *On the Complexity of Timetable and Multicommodity Flow Problems*, SIAM Journal on Computing, Vol.5, No.4, 1976.

[54]Fogarty, T.C., *Varying the Probability of Mutation in the Genetic Algorithm*, in [160], pp.104–109.

[55]Fogel, D.B., *An Evolutionary Approach to the Traveling Salesman Problem*, Biol. Cybern., Vol.60, pp.139–144, 1988.

[56]Fogel, D.B. and Atmar, J.W., *Comparing Genetic Operators with Gaussian Mutation in Simulated Evolutionary Process Using Linear Systems*, Biol. Cybern., Vol.63, pp.111-114, 1990.

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

[58]Forrest, S., *Implementing Semantic Networks Structures using the Classifier System*, in [78], pp.24–44.

[59]Fox, B.R., and McMahon, M.B., *Genetic Operators for Sequencing Problems*, in [148], pp.284–300.

[60]Fox, M.S., *Constraint–Directed Search: A Case Study of Job–Shop Scheduling*, Morgan Kaufmann Publishers, Los Altos, CA, 1987.

[61]Garey, M. and Johnson, D., *Computers and Intractability*, W.H. Freeman, San Francisco, 1979.

[62]Gillies, A.M., *Machine Learning Procedures for Generating Image Domain Feature Detectors*, Doctoral Dissertation, University of Michigan, 1985.

[63]Glover, D.E., *Solving a Complex Keyboard Configuration Problem Through Generalized Adaptive Search*, in [33], pp.12–27.

[64]Glover, F., *Heuristics for Integer Programming Using Surrogate Constraints*, Decision Sciences, Vol.8, No.1, pp.156–166, 1977.

[65]Glover, F., *Tabu Search — Part I*, ORSA Journal on Computing, Vol.1, No.3, pp.190–206, 1989.

[66]Glover, F., *Tabu Search — Part II*, ORSA Journal on Computing, Vol.2, No.1, pp.4–32, 1990.

[67]Glover, F., *Tabu Search for Nonlinear and Parametric Optimization*, Technical Report, Graduate School of Business, University of Colorado at Boulder; preliminary version presented at the EPFL Seminar on OR and AI Search Methods for Optimization Problems, Lausanne, Switzerland, November 1990.

[68]Goldberg, D.E., *Genetic Algorithm and Rule Learning in Dynamic Control System*, in [78], pp.8–15.

[69]Goldberg, D.E., *Dynamic System Control Using Rule Learning and Genetics Algorithms*, in Proceedings of the International Joint Conference on Artificial Intelligence, 9, pp.588–592, 1985.

[70]Goldberg, D.E. and Lingle, R., *Alleles, Loci, and the TSP*, in [78], pp.154–159.

[71]Goldberg, D.E., *Sizing Populations for Serial and Parallel Genetic Algorithms*, in [160], pp.70–79.

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

[73]Goldberg, D.E., *Messy Genetic Algorithms: Motivation, Analysis, and First Results*, Complex Systems, Vol.3, pp.493–530, 1989.

[74]Goldberg, D.E., *Zen and the Art of Genetic Algorithms*, in [160], pp.80–85.

[75]Goldberg, D.E., *Real-Coded Genetic Algorithms, Virtual Alphabets, and Blocking*, University of Illinois at Urbana-Champaign, Technical Report No. 90001, September 1990.

[76]Goldberg, D.E., *Messy Genetic Algorithms Revisited: Studies in Mixed Size and Scale*, Complex Systems, Vol.4, pp.415–444, 1990.

[77]Goldberg, D.E., Deb, K., and Korb, B., *Do not Worry, Be Messy*, in [13], pp.24–30.

[78]Grefenstette, J.J., (Editor), Proceedings of the First International Conference on Genetic Algorithms, Lawrence Erlbaum Associates, Hillsdale, NJ, 1985.

[79]Grefenstette, J.J., Gopal, R., Rosmaita, B., and Van Gucht, D., *Genetic Algorithm for the TSP*, in [78], pp.160–168.

[80]Grefenstette, J.J., *Optimization of Control Parameters for Genetic Algorithms*, IEEE Transactions on Systems, Man, and Cybernetics, Vol. 16, No.1, pp.122–128, 1986.

[81]Grefenstette, J.J., *Incorporating Problem Specific Knowledge into Genetic Algorithms*, in [33], pp.42–60.

[82]Grefenstette, J.J., (Editor), Proceedings of the Second International Conference on Genetic Algorithms, Lawrence Erlbaum Associates, Hillsdale, NJ, 1987.

[83]Groves, L., Michalewicz, Z., Elia, P., Janikow, C., *Genetic Algorithms for Drawing Directed Graphs*, Proceedings of the Fifth International Symposium on Methodologies of Intelligent Systems, North-Holland, Amsterdam, pp.268–276, 1990.

[84]Herdy, M., *Application of the Evolution Strategy to Discrete Optimization Problems*, in [164], pp.188–192.

[85]Hesser, J., Männer, R., and Stucky, O., *Optimization of Steiner Trees Using Genetic Algorithms*, in [160], pp.231–236.

[86]Hillier, F.S. and Lieberman, G.J., *Introduction to Operations Research*, Holden–Day, San Francisco, CA, 1967.

[87]Hobbs, M.F., *Genetic Algorithms, Annealing, and Dimension Alleles*, MSc. Thesis, Victoria University of Wellington, 1991.

[88]Hoffmeister, F. and Bäck, T., *Genetic Algorithms and Evolution Strategies*, in [164], pp.455–470.

[89]Holland, J., *Adaptation in Natural and Artificial Systems*, University of Michigan Press, Ann Arbor, 1975.

[90]Holland, J.H. and Reitman, J.S., *Cognitive Systems Based on Adaptive Algorithms*, in R.S. Michalski, J.G. Carbonell, T.M. Mitchell (Editors), *Machine Learning: An Artificial Intelligence Approach*, Los Altos, CA, Morgan Kaufmann Publishers, Los Altos, CA, 1983.

[91]Holland, J., *Properties of the Bucket Brigade*, in [78], pp.1–7.

[92]Holland, J., *Escaping Brittleness*, in R.S. Michalski, J.G. Carbonell, T.M. Mitchell (Editors), *Machine Learning II*, Morgan Kaufmann Publishers, Los Altos, CA, 1986.

[93]Holland, J.H., Holyoak, K.J., Nisbett, R.E., and Thagard, P.R., *Induction*, MIT Press, Cambridge, MA, 1986.

[94]Homaifar, A. and Guan, S., *A New Approach on the Traveling Salesman Problem by Genetic Algorithm*, Technical Report, North Carolina A & T State University, 1991.

[95]Horowitz, E. and Sahni, S., *Fundamentals of Computer Algorithms*, Computer Science Press, Potomac, MD, 1978.

[96]Husbands, P., Mill, F., and Warrington, S., *Genetic Algorithms, Production Plan Optimization, and Scheduling*, in [164], pp.80–84.

[97]Janikow, C., *Inductive Learning of Decision Rules in Attribute-Based Examples: a Knowledge-Intensive Genetic Algorithm Approach*, PhD Dissertation, University of North Carolina at Chapel Hill, July 1991.

[98]Janikow, C., and Michalewicz, Z., *Specialized Genetic Algorithms for Numerical Optimization Problems*, Proceedings of the International Conference on Tools for AI, pp.798–804, 1990.

[99]Janikow, C., and Michalewicz, Z., *On the Convergence Problem in Genetic Algorithms*, UNCC Technical Report, 1990.

[100]Janikow, C. and Michalewicz, Z., *An Experimental Comparision of Binary and Floating Point Representations in Genetic Algorithms*, in [13], pp.31–36.

[101]Jankowski, A., Michalewicz, Z., Ras, Z., and Shoff, D., *Issues on Evolution Programming*, in Computing and Information, (R. Janicki and W.W. Koczkodaj, Editors), North-Holland, Amsterdam, 1989, pp.459–463.

[102]Jog, P., Suh, J.Y., Gucht, D.V., *The Effects of Population Size, Heuristic Crossover, and Local Improvement on a Genetic Algorithm for the Traveling Salesman Problem*, in [160], pp.110–115.

[103]Johnson, D.S., *Local Optimization and the Traveling Salesman Problem*, in M.S. Paterson (Editor), Proceedings of the 17th Colloquium on Automata, Languages, and Programming, Springer-Verlag, Lecture Notes in Computer Science, Vol.443, pp.446–461, 1990.

[104]Jones, D.R. and Beltramo, M.A., *Solving Partitioning Problems with Genetic Algorithms*, in [13], pp.442–449.

[105]Kaufman, K.A., Michalski, R.S., and Scultz, A.C., *EMERALD 1: An Integrated System for Machine Learning and Discovery Programs for Education and Research*, Center for AI, George Mason University, User's Guide, No. MLI–89–12, 1989.

[106]Kodratoff, Y. and Michalski, R., *Machine Learning: An Artificial Intelligence Approach*, Vol.3, Morgan Kaufmann Publishers, Los Altos, CA, 1990.

[107]Korte, B., *Applications of Combinatorial Optimization*, talk at the 13th International Mathematical Programming Symposium, Tokyo, 1988.

[108]Koza, J.R., *Genetic Programming: A Paradigm for Genetically Breeding Populations of Computer Programs to Solve Problems*, Report No. STAN–CS–90–1314, Stanford University, 1990.

[109]Koza, J.R., *A Hierarchical Approach to Learning the Boolean Multiplexer Function*, in [148], pp.171–192.

[110]Koza, J.R., *Evolving a Computer Program to Generate Random Numbers Using the Genetic Programming Paradigm*, in [13], pp.37–44.

[111]Koza, J.R., *Genetic Programming*, MIT Press, Cambridge, MA, 1991.

[112]Laarhoven, P.J.M. van, and Aarts, E.H.L., *Simulated Annealing: Theory and Applications*, D. Reidel, Dordrecht, Holland, 1987.

[113]Langley, P., *On Machine Learning*, Machine Learning, Vol.1, No.1, pp.5–10, 1986.

[114]von Laszewski, G., *Intelligent Structural Operators for the k-way Graph Partitioning Problem*, in [13], pp.45–52.

[115]Lawer, E.L., Lenstra, J.K., Rinnooy Kan, A.H.G., and Shmoys, D.B., *The Traveling Salesman Problem*, Wiley, Chichester, UK, 1985.

[116]Leler, W., *Constraint Programming Languages: Their Specification and Generation*, Addison Wesley, Reading, MA, 1988.

[117]Lidd, M.L., *Traveling Salesman Problem Domain Application of a Fundamentally New Approach to Utilizing Genetic Algorithms*, Technical Report, MITRE Corporation, 1991.

[118]Liepins, G.E., and Vose, M.D., *Representational Issues in Genetic Optimization*, Journal of Experimental and Theoretical Artificial Intelligence, Vol.2, No.2, pp.4–30, 1990.

[119]Lin, S. and Kernighan, B.W., *An Effective Heuristic Algorithm for the Traveling Salesman Problem*, Operations Research, pp.498–516, 1972.

[120]Litke, J.D., *An Improved Solution to the Traveling Salesman Problem with Thousands of Nodes*, Communications of the ACM, Vol.27, No.12, pp.1227–1236, 1984.

[121]Messa, K. and Lybanon, M., *A New Technique for Curve Fitting*, Naval Oceanographic and Atmospheric Research Report JA 321:021:91, 1991.

[122]Michalewicz, Z. (Editor), Proceedings of the 5th International Conference on Statistical and Scientific Databases, Springer Verlag, Lecture Notes in Computer Science, Vol.420, 1990.

[123]Michalewicz, Z., *A Genetic Algorithm for Statistical Database Security*, IEEE Bulletin on Database Engineering, Vol.13, No.3, September 1990, pp.19–26.

[124]Michalewicz, Z., *EVA Programming Environment*, UNCC Technical Report, 1990.

[125]Michalewicz, Z., *Optimization of Communication Networks*, Proceedings of the SPIE International Symposium on Optical Engineering and Photonics in Aerospace Sensing, SPIE, Bellingham, WA, 1991, pp.112–122.

[126]Michalewicz, Z. (Editor), *Statistical and Scientific Databases*, Ellis Horwood, London, 1991.

[127]Michalewicz, Z. and Janikow, C., *Genetic Algorithms for Numerical Optimization*, Statistics and Computing, Vol.1, No.1, 1991.

[128]Michalewicz, Z. and Janikow, C., *Handling Constraints in Genetic Algorithms*, in [13], pp.151–157.

[129]Michalewicz, Z. and Janikow, C., *GENOCOP: A Genetic Algorithm for Numerical Optimization Problems with Linear Constraints*, Communications of the ACM, 1992.

[130]Michalewicz, Z., Janikow, C., and Krawczyk, J., *A Modified Genetic Algorithm for Optimal Control Problems*, Computers & Mathematics with Applications, Vol.23, No.12, pp.83–94, 1992.

[131]Michalewicz, Z., Jankowski, A., Vignaux, G.A., *The Constraints Problem in Genetic Algorithms*, in *Methodologies of Intelligent Systems: Selected Papers*, M.L. Emrich, M.S. Phifer, B. Huber, M. Zemankova, Z. Ras (Editors), ICAIT, Knoxville, TN, pp.142–157, 1990.

[132]Michalewicz, Z., Krawczyk, J., Kazemi, M., Janikow, C., *Genetic Algorithms and Optimal Control Problems*, Proceedings of the 29th IEEE Conference on Decision and Control, Honolulu, pp.1664–1666, 1990.

[133]Michalewicz, Z., Vignaux, G.A., Groves, L., *Genetic Algorithms for Optimization Problems*, Proceedings of the 11th NZ Computer Conference, Wellington, New Zealand, pp.211–223, 1989.

[134]Michalewicz, Z., Vignaux, G.A., Hobbs, M., *A Non-Standard Genetic Algorithm for the Nonlinear Transportation Problem*, ORSA Journal on Computing, Vol.3, No.4, pp.307–316, 1991.

[135]Michalewicz, Z., Schell, J., and Seniw, D., *Data Structures + Genetic Operators = Evolution Programs*, Proceedings of the Sixth International Symposium on Methodologies for Intelligent Systems (Poster Session), ORNL, Knoxville, TN, pp.107–118, 1991.

[136]Michalski, R., *A Theory and Methodology of Inductive Learning*, in R. Michalski, J. Carbonell, T. Mitchell (Editors), *Machine Learning: An Artificial Intelligence Approach*, Vol.1, Tioga Publishing Co., Palo Alto, CA, 1983.

[137]Michalski, R., Mozetic, I., Hong, J., Lavrac, N., *The AQ15 Inductive Learning System: An Overview and Experiments*, Technical Report, Department of Computer Science, University of Illinois at Urbana–Champaign, 1986.

[138]Michalski, R. and Watanabe, L., *Constructive Closed-Loop Learning: Fundamental Ideas and Examples*, MLI–88, George Mason University, 1988.

[139]Michalski, R., *A Methodological Framework for Multistrategy Task-Adaptive Learning*, Methodologies for Intelligent Systems, Vol.5, in Z. Ras, M. Zemankowa, M. Emrich (Editors), North-Holland, Amsterdam, 1990.

[140]Michalski, R. and Kodratoff, Y., *Research in Machine Learning: Recent Progress, Classification of Methods, and Future Directions*, in [106], pp.3–30.

[141]Montana, D.J., and Davis, L., *Training Feedforward Neural Networks Using Genetic Algorithms*, Proceedings of the 1989 International Joint Conference on Artificial Intelligence, Morgan Kaufmann Publishers, Los Altos, CA, 1989.

[142]Mühlenbein, H., Gorges-Schleuter, M., Krämer, O., *Evolution Algorithms in Combinatorial Optimization*, Parallel Computing, Vol.7, pp.65–85, 1988.

[143]Murtagh, B.A. and Saunders, M.A., *MINOS 5.1 User's Guide*, Report SOL 83-20R, December 1983, revised January 1987, Stanford University.

[144]von Neumann, J., *Theory of Self-Reproducing Automata*, edited by Burks, University of Illinois Press, 1966.

[145]Oliver, I.M., Smith, D.J., and Holland, J.R.C., *A Study of Permutation Crossover Operators on the Traveling Salesman Problem*, in [82], pp.224–230.

[146]Padberg, M., and Rinaldi, G., *Optimization of a 532-City Symmetric Travelling Salasman Problem*, Technical Report IASI–CNR, Italy, 1986.

[147]Quinlann, J.R., *Induction of Decision Trees*, Machine Learning, Vol.1, No.1, 1986.

[148]Rawlins, G., *Foundations of Genetic Algorithms*, First Workshop on the Foundations of Genetic Algorithms and Classifier Systems, Morgan Kaufmann Publishers, Los Altos, CA, 1991.

[149]Rechenberg, I., *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*, Frommann–Holzboog Verlag, Stuttgart, 1973.

[150]Reinelt, G., *TSPLIB – A Traveling Salesman Problem Library*, ORSA Journal on Computing, Vol.3, No.4, pp.376–384, 1991.

[151]Reinke, R.E. and Michalski, R., *Incremental Learning of Concept Descriptions*, in D. Michie (Editor), *Machine Intelligence XI*, 1985.

[152]Rendell, Larry A., *Genetic Plans and the Probabilistic Learning System: Synthesis and Results*, in [78], pp.60–73.

[153]Richardson, J.T., Palmer, M.R., Liepins, G., and Hilliard, M., *Some Guidelines for Genetic Algorithms with Penalty Functions*, in [160], pp.191–197.

[154]Rumelhart, D. and McClelland, J., *Parallel Distributed Processing: Exploration in the Microstructure of Cognition*, Vol.1, MIT Press, Cambridge, MA, 1986.

[155]Sasieni, M., Yaspan, A., and Friedman, L., *Operations Research Methods and Problems*, Wiley, Chichester, UK, 1959.

[156]Schaffer, J.D., *Learning Multiclass Pattern Discrimination*, in [78], pp.74–79.

[157]Schaffer, J.D. and Morishima, A., *An Adaptive Crossover Distribution Mechanism for Genetic Algorithms*, in [82], pp.36–40.

[158]Schaffer, J.D., *Some Effects of Selection Procedures on Hyperplane Sampling by Genetic Algorithms*, in [33], pp.89–103.

[159]Schaffer, J., Caruana, R., Eshelman, L., and Das, R., *A Study of Control Parameters Affecting Online Performance of Genetic Algorithms for Function Optimization*, in [160], pp.51–60.

[160]Schaffer, J., (Editor), Proceedings of the Third International Conference on Genetic Algorithms, Morgan Kaufmann Publishers, Los Altos, CA, 1989.

[161]Schraudolph, N. and Belew, R., *Dynamic Parameter Encoding for Genetic Algorithms*, CSE Technical Report #CS90–175, University of San Diego, La Jolla, 1990.

[162]Schwefel, H.-P., *Numerical Optimization for Computer Models*, Wiley, Chichester, UK, 1981.

[163]Schwefel, H.-P., *Evolution Strategies: A Family of Non-Linear Optimization Techniques Based on Imitating Some Principles of Organic Evolution*, Annals of Operations Research, Vol.1, pp.165–167, 1984.

[164]Schwefel, H.-P. and Männer, R. (Editors), Proceedings of the First International Conference on Parallel Problem Solving from Nature (PPSN), Dortmund, Germany, 1990.

[165]Schwefel, H.-P., Private communication, 1991.

[166]Seniw, D., *A Genetic Algorithm for the Traveling Salesman Problem*, MSc Thesis, University of North Carolina at Charlotte, 1991.

[167]Shaefer, C.G., *The ARGOT Strategy: Adaptive Representation Genetic Optimizer Technique*, in [82], pp.50–55.

[168]Shonkwiler, R. and Van Vleck, E., *Parallel Speed-up of Monte Carlo Methods for Global Optimization*, unpublished manuscript, 1991.

[169]Siedlecki, W. and Sklanski, J., *Constrained Genetic Optimization via Dynamic Reward–Penalty Balancing and Its Use in Pattern Recognition*, in [160], pp.141–150.

[170]Sirag, D.J. and Weisser, P.T., *Toward a Unified Thermodynamic Genetic Operator*, in [82], pp.116–122.

[171]Smith, D., *Bin Packing with Adaptive Search*, in [78], pp.202–207.

[172]Smith, S.F., *A Learning System Based on Genetic Algorithms*, PhD Dissertation, University of Pittsburgh, 1980.

[173]Smith, S.F., *Flexible Learning of Problem Solving Heuristics through Adaptive Search*, Proceedings of the Eighth International Conference on Artificial Intelligence, Morgan Kaufman Publishers, Los Altos, CA, 1983.

[174]Spears, W.M. and De Jong, K.A., *On the Virtues of Parametrized Uniform Crossover*, in [13], pp.230–236.

[175]Starkweather, T., McDaniel, S., Mathias, K., Whitley, C., and Whitley, D., *A Comparison of Genetic Sequencing Operators*, in [13], pp.69–76.

[176]Stebbins, G.L., *Darwin to DNA, Molecules to Humanity*, W.H. Freeman, New York, 1982.

[177]Stein, D., *Scheduling Dial a Ride Transportation Systems: An Asymptotic Approach*, PhD Dissertation, Harvard University, 1977.

[178]Stevens, J., *A Genetic Algorithm for the Minimum Spanning Tree Problem*, MSc Thesis, University of North Carolina at Charlotte, 1991.

[179]Suh, J.-Y. and Gucht, Van D., *Incorporating Heuristic Information into Genetic Search*, in [82], pp.100–107.

[180]Svirezhev, Yu.M., and Passekov, V.P., *Fundamentals of Mathematical Evolutionary Genetics*, Kluwer Academic Publishers, Mathematics and Its Applications (Soviet Series), Vol.22, London, 1989.

[181]Sysło, M.M., Deo, N., Kowalik, J.S., *Discrete Optimization Algorithms*, Prentice–Hall, Englewood Cliffs, NJ, 1983.

[182]Syswerda, G., *Uniform Crossover in Genetic Algorithms*, in [160], pp.2–9.

[183]Syswerda, G., *Schedule Optimization Using Genetic Algorithms*, in [38], pp.332–349.

[184]Syswerda, G. and Palmucci, J., *The Application of Genetic Algorithms to Resource Scheduling*, in [13], pp.502–508.

[185]Suh, J.-Y. and Lee C.-D., *Operator–Oriented Genetic Algorithm and Its Application to Sliding Block Puzzle Problem*, in [164], pp.98–103.

[186]Taha, H.A., *Operations Research: An Introduction*, 4th ed, Collier Macmillan, London, 1987.

[187]Tamassia, R., Di Battista, G. and Batini, C., *Automatic Graph Drawing and Readability of Diagrams*, IEEE Transactions Systems, Man, and Cybernetics, Vol.18, No.1, pp.61–79, 1988.

[188]Ulder, N.L.J., Aarts, E.H.L., Bandelt, H.-J., van Laarhoven, P.J.M., Pesch, E., *Genetic Local Search Algorithms for the Traveling Salesman Problem*, in [164], pp.109–116.

[189]Vignaux, G.A. and Michalewicz, Z., *Genetic Algorithms for the Transportation Problem*, Proceedings of the 4th International Symposium on Methodologies for Intelligent Systems, North-Holland, Amsterdam, pp.252–259, 1989.

[190]Vignaux, G.A. and Michalewicz, Z., *A Genetic Algorithm for the Linear Transportation Problem*, IEEE Transactions on Systems, Man, and Cybernetics, Vol.21, No.2, pp.445–452, 1991.

[191]Whitley, D., *GENITOR: A Different Genetic Algorithm*, Proceedings of the Rocky Mountain Conference on Artificial Intelligence, Denver, 1988.

[192]Whitley, D., *The GENITOR Algorithm and Selection Pressure: Why Rank-Based Allocation of Reproductive Trials is Best*, in [160], pp.116–121.

[193]Whitley, D., Starkweather, T., and Fuquay, D'A., *Scheduling Problems and Traveling Salesman: The Genetic Edge Recombination Operator*, in [160], pp.133–140.

[194]Whitley, D., Starkweather, T., and Shaner, D., *Traveling Salesman and Sequence Scheduling: Quality Solutions Using Genetic Edge Recombination*, in [38], pp.350–372.

[195]Whitley, D., Mathias, K., Fitzhorn, P., *Delta Coding: An Iterative Search Strategy for Genetic Algorithms*, in [13], pp.77–84.

[196]Winston, W.L., *Operations Research: Applications and Algorithms*, Duxbury, Boston, 1987.

[197]Wirth, N., *Algorithms + Data Structures = Programs*, Prentice-Hall, Englewood Cliffs, NJ, 1976.

[198]Wnęk, J., Sarma, J., Wahab, A.A., and Michalski, R.S., *Comparing Learning Paradigms via Diagramatic Visualization*, Proceedings of the 5th International Symposium on Methodologies for Intelligent Systems, North-Holland, Amsterdam, pp.428–437, 1990.

Springer Series
Artificial Intelligence

N. J. Nilsson: Principles of Artificial Intelligence. XV, 476 pages, 139 figs., 1982

J. H. Siekmann, G.Wrightson (Eds.): Automation of Reasoning 1. Classical Papers on Computational Logic 1957–1966. XXII, 525 pages, 1983

J. H. Siekmann, G. Wrightson (Eds.): Automation of Reasoning 2. Classical Papers on Computational Logic 1967–1970. XXII, 638 pages, 1983

L. Bolc (Ed.): The Design of Interpreters, Compilers, and Editors for Augmented Transition Networks. XI, 214 pages, 72 figs., 1983

M. M. Botvinnik: Computers in Chess. Solving Inexact Search Problems. With contributions by A. I. Reznitsky, B. M. Stilman, M. A. Tsfasman, A. D.Yudin. Translated from the Russian by A. A. Brown. XIV, 158 pages, 48 figs., 1984

L. Bolc (Ed.): Natural Language Communication with Pictorial Information Systems. VII, 327 pages, 67 figs., 1984

R. S. Michalski, J. G. Carbonell, T. M. Mitchell (Eds.): Machine Learning. An Artificial Intelligence Approach. XI, 572 pages, 1984

C. Blume, W. Jakob: Programming Languages for Industrial Robots. XIII, 376 pages, 145 figs., 1986

J. W. Lloyd: Foundations of Logic Programming. Second, extended edition. XII, 212 pages, 1987

L. Bolc (Ed.): Computational Models of Learning. IX, 208 pages, 34 figs., 1987

L. Bolc (Ed.): Natural Language Parsing Systems. XVIII, 367 pages, 151 figs., 1987

N. Cercone, G. McCalla (Eds.): The Knowledge Frontier. Essays in the Representation of Knowledge. XXXV, 512 pages, 93 figs., 1987

G. Rayna: REDUCE. Software for Algebraic Computation. IX, 329 pages, 1987

D. D. McDonald, L. Bolc (Eds.): Natural Language Generation Systems. XI, 389 pages, 84 figs., 1988

L. Bolc, M. J. Coombs (Eds.): Expert System Applications. IX, 471 pages, 84 figs., 1988

Springer Series
Artificial Intelligence

C.-H. Tzeng: A Theory of Heuristic Information in Game-Tree Search. X, 107 pages, 22 figs., 1988

H. Coelho, J. C. Cotta: Prolog by Example. How to Learn, Teach and Use It. X, 382 pages, 68 figs., 1988

L. Kanal, V. Kumar (Eds.): Search in Artificial Intelligence. X, 482 pages, 67 figs., 1988

H. Abramson, V. Dahl: Logic Grammars. XIV, 234 pages, 40 figs., 1989

R. Hausser: Computation of Language. An Essay on Syntax, Semantics, and Pragmatics in Natural Man-Machine Communication. XVI, 425 pages, 1989

P. Besnard: An Introduction to Default Logic. XI, 201 pages, 1989

A. Kobsa, W. Wahlster (Eds.): User Models in Dialog Systems. XI, 471 pages, 113 figs., 1989

B. D'Ambrosio: Qualitative Process Theory Using Linguistic Variables. X, 156 pages, 22 figs., 1989

V. Kumar, P. S. Gopalakrishnan, L. N. Kanal (Eds.) Parallel Algorithms for Machine Intelligence and Vision. XI, 433 pages, 148 figs., 1990

Y. Peng, J. A. Reggia: Abductive Inference Models for Diagnostic Problem-Solving. XII, 284 pages, 25 figs., 1990

A. Bundy (Ed.): Catalogue of Artificial Intelligence Techniques. Third, revised edition. XV, 179 pages, 1990

D. Navinchandra: Exploration and Innovation in Design. XI, 196 pages, 51 figs., 1991

R. Kruse, E. Schwecke, J. Heinsohn: Uncertainty and Vagueness in Knowledge Based Systems. Numerical Methods. XI, 491 pages, 59 figs., 1991

Z. Michalewicz: Genetic Algorithms + Data Structures = Evolution Programs. XVII, 250 pages, 48 figs., 1992