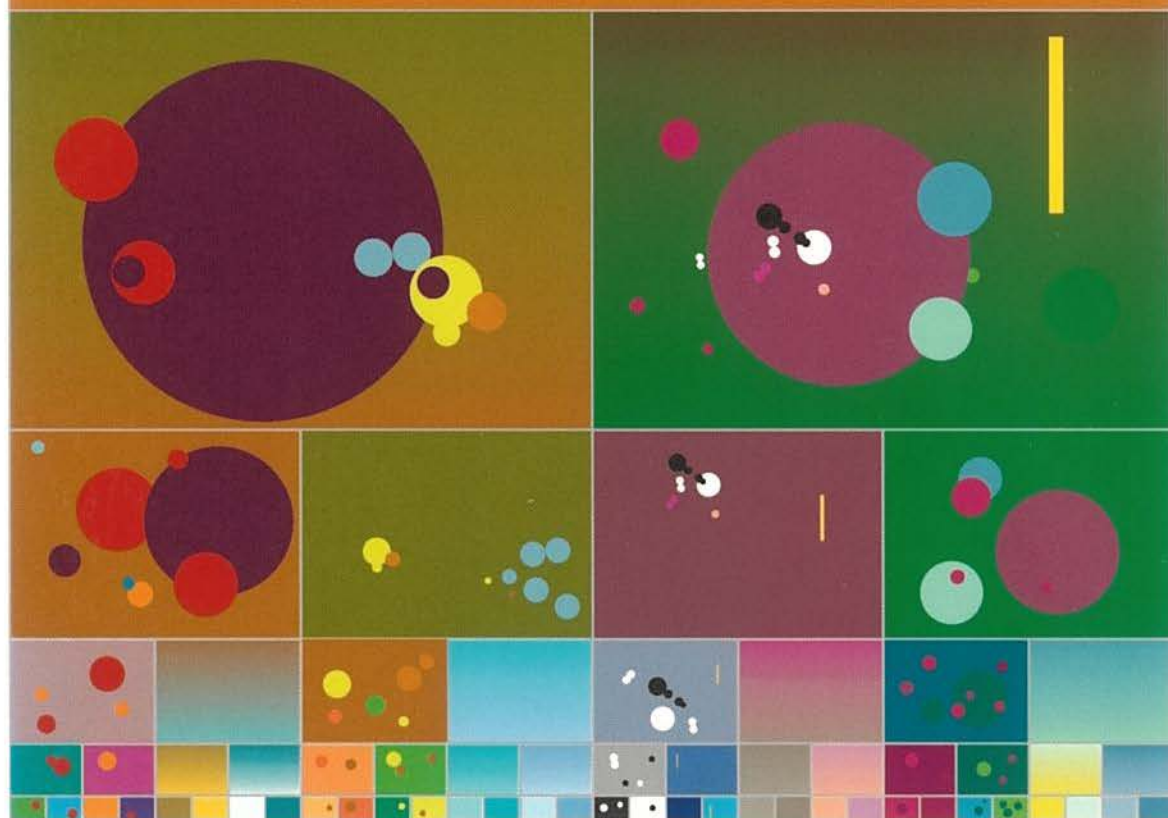


# Recombinative Evolutionary Search

Cees H.M. van Kemenade



# Stellingen

bij het proefschrift getiteld  
“Recombinative Evolutionary Search”  
van Cees H.M. van Kemenade

1. Recombinatie-gebaseerde evolutionaire algoritmen hebben het potentieel om mutatie-gebaseerde algoritmen te verslaan door het uitbuiten van het zogenaamde impliciet parallelisme<sup>1</sup>.
2. Recombinatieve evolutionaire algoritmen verrichten een globale exploratie over de ruimte die opgespannen wordt door de gehele populatie, en kunnen daarom baat hebben bij lokale zoekmethoden voor het beter exploreren van de directe omgeving van individuen<sup>2</sup>.
3. Elitist recombination kan beschouwd worden als een adaptief selectiemechanisme dat de selectieve druk aanpast aan de mate van succes van de evolutionaire operatoren<sup>3</sup>.
4. De balans tussen exploratie en exploitatie binnen een evolutionair algoritme is belangrijk en een nadruk op snelle resultaten, corresponderend met een voorkeur voor exploitatie boven exploratie, kan eenvoudig resulteren in snelle duplicatie van sub-optimale oplossingen<sup>4</sup>.
5. Biases in evolutionaire operatoren kunnen snellere convergentie van zoekmethoden mogelijk maken door het zoekproces primair te richten op bepaalde delen van de totale zoekruimte, waarbij het echter van groot belang is zulke biases zorgvuldig te kiezen aangezien een verkeerd gekozen bias de kans op het vinden van het (globale) optimum kan reduceren<sup>5</sup>.
6. Als men niet tracht strikt het proces der natuurlijke evolutie na te bootsen, dan zijn er veel innovatieve, niet-traditionele adaptieve methoden die nog steeds gebruik maken van (krachtige) principes overgenomen van het proces der natuurlijke evolutie<sup>6</sup>.

---

<sup>1</sup>Zie onder andere J.H. Holland (“Adaptation in natural and artificial systems” 1975/1992 en “Hidden order: How adaptation builds complexity”), D.E. Goldberg (“Genetic algorithms in search, optimization and machine learning” 1989), M. Mitchel “An introduction to genetic algorithms” 1996) en dit proefschrift.

<sup>2</sup>Zie hoofdstukken 9 en 10 van dit proefschrift

<sup>3</sup>Voorbeelden van de succesvolle toepassingen van zulk elitisme kunnen onder andere gevonden worden in dit proefschrift, hoofdstukken 4, 5, 7, 9, 10 en 12.

<sup>4</sup>Zie proefschrift.

<sup>5</sup>Zie hoofdstukken 1–3 van proefschrift en L.J. Eshelman and J.D. Schaffer, Productive recombination and propagating and preserving schemata. In L.D. Whitley and M.D. Vose, editors, *Foundations of Genetic Algorithms-3*, pages 299–313. Morgan Kaufmann, 1995.

<sup>6</sup>Zie hoofdstukken 4, 6, 7, 9, 11 en 12 van het proefschrift en vele andere publicaties die voorbeelden van niet-natuurlijke varianten op natuurlijke evolutie tonen.



7. In (unsupervised) remote-sensing beeldverwerking zijn significant betere classificaties te verkrijgen door het uitbuiten van dichtheids-gebaseerde separatierregels voor klassen en het uitbuiten van spatiële kennis gedurende het classificatieproces<sup>7</sup>.
8. Aangezien uitdagende, maar haalbare, doelen een belangrijke impuls kunnen vormen voor verder onderzoek is het zinvol als iedere onderzoeker (in opleiding) een deel van zijn/haar tijd besteedt aan het formuleren van uitdagende probleemstellingen die de grenzen van de “state-of-the-art” methoden aftasten.
9. Toename van capaciteiten van computers maakt het aanpakken van grotere problemen mogelijk, doch de meer fundamentele grenzen aan de toepasbaarheid van computers worden grotendeels bepaald door het vakmanschap van onderzoekers en ontwikkelaars.
10. Het volle potentieel van technieken uit het gebied der computationele intelligentie wordt zichtbaar als men zich bezig gaat houden met toepassingsgebieden met dynamisch veranderende gegevens, meerdere optimalisatiecriteria, vage restricties, mogelijk incorrecte informatie en onvolledige kennis, aangezien in deze gebieden mogelijk geen optimale oplossingen bestaan en men zich dient te richten op robuuste oplossingen die een redelijk compromis bieden met betrekking tot de optimalisatiecriteria en de gegeven restricties.
11. Mensen onderschatten de waarde van vaardigheden die ze zelf niet bezitten, wat eenvoudig leidt tot een te positief zelfbeeld (en een te negatief beeld van anderen).
12. Verkregen kennis en vaardigheden verworden tot een trivialiteit dan wel automatisme, wat eenvoudig leidt tot een onderschatting van de waarde van de eigen capaciteiten, en daardoor tot een te negatief zelfbeeld.
13. De wetenschap dient zich voornamelijk bezig te houden met het verkrijgen van praktisch toepasbare inzichten en niet te veel tijd te besteden aan praktische oplossingen voor specifieke problemen.
14. Het onderhouden van een kat laat je alvast wennen aan bepaalde facetten van het ouderschap, want zowel katten als kinderen vragen expliciet om (veel) aandacht, alle twee creëren een berg rommel en geen van de twee ruimt uit eigen initiatief deze rommel op. (Dat ze zich altijd netjes aan de huisregels houden kun je al helemaal vergeten.)

---

<sup>7</sup>C.H.M. van Kemenade, J.A. La Poutré, and R.J. Mokken. Density-based unsupervised classification for remote sensing. In *Proceedings of Machine Vision In Remotely sensed Image Comprehension (MaVIRIC)*, 1998.

Recombinative  
Evolutionary  
Search

The work reported in this dissertation has been carried out at the Centre for Mathematics and Computer Science (CWI) in Amsterdam and at Leiden University under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

IPA Dissertation Series 1999-04

Cover design: Tobias Baanders

# Recombinative Evolutionary Search

PROEFSCHRIFT

TER VERKRIJGING VAN DE GRAAD VAN DOCTOR  
AAN DE UNIVERSITEIT LEIDEN  
OP GEZAG VAN DE RECTOR MAGNIFICUS DR. W.A. WAGENAAR,  
HOOGLERAAR IN DE FACULTEIT DER SOCIALE WETENSCHAPPEN,  
VOLGENS BESLUIT VAN HET COLLEGE VOOR PROMOTIES  
TE VERDEDIGEN OP DONDERDAG 18 MAART 1999  
TE KLOKKE 15.15 UUR

DOOR

CORNELIS HENDRICUS MARIA VAN KEMENADE

GEBOREN TE UDEN  
IN 1969



## **Promotiecommissie:**

Promotor: Prof. dr. J.N. Kok

Referent: Prof. dr. Z. Michalewicz (University of North Carolina, USA)

Overige leden: Dr. Th. Bäck  
Prof. dr. J.W. de Bakker (Vrije Universiteit Amsterdam/CWI)  
Dr. A.E. Eiben  
Dr. J.A. La Poutré (Vrije Universiteit Amsterdam/CWI)  
Prof. dr. G. Rozenberg  
Prof. dr. H.A.G. Wijshoff

# Preface

In this dissertation I present the results of my research performed between September 1994 and September 1998 at the department SEN (Software ENgineering) of the Centre for Mathematics and Computer Science (CWI) in Amsterdam. This research was supervised by Prof. dr. J.N. Kok of Leiden University.

This dissertation is divided in two parts. The first part (chapter 1 through 5) is about the theory of genetic algorithms (GA's), and the second part (chapter 6 through 12) deals with empirical and applied research. The focus of the dissertation is on recombination-based search.

An overview of the field of evolutionary computation is given in *chapter 1*, which is followed by a more detailed introduction in *chapter 2*. In *chapter 3* a simple problem is introduced, and the behaviour of evolutionary algorithms when solving such a simple problem is studied by means of some quantitative models. *Chapter 4* describes the so-called transmission function framework, and contains implementations of transmission function models for a broad range of genetic algorithms (GA's). These models describe GA's with a population of infinite size. Real GA's always use a finite population, and therefore extensions of the transmission function framework for modelling of GA's with finite population size are considered in *chapter 5*. *Chapter 6* describes the mixing evolutionary algorithm (MixEA). This algorithm places more emphasis on the mixing of building blocks than the more traditional GA's. In *chapter 7* a three-stage method, the bbf-GA is introduced, where bbf-GA is an acronym for building block filtering genetic algorithm. This algorithm separates the exploration and exploitation processes that are performed simultaneously in the more traditional GA's. In *chapter 8* a test-suite of binary optimization problems is given. This test-suite is used to compare the performance of the canonical genetic algorithm, the generational genetic algorithms with tournament selection, the random-mutation hill-climber, the elitist recombination, the triple-competition, the mixEA, and the bbf-GA. One of the interesting properties of evolutionary algorithms is that these algorithms can easily incorporate heuristics and local optimization methods. In *chapter 9* the cluster evolution strategy (CLES) for evolutionary optimization is introduced. The CLES discriminates explicitly between global search (exploration) and local search (exploitation). The global search is provided by an evolutionary algorithm, and a local search method is used for the exploitation. *Chapter 10* discusses the use of evolutionary algorithms for constrained numerical optimization, and compares the performance of CLES and a number of other selection schemes on a set of constrained optimization problems. In *chapter 11* we give an

application of GA's with diagonal crossover operators for numerical optimization. The diagonal crossover is a multi-parent recombination operator that uses a set of  $n \geq 2$  parents, where  $n$  is adaptable. An application of GA's to air traffic flow management is given in *chapter 12*. A recombination-based evolutionary algorithm is used for this problem, and it is discussed how recombination improves the efficiency of the evolutionary algorithm.

Preliminary results from chapter 4 and chapter 5 were presented at the seventh "International Conference on Genetic Algorithms" [vK97a] and at the third "Nordic Workshop on Genetic Algorithms" [vK97c]. The MixEA described in chapter 6 was first introduced at the fourth "IEEE Conference on Evolutionary Computation" [vK97b]. Chapter 7 is based on a paper published in the proceedings of the "IEEE World Congress on Computational Intelligence" [vK98b], and the building block filtering method used in this chapter was first presented at "Parallel Problem Solving from Nature IV" [vK96c]. Chapter 9 is based on a paper presented at the third "IEEE conference on Evolutionary Computation" [vK96a]. Chapter 10 is based on a paper presented at the seventh "Dutch Conference on Artificial Intelligence" [vK96b]. Chapter 11 is based on a paper for the journal "Control & Cybernetics" [EvK97]. We would like to thank the editors for their kind permission to add the paper to this dissertation. Earlier work in this direction was presented at the third "European Conference on Artificial Life" [EvKK95], and at the sixth "Dutch Conference on Artificial Intelligence" [vKE95]. Chapter 12 is a paper published in the "Handbook on Evolutionary Computation" [vKKvdA97]. We would like to thank the editors for their permission to add the paper to this dissertation. Earlier work on the same problem was presented at the sixth "International Conference on Genetic Algorithms" [vKHHK95] and at "Parallel Problem Solving from Nature IV" [vKvdAK96]. Furthermore I did research on unsupervised classification of satellite imagery for more than two years. The results are outside the scope of this dissertation, however these results were published in the Proceedings of "MACHINE Vision In Remotely sensed Image Comprehension (MAVIRIC)" [vKPM99a], and in a book entitled "Spatial Statistics and Remote Sensing" [vKPM99b].

Many of my friends, family, and colleagues supported me during my research. I am grateful for their support. Some of them I would like to mention especially. My co-authors helped during research and in presenting my work. I had many discussions with Dirk Thierens about building block processing, mixing, deceptiveness, and evolutionary computation in general. These discussions have strongly influenced and enhanced my work. I discussed a lot with Han La Poutré on algorithmics and remote sensing, and he shared his vision on several areas from the field of computational intelligence and their applications. Eva was always there for (non-scientific) support and was patient with me. On several occasions holidays had to be postponed due to approaching submission deadlines. Although it might seem the other way round, she is much more important to me than my work.

Cees H.M. van Kemenade  
Amsterdam, 1999.

# Contents

|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>Introduction</b>   | <b>1</b>  |
| 1.1      | History of evolutionary computation . . . . .               | 2         |
| 1.2      | Basic terminology . . . . .                                 | 3         |
| 1.3      | Complex adaptive systems . . . . .                          | 5         |
| 1.4      | Evolution and optimization . . . . .                        | 6         |
| 1.5      | Mutation-based evolutionary algorithm . . . . .             | 8         |
| 1.6      | Recombination-based evolutionary algorithms . . . . .       | 10        |
| 1.7      | Taxonomy of evolutionary computation . . . . .              | 11        |
| 1.8      | Applications of evolutionary computation . . . . .          | 13        |
| 1.9      | Overview . . . . .  | 15        |
| <b>2</b> | <b>Concepts of evolutionary algorithms</b>                  | <b>19</b> |
| 2.1      | Functions of unitation . . . . .                            | 19        |
| 2.2      | Evolutionary operators . . . . .                            | 20        |
| 2.3      | Selection schemes . . . . .                                 | 23        |
| 2.4      | Schema theorem . . . . .                                    | 27        |
| 2.5      | Forma analysis . . . . .                                    | 28        |
| 2.6      | Building blocks and deception . . . . .                     | 28        |
| 2.7      | Genetic hitch-hiking . . . . .                              | 31        |
| 2.8      | Genetic drift . . . . .                                     | 32        |
| 2.9      | Stochastic sampling errors . . . . .                        | 32        |
| 2.10     | Balancing exploration and exploitation . . . . .            | 34        |
| 2.11     | NK-landscapes . . . . .                                     | 36        |
| <b>3</b> | <b>Models of Building block processing</b>                  | <b>37</b> |
| 3.1      | Efficiency of the traditional crossover operators . . . . . | 37        |
| 3.2      | Mixing of building blocks by uniform crossover . . . . .    | 40        |
| 3.3      | Cross-competition between building blocks . . . . .         | 42        |
| 3.4      | Mixing vs. generation of building blocks . . . . .          | 47        |
| 3.5      | Summary . . . . .   | 52        |



|          |  |            |
|----------|--|------------|
| <b>4</b> | <b>Infinite population models</b>                  | <b>53</b>  |
| 4.1      | Transmission models of selection schemes . . . . . | 54         |
| 4.2      | Infinite population models . . . . .               | 61         |
| 4.3      | Equivalence classes . . . . .                      | 62         |
| 4.4      | Cross-competition problem . . . . .                | 64         |
| 4.5      | Population flow diagrams . . . . .                 | 65         |
| 4.6      | Summary . . . . .                                  | 73         |
| <b>5</b> | <b>Finite population models</b>                    | <b>75</b>  |
| 5.1      | Finite population GA's . . . . .                   | 75         |
| 5.2      | Modified transmission function models . . . . .    | 80         |
| 5.3      | Finite population models . . . . .                 | 84         |
| 5.4      | Experiment setup . . . . .                         | 90         |
| 5.5      | Experiments . . . . .                              | 91         |
| 5.6      | Summary . . . . .                                  | 107        |
| <b>6</b> | <b>Mixing Evolutionary Algorithm</b>               | <b>109</b> |
| 6.1      | Mixing of building blocks . . . . .                | 109        |
| 6.2      | Mixing evolutionary algorithm . . . . .            | 112        |
| 6.3      | Experiments . . . . .                              | 116        |
| 6.4      | Summary . . . . .                                  | 120        |
| <b>7</b> | <b>Building Block Filtering and Mixing</b>         | <b>121</b> |
| 7.1      | Outline of bbf-GA . . . . .                        | 121        |
| 7.2      | Filtering of building blocks . . . . .             | 122        |
| 7.3      | Mixing with masked uniform crossover . . . . .     | 126        |
| 7.4      | The hybrid bbf-GA . . . . .                        | 127        |
| 7.5      | Test-problems . . . . .                            | 129        |
| 7.6      | Experiments . . . . .                              | 132        |
| 7.7      | Enhancements to the bbf-GA . . . . .               | 134        |
| 7.8      | Summary . . . . .                                  | 136        |
| <b>8</b> | <b>Comparison of genetic algorithms</b>            | <b>137</b> |
| 8.1      | Test-suite . . . . .                               | 137        |
| 8.2      | Compared methods . . . . .                         | 145        |
| 8.3      | Results . . . . .                                  | 146        |
| 8.4      | Summary . . . . .                                  | 173        |
| <b>9</b> | <b>Cluster Evolution Strategies</b>                | <b>175</b> |
| 9.1      | Randomized sampling methods . . . . .              | 175        |
| 9.2      | Evolutionary numerical optimization . . . . .      | 178        |
| 9.3      | Outline of CLES . . . . .                          | 180        |
| 9.4      | Clustering method . . . . .                        | 182        |

|           |   |            |
|-----------|---|------------|
| 9.5       | Evolutionary operators . . . . .                        | 184        |
| 9.6       | Local search strategy . . . . .                         | 187        |
| 9.7       | Test-suite . . . . .                                    | 188        |
| 9.8       | Results . . . . .                                       | 189        |
| 9.9       | Summary . . . . .                                       | 192        |
| <b>10</b> | <b>Constrained Numerical Optimization</b>               | <b>193</b> |
| 10.1      | Numerical constrained optimization . . . . .            | 194        |
| 10.2      | Regions of attraction . . . . .                         | 195        |
| 10.3      | Selection schemes . . . . .                             | 197        |
| 10.4      | Experiments . . . . .                                   | 198        |
| 10.5      | Larger test-suite . . . . .                             | 201        |
| 10.6      | Comparison to other methods . . . . .                   | 213        |
| 10.7      | Discussion of results on the large test-suite . . . . . | 219        |
| 10.8      | Summary . . . . .                                       | 220        |
| <b>11</b> | <b>Diagonal Crossover for Numerical Optimization</b>    | <b>221</b> |
| 11.1      | Introduction . . . . .                                  | 221        |
| 11.2      | Multi-parent recombination . . . . .                    | 222        |
| 11.3      | Experiment setup . . . . .                              | 224        |
| 11.4      | Experiments . . . . .                                   | 225        |
| 11.5      | Conclusions . . . . .                                   | 237        |
| <b>12</b> | <b>Evolutionary 3D-Air Traffic Flow Management</b>      | <b>239</b> |
| 12.1      | Project overview . . . . .                              | 239        |
| 12.2      | Air Traffic Flow Management Problem . . . . .           | 240        |
| 12.3      | Design process . . . . .                                | 241        |
| 12.4      | Development and implementation . . . . .                | 247        |
| 12.5      | Handling large-scale problems . . . . .                 | 248        |
| 12.6      | Results . . . . .                                       | 250        |
| 12.7      | Summary . . . . .                                       | 253        |
|           | <b>Bibliography</b>                                     | <b>255</b> |
|           | <b>Summary (in Dutch)</b>                               | <b>269</b> |
|           | <b>Curriculum Vitae</b>                                 | <b>271</b> |
|           | <b>Index</b>  | <b>273</b> |



# Chapter 1

## Introduction

Evolutionary computation is about the use of simulated evolution in a computer in order to solve optimization problems, in order to obtain adaptive systems, and in order to model evolution itself. Evolutionary computation methods typically use a population of individuals. These individuals have to compete for scarce resources. The distribution of these resources is based on the relative fitness of the different individuals, where the fitter individuals are likely to get more resources.

The theme of this dissertation is recombinative evolutionary search. One of the reasons that evolutionary search is so successful in nature is the highly parallel nature of this search mechanism. This parallelism really starts to pay off when recombination is used, because then it is possible to discover different traits in parallel and recombine them afterwards. Recombination-based evolutionary search promises to apply the same principles for artificial evolution. The primary components of recombinative evolutionary search are a fitness-based selection and a recombination operation. These two components are equally important for a reliable evolutionary search. In this dissertation we study the conditions for the reliability of selection schemes, we search for the reasons that traditional recombination operators fail on certain binary optimization problems, we show how to extend the range of applicability of recombination for these problems, and we show successful applications of recombination-based evolutionary search. In particular we study models for building block discovery and mixing processes, we develop more effective recombination operators by exploiting additional information, and we design selection mechanisms and recombination operators for several domains of application.

The plan of this introduction is as follows. A historical perspective on the field of evolutionary computation is given in section 1.1, followed by a brief overview of the vocabulary used in this field in section 1.2. Two perspectives on evolution are presented. First, natural evolution is described as a method to manage a complex adaptive system in section 1.3. Next, in section 1.4 natural evolution is described as an optimization process, and it is shown how the two perspectives can fit together. In section 1.5 a simple random search method is step-wise refined until a mutation-based evolutionary system for optimization is obtained. Section 1.6 discusses the decomposition of complex problems in a set of simpler problems as a strategy that can be used both in engineering problems and in natural



evolution, and shows how recombination-based evolutionary algorithms exploit a similar principle. A taxonomy of the field of evolutionary computation is given in section 1.7, and some interesting applications of evolutionary computation are discussed in section 1.8. We conclude this introduction with an overview of the rest of this dissertation in section 1.9.

## 1.1 History of evolutionary computation

The basis of the theory of natural evolution was laid by Darwin and Mendel. Darwin was mainly interested in how species originate and develop over generations. He visited the Galapagos islands, sailing on the *Beagle*, and observed how species had evolved differently on each of the isolated islands. In 1859 he published “The origin of species” [Dar59]. Mendel, an Austrian monk, was mainly interested in the evolution within a single species, and how traits of different parents are combined in an offspring by sexual reproduction. He did cross-fertilization between pea-plants, and discovered the rules for inheritance of traits (dominant and recessive). In 1865 he published “Experiments in Plant Hybridization”. He even tried to catch the attention of Darwin for his work. Unfortunately the Mendel-rules got forgotten until these rules were rediscovered by de Vries, Correns and Tschermak in 1900.

In evolutionary computation, principles from natural evolution are applied in computational models. Usually, such models only use a small subset of the ingredients of natural evolution. Both evolution of different species, as first studied by Darwin, and the evolution within a single species, as studied by Mendel, are considered in the field of evolutionary algorithms.

The field of evolutionary computation almost dates back to the time of the first electronic computers. Some of the first publications in this area are a masters’s thesis by Friedman on the evolution of control circuits for autonomous robots [Fri56], a paper on simulation of the evolution of genetic systems by Fraser [Fra57], a proposal for a (non-automated) evolutionary approach to increase industrial productivity by Box [Box57], papers on the evolution of computer programs by Friedberg [Fri58, FDN59], and work by Barricelli on artificial life and evolving strategies [Bar62, Bar63]. Many of these early publications can be found in the book “Evolutionary Computation, the fossil Record” [Fog98]. Although these early papers certainly contain many interesting ideas, the field did not really get momentum at that time. The turning point seems to be due to the continued work of two groups. The first group was centred around Holland at Michigan University (Ann Harbor) in the United States of America. This group focussed on adaptive systems and developed genetic plans, later called genetic algorithms. This work resulted in the influential book “Adaptation in natural and artificial systems” by Holland [Hol75, Hol92] and the dissertation of DeJong [DeJ75] titled “An analysis of the behaviour of a class of genetic adaptive systems”. The second group involved Schwefel and Rechenberg in Western-Germany. This group worked on the application of the principles of evolution for optimization and developed the so-called evolution strategy (Evolutionsstrategie in German). The first book on evolution strategies, written in the German language, is “Evolutionsstrategie ’73” by

Rechenberg [Rec73]. Both groups used a different perspective on evolution. The American group was mainly interested in adaptive systems, while the German group was aiming at optimization methods to be used in engineering applications. These different perspectives might have been the reason why the two research groups did not discover each others work until the 80's.

Due to the continuous efforts of these two groups, and the rise of faster computer systems, the field of evolutionary computation started to gain momentum. This resulted in the initiation of two important series of conferences. The first being the "International Conference on Genetic Algorithms", that was first organized by Holland and Grefenstette in Pittsburgh in 1985 [Gre85], and the second being the "Parallel Problem Solving from Nature" that was first organized by Männer and Schwefel in Dortmund (Germany) in 1990 [SM91]. Subsequent the workshop "Foundations of Genetic Algorithms", the conference "Evolutionary Programming", the conference "Genetic Programming" and the "IEEE conference on Evolutionary Computation" followed, each of which is devoted primarily to topics in the field of evolutionary computation. Currently evolutionary computation is an area of active research. There are two journals: the "Journal on Evolutionary Computation" by MIT-press that first appeared in 1993 and the "IEEE transactions on Evolutionary Computation" that started in 1997.

## 1.2 Basic terminology

In this section some of the basics of evolution are described in order to introduce the terminology that the field of evolutionary computation borrowed from biology.

*Evolution* is a gradual development, especially from simple to more complex forms. Natural evolution is a process by which species develop from earlier forms, as an explanation of their origins. This process is driven by means of a selection based on fitness. The *fitness* of individual is a measure for the potential of this individual to survive and to reproduce. The evolution process strives to maximize the fitness of the individuals. In natural evolution an individual has a genotype and a phenotype. The *genotype* is the genetic blueprint of the individual, which is given in its genome. The *phenotype* of an individual is given by the set of characteristics that an individual possesses. The phenotype is the actual expression of the genotype. The genotype-phenotype mapping is a many-to-one mapping, so each genotype results in a specific phenotype. However a phenotype can correspond to many different genotypes. The *genome* is the complete set of genetic information within a single cell. This genome consists of a set of chromosomes. A single *chromosome* is formed by double-stranded DNA. *DNA* is an abbreviation for deoxyribonucleic acid. It is a self-replicating material that is present in nearly all living organisms. A DNA strand consists of a sequence of genes, where a *gene* codes a single property, typically a certain protein that can be generated by the cell. These proteins express themselves by their influence on the characteristics of the individual. In genetics, each gene has a locus, and it can take a number of forms, called alleles. The *locus* of a gene is the location of the gene on the chromosome. An *allele* corresponds to a certain phenotypical characteristic. For

example, given a gene that codes the colour of the eye, its alleles are green, brown, grey, and black. The genome of a parent is transmitted to the offspring during reproduction. The simplest type of reproduction is *asexual reproduction*, where a single parent produces a single offspring. During such a reproduction the complete genome of the parent is copied to the offspring. It is possible that errors appear during this duplication process. Such an error is called a *mutation*. A random mutation can have either a positive or a negative influence on the fitness of an individual. These mutations play an important role in the evolution process, because mutations allow different types of individuals to arise. In natural evolution, the copying process is very precise, and the mutation-rates are generally very low, about  $10^{-5}$  or  $10^{-6}$  per generation for most loci in most organisms [FM96]. A more complex process is sexual reproduction. During *sexual reproduction* two parents mate to produce a set of offspring. The genome of the offspring is formed by means of a crossover. During *crossover*, also called *recombination*, an amalgam of the genomes of (typically) two parents is formed; Therefore, an offspring obtains a blend of the characteristics of their two parents. So, while mutation can generate new characteristics that were not present within the population, the recombination is able to bring together characteristics that arose in different individuals, and therefore is able to produce new combinations of characteristics.

The proteins that are produced can have quite complex interactions. Even proteins that seem to code unrelated characteristics can influence each other, for example due to competition for the same basic resources. So, even though we have assumed that genes are independent of each other in the genotype, during the actual expression phase nonlinear interactions might appear. The resulting nonlinear (non-additive) interaction between genes is called the *epistasis*, or epistatic interaction. Such epistasis complicates the adaptive system, and gives rise to many different effects. Alleles can both increase and decrease the fitness of an individual, depending on the context given by the other alleles present in the individual. Recombination becomes more important due to these epistatic effects. Without epistasis a recombination can only combine existing characteristics. When epistasis is present, recombination can also result in individuals having characteristics that were not expressed before in any individual even though the coding gene was present already. So the mixing of alleles done by recombination is important for the proper assessment of the performance of an allele. Genes on the same chromosome are more likely to be taken from the same parent. This tendency is referred to as *linkage* of genes.

Evolution is driven by a selective pressure. In nature two types of selective pressure are used. The first type selection is based on fertility. Assuming a fixed lifetime of individuals, the number of offspring is related to the *fertility* of the individual. Individuals with high fertility are able to produce more offspring, and therefore to spread more copies of their genetic content. This type of selective pressure applies for example to a predator at the high end of the food chain. Such a predator does not have natural enemies, and the amount of offspring this predator produces is mainly related to its fertility, where fertility is interpreted in a broad sense. So, the ability to catch food is also incorporated in the measure of fertility. The other interpretation of selective pressure is in terms of *viability*. The viability of an individual is its capability of living and developing normally under particular environmental conditions. In this interpretation the lifetime of an individual is

variable and the longer an individual lives the more likely it is that it can create offspring. So, while the first interpretation puts emphasis on the number of offspring produced, the second interpretation is in terms of the probability that an individual reaches maturity and is able to reproduce. This type of selective pressure dominates at the low end of the food chain. Typical examples are small animals that produce dozens of offspring once being mature. Here, the selective pressure stems from the ability to reach maturity. Another example is an oak-tree. During its lifetime it produces thousands of seeds. On average only one of these seeds produces a new, fully grown tree. Given a selection scheme, the *response* to selection is the change of the population means due to the selective pressure. In natural evolution mutations appear at each generation, but it may take twenty generations before mutation begins to contribute appreciably to the response, and much longer before the rate of response becomes constant. Thus mutation is important only for the long-term responses [FM96, Hil82].

*Ecology* is a branch of biology, dealing with the relations of organisms to each other and to their physical environment. A *species* is a group of living organisms consisting of related similar individuals capable of exchanging genes or interbreeding. A *niche* is a position or a role taken by a species within its community.

### 1.3 Complex adaptive systems

In this section two complex adaptive systems are discussed: the first one is a natural ecosystem and the second one is an economic system. Natural ecosystems are steered by means of evolution. We indicate that some of the principles of natural evolution also apply to certain economic systems.

Natural evolution is an amazingly efficient method for steering complex adaptive systems such that these perform robustly in a changing environment. An ecosystem is an adaptive system, composed of a large set of different individuals together with their environment, in which individuals have to compete for the scarce resources available. The robustness of such an adaptive system is mainly due to the diversity of the individuals present in the system. Given a specific environment, certain individuals perform better than others. However, if the environment changes, then other individuals can become the better players. So in general the population does not have an all-time “superstar” that performs best under all circumstances: the robustness of the system stems from the fact that individuals tailored to different circumstances are present in the system, thereby anticipating unknown future changes of the environment.

In a small system involving two species, one can already get interesting dynamics. Our ecosystem contains thousands of species, forming a highly complex adaptive system. Although a single individual can easily die due to a change in the environment or due to bad luck, the whole system is likely to survive even when there are drastic changes of the environment. In such a complex adaptive system, many different niches are present. Each niche is filled by only a small number of species. The boundaries of the niche determine the role of these species in the ecosystem, and the conditions under which these species



have to survive. The species adapts to its niche, resulting in a specialization of the species. If a species disappears from the ecosystem, then it leaves an empty niche. This hole in the system will be filled by other species. Natural evolution is mainly a highly parallel trial and error search process, driven by the survival of the fittest. It uses a diverse population of individuals in order to be prepared for changes in the environment. Even complex optimization problems are futile compared to the set of problems of survival solved in parallel by natural evolution.

Another complex adaptive system is an economy. An interesting example is given in the book “Hidden order” by Holland [Hol95]. He describes an inhabitant of New York who goes for some shopping. Like most other inhabitants she expects that everything she wants to buy is available. To quote Holland [Hol95]

“It’s a sort of magic that everywhere is taken for granted. Yet these cities have no central planning commissions that solve the problem of purchasing and distributing supplies. Nor do they maintain large reserves to buffer fluctuations.”

This economic system involves many agents that all operate in parallel, and all compete for the same scarce resource. If an agent makes a lot of money, then other agents notice this. The others will try to mimic the winning strategy. As a result the number of copies of (parts of) the winning strategy increases. Furthermore, each agent tries to attain a competitive edge by improving the strategy. To do so, the agent modifies the strategy or combines it with other successful strategies that were used in the past. The rate of evolution in such an economic system is much faster than in the evolution of the human species. As a result its evolution is more visible. The  $n$ -ary recombination operator of evolution can be compared to the process of combining a set of different strategies. The unary mutation operator here is the process of modifying a single strategy in order to improve it, or to use it in a different setting. Fitness of a strategy is measured in terms of return, i.e. money or units of purchasing power. Selective pressure is given by the drive of each of the agents to increase their income.

Of course, the natural ecosystem and the economy of New York are two completely different complex adaptive systems; However, both systems seem to use similar underlying evolutionary principles in order to attain a robust system that adapts itself to changes in the environment.

## 1.4 Evolution and optimization

The previous section described evolution in complex adaptive systems. Another way of looking at evolution is in terms of optimization. These two perspectives are quite different. In case of optimization one is looking for a single best performing individual, while in case of the adaptive systems perspective the performance and robustness of the population as a whole are of interest. In case of optimization the individuals compete on an external optimization task, and fitness is measured in terms of performance on this task. In adaptive systems we can have a direct interaction among the individuals, because they are competing

for the same scarce resources. Most optimization problems are static problems where fitness can be evaluated without noise. The typical adaptive system is highly dynamic, involves a changing environment, and is plagued by noise.

However, if we look at a single species and use a finer time scale, then optimization is also present in natural evolution. On a short time scale the environment of a single species is more or less unchanged. The environment does not change too much and the average interaction with the other species in the ecosystem remains more or less the same. Under these circumstances the evolution of a single species can be seen as an optimization problem. Natural evolution searches for the best possible individual of this species given the current environment. Due to the selective pressure, the better individuals get more duplicates of their genes. Of course the other species are evolving too, but it will take time before a successful trait is spread throughout the population of a species; Even though the best individual of other species might evolve rapidly, the average behaviour of other species changes only slowly. Hence, given an appropriate time scale, the environment of a species can be considered more or less unchanged, and optimization is seen as the adaptation of a single species. There has to be time for optimization, otherwise adaptation of the species would not be possible. Adaptation of a species requires that a new successful trait to be discovered and spread over a significant part of the population of this species. If the environment would change too fast (evolve faster than the species itself), then the species would not be able to adapt, and therefore the species is likely to get extinct in time.

Hence, one can have two perspectives on natural evolution. When looking at a small time-scale and a single species, an optimization process is observed. At larger time-scales and looking at the evolution of a complete ecosystem, one observes a highly complex adaptive system.

Evolutionary computation tries to exploit the principles of evolution by mimicking natural evolution within a computer. Given the two different perspectives, it is clear that we have to be careful when mimicking nature. Within evolutionary computation this mimicking of evolution is not a goal in itself. The actual goals of evolutionary computation are optimization and generation of (small) adaptive systems. Currently, we do not understand natural evolution well enough to know which subset of ingredients of natural evolution is necessary to solve optimization problems. Many evolutionary computation models incorporate other mechanisms. For example, the breeder genetic algorithm [MSV94] models the operation of a breeder that chooses who is allowed to mate with whom, and uses a deterministic selection scheme instead of probabilistic selection. Another modification that is used in evolutionary computation is to have more than two parents during recombination, in the multi-parent or gene-pool recombination operators [EvK97, MV95]. Other ways in which evolutionary computation differs from natural evolution are the use of local search methods, the incorporation of heuristics, and the use of problem-specific representations and operators, e.g. [Mic92, Mic94, Mic96].

The optimization perspective is used more often than the adaptive systems perspective in evolutionary computation. The majority of research in this area is related to optimization, and many extensions have appeared in evolutionary computation that do not have an equivalent in natural evolution.

## 1.5 Mutation-based evolutionary algorithms for optimization

In this section the usage of evolutionary algorithms for optimization is discussed and we introduce the notion of black box optimization problems. Next, a number of optimization methods are discussed, starting from a simple random search to a mutation-based evolutionary algorithm.

Consider a black-box optimization problem [Gol89b, Kar95]. Such a problem involves a black-box device with a bank of input switches. For every setting of these switches, there is an output signal  $f(s)$ , where  $s$  encodes the setting of the switches. Now the objective is to find the setting such that the output of the box is maximized. Because the way in which the box operates is unknown, a direct optimization method seems to be most appropriate. Direct methods only need to be able to evaluate the output signal given a parameter setting [Sch95], and therefore only use little knowledge about the underlying problem. Indirect methods use a model for the problem under consideration, and additional information is extracted to fit this model to a specific problem instance. If the model-assumptions are correct, then an indirect method is likely to locate well-performing solutions fast. Evolutionary algorithms are direct optimization methods [Sch95]; An evolutionary algorithm only needs a qualitative measure to compare the performance of the different individuals in order to proceed. Taking function optimization as an example, an evolutionary algorithm can do direct optimization. This means that one only needs to be able to evaluate the function in order to do the optimization, but one does not need to evaluate any derivatives of the function. An example of an indirect method is a gradient-based optimization method. Exploitation of an assumption corresponds to a bias in the search process. To quote Eshelman and Schaffer on biased search [ES95]:

“Only two search algorithms are bias free: random search and exhaustive enumeration. They work equally well, but equally inefficiently, on all problems. To improve efficiency of search one introduces some means of exploiting the information in previous samples to bias future samples. Such a practice will invariably divide the space of problems into two classes: those in which the bias is effective (or at least not ineffective), and those on which the bias will be deceived (the Achilles’ heel of the bias).”

Now, let us delve a little bit further into the details of the evolutionary computation, when used for optimization. Given a black-box optimization problem, the simplest approach is an enumeration of the search space, or to visit at least a large part of the search-space by means of random search. An advantage of such an enumerative method is its generality, because it does not use problem-specific knowledge. The disadvantage is that the number of computational steps required to locate the optimum is proportional to the cardinality of the search space. For a large search space such an enumerative approach is just too slow. Faster methods can be obtained by exploiting assumptions about the structure of the search space. An example is the assumption that a neighbourhood

of a well-performing solution is likely to contain even better solutions. For a search space that adheres to this (local) property, a hill-climbing method can locate a local optimum relatively fast. Such a hill-climber is a path-oriented search method that climbs a nearby hill, and locates its peak. Of course, a hill-climber is not a reliable global optimizer. The hill-climber converges to the nearest local optimum, and it only evaluates the function along the path it travels. A better behaviour can be obtained by doing multiple restarts of the local hill-climber, each time using a different starting point. Each hill-climber does a path-oriented search. The combination of all these paths results in a more or less volume-oriented search. Given that the local hill-climber is efficient, such a method is likely to be significantly faster than an enumerative method. When using such an approach, it is tricky to determine how long one should proceed with hill-climbing before selecting a new starting point. Some global knowledge is needed to determine whether a single hill-climber should be continued or not, because single hill-climber does not provide this kind of global information. To obtain this information, one can run a number of hill-climbers in parallel. Now by comparing the performance of the different hill-climbers, it is possible to evaluate the potential of the individual hill-climbers. The hill-climbers that perform (far) below average are discarded. The low performance of a hill-climber is thus used as an indication that the nearest local optimum is not very good. This approach, which corresponds to a shrinking population of hill-climbers, is also used for global optimization [TŽ89]. Apart from fitness-based pruning, one can also apply an intensification by creating new hill-climbers in the neighbourhood of successful hill-climbers. If such a new hill-climber is instantiated, based on the information present in a single existing hill-climber, then this algorithm is a mutation-based evolutionary algorithm.

Most evolutionary algorithms use a fixed population size, and apply operators that produce new individuals based on the current individuals. The mutation is an unary evolutionary operator. It takes a single parent as an input, and generates one new offspring. To get this offspring a copy of the parent is taken, and some, usually random, changes are made to this copy. Most evolutionary algorithms do not explicitly use a hill-climber, but local search is performed implicitly in the algorithm because the evolutionary operators are producing offspring that contain parts of the existing individuals in the population. Due to the usage of a population a more-or-less volume-oriented search method is obtained.

Some evolutionary algorithms also evolve internal models of the search space by means of evolution. These internal models are exploited by the evolutionary operators, allowing a more efficient generation of offspring. Examples are Evolution Strategies [Sch95, Bäck96] and Evolutionary Programming [Fog95] for numerical optimization. Both systems use Gaussian distributed noise during the mutation. The width of the Gaussian kernel is determined by means of evolution. During the initial stages of evolution the kernel is relatively wide: such a wide kernel corresponds to a global search. In the final stages of the search the width of this kernel is usually quite small, resulting in a local search in a small neighbourhood around the current best individuals. The width of the kernels is adapted by means of evolution.

## 1.6 Recombination-based evolutionary algorithms

Interesting optimization problems typically have a search space that is much too large to be searched fast by means of enumerative methods. Under these circumstances, one is forced to make assumptions about the structure of the search space. These assumptions can be used to create a more efficient search strategy for the problem at hand. The price to be paid is that the search strategy becomes tailored towards solving problems that have a search space in which the assumptions are valid.

A typical example of exploitation of structure is found in the area of numerical function optimization. Given a function one often assumes that the function has a continuous first derivative. In that case gradient-based methods can be used to do a fast local search. An example of such a method is a steepest ascent hill-climber, which moves uphill in the direction of the strongest gradient.

Another assumption that is often used is the decomposability of a function. If a function  $f : \mathbb{R}^n \rightarrow \mathbb{R}$  that maps a vector  $\vec{x}$  to a real value is separable, then it can be written as  $f(\vec{x}) = \sum_{i=1}^n f_i(x_i)$ . So, the function can be decomposed in the sum of a set of independent functions. Now, if we know the optimal values  $x_i^{opt}$  for the independent functions  $f_i$ , then we also know the optimum of the complete problem. Decomposition of complex problems in a set of simpler problems is often used in the area of engineering. Goldberg uses design of the first engine-propelled aircraft by the Wright brothers to explain his idea's on the design of genetic algorithms [Gol93]. They recognized two subproblems, i.e. the problem of getting lift and the problem of propelling the aircraft. They solved these problems separately, combined the solutions, and were able to fly. In natural evolution we can also observe examples of decomposing the complex problem of survival in many smaller independent problems. The main advantage of sexual reproduction (recombination) over asexual reproduction (mutation) is that sexual reproduction allows the combination of successful traits of different parents.

The same principle is exploited implicitly by recombination-based evolutionary algorithms. The building block hypothesis is often used as a possible explanation for the working of genetic algorithms, a recombination-based evolutionary algorithm. To quote Goldberg [Gol89b]:

“... so does a genetic algorithm seek near optimal performance through the juxtaposition of short, low-order, high-performance schemata, or building blocks.”

Given a problem, in which the optimal solution is composed of a set of building blocks, the building blocks are assumed to represent the most important interactions between loci for this problem. Loci that belong to different building blocks can interact, but it is assumed that the strongest interactions are between loci that belong to the same building block. The efficiency of an evolutionary algorithm is strongly influenced by the efficiency of the recombination operator. An efficient recombination operator should be able to mix the different building blocks rapidly. Simultaneously, this operator should not be too disruptive to the existing building blocks, in the sense that building blocks have to survive at least the recombination in order to get another opportunity to be mixed. The standard

recombination operators are randomized operators. These operators are blind to where the building blocks are located, so it is a matter of chance whether building blocks are preserved and recombined. For small building blocks and an appropriate recombination operator this process works fine. For large building blocks such randomized recombination operators become inefficient. There are several papers showing these linkage problems [ECS89, FA90, SEO91, ES93, FS94]. Recombination operators that are more efficient at processing building blocks are needed. The efficiency of the operator depends on the basis that one chooses for the search-space. Given a basis such that the building blocks are processed in one piece by the recombination operator, the evolutionary algorithm is likely to perform well. Which basis performs best is problem-specific, and sometimes even depends on the specific problems instances that one considers. Therefore, automated procedures are needed to find a basis for the search space such that recombination performs well. A number of different approaches along this line have been investigated. The messy genetic algorithms [GKD89, GDK90] use a representation consisting of lists of tuples, where each tuple contains a locus and a bit-value. The messy genetic algorithm processes strings that only describe part of a solution; Such partial solutions can represent a single building block. Specialized operators are used to process these partial solutions. This method was followed by the fast messy genetic algorithm [GDKH93, Kar95], and the Gene Expression Messy genetic algorithm (GEMGA) [Kar96c, Kar96a, BKW98]. Another approach was taken in the Linkage Learning Genetic Algorithm, where one searches for an ordering such that related genes get close to one another. The search for this ordering is done in an evolutionary manner [Har95].

Many applications of evolutionary algorithms are mutation-based. This is due to the fact that when one uses an arbitrary basis, it is likely that the optimal solution to the problem cannot be decomposed into a set of (independent) building blocks. Therefore, it is difficult to find an efficient crossover. Even for problems that are decomposable, it might be difficult to find independent building blocks without some method for learning the linkage.

## 1.7 Taxonomy of evolutionary computation

In this section we discuss the different branches in the area of evolutionary computation, and we give the similarities and differences between these main branches. The four main branches are genetic algorithms, evolution strategies, evolutionary programming, and genetic programming.

*Genetic algorithms*, originally called genetic plans, were introduced by Holland [Hol75]. The genetic algorithms were originally introduced as adaptive systems, though genetic algorithms were used for optimization too [DeJ75]. The original genetic algorithms typically use a binary coding of fixed length, a one-point crossover, a generational replacement scheme, and a fitness proportional selection. In a generational scheme there is no overlap between subsequent generations. So, the individuals of generation  $G_t$  produce offspring, and this offspring forms generation  $G_{t+1}$ . After production of offspring the parents of

$G_t$  are discarded. Fitness proportional selection implies that the expected number of offspring of an individual is proportional to its fitness. Many ideas used in genetic algorithms like the one-point crossover, binary coding, and fitness proportional selection were inspired by natural evolution. More details on genetic algorithms can be obtained from Holland [Hol75, Hol92, Hol95], Goldberg [Gol89b], Michalewicz [Mic92, Mic94, Mic96], Mitchel [Mit96], and Bäck [Bäc96].

The *evolution strategies* were developed by Rechenberg and Schwefel in Berlin, Germany, in 1965. At that moment the evolution strategy was basically an experimental strategy that was performed by hand. One of the first problems was finding an optimal setting for a parametrized shape, where the goal was to minimize the drag of this shape. Each fitness evaluation required the adjustment of the shape by hand, and performing an experiment in a wind-tunnel. The first numerical evolution strategies were developed in the 70's. The original strategy involved a kind of mutation-based hill-climbing, where the step size was determined by the fraction of successful mutations. If the number of successful mutations was too large, then the step-size was increased, if it was too small, then the step-size was decreased. Evolution strategies basically come in two variants. The first is the  $(\mu + \lambda)$  strategy. In this strategy one uses a population of  $\mu$  parent individuals. These  $\mu$  parents are used to produce  $\lambda$  offspring, where a uniform selection of parents is applied. Next the best  $\mu$  individuals from the set involving both the parents and the offspring, are transferred to the next generation. The second variant is the  $(\mu, \lambda)$  strategy. In this strategy the  $\mu$  parents of the next generation are obtained by selecting the  $\mu$  best among the  $\lambda$  offspring only. So, in case of a  $(\mu + \lambda)$ -strategy the parents can survive, while they cannot survive in case of the  $(\mu, \lambda)$ -strategy. The first evolution strategy, which involved a wind-tunnel experiment for each fitness-evaluation, was a  $(1 + 1)$ -strategy. The evolution strategies are very successful on numerical optimization problems. The evolution strategies evolve a vector of real values, and use a Gaussian kernel during the mutation operation. This Gaussian model is adapted during the evolution. In case of the  $(\mu, \lambda)$ -strategy, this model is adjusted by means of evolution. Mutation is the main operator in the evolution strategies, although recombination is used too. Further information about evolution strategies can be obtained from Rechenberg [Rec73, Rec94], Schwefel [Sch81, Sch95], and Bäck [BHmS91, Bäc96].

Originally *evolutionary programming* was used to evolve finite-state automata for a machine learning task. These evolution programs were developed by Fogel<sup>1</sup>, Owens, and Walsh in 1965 [FOW65]. In the 1990's Fogel<sup>2</sup> developed a variant of the evolution programs, specifically tailored for numerical optimization [Fog95]. This method is purely mutation-based, and it uses a Gaussian kernel for mutation.

*Genetic programming* was introduced by Koza and Rice [KR92, Koz94]. In genetic programming an individual typically consists of a Lisp-expression of variable length. Such a Lisp-expression corresponds to a computer program. This general representation makes genetic programming suitable for machine-learning tasks. Genetic programming typically

---

<sup>1</sup>L.J. Fogel

<sup>2</sup>D.B. Fogel



uses mutation and recombination. Some research suggests that recombination is not an efficient operator for genetic programming, and experiments show that a simulated-annealing based search through the space of Lisp expressions performs better than genetic programming on a number of tasks [OO94, Mit96], thereby suggesting that the power of genetic programming is mainly due to its representation. Many successful applications of genetic programming are given in "Advances in Genetic Programming" [Kin94, KA96].

## 1.8 Applications of evolutionary computation

There are many examples of successful applications of evolutionary computation. Here, we mention a few successful applications; Next, we give a more detailed description of the features that make evolutionary techniques successful, and the characteristics of problems where evolutionary algorithms are likely to outperform other methods. More interesting applications can be obtained from the "Handbook of Evolutionary Computation" [BFM97], and from an overview paper by Goldberg [Gol94].

One of the early complex engineering applications was the design of a nozzle [Rec94]. This nozzle should produce a high-speed, vapourized jet. Given the complexity of this system, involving both fluid and damp phases, it was not possible to design the optimal shape of the nozzle by means of traditional techniques. The evolutionary approach resulted in a nozzle with an efficiency of 80%, while the traditional design had an efficiency of approximately 50%. The evolution strategy used in that case involved real experiments, in the sense that a parametrized shape was used and fitness was determined by actually measuring the performance of the shape.

A completely different application is the Faceprints system, which is used to compose photographs of suspects [CJ91]. When using the traditional method of composing a photograph by means of a transparency-based sets of facial features, the witness has to explain why a composed photograph does not match. This requires the victim to be able to recall the individual features of the face. The Faceprint system uses an evolutionary algorithm to compose 20 different photographs. The witness has to rate these photographs on a 10-point subjective scale. A genetic algorithm is then used to take this information through normal selection, crossover, and mutation in order to generate a new set of photographs after which a new evaluation of the photographs by the witness follows.

Evolutionary algorithms have also been applied successfully in the area of engineering design. Deb and Goyal [DG97] compared a genetic algorithm with an augmented Lagrange method, and a branch-and-bound method, on a set of four engineering design problems. The test-suite involved nonlinear programming problems that were nonlinear both in the objective function and in the constraints. In their paper the genetic algorithm shows to be a flexible yet efficient optimization technique that handles mixed variables.

An interesting area for evolutionary computation is multi-objective optimization, because a population of individuals can easily follow different objectives. An overview is given by Fonseca and Fleming [FF95]. Furthermore evolutionary algorithms have been applied to combinatorial optimization problems, to evolve hardware, to design circuits,



and to evolve neural networks with non-traditional architectures.

Evolutionary algorithms are direct and flexible methods that perform well on many complex problems involving nonlinear objectives, nonlinear constraints, multiple objectives, dynamically changing problems, and in noisy environments. Evolutionary algorithms are direct methods. A large variety of problems can be handled directly by evolving a representation that is natural to the problem at hand. So, the evolutionary approach does not require one to recast the problem in a specific mathematical framework, nor does it require simplifying assumptions, such as linearization of the constraints. Evolutionary algorithms are flexible: the addition of new constraints or an additional objective does not require a change of the evolutionary method. Problem-specific solving methods can easily break down when the problem changes slightly. It is easy to incorporate problem-specific knowledge, heuristics, or local optimization methods within the evolutionary framework, resulting in so-called hybrid evolutionary algorithms. These hybrid algorithms combine the advantages of the problem-specific methods and the evolutionary framework.

Problem-specific optimization methods are usually quite fast. These methods use a model for the problem under consideration; Such a model contains the assumptions about the problem-domain that the method is tailored to. Evolutionary methods are general methods, and therefore these methods cannot exploit such problem-specific information. As a result, the evolutionary methods usually cannot compete with problem-specific methods on well-understood, structured problems. For example on problems only involving linear objectives and linear constraints, a method that exploits this knowledge such as the simplex-method, can search efficiently for the optimal solution. Due to the linearity the feasible region is a convex hull. The simplex-method locates the optimum by jumping over adjacent vertices, such that the objective value increases along the path followed. So, the simplex-method only visits a tiny subset of the complete search space. On such linear problems a general evolutionary method cannot be competitive. Now, if we change the problem by adding a nonlinear constraint, or by adding a nonlinear term to the objective, then the simplex method is not applicable anymore, because the linearity assumption is violated. The evolutionary method is still applicable: it does not make use of the linearity assumption.

Developing useful theory for evolutionary algorithms is difficult because evolutionary algorithms are general optimization methods. General theory of evolutionary algorithms investigates the operation of an evolutionary algorithm on an arbitrary (optimization) problem. The results predicted by such theories are of little practical value because no restrictions are imposed on the optimization problem under consideration. More applicable theoretical results can be obtained by restricting the class of optimization problems that is handled by an evolutionary algorithm. Examples of restricted classes are the functions of unimodal (such as oneMax), spherical functions for real-values, and linear functions. The results obtained from such a theoretical approach can give more insight in the actual operation of evolutionary algorithms, and the theoretical results can be verified by experiments. Of course one has to be very careful when applying such results to problems that do not adhere to the restrictions imposed during derivation of the results.

## 1.9 Overview

This dissertation is divided in two parts. The first part (chapter 1 through 5) is about the theory of genetic algorithms (GA's), and the second part (chapter 6 through 12) deals with empirical and applied research. The focus of the dissertation is on recombination-based search.

In *chapter 2* a more detailed introduction into the area of evolutionary computation is given. A number of different evolutionary algorithms are introduced, genetic operators are described, and topics related to the reliability of convergence of evolutionary algorithms are discussed. These preliminaries form the basis for later chapters.

In *chapter 3* a simple problem is introduced. The global optimum to this problem can be decomposed in two building blocks. Some of the factors that influence the probability that the global optimum is discovered are investigated. For this problem it is shown that the traditional crossover operators are not always efficient in transferring the building blocks with arbitrary linkage. The mixing of the building blocks is studied, and it is shown that cross-competition takes place between these building blocks. Such cross-competition can easily lead to the situation that one of the two building blocks is being pushed out of the population, and therefore it reduces the probability that the global optimum is found. If a building block is lost, then it has to be created again. The probability that this takes place is investigated.

In *chapter 4* the so-called transmission function framework is described, and implementations of transmission function models are given for a broad range of genetic algorithms. These models describe GA's with a population of infinite size. An actual implementation of these models for a non-trivial problem involving deception is given, these models are traced, and the results are visualized by means of population flow diagrams. These diagrams show that cross-competition between different parts of the optimal solution takes place.

Infinite population models show a deterministic behaviour. Genetic algorithms with finite populations behave non-deterministically. For small population sizes, the results obtained with these models differ strongly from the results predicted by the infinite population model. When the population size is increased towards infinity, a convergence to the results predicted by the infinite population models is observed. In real GA's random decisions are used during the run of the GA. These random decisions can lead to a behaviour that results in a deviation of the GA from the expected path of evolution. In *chapter 5* four sources of non-determinism are identified. Finite population models are generated by explicitly modelling two of these sources. When comparing the results to runs of actual genetic algorithms, similar results are obtained. Hence, this model shows what are the most important sources of non-determinism in the GA for the problem at hand.

*Chapter 6* describes the mixing evolutionary algorithm (MixEA). This algorithm is specifically designed to handle building block oriented problems, where the linkage is not known. Such problems are difficult to handle when the optimal building blocks are larger (high-order). Given a problem with arbitrary linkage the traditional recombination operators are not efficient. Therefore the transfer of high-order building blocks is not efficient,

and thus these building blocks will propagate and mix slowly. Suboptimal building blocks of low-order can easily generate many duplicates, and these low-order suboptimal blocks recombine easily to suboptimal solutions with a relatively high fitness. Individuals containing optimal building blocks can be pushed out of the population, and as a result the probability that the optimum is found decreases. The MixEA prevents such problems by using a more flexible allocation of trials. In this way it compensates for the low efficiency of the recombination operator, and it prevents the rapid duplication of lower-order building blocks. The population size of the MixEA decreases during the evolution. Simultaneously, the maximal number of recombination operations per individual increases. The idea exploited here is that during evolution better individuals containing more building blocks are found. As a result more recombinative trials are required to propagate these building blocks, because of the increase of the number of bits that have to be propagated. The large initial population of the MixEA allows for a good explorative search at the start of the evolution; The small populations during the later phases of evolution allow for more recombinative trials per individual such that a good exploitation and mixing of the building blocks present in these individuals is obtained.

Recombination works fine on many artificial problems involving tightly linked building blocks. On problems involving loose linkage, traditional recombination operators perform less well. The MixEA can prevent premature convergence due to the inefficient recombination operation and therefore increases the probability that the global optimum is found; However, the speed of convergence of the MixEA is still bound by the low probability of success of a recombination event, and therefore it converges only slowly. To make a more efficient recombination possible, one has to be able to discover the structure of the underlying problem. We have studied binary problems, where we try to discover and learn the linkage in the search-space during evolution. In *chapter 7* a three-stage method, the bbfg-GA is introduced, where bbfg-GA is an acronym for building block filtering genetic algorithm. During the first stage, an ensemble of fast evolutionary algorithms is used to explore the search space. The best individual found by each of these evolutionary algorithms is propagated to the next phase. During the second stage, building block filtering is used to extract the essential parts of each of these local optimal strings, and masks these essential parts. During the third stage, a single evolutionary algorithm is used to find the global optimum by recombining the masked strings. For this purpose we use a special recombination operator that exploits the information stored in the masks. Given an appropriate basis, such that partial solutions can be discovered and evaluated in parallel and be combined afterwards, a recombination-based evolutionary algorithm can be very efficient. Therefore, learning of the structure of problem-spaces is important to make a more efficient recombination possible. The bbfg-GA is a first step along this line for binary search spaces and problems that adhere to the building block hypothesis.

In *chapter 8* a test-suite of binary optimization problems is given. This test-suite is used to compare the performance of the canonical genetic algorithm, the generational genetic algorithms with tournament selection, the random-mutation hill-climber, the elitist recombination, the triple-competition, the mixEA, and the bbfg-GA. Several performance measures are given and the results are discussed.

One of the interesting properties of evolutionary algorithms is that these algorithms can easily incorporate existing heuristics and local optimization methods. In *chapter 9* the cluster evolution strategy (CLES) for numerical optimization is introduced. The CLES discriminates explicitly between global search (exploration) and local search (exploitation). The global search is provided by an evolutionary algorithm, and a local search method is used for the exploitation. Applying local search directly in an evolutionary algorithm can sometimes lead to an inefficient method, because many individuals have to be optimized, and the same local optima are likely to be discovered several times. In such hybrid evolutionary algorithms it is often difficult to find an appropriate population size. A small population is needed to prevent that a large amount of computation is spent on local optimization, while a large population is required to get a reliable global search. The CLES resolves this apparent contradiction by using a large population during the recombination phase, and pruning this population before applying local search. An additional reason for pruning is that it helps in obtaining a more reliable global search. Recombination tends to focus on regions that already contain relatively many individuals. Due to this effect the search can easily focus on broad suboptimal peaks, because these are discovered easily. The density-based pruning also helps in reducing this source of premature convergence. The CLES is tested on a set of unconstrained optimization problems.

*Chapter 10* discusses the use of numerical optimization methods for constrained numerical optimization, and compares the performance of CLES and a number of other selection schemes on a constrained optimization problem. The dimension of this problem is adjustable, and it is shown that this problem becomes more difficult as the dimension increases. Large differences are observed between the performance of the different selection schemes. The CLES outperforms the other selection schemes on this problem. Next, a set of evolutionary algorithms is applied to a larger test-suite involving eight numerical constrained optimization problems, and it is shown the proposed penalty-approach combined with the evolutionary algorithms used results in a well performing system. On all problems our GA's are very competitive with other evolutionary systems, and on two problems we obtained results that exceed the best results reported in literature.

In *chapter 11* we give an application of GA's with diagonal crossover operators for numerical optimization. The diagonal is a multi-parent recombination operator that uses a set of  $n \geq 2$  parents, where  $n$  is adaptable. This operator is shown to outperform the more traditional recombination operators on a test-suite.

An application of GA's to air traffic flow management is given in *chapter 12*. A recombination-based evolutionary algorithm is used for this problem, and it is discussed how recombination improves the efficiency of the evolutionary algorithm. The recombination operator used in this application was specifically designed for this problem. During the design we paid special attention to obtain a proper balance between exploration and exploitation. The recombination operator should be such that enough information is preserved to be effective, and at the same time the recombination operator should be disruptive enough to provide good global search.



## Chapter 2

# Concepts of evolutionary algorithms

In this chapter a more detailed introduction to evolutionary algorithms is given. Its primary aim is to present the basic knowledge about evolutionary algorithms needed in the rest of this dissertation.

Most work presented in this dissertation is about genetic algorithms (GA's), and unless stated otherwise, an individual is represented by a fixed length string of bits. In section 2.1 functions of unitation are described. Several test-functions used in theory and experiments of evolutionary algorithms are based on these functions. In section 2.2 the commonly used evolutionary operators for bit-string representations are given. Evolutionary operators are used to generate new individuals based on a set of parent individuals. These operators provide the actual exploration and discovery of new (possibly superior) individuals. In section 2.3 different selection schemes are discussed. The selection scheme determines the fertility and viability of individuals based on their (relative) fitness, and hence the selection scheme is the actual driving force of the evolutionary process. In section 2.4 the schema theorem is given, together with an intuitive explanation of this theorem. Section 2.5 discusses the extension of the schema theorem to more general representations by means of formae. Section 2.6 discusses the building block hypothesis, the notion of deception, and the actual definition of a building block, as used in this dissertation. Genetic hitch-hiking is discussed in section 2.7, genetic drift is discussed in section 2.8, and the stochastic sampling errors are discussed in section 2.9. In section 2.10 we look at the balance between exploration and exploitation. During a GA-search, an appropriate balance between these two processes is necessary. If this balance is not good, then the GA converges slowly, or it converges to suboptimal solutions. This balance is drawn implicitly in many genetic algorithms. In section 2.11 the quasi-random NK-landscapes are introduced. These landscapes are often used to test the performance of evolutionary algorithms.

### 2.1 Functions of unitation

A GA maximizes the fitness values of the individuals in its population. When using a GA with fitness proportional selection the fitness is required to be a positive value. Function

optimization involves the task of finding a value  $x$ , such that the objective function  $f(x)$  is maximized or minimized. When using a GA with fitness proportional selection as a function optimizer one chooses a representation such that an individual codes a value  $x$ , and one usually sets the fitness equal to  $(\alpha f(x) + \beta)$ , where  $\alpha$  is negative for a minimization problem and  $\alpha$  is positive for a maximization problem, and  $\beta$  is chosen such that fitness is always positive.

The path of evolution taken by an evolutionary algorithm depends on the fitness landscape defined by the fitness function. It is possible to develop general theory for an evolutionary algorithm, but then no assumptions can be made about the underlying problem and therefore about the fitness landscape. As a consequence the results of such a theory have to be quite general. Another approach is to assume a class of optimization problems and develop theory for an evolutionary algorithm, when applied to this class of problems.

An example of such a class is the class of the functions of unitation. This class is defined as follows. Let  $x$  be a bit-string of length  $l$ , and let  $u(x)$  define the number of 1-bits in  $x$ . A function of unitation is only dependent upon the number of 1-bits in a string. Now, if  $f : \mathcal{S} \rightarrow \mathbb{R}$  is a function of unitation, then there has to exist a function  $g : [0, |\mathcal{S}|] \rightarrow \mathbb{R}$  such that

$$f(x) = (g \circ u)(x).$$

A problem of unitation that is often used is the oneMax problem. This problem is about the maximization of the number of 1-bits in a string, with  $f(x) = u(x)$ . The optimum is a string containing only 1-bits, which has fitness  $l$ , where  $l$  is the total number of loci.

## 2.2 Evolutionary operators

Evolutionary algorithms are population based search methods. Evolutionary operators are used to explore the search-space, by generating offspring from parents. One can discriminate between two classes of operators. The first class contains mutation operators. These are unary operators, which means that the operators take a single parent as input and produce a single offspring. The second class contains the recombination operators, also called the crossover operators. These are  $p$ -ary operators, which means that these operators use  $p$  parents as input. Most recombination operators use two parents as input, just like in natural evolution, but it is possible to use different numbers of parents, under the restriction that  $p \geq 2$ .

In the remainder of this section a description is given of the most commonly used evolutionary operators for bit-string representations. More specific evolutionary operators are introduced in later chapters. A description is given of the  $n$ -point crossover, the uniform recombination, the multi-parent recombination, and the mutation operator.

### 2.2.1 $n$ -Point recombination operators

Holland [Hol75] introduced the one-point crossover operator. This operator takes two individuals, represented as bit-strings, as input. A random position between two loci is

selected as the crossover point, and both strings are split at this crossover point. Next, the first offspring is generated by combining the first part of first parent and the second part of the second parent. The second offspring is generated by combining the second part of the first parent with the first part of the second parent. Given a pair of parents,  $(l - 1)$  different crossover points are possible, and therefore at most  $(l - 1)$  different pairs of offspring can be obtained.

A generalization of the one-point crossover is the  $n$ -point crossover [DeJ75, DS91, DS93]. This crossover uses two parent individuals. It chooses  $n$  crossover points at random, splits both strings at these  $n$  points. The parts are numbered from 1 to  $(n + 1)$ . Two offspring are generated; The first offspring combines the odd-numbered parts of the first parent with the even numbered parts of the second parent; The second offspring is generated from the even numbered parts of the first parent and the odd numbered parts of the second parent.

### 2.2.2 Uniform recombination operators

The uniform crossover was introduced by Syswerda [Sys89]. It uses two parent individuals as input and creates two offspring. For each bit-position a separate coin is flipped. Depending on the side of the coin, the first offspring inherits the bit-value of the first parent or the second parent. The second offspring inherits the bit-value of the other parent. In case of uniform crossover the probability that two adjacent bit-values are obtained from the same parent is 0.5. Thus, the uniform crossover behaves roughly the same as an  $n$ -point crossover with  $(l - 1)/2$  different crossover points.

Of course, one can also use a biased coin during the uniform crossover. Given that  $\alpha$  is the probability of obtaining heads, such a crossover takes approximately  $\alpha l$  bit-values from the first parent and  $(1 - \alpha)l$  bit-values from the second parent, where  $l$  is the length of the bit-string.

### 2.2.3 Multi-parent/Gene-pool recombination operators

The  $p$ -parent generalizations of the traditional one-point crossover and uniform crossover in GA's were introduced in [ERR94]. Traditional crossover creates two children from two parents by splicing the parents along the single crossover point and exchanging the 'tails'. The basic idea behind diagonal crossover is to generalize this mechanism to an  $p$ -ary crossover. Diagonal crossover selects  $(p - 1)$  crossover points resulting in  $p$  chromosome segments in each of the  $p$  parents and composes  $n$  children by taking the pieces from the parents 'along the diagonals'. For instance, the first child is composed by taking substring<sub>1</sub> from parent<sub>1</sub>, substring<sub>2</sub> from parent<sub>2</sub>, etc., while the second child would have substring<sub>1</sub> from parent<sub>2</sub>, substring<sub>2</sub> from parent<sub>3</sub>, etc. Figure 2.1 illustrates this idea.

Notice that for  $p = 2$  diagonal crossover coincides with the traditional one-point crossover. The reason to expect that the use of more parents in diagonal crossover leads to improved GA performance is basically that the search becomes more explorative. Due



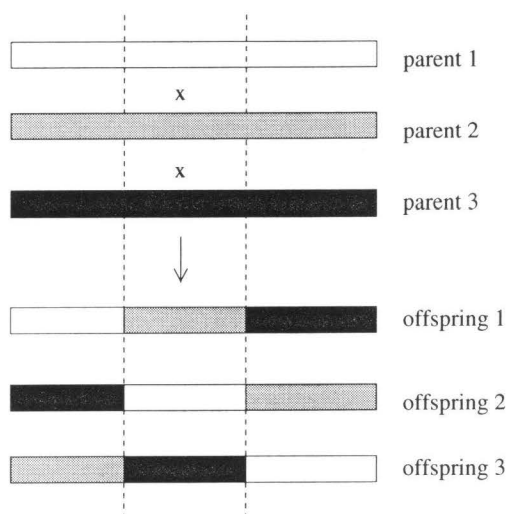


Figure 2.1: Application of diagonal crossover with three parents

to the larger number of parents, mixing of building blocks of low defining length (see section 2.4) is more effective than with the  $n$ -point crossover, and the probability of producing clones (duplicates of existing individuals) decreases. An overview on multi-parent recombination operators can be found in the “Handbook of Evolutionary Computation” [Eib98].

## 2.2.4 Mutation operators

The mutation operator takes a single parent individual as input and produces one offspring. If individuals are represented by bit-strings, then each bit is flipped, changed to the opposite value, with a certain small probability  $p_m$ ; So a biased coin is flipped for each locus. For the oneMax problem it has been shown that the optimal mutation probability is given by  $p_m = 1/l$ , where  $l$  is the length of the bit-string [Müh92]. For complex problems it is not known which value for  $p_m$  is optimal. As mutation is a disruptive operator, a large value of  $p_m$  can prevent the convergence of the evolutionary algorithm. For some optimization problems it has been shown that it is best to have an adaptive mutation rate [Fog89], where  $p_m$  is large during the initial, explorative phase of evolution, and  $p_m$  decreases as the evolution proceeds [Bäc92, Bäc93].

## 2.3 Selection schemes

The selection scheme enforces a selective pressure by adjusting the fertility and the viability of individuals based on their fitness. The selection scheme determines who is allowed to reproduce, how often individuals are allowed to reproduce, and which individuals are allowed to survive and thus are propagated to the next generation. Here, we discriminate between three types of selection schemes: the generational schemes, the steady-state schemes, and the local competition schemes. A separate subsection is devoted to each of these schemes.

### 2.3.1 Generational selection schemes

In generational evolutionary algorithms a sequence of generations is created. The individuals in generation  $G_t$  are the parents of the individuals in the next generation  $G_{t+1}$ . There is no overlap between subsequent generations in the sense that none of the individuals of generation  $G_t$  are transferred to the next generation, and thus all individuals in generation  $G_{t+1}$  are obtained by application of the evolutionary operators to the parents selected from generation  $G_t$ . We use three different types of generational schemes: the selection scheme of the canonical genetic algorithm, the generational genetic algorithm with tournament selection, and the  $(\mu, \lambda)$  selection scheme.

In a generational genetic algorithm, a selection method is used to select a number of parents. Next, a recombination operator is applied to these parents to get two offspring individuals. The mutation operator is applied to these offspring. These mutated offspring are put in the population  $P_{t+1}$ . This process is continued until the next population is completely filled.

Usually recombination is only applied with a certain probability given by the crossover probability, often denoted by  $p_c$ . If no crossover is applied, then the parents are duplicated, and mutation is applied to the resulting copies, to get the offspring.

During the selection step, the generational genetic algorithm needs a selection method. Two fitness-based selection methods are discussed here. The first is the fitness proportional selection. When using this type of selection, the probability that an individual  $x$  is selected is proportional to its fitness. So given the average fitness  $\bar{f}$  over the population  $P_t$ , the number of copies of individual  $x$  is  $f(x)/\bar{f}$ . The process of the selection of an individual can be visualized by means of a roulette-wheel with a pointer. The roulette-wheel is divided in a number of parts, one for each individual; The part corresponding to individual  $x$  covers a fraction  $f(x)/\bar{f}$  of the roulette-wheel. Now a single individual is selected by turning the roulette-wheel. When the rotation stops, the pointer points at a certain part, and the corresponding individual is selected. If we have to select  $n$  individuals, then one has to spin the wheel  $n$  times. On average, individual  $x$  is selected  $nf(x)/\bar{f}$  times, but in practice it can be selected more often, or less often. This spread in the number of copies is a sampling error during selection. The sampling error of fitness proportional selection can be reduced by using "Stochastic Universal Sampling" [Bak87]. This selection method is obtained by means of a small modification to the previous method. Instead of one pointer,

this method use  $n$  pointers, where the angle between two subsequent pointers is  $2\pi/n$ ; Now, a single spin of the wheel results in the selection of all  $n$  individuals. The sampling error is much lower now. The actual number of copies of an individual  $c_x$  in this case is bounded by

$$\lfloor nf(x)/\bar{f} \rfloor \leq c_x \leq \lceil nf(x)/\bar{f} \rceil.$$

Fitness proportional selection is a non-deterministic selection method. This selection method only modifies the probability of selection of the different individuals. The fitness proportional selection requires that the fitness of individuals is given by positive values, because the probabilities of selecting an individual are proportional to fitness, and these probabilities cannot be negative. Furthermore, the fitness proportional selection is not invariant under scaling of fitness values. For example, if the fitness of all individuals is changed by adding a large positive constant  $a$ , then the average fitness is increased by  $a$ , and the selective pressure of fitness proportional selection is decreased. Due to the same effect the selective pressure of fitness proportional selection decreases when most individuals in the population have a near-optimal fitness.

An example of a generational genetic algorithm with fitness proportional selection is the canonical genetic algorithm that was introduced by Holland [Hol75]. It is also described by Goldberg [Gol89b] under the name simple genetic algorithm.

Another fitness-based selection scheme is tournament selection. A form of tournament selection was already studied in Brindle's dissertation [Bri81]. A tournament selection with tournament size  $k$ , where  $k \geq 2$ , is obtained by selecting  $k$  individuals uniform at random from the population. The best out of these  $k$  individuals is selected. After tournament selection the best individual is expected to have  $k$  copies, the median individual is expected to have  $(1/2)^{k-1}$  copies, and the worst individual is expected to have no copies. Larger values of the tournament size  $k$  result in a stronger selective pressure and therefore in more duplicates of the best few individuals. Tournament selection is a non-deterministic selection method. In this selection scheme it is possible to reduce the sampling error during selection by means of an intermediate population. Given that a single recombination operation uses two parents to produce two offspring, an intermediate population of size  $kn$  is generated by duplicating all individuals in the parent population  $k$  times. Now a parent is selected by drawing  $k$  individuals without replacement from this intermediate population and setting up the  $k$ -tournament. The sampling error is reduced as each individual participates in exactly  $k$  tournaments.

The  $(\mu, \lambda)$  selection scheme from evolution strategies is a generational scheme too. A parent population of size  $\mu$  is used to generate  $\lambda$  offspring. Next, the  $\mu$  best individuals out of the  $\lambda$  offspring are used as parents for the next generation. This selection scheme is deterministic. In this selection scheme it is also possible to reduce the sampling error during selection by means of an intermediate population. This intermediate population is filled with the number of parents needed. Next, the actual parent pairs are selected without replacement from this intermediate population. Now given that a single recombination operation uses two parents to produce two offspring, the number of times a parent is selected for reproduction is bounded by  $\lfloor \lambda/\mu \rfloor \leq c_x \leq \lceil \lambda/\mu \rceil$ . A similar selection scheme

is the  $T\%$ -truncation selection, which is a deterministic selection scheme that retains the  $T\%$  best individuals. Truncation selection is used in Breeder Genetic Algorithm [MSV94].

### 2.3.2 Steady-state selection schemes

Steady-state selection schemes typically replace only a limited number of individuals. An individual can live for more than one generation, and thus there is an overlap between subsequent generations. A steady-state selection scheme requires a selection method and a replacement method. The selection method determines which individuals are allowed to reproduce. The resulting offspring has to be added to the population. In order to keep the population size constant, the offspring will replace individuals from the population. The replacement method determines which individual is replaced. So the selection method determines the fertility of individuals, while the reduction scheme determines the viability of the individuals. Both the selection method and the reduction method can be used to obtain a selective pressure. An example of the usage of the selection method is a selection scheme where tournament selection is used to select individuals for reproduction, and the individual to be replaced is obtained by a uniform selection. In this selection scheme the replacement is not by a fitness-based selection, and therefore the selective pressure is solely due to the selection method. Another example is a selection scheme where a uniform selection over the population is used to obtain the individuals that are allowed to reproduce, and the replacement method always replaces the worst individual of the population. In such a selection scheme, the selective pressure is solely enforced by the replacement method. Steady-state genetic algorithms are used in Genitor [Whi89, WS90] and several other genetic algorithm [Sys89, vKKE95].

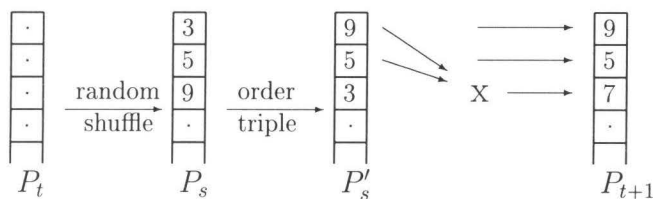


Figure 2.2: Schematic representation of triple-competition selection

A different steady-state scheme is the triple-competition selection scheme [vK97c]. Figure 2.2 shows how the next population  $P_{t+1}$  is produced from the current population  $P_t$ . On the left we see the current population  $P_t$ , where each box represents a single individual. The values in the boxes denote the fitness of the corresponding individuals. An intermediate population  $P_s$  is generated by doing a random shuffle on  $P_t$ . Population  $P_s$  is partitioned in a set of triples. Within each set of three individuals the two best performing individuals are allowed to create one offspring. This offspring replaces the third individual.

This modified triple is propagated to the next generation. So two-third of the individuals are propagated unmodified to the next generation. The best two individuals are never lost when using this selection scheme. It has been observed that triple-competition selection tends to result in relatively fast convergence of the population.

The  $(\mu + \lambda)$  selection of evolution strategies is considered to be a steady-state selection scheme too. A parent population of size  $\mu$  is used to generate  $\lambda$  offspring. Next, the best  $\mu$  individuals out of the  $\mu$  parents and the  $\lambda$  offspring are used as parents for the next generation.

### 2.3.3 Local competition selection schemes

Local competition evolutionary algorithms use a local competition between the parents and their direct offspring. The winners of this competition are transferred to the next population. An example of a local competition selection scheme is the Elitist recom-

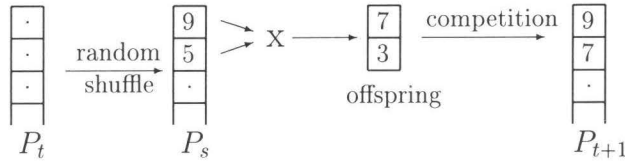


Figure 2.3: Schematic representation of Elitist recombination

ination [TG94]. It selects parents by creating random pairs of individuals. Because all parents have exactly the same probability of being selected this corresponds to a uniform selection. The sampling error during this selection is reduced because each individual participates exactly once in a competition during a single generation. Figure 2.3 shows how the next population  $P_{t+1}$  is produced from the current population  $P_t$ . On the left we see the current population  $P_t$ , where each box represents a single individual. The values in the boxes denote the fitness of the corresponding individuals. An intermediate population  $P_s$  is generated by doing a random shuffle on  $P_t$ . Population  $P_s$  is partitioned in a set of adjacent pairs and for each pair the recombination operator is applied to obtain two offspring. Next, a competition is held between the two offspring and their two parents, and the two winners are transferred to the next population  $P_{t+1}$ . In the example one parent and one offspring are transferred to  $P_{t+1}$ . Elitism is used because parents can survive their own offspring. Elitist recombination has been shown to be more efficient than some other GA's on a problem involving a high-order building block [vK97a].

Deterministic crowding is almost similar to elitist recombination. It uses local competition between parents and offspring too [Mah92]. The only difference is the way in which the local competition is set up. In order to get a competition two pairs are formed, each consisting of one parent and one offspring. These pairs are formed such that the similarity between the individuals in the pairs is maximized. For each pair a competition is held, and

the winner is propagated to the next generation. Due to the similarity based matching of parent-offspring pairs, a too rapid duplication of well-performing individuals is prevented. Deterministic crowding is likely to converge more slowly than elitist recombination, but it is less sensitive to premature convergence.

Other selection schemes that use a local competition are parallel simulated annealing [MG92, MG95], and the Gene Invariant GA (GIGA) [Cul93].

## 2.4 Schema theorem

The schema theorem by Holland [Hol75] gives lower bounds on the expected proportion of strings that contain a schema in the next generation. A binary coding is assumed. Given that a single solution consists of a sequence of  $l$  bits, a schema is represented by a string of length  $l$  over the alphabet  $\{0, 1, \#\}$ . Here 0 and 1 denote the values of a the bit, and  $\#$  can represent either a 0 or a 1. A schema that contains  $m$   $\#$ -symbols represents  $2^m$  different bit-strings. A defined locus in a schema is a position where the schema contains either a 0 or a 1 symbol. Now, let the defined loci of a schema  $H$  be given by  $i_1, \dots, i_h$ , such that  $i_1 < i_2 < \dots < i_h$ , where each  $i_k$  denotes a position of locus in the bit-string. The order of a schema, denoted by  $o(H)$  is  $h$ , and the defining length of the schema, denoted by  $\delta(H)$  is  $(i_h - i_1)$ . Holland [Hol75] derived a lower bound on the growth of the proportion of strings that adhere to a certain schema over a single generation, when using the canonical genetic algorithm (a generational genetic algorithm with fitness proportional selection, one-point crossover, and a low probability of mutation):

$$m(H, t+1) \geq m(H, t) \frac{f(H)}{\bar{f}} \left[ 1 - p_c \frac{\delta(H)}{l-1} - o(H)p_m \right].$$

This inequality is called the schema theorem [Gol89b]. The proportion of strings that contain schema  $H$  in generation  $t$  is given by  $m(H, t)$ ,  $f(H)$  denotes the (observed) schema fitness of schema  $H$ ,  $\bar{f}$  denotes the average (observed) schema fitness,  $p_c$  denotes the probability that recombination is applied, and  $p_m$  gives the probability that a bit is mutated. The right-hand side of the equation contains three factors. The first factor is the proportion of instances of the schema in the previous generation, the second factor gives the growth of the schema due to fitness proportional selection, and the third factor (within the square brackets) gives the probability that the schema  $H$  is not disrupted by the genetic operators. Within the square brackets, the second term gives the probability of disruption by a one-point crossover, and the third term gives the probability of disruption by means of mutation. For a derivation of the schema theorem, see Holland [Hol75, Hol95], Goldberg [Gol89b], or Mitchel [Mit96].

The schema theorem implies that up to  $3^l$  schemata are processed simultaneously. This is called the implicit parallelism of the genetic algorithm. In older work this effect is called intrinsic parallelism. However, currently one discriminates between these two, and intrinsic parallelism implies that GA's are relatively easy to parallelize, as the different individuals in the population can be evaluated independently from each other [Hol92].

## 2.5 Forma analysis

Radcliff showed that implicit parallelism has application areas beyond the binary domain by introducing formae and general operators that manipulate these formae [Rad91b, Rad91a]. A forma describes a combination of more elementary properties that are present within an individual. In fact, if each of the different properties of an individual are encoded in a single two-valued bit, then the set of all formae that an individual is a member of, corresponds to the set of all schemata that an individual is a member of. Formae have been applied to set-based representations [Rad93], to representations for the travelling salesperson problem (TSP) [Rad91b, RS95], and to real-valued representations [RS97].

Given these formae one can take a look at properties of recombination operators. A recombination operator is said to be respectful if offspring are always members of all formae present in all of its parents. The  $n$ -point crossover, uniform crossover, and multi-parent crossover operators are respectful operators in the binary domain.

A recombination operator is said to properly assort a set of formae if the offspring can become a member of an arbitrary combination of compatible formae that either of the parents is a member of. It can easily be seen that that one-point crossover does not properly assort. Given the parent strings 111 and 000 one can not obtain the string 010 in one step by means of one-point crossover. So the formae 0##, #1#, and ##0 that are present in one of the parents can not be mixed in one step. Uniform crossover properly assort in this case. A recombination is said to weakly assort if assortment can be obtained by means of a repeated incestuous recombination. One-point crossover does have this weak assortment property.

Radcliff introduces the random, respectful recombination  $R^3$  [Rad91b], which is an operator that produces an offspring by means of selection of uniform selections amongst all possible offspring that belong to the similarity set of the two parents, where this similarity set is defined as the highest precision forma which contains both of the parents.

## 2.6 Building blocks and deception

Goldberg [Gol89b] introduces the notion of a building block in a qualitative manner. He states that building blocks are short, low order schemata with high average fitness, such that near-optimal solutions can be constructed by means of the composition of a set of such building blocks.

Here a more specific definition of building block is used. We first give some additional definitions. Let the fitness  $f(s)$  of schema  $s$  be defined as the average fitness over all possible instantiations of this schema, and let  $d(s)$  denote the set of defined positions of the schema  $s$ . We define a schema  $s$  to be a sub-schema of schema  $t$  if  $d(s) \subseteq d(t)$  and  $\forall i \in d(s) : s_i = t_i$ . A partition is a string over the alphabet  $\{f, \#\}$ . A partition that contains  $k$   $f$ -symbols divides the space of strings in  $2^k$  parts, where each part is labelled by the corresponding schema. For example, the partition  $\#f\#f\#\#$  corresponds to the partition where the parts are labelled by the four schemata  $\#0\#0\#\#$ ,  $\#0\#1\#\#$ ,

#1#0##, and #1#1##. Now, we define a first-order building block as the order-one schema containing the optimal bit for a locus. Higher-order building blocks are given by schemata that adhere to the following two conditions. First, the schema needs to have the highest possible schema-fitness within its partition, and second the schema should not be fully covered by sub-schemata that form a building block. This definition is based on the idea that the building blocks are given by high-performance schemata (first condition) that have to be processed in one piece (second condition). If a schema is fully covered by lower-order building blocks, then this schema is not considered to be a building block because instances of this schema are likely to be generated due to the composition of the lower-order building blocks. Note that we do not require a building block to be represented by a short schema, this in contrast to the description given by Goldberg [Gol89b].

Two schemata  $s$  and  $t$  are called non-overlapping if  $d(s) \cap d(t) = \emptyset$ , and we call two schemata compatible if  $\forall i \in d(s) \cap d(t) : s_i = t_i$ .

In many optimization problems the linkage of loci is unknown; Therefore, we would like to find highly fit schemata independent of the length of these schemata. In a completely linear problem the largest possible building block is of order one, because in case of linear problems one only has additive interactions. Only in case of problems involving epistasis, the building blocks can become of a higher order. A problem involving epistasis does not have to be difficult. However, such a problem becomes difficult when there is some deception in the problem. Deception is present when lower order interactions are such that at some loci non-optimal bit-values are preferred.

A specific class of epistatic problems is the class of fully deceptive problems [Gol89a, DG93, DG94]. A partition of order  $k$  is fully deceptive, when all lower order schemata are such that the search is led to the bit-wise complement of the optimum. For example, in case of a three-bit optimum #111#, the problem is deceptive if the following twelve conditions hold:

$$\begin{array}{ll} f(\#0####) > f(\#1####) & f(\#11##) > f(\#01##), f(\#10##), f(\#00##) \\ f(\###0##) > f(\###1##) & f(\###11#) > f(\###01#), f(\###10#), f(\###00#) \\ f(\##0#) > f(\##1#) & f(\#1#1#) > f(\#0#1#), f(\#1#0#), f(\#0#0#). \end{array}$$

So, when using first-order and second-order statistics only, it seems that a value of 0 is the preferred value for all three defined positions. However, when observing the fitness of the complete building block #111#, it is clear that a value of 1 should be used at these loci. So the genetic algorithm has to discover instances of the schema #111# in order to assess the schema fitness properly. Deceptive partitions of order three usually do not cause a problem to genetic algorithms, because one out of eight randomly generated individuals is an instance of such a low order schema. Of course, if the global optimum contains many deceptive partitions of order three, then the problem can become difficult. Deception of higher order is more difficult to handle.

An example of a fully deceptive function is the order- $k$  trap-function [DG93]. The fitness contribution for a set of  $k$  bits that together compose a building block is calculated



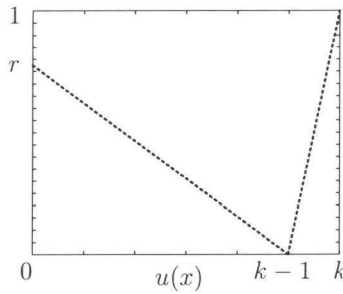


Figure 2.4: Fitness calculation for a single building block. The function  $u(x)$  counts the number of 1-bits in the string  $x$ .

according to the following formula,

$$f(x) = \begin{cases} 1 & \text{if } u(x) = k \\ r^{\frac{k-1-u(x)}{k-1}} & \text{otherwise} \end{cases}$$

where  $u(x)$  counts the number of 1-bits in string  $x$ , and  $r < 1$  denotes the fitness ratio between the optimal and the suboptimal solution. An example is shown in Figure 2.4. The horizontal axis represents the number of 1-bits in the bit-string, and the vertical axis denotes the fitness contribution. The optimal string contains  $k$  1-bits, and the deceptive attractor is the string containing only 0-bits. This problem is deceptive (given that  $r$  is not too small). For all the  $(2^k - 1)$  non-optimal strings the fitness is increased when the number of 1-bits is decreased. Only when processing schemata of order  $k$ , one is able to detect the optimum. This is a necessary, but not sufficient condition for a function to be fully deceptive.

There has been criticism on the assumption that deception is the main thing that makes a problem difficult to solve for a GA [Gre93]. Non-deceptive functions were constructed, that were in fact difficult to solve. These examples exploited the fact that the definition of deception is based on schema fitness. The behaviour of a GA is guided by the observed schema fitness, where these observations are based on the individuals in the current population [Gre93]. For a GA with an infinite population the observed schema fitness will correspond to the actual schema fitness, but for finite population GA's these two can differ. This discrepancy between observed schema fitness and the real schema fitness is important to consider. For example, let us take a look at the deceptive trap-function defined earlier in this section. If we take  $r < 2^{1-k}$ , then this problem is without any deception. The optimal schema containing 1-bits only has such a high fitness that  $f(\#1\#\#\#) > f(\#0\#\#\#)$ ; The same equation also holds for all other first-order schemata. However, in the case that the global optimal string is not present in the population, which is very likely for large values of  $k$ , the observed schema fitnesses still point towards the deceptive attractor, because an individual corresponding to the schema 11111 has to be present in the population in

order to get a reasonable estimate of the schema fitnesses. This example shows that it is important to keep in mind that real genetic algorithms are guided by the observed schema fitness and not by the real schema fitness.

Difficult problems typically involve some kind of deception. So, even though the formal definition of deception in terms of schema fitness does have its problems, the concept of deception in terms of lower-order statistics leading to suboptimal solutions, is important. If no deception is present, then the optimum can already be found by the first-order statistics, and therefore the optimum can be found easily and efficiently by means of a hill-climbing or by bit-wise optimization [DW91, Whi91]. The fully deceptive functions themselves are rather artificial, but these functions form an interesting test-bed for getting a better understanding about how evolutionary algorithms can properly delineate a problem, even when  $k - 1$  order interactions point away from the optimal solution. Such problems involving high-order interactions are likely to deceive local optimization methods. When only considering low-order interactions, the method likely converges to the deceptive attractor. In fully deceptive problems the optimal string is always bit-wise complement of the deceptive attractor. Therefore, if it is known that a problem is fully deceptive, then this problem can easily be optimized by locating this deceptive attractor and next complementing all bits, as noted by Grefenstette [Gre93]. He correctly notes that the more usual cases might correspond to “partially deceptive” problems.

## 2.7 Genetic hitch-hiking

The selection scheme makes multiple copies of parts of individuals that have a relatively high fitness. The high performance of an individual can be due to a small building block in the individual. Each time an individual is selected, a complete copy of the individual is made and hence other parts of the individual that do not contribute to the fitness are duplicated too. So, the schemata that do not contribute to the fitness can get many copies when combined with a highly fit schema. This effect is called genetic hitch-hiking, because a not-so-fit schema gets a lift from the highly fit schema.

Genetic hitch-hiking is a result of a high selective pressure, combined with a recombination that has a low disruptiveness. A high selective pressure results in many duplicates of the best performing individuals and therefore in many opportunities for some schemata to get a ride by a well-performing schema. A non-disruptive crossover, such as  $n$ -point crossover for low values of  $n$ , is sensitive to genetic hitch-hiking. If the joint schema of the low-performance and the high-performance schema is of short length, then the hitch can take quite a few generations. Highly disruptive recombination operators, such as the unbiased uniform crossover, strongly reduce the effect of genetic hitch-hiking.

## 2.8 Genetic drift

Genetic drift is the convergence of genes without the presence of selective pressure. Genetic drift is clearly present in a genetic algorithm with uniform selection of parents and recombination as the only evolutionary operator. Once an allele is lost, it is never rediscovered, because mutation is needed to introduce the alleles that are lost. Hence, in such a genetic algorithm each locus converges to either a one-value or a zero value. This effect is called genetic drift. It has been shown by Goldberg and Segrest [GS87] and later by Asoh and Mühlenbein [AM94] that the convergence time due to genetic drift is proportional to the population size.

In practice, one will never use a genetic algorithm without selective pressure; However, it can easily be the case that no selective pressure is induced for certain loci. This can happen for many problems, where some loci have a much stronger influence on the fitness than other loci. In such a case a genetic algorithm is mainly busy to find good values for the important loci. The less important loci have no effect on the fertility or viability of the individual, and therefore genetic drift can appear for these loci. An interesting example is the domino-convergence observed on the BinInt problem [TGP98]. The BinInt problem is defined as:

$$f(x) = \sum_{i=1}^l x_i 2^{l-i} \quad x_i \in \{0, 1\}.$$

The marginal fitness contribution of the loci decreases exponentially with  $i$ . Therefore, the value of locus  $i$  has almost no influence on the selection probability of an individual when the population has not converged to a unique value for all  $j < i$ . Thus, the loci with low marginal fitness contributions tend to drift and settle for a random bit-value.

## 2.9 Stochastic sampling errors

Stochastic sampling errors influence the behaviour of an evolutionary algorithm. These errors are one of the important reasons why real evolutionary algorithms with a finite population behave differently from the ideal evolutionary algorithms with infinite population size. These sampling errors can have either a positive or a negative effect. If the ideal evolutionary algorithm converges to the optimum, then a real evolutionary algorithm can converge to a suboptimum due to these errors. However, if the ideal evolutionary algorithm converges to a suboptimum, then the real evolutionary algorithm might be able to locate the optimum due to the sampling errors. In the first case the sampling errors are a negative influence, however in the second case effect of the sampling errors can have a positive influence. Sampling errors can appear during the selection step and during the recombination step. Both types of sampling errors are discussed here.

Due to sampling errors during selection, the actual number of times an individual is selected for reproduction can deviate from the expected value of this number. Let

$$\gamma_i = \frac{\text{actual number of copies of individual } i}{\text{expected number of copies of individual } i}.$$

The numerator of this ratio is an integer value in the range  $0, \dots, N$ , where  $N$  is the population size. If the sampling errors are small, then  $\gamma_i \approx 1$ . If the expected number of copies of the individual is large, then this value is likely to be close to one. Therefore, an increase of the population size helps in reducing the sampling errors due to selection. Furthermore, one can use variants of the standard selection methods. An example is the "Stochastic Universal Sampling", which can replace the standard roulette wheel selection. Another method that can help in reducing the sampling error during selection is the use of recombination operators that produce only one offspring. For all recombination operators described in section 2.2, the number of offspring is equal to the number of parents. An operator that produces one offspring is obtained by discarding all offspring except for the first offspring. If the recombination operator produces only one offspring instead of two, then the selection step has to be performed twice as often. For the ratio  $\gamma_i$  this means that the numerator is in the range  $0, \dots, 2N$ , and the denominator becomes twice as large. The numerator is still restricted to the integer values, but as the range is twice as large, the ratio is likely to be closer to the ideal value  $\gamma_i = 1$ . For the usual recombination operators for bit-strings the number of offspring is equal to the number of parents. These operators are defined such that all bit-values of each of the parents are transferred exactly once to one of the offspring. So the frequency of one bits for a specific locus, does not change due to the application of such a recombination operator. This can be an advantage during analysis, because only mutation and selection can change the frequency of one-bits in that case. Another advantage is present when the building blocks and recombination operator are tuned to each other, such that the recombination is unlikely to disrupt building blocks. In that case the building blocks that are present in any of the parents are likely to be transferred to one of the offspring. If only one offspring is produced, then the building block is only transferred with a certain probability  $p$ , which leads to another step where sampling errors can appear. When using highly disruptive recombination operators, like the uniform crossover, this advantage is not present anymore because the probability that a building block is propagated is small anyway.

Sampling errors are also present during the application of the evolutionary operators. Even when the individuals that contain building blocks are selected in the appropriate proportions, this does not mean that the building blocks are propagated to the next generation. During the recombination step sampling errors are present too. Due to these sampling errors, the actual number of building blocks in the offspring population and the expected number of building blocks in this population can differ.

Let us assume that the selection of parents appeared in exactly the right proportions. Next, the parents have to be partitioned such that tuples of  $p$  elements are generated, where  $p$  is the arity of the recombination operator. The actual partition of the population that is chosen influences the expected distribution of the offspring. Only if parents contain different (but compatible) building blocks, then a successful mixing event can be obtained. When using large populations, the sampling errors introduced by the choice of the actual partition will be cancelled out. However in case of small populations, the actual partition that is selected can strongly influence the distribution of the offspring.

## 2.10 Balancing exploration and exploitation

Genetic algorithms used for optimization purposes have to do both exploration and exploitation. The exploration phase involves an unstructured search. An analysis of the search space is performed and (unknown) structures in the search space are uncovered. During the exploitation phase the discovered structures are used to construct good solutions. In genetic algorithms exploration and exploitation are performed simultaneously, but the balance between exploration and exploitation varies in time. At the start of the search the GA focuses primarily on exploration. In later generations, the balance usually moves towards exploitation. A successful application of genetic algorithms requires a good balance between exploration and exploitation. A GA that sticks to exploration too long is not able to construct good solutions. Such a GA will mainly do a kind of an unstructured random search over the complete search space. A GA that puts too much emphasis on exploitation does not have the time to discover enough structure in the search space and therefore does not get enough information for an effective construction of good solutions. A GA that makes a fast transition to exploitation usually converges fast to a suboptimal solution (premature convergence). How fast the balance moves from exploration to exploitation is dependent upon the GA parameters and the problem being tackled. Hence, problem-specific parameter tuning is often required.

A GA is robust when no extensive fine-tuning of GA-parameters is needed in order to obtain convergence to the optimal solution. So there is a broad range of parameter settings such that convergence to the optimal solution is obtained. Fine tuning of parameters might result in a faster convergence, but the speed of convergence is not the most important issue when considering robustness.

Robustness of GA's was investigated by means of a dimensional analysis by Deb, Goldberg and Thierens [GDT93, Thi95, Thi96]. A theoretical model for a problem involving the mixing of two  $k$ -order building blocks was generated and validated by experiments [Thi95]. Figure 2.5 shows the influence of the selective pressure and the crossover probability on the probability of convergence to the optimal solution. A (generational) GA will only be able to find the optimum reliably when the parameters are chosen such that the corresponding point is located in the interior region (also called the sweet-spot) given by these three constraining boundaries. These three boundaries correspond to the following processes:

1. too high selective pressure results in cross-competition between building blocks (vertical line),
2. too little recombination means that building blocks's are not mixed (mixing boundary),
3. too little selective pressure means that selection cannot compensate for the disruption of building blocks by the recombination operator (selection boundary).

The boundaries 2 and 3 shift upwards when the selective pressure increases. Details about the actual model can be obtained from the original publication [Thi95]. Cross-competition

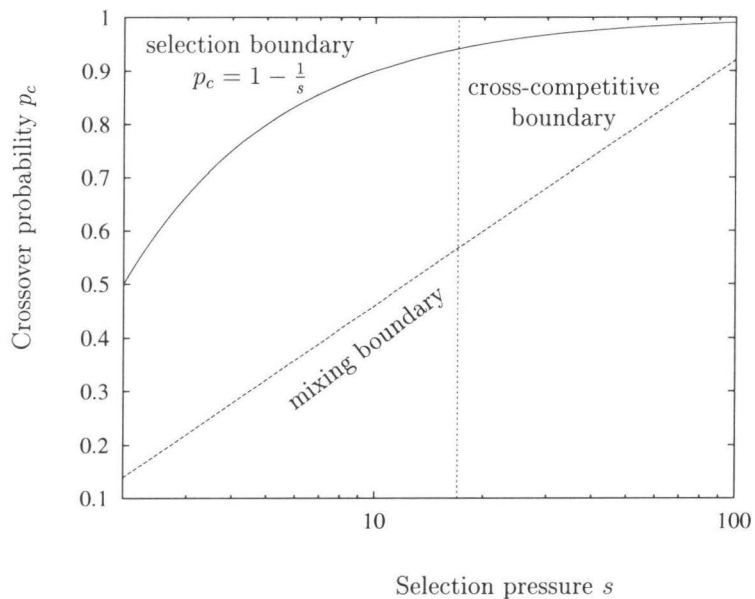


Figure 2.5: Dimensional analysis for problem involving the mixing of two order- $k$  building blocks

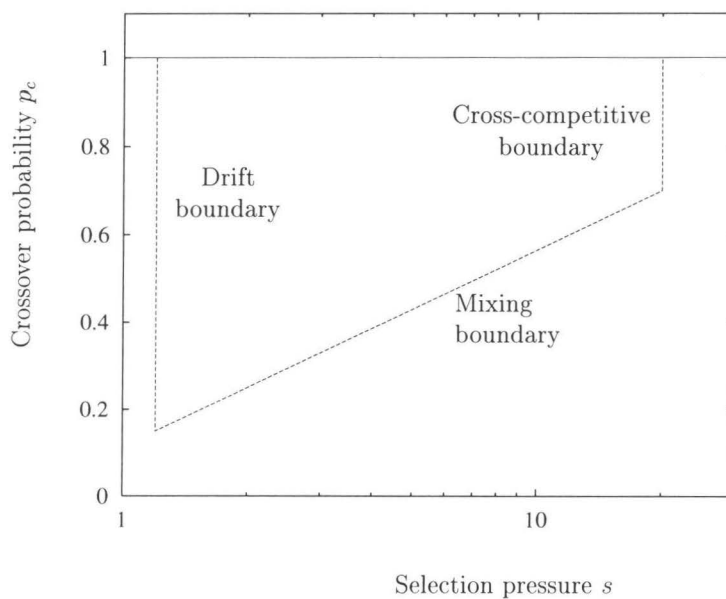


Figure 2.6: Dimensional analysis for problem involving the mixing in the oneMax problem

is discussed in section 3.3. A robust GA will have a large region enclosed by these three lines. Tuning of parameters becomes more important when this region gets smaller. The region of reliable convergence vanishes when for example the selection boundary falls below the mixing boundary. In that case there is no parameter setting where we get a reliable convergence of the GA. In that case it will always be a matter of chance whether the optimal solution is found or not.

A similar analysis was applied to the oneMax problem [GDT93, Thi96]. Figure 2.6 taken from [GDT93] shows the corresponding control map. Just like the previous map in figure 2.5, this map contains a mixing boundary and a cross-competitive boundary. This map does not have a selection boundary, because all bits are independent, so crossover cannot disrupt building blocks. On the left the drift boundary is shown. If the selective pressure is too low, then genetic drift dominates, and the GA does not converge to the optimum anymore.

## 2.11 NK-landscapes

The NK-landscapes have been introduced by Kauffman [Kau93]. These are quasi-random landscapes that take a bit-string as input and compute the fitness value. The fitness of a string is given by the average fitness contribution over all loci in the bit-string. The fitness contribution of a locus  $i$  is determined by a random function  $f_i(\cdot)$ . This function uses a bit-string of length  $(k+1)$  that is obtained by concatenating the value  $x_i$  of locus  $i$  and the bit-string  $N_i$  that contains the values of  $k$  other loci that are in a neighbourhood of locus  $i$ . Both the random function  $f_i(\cdot)$  and the neighbourhood  $N_i$  are chosen independently for each of the loci. Two types of NK-landscape exist based on the way the neighbours are selected. The first type uses nearest-neighbour interaction: each locus is dependent on its  $k$  nearest other loci. Hence, the expected correlation between loci decreases with distance. The second type of NK-landscape uses a randomly selected neighbourhood. In the case the expected correlation between loci is independent of their relative position on the bit-string and loci that are far apart will usually be stronger correlated than in case of the nearest-neighbour interaction.

The fitness of a NK-landscape is given by the formula

$$f_{\text{NK}}(\vec{x}) = \sum_{i=1}^l f_i(x_i \cdot N_i),$$

where  $f_i$  is a random function,  $x_i$  denotes the value of locus  $i$  of individual  $x$ ,  $N_i$  is the bit-string containing the values of the loci that form the neighbourhood of bit  $i$ ,  $\cdot$  is the concatenation operator, and  $k$  is a parameter of the landscape that takes a value in the range 0 to  $(l-1)$ . The function  $f_i : I \rightarrow [0, 1]$ , where  $I$  denotes the integer range from 0 to  $(2^{k+1} - 1)$ ; So, this is a random function that maps a  $(k+1)$ -bits integer to a real value between zero and one.

## Chapter 3

# Models of Building block processing

In this section some simple models of building block processing are introduced. These models are used to gain intuition about the behaviour of a genetic algorithm on simple problems. The efficiency of the uniform crossover operator is studied in section 3.1. Section 3.2 investigates the influence of the bias-parameter of uniform crossover on the mixing probabilities. Cross-competition between non-overlapping building blocks can influence the reliability of evolutionary algorithms. Section 3.3 shows this on a simple problem involving cross-competition. The power of a recombination based evolutionary algorithm is that building blocks can be discovered separately, and combined afterwards by means of recombination. It is also possible that an optimal building block is generated, which means that the offspring contains the building block, while neither of the parents contained it. In section 3.4 the balance between these two processes (mixing and generation of building blocks) is investigated.

### 3.1 Efficiency of the traditional crossover operators

A crossover operator is a  $n$ -ary operator that takes  $n$  individuals as an input, and produces a number of offspring individuals. The primary advantage of a  $n$ -ary crossover operator over the unary mutation operator is that the crossover mixes parts of its parent individuals to create the offspring, and therefore is able to mix partial solutions that have been found independently of each other. If the optimal parts of the different parents are mixed, then a superior offspring can be produced. Crossover is efficient when the structure exploited by the crossover matches the structures present in the search space. Which crossover operator performs best depends among other things on the linkage of the loci that correspond to the same building block.

In most practical applications the linkage of the bits is not known beforehand. Therefore, we focus here on problem with unknown linkage. The uniform crossover operator is used because this operator does not make any assumptions about the linkage of loci.

Next, a simple problem is introduced, and this problem is used to study the behaviour of GA's with uniform crossover.



Let us assume that we have a problem where the optimal solution is partitioned in a set of building blocks. A partition consists of a set of non-overlapping blocks. Let the problem have the following properties:

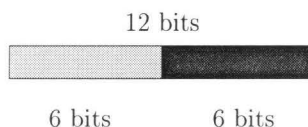
- linkage is unknown, thus the loci of a building block can be scattered arbitrarily over the bit-string,
- $f(x) = \sum f_i(x_i)$ , where index  $i$  runs over all building blocks and  $x_i$  is bit-string  $x$  limited to the loci that belong to building block  $i$ , and

$$f_i(x_i) = \begin{cases} 2 & \text{if } x_i = x_i^{(opt)} \\ \text{lookup}(x_i) \in U[0, 1] & \text{otherwise} \end{cases}$$

where  $\text{lookup}(x_i)$  denotes a lookup table of size  $(2^{|x_i|} - 1)$  with the elements drawn randomly from the range  $[0, 1]$ . So the only structure is given by the optimal building block  $x^{(opt)}$ . The actual value of this optimal block is assumed to be unknown.

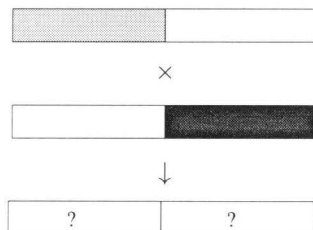
Moreover, let us assume that two parents are picked from the current population, and that one parent contains building block  $i$ , while the other parent does not contain this building block. Within the part corresponding to the building block, the second parent is expected to match the first parent for approximately half of the loci. If a single offspring is generated by uniform crossover, then the probability that this offspring contains the building block is approximately  $0.5^{(k/2)}$ . Here  $k$  denotes the order of the building block. So, the probability that a building block is preserved under crossover decreases exponentially in the order of the building block.

Next, the mixing of two building blocks is investigated. A simple problem consisting of a concatenation of two building blocks of order six is used. A single individual can be represented by the following picture.

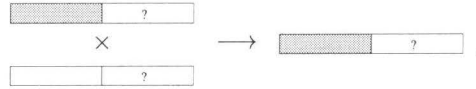


The left-hand part of the string is optimal when it is coloured light-grey, and the right-hand part is optimal when it is coloured dark-grey. Non-optimal parts are coloured white. Only during visualization it is assumed that the loci corresponding to a single part are adjacent. During the analysis no assumptions are made with respect to the linkage of the loci.

Let us examine the following application of uniform crossover:



The first parent contains the optimal building block in the left-hand part of the string and a non-optimal solution in the right-hand part. For the second parent it is the other way round. Let  $x_i$  denote the schema corresponding to optimal building block for part  $i$ , so  $x_1$  represents the optimal building block for the left-hand part, and  $x_2$  represents the optimal building block for the right-hand part. In the case that the optimal building block, denoted by  $x_i$ , is not present, let  $p_i^{(opt)}$  denote the average probability that a locus defined by  $x_i$  contains the corresponding bit-value. Note that  $p_i^{(opt)}$  only applies to the individuals that do not contain  $x_i$ . The probability of transferring each of the two building blocks can be computed separately. First, the transfer of the left-hand building block is examined. This is denoted by



So, given that exactly one of the parent individuals contains the optimal building block in the left-hand part, what is the probability that the offspring will contain this optimal building block too? The probability that building block  $x_i$  is transferred to the offspring is given by:



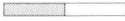
$$P_{tr}(x, p^{(opt)}) = \left( p^{(opt)} + (1 - p^{(opt)}) \frac{1}{2} \right)^{order(x)}.$$



Here, the first term corresponds to the case that the second individual contains the optimal bit-value at a specific locus, and the second term corresponds to the case that the optimal value is not present at the given locus. To validate the formula of  $P_{tr}(x, p^{(opt)})$  the full computation for an example, involving a building block of order six and  $p^{(opt)} = \frac{1}{2}$ , is shown.

| $i$ | $\text{Bin}(6, i, 0.5)$ | $P_{tr} = (\frac{1}{2})^{6-i}$ | $P$         |
|-----|-------------------------|--------------------------------|-------------|
| 0   | $\frac{1}{64}$          | $(\frac{1}{2})^6$              | 0.000244141 |
| 1   | $\frac{6}{64}$          | $(\frac{1}{2})^5$              | 0.002929688 |
| 2   | $\frac{15}{64}$         | $(\frac{1}{2})^4$              | 0.014648438 |
| 3   | $\frac{20}{64}$         | $(\frac{1}{2})^3$              | 0.0390625   |
| 4   | $\frac{15}{64}$         | $(\frac{1}{2})^2$              | 0.05859375  |
| 5   | $\frac{6}{64}$          | $(\frac{1}{2})^1$              | 0.046875    |
| 6   | $\frac{1}{64}$          | $(\frac{1}{2})^0$              | 0.015625    |
|     | 1                       |                                | 0.177978516 |

The first column gives the number of loci where the second parent contains the optimal bit-value, the second column gives the probability that the second parent contains this number of optimal bits (computed by means of the binomial distribution), the third column gives the probability that the offspring contains the building block in the corresponding case, and the fourth column shows the actual probability of the corresponding case. The sum over the fourth column corresponds to the value given by the formula, i.e.  $(3/4)^6$ . Note that a quick guess given by formula  $(1/2)^{6/2}$ , corresponding to the observation that approximately

half of the bits are not correct, gives an incorrect value for the probability of the event  $((1/2)^{6/2} = 0.125$  while the correct value is approximately 0.18).

So, if we assume that  $p^{(opt)} = \frac{1}{2}$ , then the probability that the building block is propagated is approximately 0.18. So, the probability that the process   $\times$   generates an offspring of type  is approximately 0.15  $(= (\frac{3}{4})^6(1 - (\frac{3}{4})^6))$ .

Now, let us return to the original question, what kind of offspring can we expect when mating two parents that contain different building blocks? Due to the symmetry in the problem and the recombination operator, the probability that an offspring of type  is obtained is 0.15 too. The probability of an optimal offspring  is only 0.032. So, the probability that either of the two already existing building blocks is propagated to the offspring is only 0.3, and the probability that these two building blocks are actually mixed is much smaller.

Given the small probability that the building block survives crossover the GA should either have:

- a low crossover probability (mutation is less likely to disrupt the building block),
- a high selective pressure (such that many duplicates of the fittest individuals are produced), or
- some form of elitism, such that well-performing parents are able to survive and therefore get more opportunities to transfer and mix building blocks.

Another possibility would be to introduce a more effective crossover operator that incorporates some knowledge about the linkage of the loci, and is therefore better able to transfer building blocks.

## 3.2 Mixing of building blocks by uniform crossover

In the previous section the transfer probability of a building block was studied for unbiased uniform crossover. In this section the mixing of two building blocks by means of biased uniform crossover is investigated, and it is shown that the unbiased uniform crossover performs best if the two building blocks have approximately the same order.

Let us assume that we have two individuals that contain different, but compatible, building blocks. One individual contains a building block of size  $K$ , and the other contains a building block of size  $M$ . Now, let us assume that  $k$  values of the first building block are missing in the individual that does not contain this block, and  $m$  values of the second building block are missing in the other individual. Given that we do not know which individual contains which size of building block, the mixing probability is given by the formula

$$P_{mix}(k, m, p) = 0.5p^k(1 - p)^m + 0.5(1 - p)^k p^m,$$

where  $p$  is the bias-parameter of the uniform crossover operator. Figure 3.1 shows  $P_{mix}(k, m, p)$  for  $k = 3$  and  $m$  between 3 and 10. The corresponding curves are either

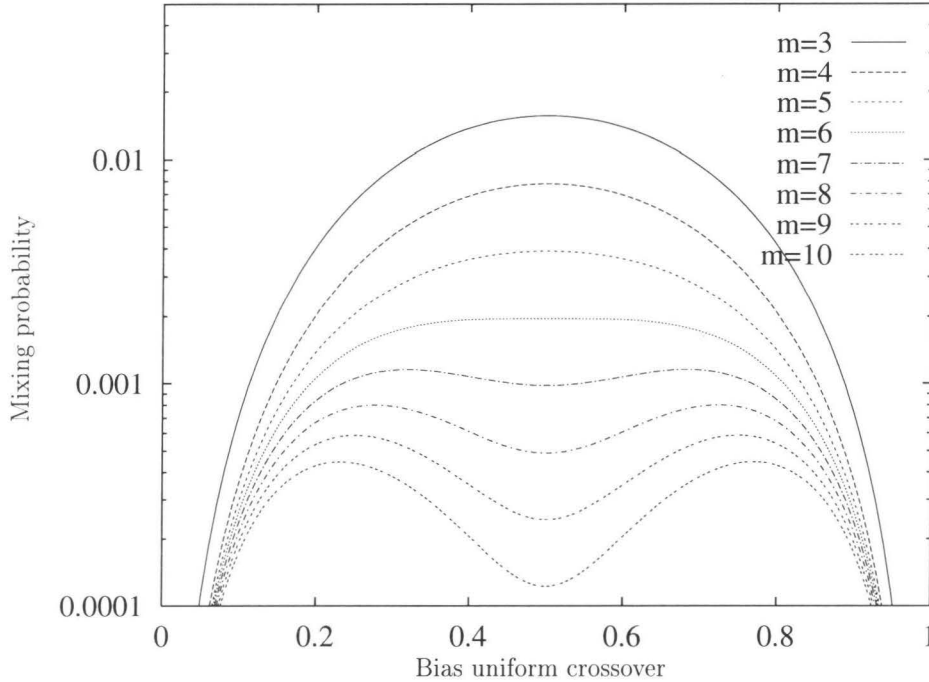


Figure 3.1: Probability of mixing two building blocks as a function of the bias-parameter of uniform crossover ( $k = 3$  and  $m = 3, \dots, 10$ ).

unimodal or bimodal. In the cases where the curves are unimodal the unbiased uniform crossover, with  $p = 0.5$ , performs best. In case of a bimodal curve a different value  $p$  performs best. Next, we investigate when a biased crossover performs best for different values of  $k$  and  $m$ . Due to symmetry considerations it suffices to show that the optimal value of  $p$  is larger than 0.5. Let us assume that  $m > k$ . Now, the derivative  $\partial P_{mix}(k, m, p)/\partial p$  is computed and evaluated at  $p = 0.5 + \epsilon$ , where  $\epsilon$  is a small positive value. If the value of this partial derivative is positive, then the optimal value of  $p$  is larger than  $0.5 + \epsilon$ , and therefore biased uniform crossover performs better than unbiased uniform crossover. Figure 3.2 shows the curves for different values of  $k$ . The curves for  $k = 2, \dots, 8$  cross the zero-line at 2.3, 2, 1.8, 1.7, 1.7, 1.6, and 1.6 respectively. These numbers indicate when biased uniform starts to outperform unbiased uniform crossover. So, in the case that  $k = 3$  an biased uniform crossover using a value  $p \neq 0.5$  starts to outperform unbiased uniform crossover when  $m/k \geq 2$ , which corresponds to  $m \geq 6$ . In Figure 3.1 we see that the curve for  $m = 6$  a value  $p = 0.5$  is still close to optimal, and we observe that the biased crossover outperforms the unbiased crossover significantly for  $m \geq 8$  (The mixing probabil-

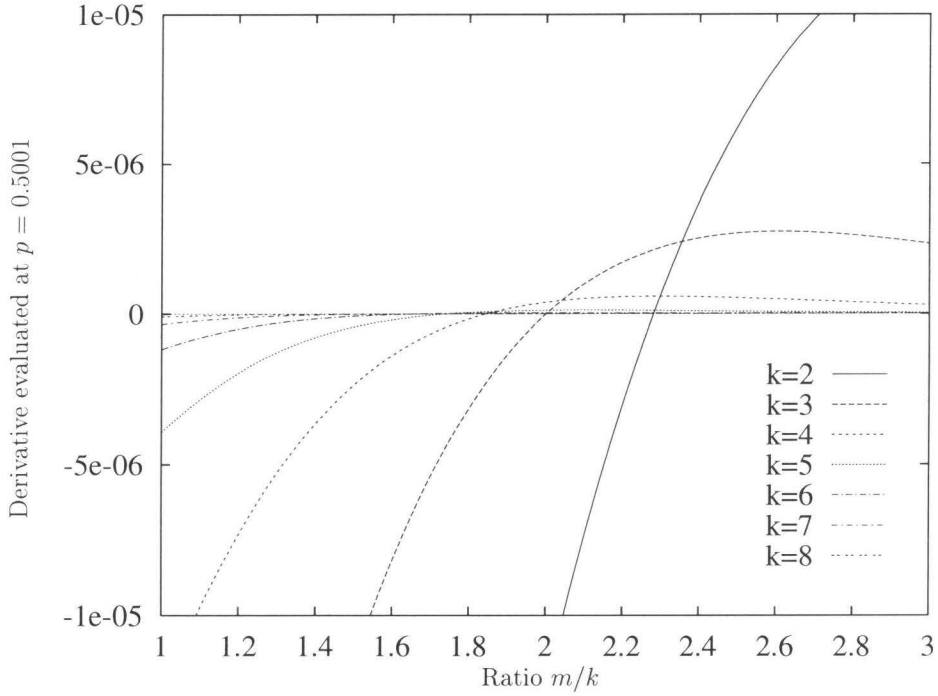


Figure 3.2: The derivative  $\frac{\partial}{\partial p} P_{mix}(k, m, p)$  as a function of the ratio  $m/k$  evaluated at  $p = 0.5001$ .

ity increases by 64% when taking  $p = 0.28$  instead of  $p = 0.5$ ). So, only when the optimal value of  $p$  is significantly larger than 0.5 the biased uniform significantly outperforms the uniform crossover.

Given these results one observes that uniform crossover with  $p = 0.5$  performs best unless the number of missing values of the blocks that have to be mixed differ a lot. Therefore, the value  $p = 0.5$  is used in the rest of this chapter.

### 3.3 Cross-competition between building blocks

Compatible building blocks do not compete directly with each other, because they do not conflict with respect to the defined positions. However, compatible building blocks can still be involved in an indirect competition, as long as these blocks are not mixed. If the individuals containing a certain building block take a larger proportion of the population, then this means that the proportion of the population that can be taken by individuals

containing other building blocks decreases. When the building blocks get mixed this type of cross-competition decreases, because an individual can contain both building blocks. So, as long as the building blocks are not mixed, additional copies of one building block will leave less opportunities for the other building block to get duplicates. This indirect competition is called cross-competition. The example problem given in the previous section is sensitive to cross-competition. This problem is studied here.

Assume that the following (intermediate) population is obtained after selection:

$$\begin{array}{cc} \text{[shaded] [white]} & p \\ \text{[white] [black]} & 1 - p \end{array}$$

So, a proportion  $p$  of the individuals contains an optimal building block of the first type, and the remaining proportion  $(1 - p)$  of the individuals contains an optimal building block of the second type. There are no optimal individuals containing both building blocks. After pairing of parent individuals, the following distribution is obtained:

$$\begin{array}{cc} \text{[shaded] [white]} \times \text{[shaded] [white]} & p^2 \\ \text{[shaded] [white]} \times \text{[white] [black]} & 2p(1 - p) \\ \text{[white] [black]} \times \text{[white] [black]} & (1 - p)^2 \end{array}$$

When computing the distribution of the population after recombination, one has to distinguish between the case that a building block is transferred, and the case that a building block is generated.

The transfer of a building block corresponds to the event

$$\begin{array}{ccc} \text{[shaded] [?]} & & \\ \times & \longrightarrow & \text{[shaded] [?]} \\ \text{[white] [?]} & & \end{array}$$

So, given that exactly one of the parent individuals contains the optimal building block in the left-hand part, what is the probability that the offspring also contains this building block? Using the equation derived in the previous section we get

$$r(i) = \left( p_i^{(opt)} + (1 - p_i^{(opt)}) \frac{1}{2} \right)^{order(x_i)}.$$

The probability of the event that the optimal building block of the left-hand part is transferred, as shown in the example above, is  $r(1)$ , the probability for the right-hand part is given by  $r(2)$ .

The generation of a building block corresponds to the event

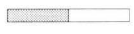






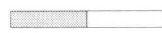

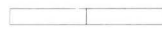
$$\begin{array}{ccc} \text{[white] [?]} & & \\ \times & \longrightarrow & \text{[shaded] [?]} \\ \text{[white] [?]} & & \end{array}$$

So, given that none of the parent individuals contain the optimal building block in the left-hand part, what is the probability that the optimal building block is generated in the offspring? This probability is given by

$$q(i) = \left( (p_i^{(opt)})^2 + p_i^{(opt)}(1 - p_i^{(opt)}) \right)^{order(x_i)}.$$

Here, the first term corresponds to the probability that both individuals contain the optimal bit-value at a locus, and the second term corresponds to the case where only one of the parents contains the optimal bit-value. If none of the parents contains the optimal bit-value, then the offspring cannot get this value either. The probability of the event that the optimal building block of the left-hand part is generated, as shown in the example above, is  $q(1)$ , and for the right-hand part the probability is given by  $q(2)$ .

The probability of obtaining each of the possible offspring given a specific pair of parents is given in the following table.

|   |  ×  |  ×  |  ×  |
|---|---|--|--|
|  | $q(2)$  | $r(1)r(2)$   | $q(1)$   |
|  | $1 - q(2)$  | $r(1) - r(1)r(2)$  | 0  |
|  | 0   | $r(2) - r(1)r(2)$  | $1 - q(1)$   |
|  | 0   | $(1 - r(1))(1 - r(2))$   | 0  |

Each column corresponds to a specific pair of parents, thus the probabilities of a column have to sum to 1.0. A row corresponds to a certain type of offspring. Given these probabilities, the distribution of the population after crossover can be computed.

$$\text{[shaded, shaded]} \quad p^2 q(2) + 2p(1-p)r(1)r(2) + (1-p)^2 q(1)$$

$$\text{[shaded, white]} \quad p^2(1 - q(2)) + 2p(1-p)(r(1) - r(1)r(2))$$

$$\text{[white, shaded]} \quad 2p(1-p)(r(2) - r(1)r(2)) + (1-p)^2(1 - q(1))$$

$$\text{[white, white]} \quad 2p(1-p)(1 - r(1))(1 - r(2))$$

It is interesting to observe the change of the ratio between the number of individuals that only contain the first building block, and the individuals that only contain the second building block. For the parents this ratio was  $p/(1-p)$ . For the offspring the ratio is:

$$\frac{\text{[shaded, white]}}{\text{[white, shaded]}} = \frac{p}{1-p} \frac{p(1 - q(2)) + 2(1-p)(r(1) - r(1)r(2))}{2p(r(2) - r(1)r(2)) + (1-p)(1 - q(1))}.$$

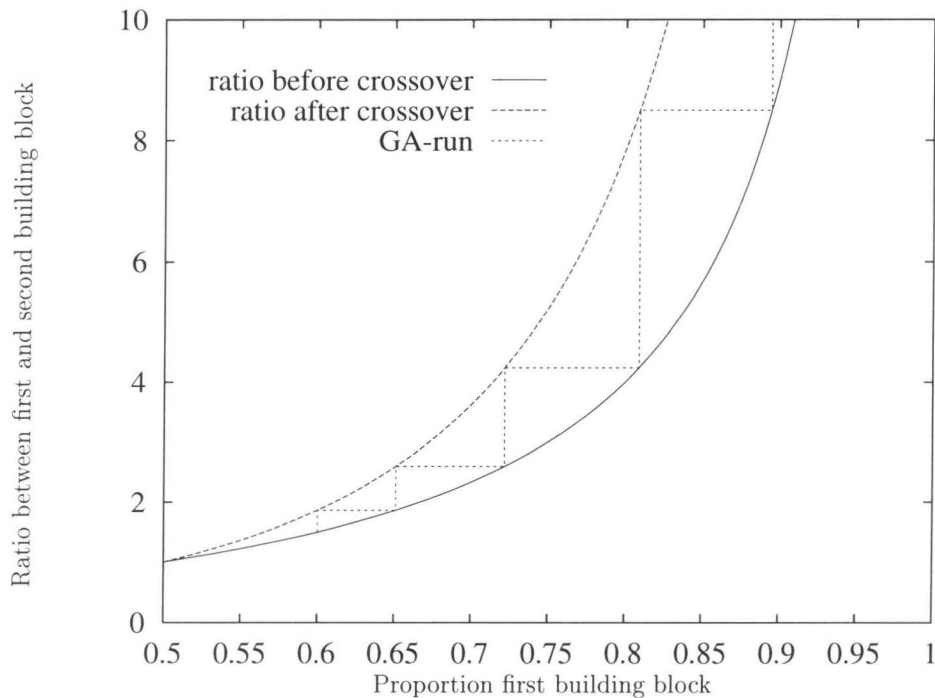




Figure 3.3: Evolution of the proportion of individuals containing the first building block when each of the individuals either contains the first of the second building block.

Consider the right-hand side of the equation. The left factor corresponds to the ratio in the parent population. The right factor denotes the change of ratio.

Let us assume that the following parent population after selection is given:

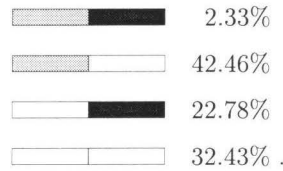
|   |     |
|---|-----|
|  | 60% |
|  | 40% |

For this parent population the distribution of pairs of parents is:

|   |     |
|---|-----|
|  ×  | 36% |
|  ×  | 48% |
|  ×  | 16% |

The expected distribution of the offspring population is





When comparing the distribution of the parents and the distribution of the offspring, a number of interesting observations can be made. First, it is clear that the crossover operator has some problems in transferring these building blocks. Even though all individuals in the parent population contain the building block, only 67.67% of the offspring contains at least one of the building blocks. Second, the probability of obtaining the optimal solution is quite small. Only 2.33% of the offspring corresponds to this optimal solution. Third, the ratio between the individuals containing only the first building block and the individuals containing only the second building block (i.e.  $\frac{\text{Black}}{\text{White}}$ ) changes from 1.5 to 1.86 approximately. This ratio will be preserved under selection because both building blocks have the same marginal fitness-contribution in the problem considered here.

Now, the evolution of a genetic algorithm on the example problem is modelled. Let us assume that a truncation selection is used, that retains the best  $2/3$  of the individuals (An arbitrary other selection scheme will give approximately the same results, because the current problem is constructed such that there are no lower-order statistics that result in a preference for either a 0 or a 1 value at any of the loci.) When using such a scheme the selection of an individual is mainly dependent upon whether it contains at least one building block. Figure 3.3 shows these curves, under the assumption that  $p^{(opt)} = \frac{1}{2}$  for all loci where the optimal building block is not present. The lower curve gives the ratio before application of crossover, the upper curve gives the ratio after crossover, and the dashed line between the two curves denotes the evolution of the genetic algorithm. Recall that 60% of the individuals in the initial population contained the first building block, so initially  $p = 0.6$ . During subsequent generations  $p$  becomes 0.6, 0.66, 0.75, 0.85. The corresponding ratios are 1.5, 1.94, 29.94, 5.64. The increase of this ratio implies that a genetic algorithm tends to converge to those building blocks that already appear relatively often in the population.

So, if one of the two building blocks is discovered before the other, or gets more copies due to some random effects, then the GA is likely to converge rapidly to the suboptimal solution containing this building block only. If all individuals of the population contain the first building block, then the search for the second building block starts again. This means that building blocks are discovered in sequence instead of in parallel. Furthermore, it is possible that the partition corresponding to the second building block has converged to a suboptimal solution, in which case the second building block is not likely to be found anymore. Thus, cross-competition can prevent the discovery of the global optimum. Note that the marginal fitness of the two building blocks is the same. So, the cross-competition between the building blocks shown here is an effect of different probabilities of disruption by recombination. This effect can be considered as a kind of genetic drift for higher order

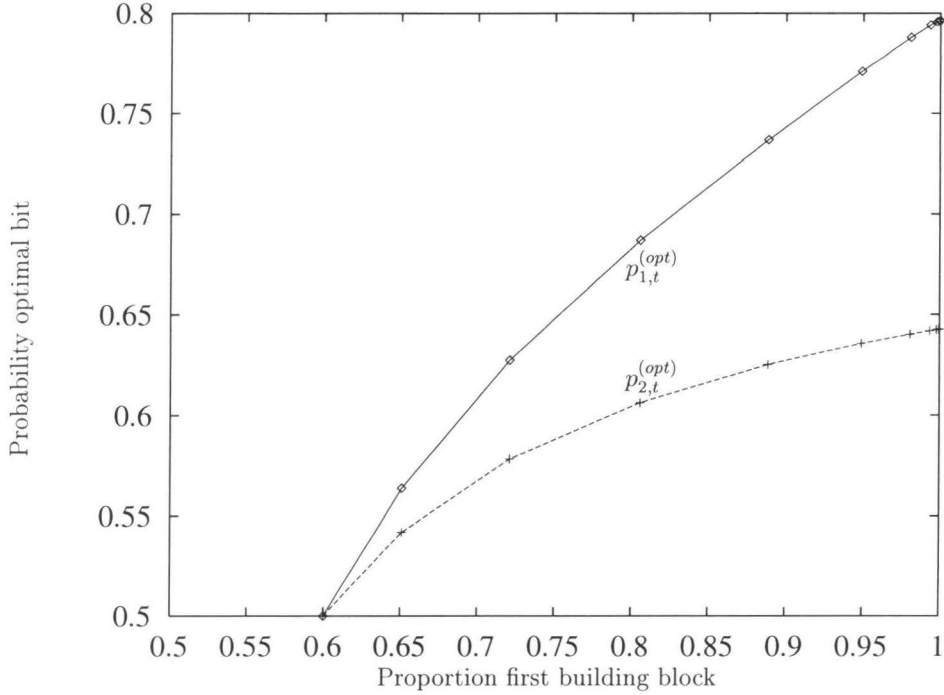


Figure 3.4: Evolution of the number of optimal bits when the building block is not present for both the left-hand part and the right-hand part.

building blocks. Recall that genetic drift results in convergence of loci to specific values even though there is no selective pressure.

### 3.4 Mixing vs. generation of building blocks

Building blocks can easily get lost due to cross-competition. If a building block gets lost, it has to be generated again. The probability that building blocks are generated is examined here.

The probability that an optimal building block is generated during crossover depends on the value of  $p_i^{(opt)}$ . This was defined as the probability of observing an optimal bit when the building block is not present (section 3.3). In the previous section, it was assumed that this value is fixed at  $p_i^{(opt)} = 0.5$  during the complete evolution. During a real GA run this value will vary. In this section the evolution of  $p_i^{(opt)}$  is included in the model.

Intuitively, it is clear that if the proportion of individuals that contain the optimal

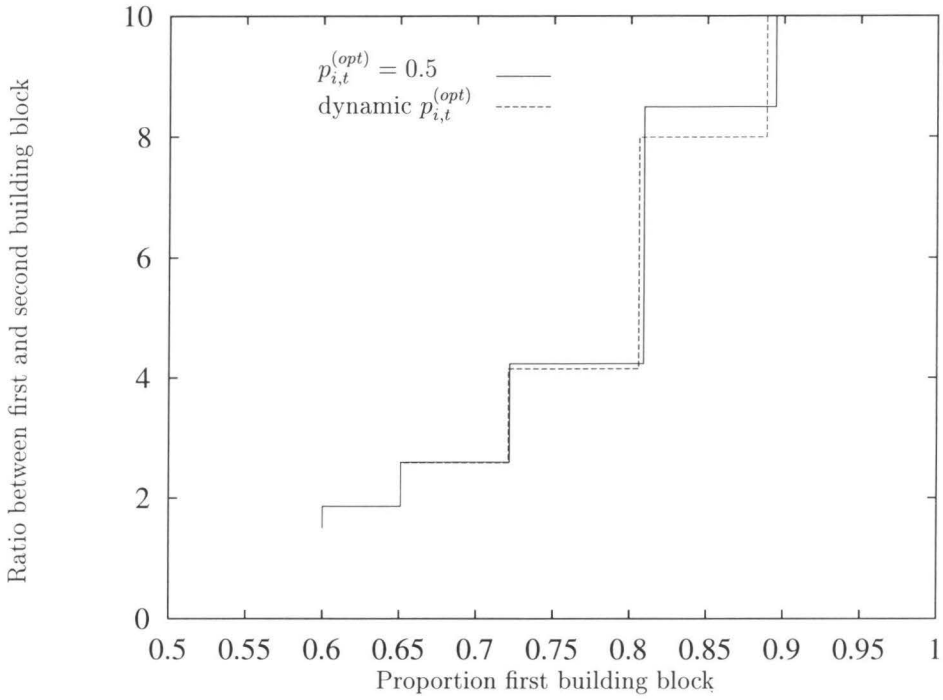



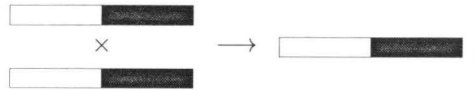


Figure 3.5: Evolution of the proportion of individuals containing the first building block when each of the individuals either contains the first of the second building block. The first curve shows the results when keeping  $p_{i,t}^{(opt)}$  is fixed at 0.5 and the second curve shows the results when updating  $p_{i,t}^{(opt)}$ .

building block in part  $i$  increases, then  $p_{i,t}^{(opt)}$  increases too.

During the computation of  $p_{i,t}^{(opt)}$  it is assumed that after selection only individuals of type  and  remain. So, if one is interested in  $p_{i,t}^{(opt)}$ , then one only has to observe the two processes that generate offspring of type . The first process corresponds to the event



Let us now assume that a proportion  $a$  of all parent pairs is of this type. The second

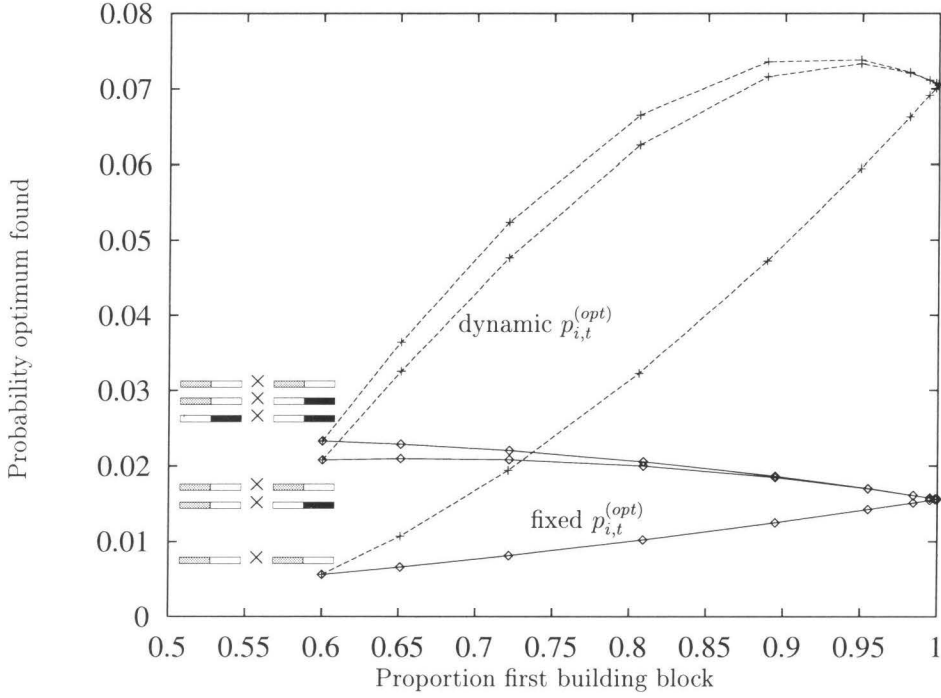
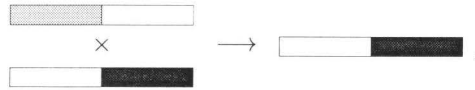


Figure 3.6: Probability of obtaining the optimal solution. The set of solid curves shows the results when keeping  $p_{i,t}^{(opt)}$  fixed at 0.5, and the set of dashed curves shows the results when updating  $p_{i,t}^{(opt)}$ .

process corresponds to the event



Let us assume that a proportion  $b$  of all parent pairs is of this type. Let the probabilities of observing an optimal bit for these two processes be denoted by  $p_a^{(opt)}$  and  $p_b^{(opt)}$ . If  $p_a^{(opt)}$  and  $p_b^{(opt)}$  are known, then  $p_{1,t+1}^{(opt)}$  is given by the following weighted sum

$$p_{1,t+1}^{(opt)} = \frac{ap_a^{(opt)} + bp_b^{(opt)}}{a + b}.$$

In case of the first process the probability of observing an optimal bit will not change, so

$$p_a^{(opt)} = p_{1,t}^{(opt)}.$$

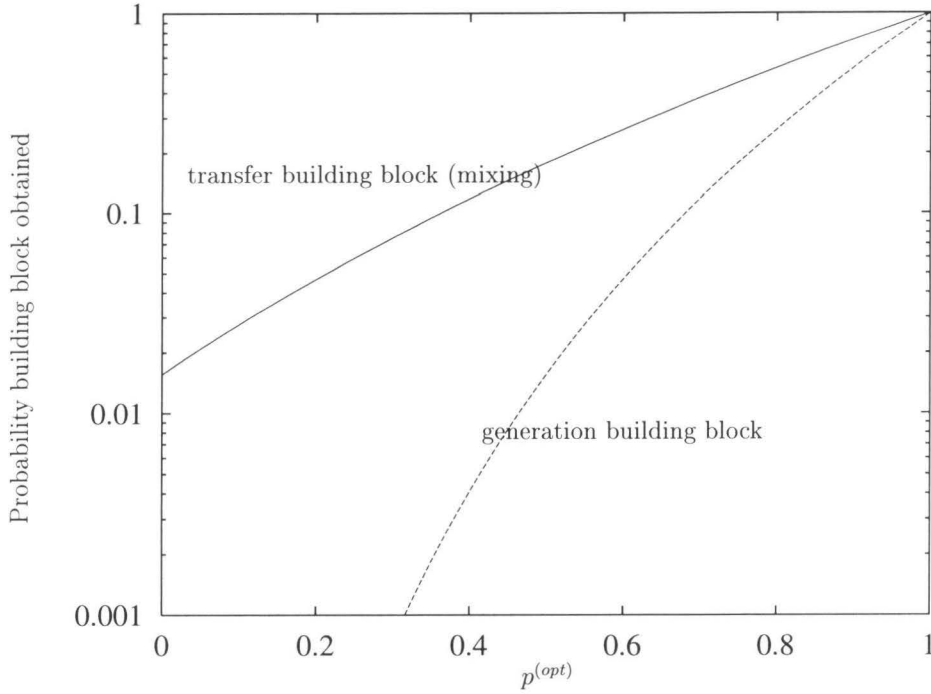


Figure 3.7: Probability of transferring a building block and of generating a building block.

In case of the second process the probability of observing an optimal bit in the offspring becomes:

$$p_b^{(opt)} = \frac{\sum_{i=0}^{k-1} \left( i + \frac{\sum_{j=0}^{k-i-1} j \cdot \text{Bin}(k-i, j, 0.5)}{\sum_{j=0}^{k-i-1} \text{Bin}(k-i, j, 0.5)} \right) \text{Bin}(k, i, p_{1,t}^{(opt)})}{\sum_{i=0}^{k-1} \text{Bin}(k, i, p_{1,t}^{(opt)})}.$$

Here,  $\text{Bin}(n, k, p)$  is the binomial distribution that gives the probability of  $k$  positive outcomes out of  $n$  tries where the probability of success is  $p$ , and  $i$  denotes the number of loci containing optimal values in the parent that does not contain the building block. For these loci it does not really matter which parent is selected, the resulting locus in the offspring always gets the optimal value. For the remaining  $(k-i)$  bits one computes the probability that  $j$  values are selected from the parent containing the optimal building block, so  $(i+j)$  out of  $k$  values of the offspring are set to the optimal value. The sum over  $j$  runs from 0 to  $(k-i-1)$ , because one should only sum over those individuals that do not contain the complete building block.

Figure 3.4 shows the evolution of  $p_{i,t}^{(opt)}$  both parts. One can easily see that the  $p_{1,t}^{(opt)}$  increases faster than  $p_{2,t}^{(opt)}$ . This is to be expected because the number of individuals

containing the optimal solution in the left-hand part is also larger than the number of individuals containing an optimal building block in the right-hand part.


Figure 3.5 shows the evolution of the ratio  for both models. Apparently, this ratio is not strongly influenced by whether  $p_{i,t}^{(opt)}$  is updated or not. Thus, the conclusion of the section 3.3, which says that cross-competition can easily result in premature convergence, is still valid.

Figure 3.6 shows the probability that the optimal solution is found for the two models. The set of solid curves corresponds to the case where  $p_{i,t}^{(opt)}$  is kept constant at 0.5. There are three processes that can generate an optimal solution. The lowest curve shows the contribution of the first process. The second curve shows the accumulated production of the first and the second process, and the last curve shows the output of all three processes. The first and the third process involve the generation of a building block that is not present in either of the parents. The second process corresponds to a mixing event where two building blocks present in different parents are mixed in the offspring. The probability that an optimal individual is generated by a the mixing process is obtained by taking the difference between the first and the second curve. At  $p = 0.6$  a fraction 0.023 of the produced offspring are optimal, 65% of these are produced by the mixing process. During the sixth generation we have  $p = 0.984$  when  $p_{i,t}^{(opt)}$  is fixed at 0.5. A fraction 0.016 off the offspring are optimal, 6% of these are produced by the mixing process. The set of dashed curves in Figure 3.6 correspond to the model where  $p_{i,t}^{(opt)}$  is updated. During the sixth generation we have  $p = 0.981$ . A fraction 0.072 off the offspring are optimal, 8% of these are produced by the mixing process, so for 92% of the optimal solutions a building block that was missing in both the parents was generated. Mixing is not efficient in this case because only a small proportion of the individuals in the population contain the second building block. Generation of non-existing building blocks is relatively easy for the current problem because the proportion of optimal bits, when a building block is not present, is significantly larger than 0.5.

Figure 3.7 shows the probability that an optimal building block is transferred, and the probability of generating a building block (again for  $k = 6$ ) as a function of the value of  $p^{(opt)}$ . Mixing is much more efficient than generation of a non-existing building block, especially when  $p^{(opt)}$  gets a low value. Note that a logarithmic scale is used for the vertical axis.

The problem investigated here does not involve any strong first-order interaction, therefore the value of  $p^{(opt)}$  does not deviate rapidly from 0.5. If the first-order interactions lead to the global optimum, then  $p^{(opt)}$  would increase rapidly, and mixing would not be very important. Such problems can also be solved easily by means of hill-climbing, and therefore do not require any building block processing. If the first-order interaction lead to the non-optimal value, then the problem involves deception. The  $p^{(opt)}$  (that starts at approximately 0.5 for a random initial population) is likely to decrease, and efficient building block processing becomes important. Hence, mixing of building blocks seems to be important on problems involving deception.

### 3.5 Summary

Traditional crossover operators become inefficient when the order of building blocks increases: the probability that a building block is transferred decreases.

Uniform crossover is unbiased with respect to the defining length of building blocks. Therefore we use uniform crossover when optimizing problems with unknown linkage. For the problems considered here, we have shown that unbiased crossover ( $p = 0.5$ ) performs best in many cases. Only when the difference in the number of bits of the different parents that have to be preserved differ a lot, the biased uniform crossover performs better.

Even when two non-overlapping building blocks have the same marginal fitness contribution, there can be a cross-competition between these building blocks. The probability that a building block is transferred depends on the order of the corresponding schema, and on the actual distribution over the search-space of the individuals in the current population. By means of a simple example, where the optimum can be decomposed in two building blocks, it was shown that if first building block has more duplicates than the second building block, then the population is off-balance, and the search tends to focus on individuals that only contain the first building block. The two building blocks have to be mixed fast, because otherwise the second building block can be pushed out of the population.

On the simple problem presented here, the generation of building blocks is possible, but on problems that are partially deceptive the generation of building blocks is unlikely. Therefore, the mixing process is important for such problems.

## Chapter 4

# Transmission function models of infinite population genetic algorithms

Altenberg used transmission functions to model generational genetic algorithms [Alt94]. We have extended these transmission function models to a broad range of genetic algorithms. Tracing these probabilistic models corresponds to running a genetic algorithm with an infinite population. The results obtained by tracing these models can be regarded as an upper bound on the probability that the optimum is found for the corresponding “real” finite population genetic algorithms when using large populations. Tracing of probability models for a simple GA was done by Whitley [Whi93]. Kok and Floréen traced probabilities using a model based on bit-products and Walsh-products [KF95]. Altenberg used a transmission function model to model generational genetic algorithms [Alt94]. All these models correspond to genetic algorithms involving an infinite population. Here, a broad range of genetic algorithms is modelled with transmission functions, and in the next chapter these models are extended such that genetic algorithms with finite population size can be handled too. We also treat elitism: in an elitist GA the parents can be propagated directly to subsequent generations, this in contrast to the generational GA where the parents are always discarded after producing enough offspring to populate the next generation. Theoretical analysis has shown that a canonical GA will not converge to the global optimum in general, but a GA that maintains the best solution will converge [Rud94, BKdGK97] and a GA involving elitism will converge to the optimum too [Rud96, BKdGK97]. Maintaining the best individuals means keeping these individuals in the population without allowing them to reproduce, while in case of elitism an individual is allowed to reproduce throughout its lifetime.

The outline of the rest of this chapter is as follows. In section 4.1 the transmission function models for different types of genetic algorithms are introduced. Section 4.2 describes how such transmission function models are used to model the behaviour of genetic algorithms with infinite population size. When restricting ourselves to functions of unitation, a formulation in terms of equivalence classes can be used to get a more efficient computation, as described in section 4.3. Section 4.4 introduces the cross-competition problem, which is used as an example of the application of the transmission function models. Population flow



diagrams are introduced in section 4.5. These diagrams are used to visualize the simulation results for the different GA's.

## 4.1 Transmission models of selection schemes

We model the canonical genetic algorithm [Hol75], the generational genetic algorithm using tournament selection [Gol89b],  $(\mu, \lambda)$  and  $(\mu + \lambda)$  selection [BHmS91, Rec94, Sch95], triple-competition [vK97c], and elitist recombination [TG94]. Furthermore, the Breeder genetic algorithm [MSV94] and the CHC algorithm [Esh91] are discussed.

A selection scheme selects the parent pairs for generation  $G_{i+1}$  from the individuals in generation  $G_i$ . A detailed description of selection schemes can be found in section 2.3.

To describe the different models we use transmission functions. Altenberg gives the following short description [Alt94]:

“It is the relationship between the transmission function and the fitness function that determines GA performance. The transmission function “screens off” [Sal71, Bra90] the effect of the choice of representation and operators, in that either affect the dynamics of the GA only through their effect on the transmission function.”

The general form of transmission-selection recursion was used at least as early as 1970 by Slatkin [Sla70].

A transmission function describes the probability distribution of offspring from every possible pair of parents. For a binary genetic operator, the transmission function is of the form  $T(i \leftarrow j, k)$  where  $j$  and  $k$  are the labels of the two parents and  $i$  is the label of the offspring. To be more specific, let  $\mathcal{S}$  be the search space; then  $T : \mathcal{S}^3 \rightarrow [0, 1]$ . We have  $\sum_i T(i \leftarrow j, k) = 1$  for all  $j, k \in \mathcal{S}$ , because  $T(i \leftarrow j, k)$  represents a distribution for fixed  $j$  and  $k$ . For a symmetric operator we have additionally  $T(i \leftarrow j, k) = T(i \leftarrow k, j)$ .

Here, the transmission function model is used to model the selection schemes. The following notation is used. The distribution of the current population is denoted by  $\vec{x}$ , and the newly generated population is denoted by  $\vec{x}'$ . In order based selection schemes, like tournament selection, it is the fitness-based rank in the population that determines the probability of being selected as a parent. The function  $frac_{<}(j, \vec{x})$  gives the fraction of individuals in distribution  $\vec{x}$  that have a fitness strictly smaller than the fitness  $f_j$  of the individual labelled  $j$ , let  $frac_{=}(j, \vec{x})$  be the function that yields the fraction of individuals that have a fitness equal to  $f_j$ , and let  $frac_{>}(j, \vec{x})$  be the function that yields the fraction of individuals that have a fitness strictly larger than  $f_j$ .

### 4.1.1 Canonical Genetic algorithm

The canonical genetic algorithm is a generational GA using fitness proportional selection. The dynamical system that describes a transition from a current population to a new

population is given by [Alt94]:

$$x'_i = \sum_{j,k} T_o(i \leftarrow j, k) \frac{f_j f_k}{\bar{f}^2} x_j x_k,$$

where  $x_i$  is the frequency of the individual labelled  $i$ , and  $x'_i$  is its frequency during the next generation,  $f_i$  is the fitness of the individual labelled  $i$ ,  $\bar{f}$  is the average fitness, and  $T_o$  is the transmission function describing the actual interaction between genetic operators and representation. The probability that a parent of type  $j$  is selected is given by  $x_j f_j / \bar{f}$ , so the probability that parents have labels  $j$  and  $k$  under a fitness proportional selection scheme is

$$\frac{f_j f_k}{\bar{f}^2} x_j x_k.$$

#### 4.1.2 Deterministic $n$ -tournament selection

The only difference between the canonical genetic algorithm and the generational genetic algorithm with tournament selection is the method used to select the parent individuals. This results in

$$x'_i = \sum_{j,k} T_o(i \leftarrow j, k) P_{tour}^{(n)}(j, \vec{x}) P_{tour}^{(n)}(k, \vec{x}),$$

where  $P_{tour}^{(n)}(j, \vec{x})$  describes the probability that an individual with label  $j$  is selected from a population with distribution  $\vec{x}$  during a  $n$ -tournament. A  $n$ -tournament selection is performed by choosing  $n$  individuals uniform at random from the population and selecting the one with the highest fitness. In case of a tie, the individual which was chosen first, wins. Given the distribution of the current population, the probability  $P_{tour}^{(n)}(j, \vec{x})$  is computed by following the choices that lead to the selected individual:

$$P_{tour}^{(n)}(j, \vec{x}) = \sum_{t=1}^n \text{frac}_{<}(j, \vec{x})^{t-1} x_j (\text{frac}_{<}(j, \vec{x}) + \text{frac}_{=}(j, \vec{x}))^{n-t}.$$

The  $t^{th}$  term of this sum is the probability that the first  $(t-1)$  selected individuals have a fitness smaller than the individual with label  $j$ , that the  $t^{th}$  individual has label  $j$ , and that all subsequent  $(n-t)$  individuals have a fitness smaller than or equal to the fitness of the individual with label  $j$ . The sum over all possible values gives the probability that an individual with label  $j$  wins the  $n$ -tournament.

#### 4.1.3 Evolution strategy $(\mu, \lambda)$ and BGA selection

When using  $(\mu, \lambda)$ -selection, a parent population containing  $\mu$  parents is used to generate  $\lambda$  offspring. To generate an offspring, two parents are selected uniform at random from

the parent population and recombination is applied. The best  $\mu$  offspring are used as the parents for the next generation, so  $\lambda \geq \mu$ .

First we introduce the truncation operator  $\text{Tr} : \mathbb{R}^n \times \mathbb{R} \rightarrow \mathbb{R}^n$ . This operator takes a distribution  $\vec{p}$  and a parameter  $\alpha \in [0, 1]$  as inputs and returns a new distribution containing the  $\alpha$  fraction of best individuals out of the original distribution. The operator selects a pivot individual  $i$  such that  $\text{frac}_{<}(i, \vec{p}) < (1 - \alpha)$  and  $(\text{frac}_{<}(i, \vec{p}) + \text{frac}_{=}(i, \vec{p})) \leq 1 - \alpha$ . If more than one value of  $i$  satisfies these constraints, then an arbitrary choice among these  $i$  is taken. The operator  $\text{Tr}(\vec{p}, \alpha)$  is defined by:

$$\text{Tr}(\vec{p}, \alpha)_j = \begin{cases} \frac{1}{\alpha} p_j & f_j > f_i \\ \frac{1}{\alpha} \frac{\alpha - \text{frac}_{>}(i, \vec{p})}{\text{frac}_{=}(i, \vec{p})} & f_j = f_i \\ 0 & f_j < f_i \end{cases}.$$

Given this truncation operator the formula for the  $(\mu, \lambda)$ -selection is:

$$\vec{x}' = \text{Tr}(\vec{y}, \frac{\mu}{\lambda}),$$

where the elements of  $\vec{y}$  are given by

$$y_i = \sum_{j,k} T_o(i \leftarrow j, k) x_j x_k.$$

The combination of the last two formulae gives the desired model.

The Breeder Genetic Algorithm uses  $T\%$  truncation selection, which means that the  $T\%$  best individuals of the current population are allowed to reproduce. Out of these  $T\%$  best individuals, the parents are selected uniform at random, and a new population is generated. Typically  $T$  is between 10 and 50. This selection scheme corresponds to the  $(\mu, \lambda)$ -selection where  $\mu = \lambda T/100$ . When applying BGA the best individual found so far will always be retained.

#### 4.1.4 Evolution strategy $(\mu + \lambda)$ and CHC selection

The  $(\mu + \lambda)$ -selection scheme is quite similar to the  $(\mu, \lambda)$ -selection scheme. The only difference is that in the  $(\mu + \lambda)$ -selection scheme, the  $\mu$  new parents are obtained by selecting the best  $\mu$  individuals from both the  $\mu$  parents and the  $\lambda$  offspring:

$$\vec{x}' = \text{Tr} \left( \text{Bl} \left( \vec{x}, \vec{y}, \frac{\mu}{\mu + \lambda} \right), \frac{\mu}{\mu + \lambda} \right),$$

where  $\text{Bl}(\vec{x}_1, \vec{x}_2, \beta)$  computes a weighted average of vectors  $\vec{x}$  and  $\vec{y}$ :

$$\text{Bl}(\vec{x}_1, \vec{x}_2, \beta) = \beta \vec{x}_1 + (1 - \beta) \vec{x}_2.$$

The vector  $\vec{y}$  is given by

$$y_i = \sum_{j,k} T_o(i \leftarrow j, k) x_j x_k.$$

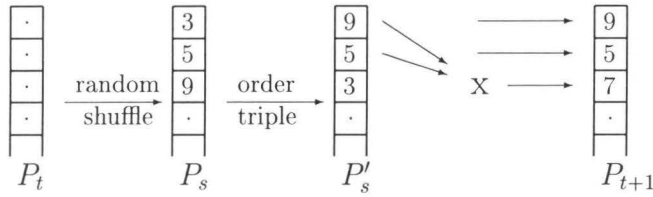


Figure 4.1: Schematic representation of triple-competition

The CHC algorithm uses unbiased selection of parents. Given a parent population of size  $N$ , a set of  $N$  offspring is produced. The next parent generation is obtained by selecting the  $N$  best among the  $N$  parents and their  $N$  offspring. This selection scheme corresponds to  $(\mu + \lambda)$ -selection, where  $\mu = \lambda = N$ .

An additional feature of the CHC selection scheme is the so-called incest-prevention. CHC does incest prevention using the Hamming distance between the two parents. If the Hamming distance is below a certain threshold, then the pair of parents is not allowed to reproduce. Typically CHC starts with a threshold  $L/4$ , where  $L$  is the length of the bit-string. We did not use this incest prevention scheme in our model because it requires knowledge about the Hamming-distances between the different types. However, for problems where this type of information is available the modelling of the incest prevention is relatively straightforward.

#### 4.1.5 Triple-competition selection

The triple-competition selection is described in section 2.3.3. It is an elitist genetic algorithm that uses a tournament-like selection of the parents, and parents can be propagated to the next generation. Figure 4.1 shows how generation  $P_{t+1}$  is generated from generation  $P_t$ : a single box corresponds to an individual, a number in a box is its fitness, and a stack of such boxes corresponds to a population. The first step involves a random shuffle of the individuals in population  $P_t$  resulting in a randomly ordered population  $P_s$ . The population  $P_s$  is partitioned in sets of three individuals, and each triple is ordered such that the best individuals are on top in each triple, resulting in an intermediate population  $P'_s$ . The two top-ranked individuals of each triple are allowed to recombine to generate a single offspring. Next, the two parents and their offspring are added to the population  $P_{t+1}$ , so only the lowest ranked individual of each triple is replaced. During a single step of this algorithm  $N/3$  offspring are generated, where  $N$  is the size of the population. This algorithm is modelled as follows

$$x'_i = \frac{1}{3} \sum_{j,k} T_o(i \leftarrow j, k) P(j, k, \vec{x}) + \frac{2}{3} \sum_k P(i, k, \vec{x}),$$

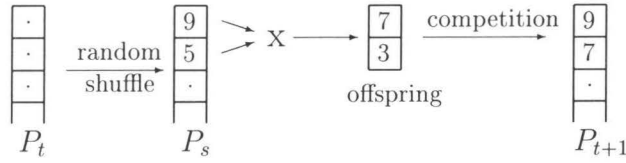


Figure 4.2: Schematic representation of Elitist recombination

where the first sum corresponds to the distribution of the newly generated offspring, and the second sum corresponds to the distribution of the surviving parents. The distribution  $\vec{x}'$  of the new population consists for one third of newly generated offspring and for two third of surviving parents. The function  $P(j, k, \vec{x})$  gives the probability that the individuals  $j$  and  $k$  are selected as parents, where the individual labelled  $j$  is selected first. To compute this probability one differentiates between the cases  $(f_j = f_k)$  and  $(f_j \neq f_k)$ , and for each of these two cases one distinguishes between the case that fitness of the third individual is smaller and the case that its fitness is equal to the fitness of the worst individual out of  $j$  and  $k$ .

$$P(j, k, \vec{x}) = \begin{cases} 3 (frac_{<}(j, \vec{x}) + frac_{=}(j, \vec{x})/3)x_j x_k & \text{if } (f_j = f_k) \\ 3 (frac_{<}(\text{Sm}(j, k), \vec{x}) + frac_{=}(\text{Sm}(j, k), \vec{x})/2)x_j x_k & \text{otherwise.} \end{cases}$$

Here, the function  $\text{Sm}(j, k)$  selects the label corresponding to the individual having the smallest fitness of  $j$  and  $k$ . The factor 3 in both formulas corresponds to the number of possible orders of the three involved individuals given that  $j$  is selected before  $k$ .

Triple-competition selection as defined above uses elitism because the best individual is always transferred to the next generation. However, it is still possible that newly produced offspring has a lower fitness than the individual that is replaced by this new offspring.

#### 4.1.6 Elitist recombination

Elitist recombination [TG94] selects parents by a random pairing of parent individuals. Figure 4.2 shows how the next population  $P_{t+1}$  is produced from the current population  $P_t$ . Just like in triple-competition selection, a random shuffle is applied. The resulting population  $P_s$  is partitioned in a set of adjacent pairs, and for each pair the recombination operator is used to obtain two offspring. Next, a competition is held among the two offspring and their two parents, and the best two out of these four are propagated to the next population  $P_{t+1}$ . In Figure 4.2 one parent, having fitness 9, and one offspring, having fitness 7, are propagated to population  $P_{t+1}$ . When using elitist recombination it is possible, but not guaranteed that parents will survive, so parents are only preserved in the case that their fitness is larger than the fitness of the offspring. The dynamical system

representing elitist recombination is:

$$x'_i = \sum_{j,k} T_{er}(i \leftarrow j, k) x_j x_k.$$

The selection of individuals that enter the next population is not visible in this model. This selection mechanism consists of a local competition between parents and their direct offspring, and therefore is located inside the transmission function  $T_{er}$ . Outside the transmission function, it is not visible which parents were used.

Here, a generalization of elitist recombination that creates a fixed number of offspring  $n \geq 2$  per generation is modelled. The original definition of elitist recombination uses  $n = 2$ . Creation of a number of offspring and accepting only the best few has been suggested by Altenberg [Alt94] under the name soft-brood selection. For the elitist recombination and its generalization a modified transmission function  $T_{er}(i \leftarrow j, k)$  is used, because the selection of survivors is done by means of a local competition between parents and offspring. This in contrast to the usual selection mechanisms that operate on the complete population. Such a local selection scheme can be modelled by modifying the original transmission function  $T_o(i \leftarrow j, k)$  that describes the interaction between the operators and the representation used. Each column of the new transmission function  $T_{er}(i \leftarrow j, k)$  can be computed independently. Assuming that we have  $C$  different elements with labels in the range 0 to  $C - 1$ , then the transmission function  $T_o(i \leftarrow j, k)$  can be represented by a matrix having  $C$  rows and  $C^2$  columns. Let  $\vec{t}_o^{jk}$  denote the column with index  $(jC + k)$  of the matrix  $T_o(i \leftarrow j, k)$ . This column represents the probability distribution of the offspring when applying recombination to parent of respectively type  $j$  and type  $k$ .

In the rest of this chapter the binomial distribution is denoted by  $\text{Bin}(n, k, p)$  where  $n$  is the total number of experiments,  $k$  is the number of successful outcomes, and  $p$  is the probability of a successful outcome.

The auxiliary function  $P_{\text{accept}}(P_<, P_=:, n, a)$  gives the probability that an offspring individual is accepted. Accepting an offspring means that the offspring is among the best two during the tournament between the two parents and their  $n$  offspring. The parameters of  $P_{\text{accept}}(P_<, P_=:, n, a)$  are as follows:  $P_<$  is the proportion of offspring having smaller fitness than the individual under consideration,  $P_=:$  is the proportion of offspring having equal fitness,  $n$  is the number of additional offspring generated, and  $a$  is the number of offspring that can be accepted apart from the current offspring. This function is computed by first considering the number of superior offspring followed by considering the number of offspring having equal fitness:

$$P_{\text{accept}}(P_<, P_=:, n, a) = \sum_{l=0}^a \text{Bin}(n, l, 1 - P_< - P_=:) \sum_{m=0}^{n-l} \text{Bin}\left(n - l, m, \frac{P_=:}{P_< + P_=:}\right) \min\left\{1, \frac{a-l+1}{m+1}\right\}.$$

Here,  $l$  denotes the number of offspring having a fitness larger than the fitness of the individual under consideration, and  $m$  denotes the number of offspring that have exactly

the same fitness. The first binomial distribution gives the probability that  $l$  out of the  $n$  other offspring have a larger fitness than the current individual. The proportion of these individuals is  $(1 - P_{<} - P_{=})$ . The second binomial distribution computes the probability that  $m$  out of  $n - l$  offspring have a fitness equal to the individual under consideration. At this point it is known that all  $n - l$  offspring have a fitness lower than or equal to the fitness of the current individual; Therefore the proportion of offspring that have equal fitness is  $P_{=}/(P_{<} + P_{=})$ . Given the values of  $l$  and  $m$ , the probability that the current individual is accepted is equal to  $\min\{1, (a - l + 1)/(m + 1)\}$ .

A column of the matrix describing  $T_o(i \leftarrow j, k)$  represents the probability density function over the space of all possible offspring  $i$  for a given pair of parents  $j$  and  $k$ . The offspring can be divided among three sets. Assume that  $f_k \leq f_j$  (otherwise exchange the roles of  $j$  and  $k$ ). We take

$$\begin{aligned} S_I &= \{i \in \mathcal{S} : f_i \geq f_j\}, \\ S_{II} &= \{i \in \mathcal{S} : f_k \leq f_i < f_j\}, \\ S_{III} &= \{i \in \mathcal{S} : f_i < f_k\}. \end{aligned}$$

Given a column  $\vec{t}_o^{jk}$  of  $T_o(i \leftarrow j, k)$  the corresponding column from  $T_{er}(i \leftarrow j, k)$  is computed as follows. Let the unnormalized distribution of the offspring (i.e. we do not require that the sum of the probabilities equals one) be denoted by  $\vec{o}$ , let the distribution of the surviving parents denoted by  $\vec{s}$ , and let the probability that an offspring of type  $i$  is obtained by recombination be denoted by  $t_i^{jk}$ . Now, the probability that the offspring is also accepted when applying elitist recombination with  $n$  offspring for each pair of parents depends on whether offspring  $i$  belongs to set  $S_I$ ,  $S_{II}$ , or  $S_{III}$ . If  $i \in S_I$ , then the offspring is accepted when at most one of the other offspring has a larger fitness,

$$o_i = t_i^{jk} n P_{accept}(frac_{<}(i, \vec{t}_o^{jk}), frac_{=}(i, \vec{t}_o^{jk}), n - 1, 1),$$

where  $frac_{<}(i, \vec{t}_o^{jk})$  is the fraction of individuals in the distribution  $\vec{t}_o^{jk}$  that have a lower fitness than an individual of type  $i$ . If  $i \in S_{II}$ , then the probability that the offspring is accepted depends on the number of offspring in  $S_I$ :

$$\begin{aligned} o_i &= t_i^{jk} n \sum_{l=0}^1 \text{Bin}(n - 1, l, 1 - f_{<}(j, \vec{t}_o^{jk})) \\ &\quad P_{accept}\left(\frac{frac_{<}(i, \vec{t}_o^{jk})}{frac_{<}(j, \vec{t}_o^{jk})}, \frac{frac_{=}(i, \vec{t}_o^{jk})}{frac_{<}(j, \vec{t}_o^{jk})}, n - l - 1, 0\right). \end{aligned}$$

Here  $l$  denotes the number of other offspring located in  $S_I$ , the binomial distribution gives the probability of having  $l$  offspring in this region, and  $P_{accept}$  computes the probability that the offspring of type  $i$  is accepted given that  $n - l - 1$  other offspring also have a fitness below  $f_j$ . At most one offspring from  $S_{II}$  is selected. If  $i \in S_{III}$ , then  $o_i = 0$  because the individual  $i$  is always rejected.

Now, the distribution of the offspring is known. Next, the distribution of the surviving parents, denoted by  $\vec{s}$ , has to be computed. Parent  $k$  is only retained when all offspring

belong to  $S_{III}$ , so

$$s_k = \text{Bin}(n, n, \text{frac}_{<}(k, \vec{t}_o^{jk})).$$

Parent  $j$  is retained when all offspring is in  $S_{II} \cup S_{III}$  or when one offspring is in  $S_I$  and the other offspring are in  $S_{III}$ . This probability is

$$s_j = \text{Bin}(n, n, \text{frac}_{<}(j, \vec{t}_o^{jk})) + \text{Bin}(n, n-1, \text{frac}_{<}(j, \vec{t}_o^{jk})) \text{Bin}\left(n-1, n-1, \frac{\text{frac}_{<}(k, \vec{t}_o^{jk})}{\text{frac}_{<}(j, \vec{t}_o^{jk})}\right).$$

Here, the first term corresponds to the case that all offspring have a fitness lower than  $f_j$ , and the second term corresponds to the case that exactly one offspring has a fitness larger than or equal to  $f_j$  while all other offspring are located in region  $S_{III}$ . In that case the superior offspring replaces parent  $k$  instead of parent  $j$ .

The vector  $\vec{o} + \vec{s}$  describes the unnormalized probability distribution of the two individuals that will be propagated to the next generation. Using these two vectors the column with index  $jC + k$  of  $T_{er}(i \leftarrow j, k)$  is given by the formula

$$\vec{t}_{er}^{jk} = \frac{1}{2}(\vec{o} + \vec{s}).$$

By applying this procedure to every column of  $T_o(i \leftarrow j, k)$  the matrix representing  $T_{er}(i \leftarrow j, k)$  is obtained.

## 4.2 Infinite population models

Given that a single application of the crossover operator produces one offspring, the transmission function model computes the expected distribution of this offspring given the distribution of the parents.

Based on the transmission function, the evolution of a genetic algorithm with an infinite population size can be modelled by iterated application of the transmission function. Let us denote a single application of the transmission function by means of the operator  $\mathcal{F} : P \rightarrow P$ . The initial population  $G_0$  is usually obtained by drawing a uniform random sample. Using a transmission function, the distribution after one step of the evolution is  $G_1 = \mathcal{F}(G_0)$ , the distribution after two generations is  $G_2 = \mathcal{F}(G_1) = \mathcal{F}(\mathcal{F}(G_0))$ , or more generally after  $t$  generations is  $G_t = \mathcal{F}^t(G_0)$ , where the superscript  $t$  denotes iterated application of the function.

Due to the law of large numbers the transmission function models the behaviour of a genetic algorithm with an infinite population size. To see this, let us assume that a population of size  $N$  is used, and let  $X_i^{(j)}$  be equal to one if sample  $i$  is of type  $j$ , and zero otherwise. The strong law of large numbers states that if  $X_1, X_2, \dots$  are independent identically distributed random variables and  $\mathbf{E}X_i = \mu$ , where  $\mu$  is the probability of an



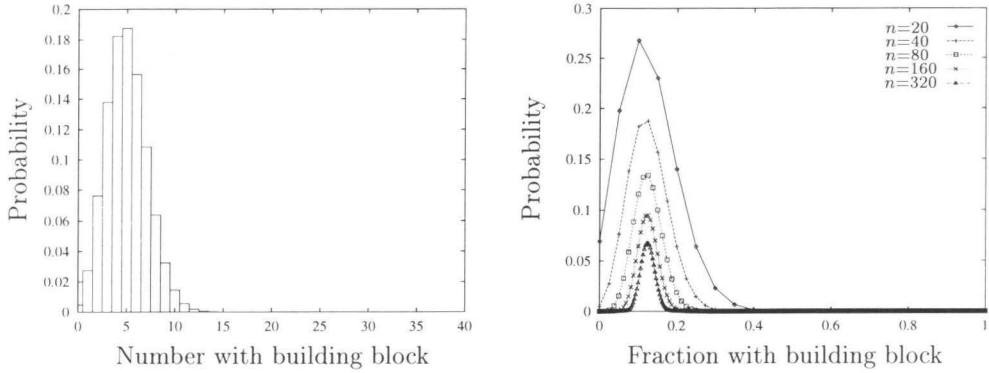


Figure 4.3: The probability distribution describing the number of individuals in the population containing the optimal building block for  $d = 3$  and  $n = 40$  (left) and the fraction of individuals containing the optimal building block for  $d = 3$ , and  $n = 20, 40, 80, 160, 320$  (right)

individual of type  $i$  in the offspring distribution, then

$$\frac{1}{n} \sum_{i=1}^n X_i \rightarrow \mu \text{ almost surely, when } n \rightarrow \infty.$$

Hence, the proportion of individuals of type  $j$  converges to the proportion predicted by the distribution when  $n \rightarrow \infty$ , and the actual distribution of an infinitely large offspring population corresponds to the distribution predicted by the transmission function model.

### 4.3 Equivalence classes

Next, the transmission function models are used to trace the evolution of genetic algorithms. Tracing the evolution of requires a lot of computation. A single application of the transmission function involves  $|\mathcal{S}|^3$  computational steps, where  $|\mathcal{S}|$  denotes the cardinality of the set  $\mathcal{S}$ . When tracing the evolution for a specific problem more efficient methods can sometimes be obtained by mapping the original search space  $\mathcal{S}$  to a more compact space  $\mathcal{V}$  where each element of  $\mathcal{V}$  represents an equivalence class containing elements from  $\mathcal{S}$ . Next, the transmission function is lifted to the space of equivalence classes  $\mathcal{V}$ , and the transmission function model is applied to  $\mathcal{V}$  instead of  $\mathcal{S}$ . Equivalence classes should satisfy the following two conditions:

- (1) all elements of an equivalence class have the same fitness, and
- (2) the distribution over the elements of an equivalence class is known and constant.

The first condition is necessary because then all individuals in a single equivalence class behave identical under selection. The second condition is necessary because then the distribution in the original space  $\mathcal{S}$  can be reconstructed solely based on knowledge of the distribution in  $\mathcal{V}$ .

Srinivas et al. [SP93] used such an equivalence-class approach to model GA's with infinite populations. Their approach assumes that the optimization problem for the GA is a single function of unitation. If the size of a string from the search space  $\mathcal{S}$  is  $l$  bits, then  $\mathcal{S}$  can be modelled by means of  $(l + 1)$  equivalence classes. The operation of an infinite population GA can be modelled exact by means of what they call a Binomially Distributed Population. The time complexity of algorithm derived from this model is  $O(l^3)$ , a significant improvement over previous models with exponential time complexities [SP93].

If the search space is divided over the equivalence classes based on the number of one-bits, then all details with respect to the distribution of one-bits are ignored; In fact, within an equivalence class all loci are assumed to have the same probability of containing a one-bit. This is not a limitation when tracing a genetic algorithm with an infinite population, and even for genetic algorithms with a small population this holds when the results are averaged over a large number of runs, because all bits within a part behave identical. However, within a single run the probability of finding a one-bit at a specific locus can easily deviate from the expected probability due to genetic drift. Because this type of deviation is ignored when using the equivalence-classes, the model actually considers a genetic algorithm without genetic drift.

In the rest of this chapter a problem that consist of a concatenation of functions of unitation is studied. Assuming that the problem consist of a concatenation of  $n$  functions of unitation, where subfunction  $i$  has a length of  $l_i$  bits, the total length is  $l = \sum_{i=1}^n l_i$ .

The space  $\mathcal{V}$  has cardinality  $|\mathcal{V}| = \prod_{j=1}^n (l_j + 1)$ . A mapping from  $\mathcal{S}$  into  $\mathcal{V}$  is

$$F : \mathcal{S} \rightarrow \mathcal{V} = \sum_{i=1}^n u_i \prod_{j=1}^{i-1} (l_j + 1),$$

where  $u_i$  is the number of one-bits in part  $i$  and  $l_i$  is the maximal number of one-bits in part  $i$ . Given an element  $v \in \mathcal{V}$  the number of one-bits in part  $k$  is

$$\alpha(v, k) = \frac{v}{\prod_{i=1}^{k-1} (l_i + 1)} \bmod (l_k + 1).$$

When using a random initial population the distribution over  $\mathcal{V}$  is

$$P(v) = \frac{1}{|\mathcal{S}|} \prod_{j=1}^n \binom{l_j}{\alpha(v, j)}.$$

If we apply uniform crossover to two parents that contain respectively  $j$  and  $k$  one-bits, then the probability that the offspring contains  $i$  one-bits is given by the recursive formula

$$p(j, k, i, l) = \begin{aligned} & \left(1 - \frac{j}{l}\right)\left(1 - \frac{k}{l}\right) p(j, k, i, l-1) + \\ & \left(1 - \frac{j}{l}\right)\frac{k}{l}\frac{1}{2} (p(j, k-1, i, l-1) + p(j, k-1, i-1, l-1)) + \\ & \frac{j}{l}\left(1 - \frac{k}{l}\right)\frac{1}{2} (p(j-1, k, i, l-1) + p(j-1, k, i-1, l-1)) + \\ & \frac{j}{l}\frac{k}{l} p(j-1, k-1, i-1, l-1), \end{aligned}$$

where  $p(j, k, i, n)$  represents the probability that an offspring string with  $i$  one-bits is obtained from two parent string having respectively  $j$  and  $k$  one-bits, and  $l$  is the number of bits in this subfunction. The boundary conditions are  $p(1, 1, 1, 1) = p(0, 0, 0, 1) = 1$ ,  $p(1, 0, 1, 1) = p(0, 1, 1, 1) = p(1, 0, 0, 1) = p(0, 1, 0, 1) = \frac{1}{2}$ ,  $p(0, 0, 1, 1) = p(1, 1, 0, 1) = 0$ , if  $j < 0$ ,  $k < 0$ , or  $i < 0$ , then  $p(j, k, i, n) = 0$ , and due to the symmetry of the uniform crossover  $p(j, k, i, n) = p(k, j, i, n)$ .

The transmission function  $T(i \leftarrow j, k)$  can be represented by a  $(n \times n^2)$ -matrix of transmission probabilities where  $n$  denotes the cardinality of  $|\mathcal{V}|$ . Because the different partitions evolve independently under uniform crossover the probability of an outcome is given by the product of the probabilities for each of the parts, which results in

$$T_o[i, j \cdot k] = \prod_{m=1}^n p(\alpha(i, m), \alpha(j, m), \alpha(k, m), l_m).$$

## 4.4 Cross-competition problem

Given a problem that contains building blocks, the compatible building blocks can be involved in a cross-competition in order to get more copies in the population. Cross-competition between building blocks can strongly influence the reliability of a GA.

In order to study this kind of effects we use a mixture of an oneMax function of length  $l$  and a deceptive trap function of length  $d$  [Gol89b], so a single individual is represented by a string of length  $(l + d)$  bits. The bits are partitioned in two sets  $O$  and  $D$ . The first partition is the oneMax part. The fitness contribution of this partition is  $f_O(b) = u_O(b)$ , where  $u_O(b)$  is a function of unitation. This function counts the number of one-bits in the partition  $O$  of string  $b$ . The fitness of the deceptive part is given by the formula,

$$f_D(b) = \begin{cases} \alpha d & \text{if } u_D(b) = 0 \\ u_D(b) & \text{otherwise} \end{cases}$$

where  $\alpha > 1$ . The global optimum of the deceptive part contains only 0-bits, which results in a fitness contribution of  $\alpha d$ .

The fitness of a string is determined by the sum of the fitness values of the two partitions ( $f = f_O + f_D$ ). The global optimal solution contains one-bits in the partition  $O$  and 0-bits in partition  $D$  and has a fitness of  $l + \alpha d$ . The second best solution is given by the string containing one-bits only that has fitness  $l + d$ .

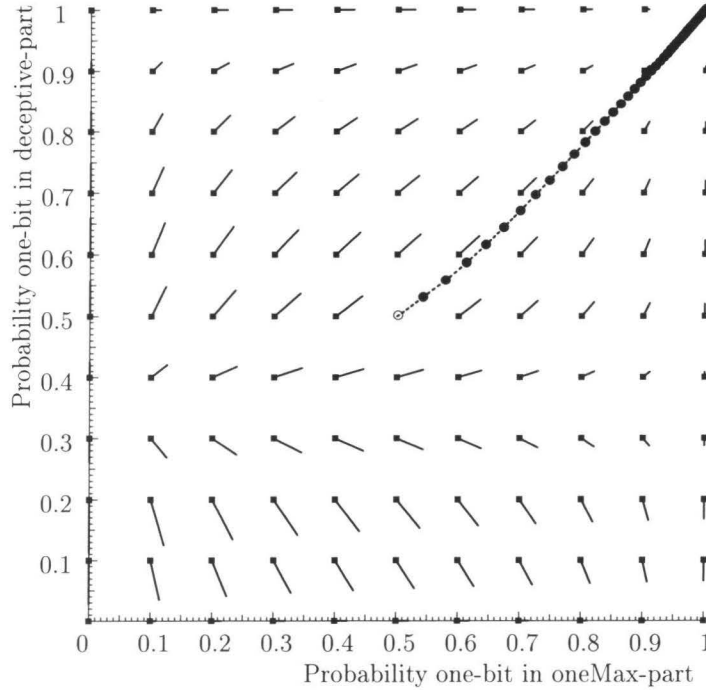


Figure 4.4: Tracing one evolutionary step for the canonical genetic algorithm

The actual linkage of the bits of the different partitions is unknown, so the loci occupied by the two partitions can be spread over the bit-string. According to the definition of a building block given in section 2.6, problem instances of the defined problem class contain one building block of order  $d$ , which is represented by the optimal schema for partition  $D$ .

This problem has been designed to compare the mixing capabilities of different genetic algorithms when confronted with a problem containing one large building block, and a set of bits that can be optimized independently of each other. The cross-competition between the building block and the bits in partition  $O$  is investigated.

## 4.5 Population flow diagrams

Here, the models described in section 4.1 are used to study the behaviour of genetic algorithms when optimizing the cross-competition problem described in section 4.4. We use a problem instance with  $l = 6$ ,  $d = 6$ , and  $\alpha = 1.5$ .

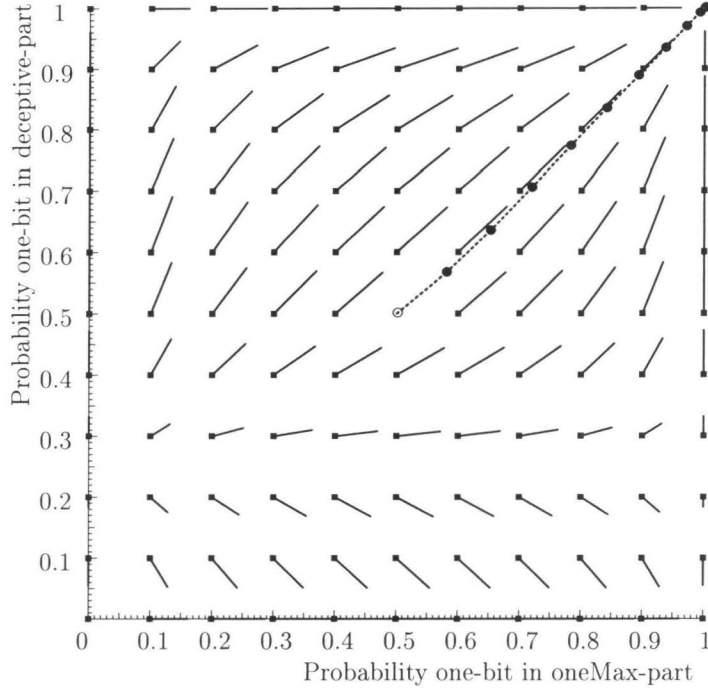


Figure 4.5: Tracing one evolutionary step for the generational genetic algorithm with tournament selection

Population flow diagrams are introduced to study the behaviour of the transmission function models. Let us assume that the population represented by distribution  $\vec{g}_t$  is transmitted to distribution  $\vec{g}_{t+1}$  after a single step of evolution. A population flow diagram contains a low-dimensional mapping of the space with all possible populations as its elements. Within a flow diagram a population is represented by a point. Many populations map to the same point. However, given a point one can construct the most typical population corresponding to this point. This population is determined by computing the distribution with the highest probability that adheres to the restrictions imposed by  $p_o$  and  $p_d$ . In case of the equivalence class model, a distribution is represented by a vector with  $(n_o \cdot n_d)$  elements, where  $n_o$  and  $n_d$  correspond to the total number of loci in the given partition. The probability-density assigned to the class labelled  $(k_o, k_d)$  is given by

$$f(k_o, k_d) = \text{Bin}(n_o, k_o, p_o) \text{Bin}(n_d, k_d, p_d),$$

where  $k_o$  and  $k_d$  correspond to the number of one-bits for the corresponding part. At

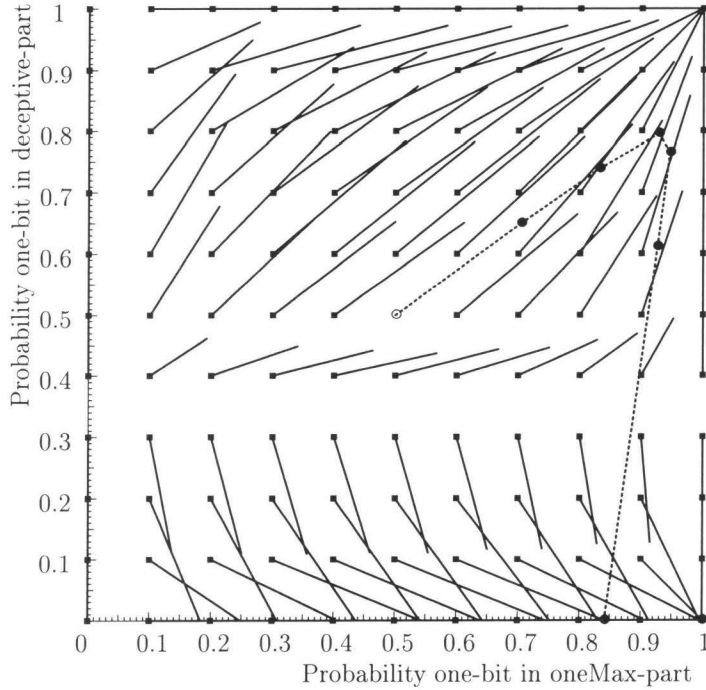


Figure 4.6: Tracing one evolutionary step for  $(\mu, \lambda)$ -selection (BGA)

each point a flow can be computed. To do so, one constructs the typical population by computing the corresponding distribution with the largest probability. This corresponds to a uniform distribution under the restrictions imposed by  $p_o$  and  $p_d$ . Let us denote this population by  $\vec{g}_0$ . Next, the corresponding point  $\vec{g}_1$  is computed by means of the finite population model. A flow is now represented by a square at  $\vec{g}_0$  and a line connecting  $\vec{g}_0$  and  $\vec{g}_1$ . Note, that this flow corresponds to the transition made by a typical population where the proportion of individuals in a class labelled  $(k_o, k_d)$  is  $f(k_o, k_d)$ .

Figure 4.4 shows a flow diagram for the application of the canonical GA to the cross-competition problem. Along the horizontal axis the probability of having a one-bit in the oneMax-part is given, and along the vertical axis the probability of a one-bit in the deceptive part is given. The population containing optimal individuals only corresponds to the point  $(1, 0)$ , which is the right-bottom corner of the diagram.

The most typical distribution  $\vec{g}_0$  is computed for 121 different points. The corresponding  $\vec{g}_1$  is obtained by applying a transmission function model.

A uniform initial distribution corresponds to the coordinate  $(0.5, 0.5)$ . The diagram

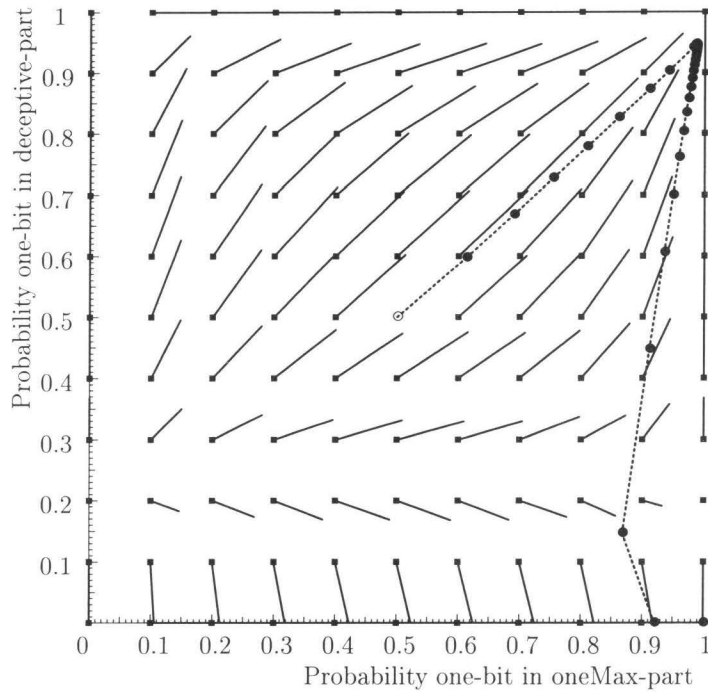


Figure 4.7: Tracing one evolutionary step for  $(\mu + \lambda)$ -selection (CHC)

contains a trace of evolution as computed by the infinite population model, denoted by the dotted line. The  $n^{th}$  bullet on the line is the point corresponding to the population after  $n$  steps of evolution.

The results for the canonical GA are shown in Figure 4.4. Given a uniform initial distribution the trace converges relatively slowly to the suboptimal solution. The convergence slows down as the distribution gets closer to the fixed-point. This could be expected because the relative fitness differences in the population get smaller. Convergence to the optimal solution can be obtained with a  $p_d$  below 0.4. When using a small population the initial population can deviate from the uniform distribution, and there is a chance that convergence to the optimum is obtained. The canonical GA closely follows the directions predicted by the flows in the plot.

Figure 4.5 shows the results for the generational genetic algorithm with tournament selection. This GA converges faster than the canonical GA. This GA also closely follows the flow as given in the diagram. To get convergence to the optimal solution,  $p_d$  has to drop below 0.3. The probability that an initial population is in the region where convergence to

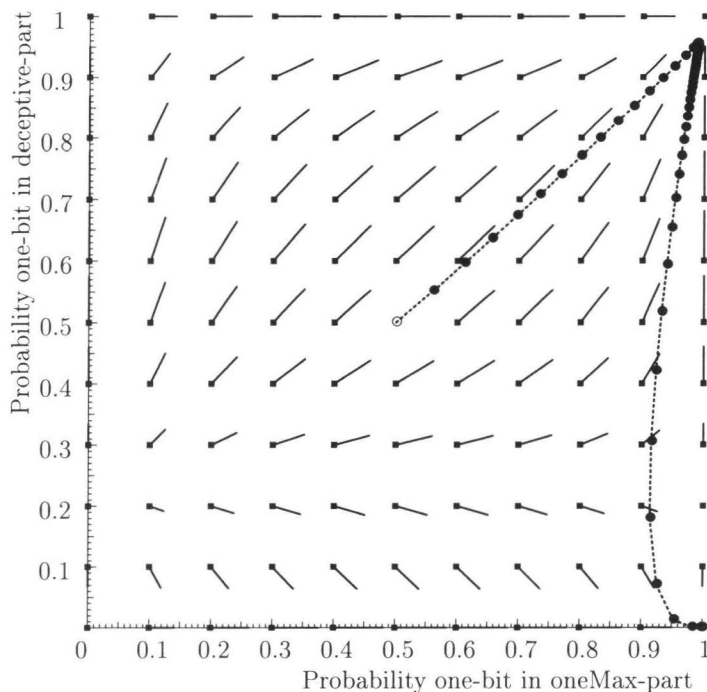


Figure 4.8: Tracing one evolutionary step for triple-competition

the optimum is obtained, is even smaller than in case of the generational genetic algorithm with fitness proportional selection.

Figure 4.6 shows the results for  $(\mu, \lambda)$ -selection. This selection scheme has a high selective pressure, resulting in large steps being taken. The  $(\mu, \lambda)$ -selection initially converges to the suboptimal solution, but after three steps of evolution the trace changes direction. Next,  $p_d$  drops fast to the optimal value of zero while  $p_o$  decreases slightly. Here the cross-competition is clearly present, hence good solutions in the oneMax part are traded for good solutions in the deceptive part. Recall that the flows predict the direction for a relatively uniform distribution. The strong selective pressure of this selection results in correlations between loci, and therefore in populations that are far from uniform. To get optimal building blocks in the deceptive part, the average fitness of the oneMax part is decreased. Once the deceptive part has converged to the line where  $p_d = 0$ , the oneMax part starts to converge again. After seven steps of evolution almost the complete population consists of optimal strings. However, recall that the  $(\mu, \lambda)$ -selection uses seven times as much offspring during a single generation as the generational GA's. After three



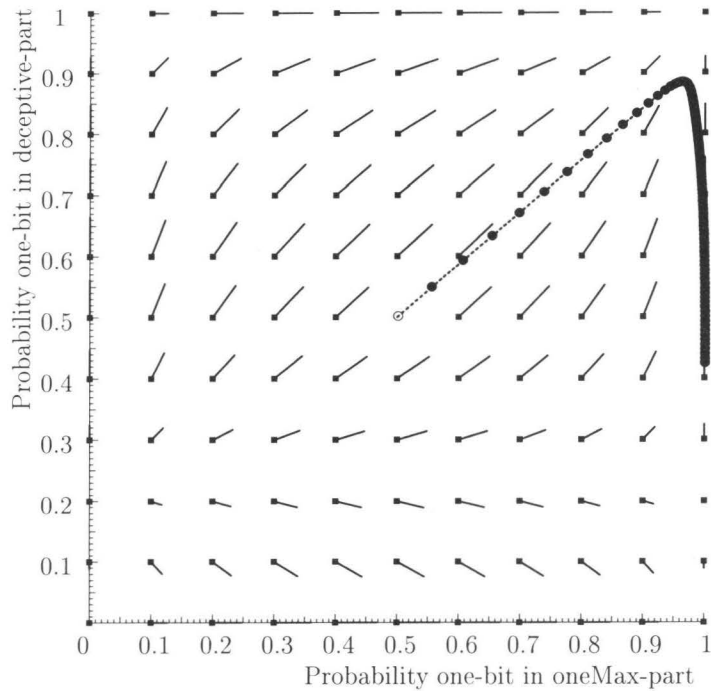


Figure 4.9: Tracing one evolutionary step for elitist recombination

steps of evolution the population is moved to (9.2, 7.9) under  $(\mu, \lambda)$ -selection. A uniform distribution with these parameters would converge to the suboptimal solution. However, the distribution of this population does not have to be uniform when applying selection. The population flow diagrams are two-dimensional mappings, while the actual model corresponds to a computation over a high-dimensional space. When using a high selective pressure non-uniform distributions can be obtained after a few generations.

Figure 4.7 show the results for  $(\mu + \lambda)$ -selection. This selection scheme initially convergence to the suboptimum too, but after some steps the trace bends and moves towards the optimal solution. The  $(\mu + \lambda)$ -selection converges more slowly than the  $(\mu, \lambda)$ -selection.

The results for triple-competition, shown in Figure 4.8, are roughly the same as for  $(\mu + \lambda)$ -selection.

Figure 4.9 shows the results for Elitist recombination. Again an initial convergence to the suboptimal solution is observed, but later the trace moves slowly towards the optimal point. Elitist recombination converges slowly, because it is difficult for an optimal individual to generate copies of itself due to the direct competition between parents and their

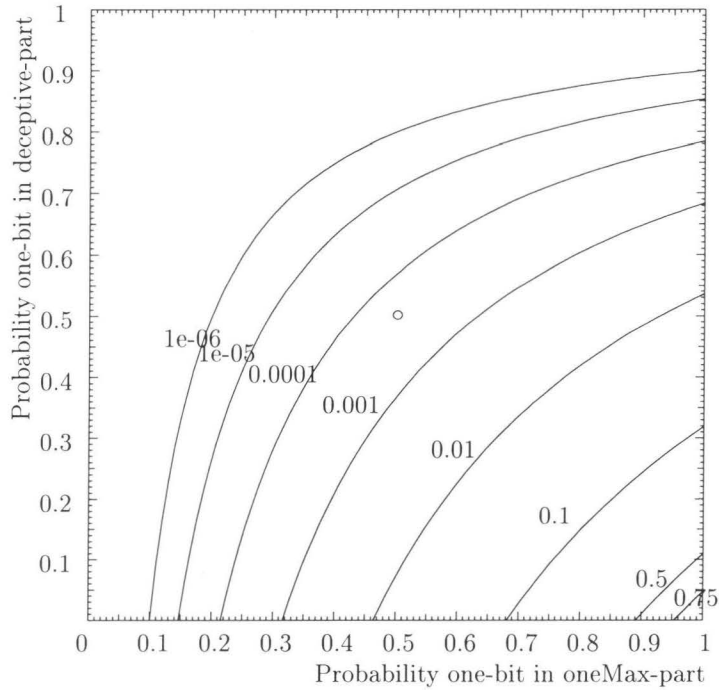


Figure 4.10: Equi-probability lines showing the chance that an optimal individual is obtained when drawing a random individual according to the distribution corresponding to the given points.

offspring.

Figure 4.10 shows the probability of obtaining an optimal solution, when generating a single offspring according to the distribution given by a point in the flow diagram. The probability that an individual drawn uniformly at random from the initial population is the optimal solution is approximately  $2.4 \cdot 10^{-4}$ .

Figure 4.11 shows a comparison of the different selection schemes by observing the evolution of the proportion of optimal solutions as a function of number of generations (top), and as a function of the equivalent number of function evaluations (bottom). The equivalent number of function evaluations is computed by scaling the number of generations proportional to the number of function evaluations used per generation. In most selection schemes the number of function evaluations is proportional to the population size. The only two exceptions are the  $(\mu, \lambda)$ -selection, that uses  $7N$  evaluations per generation, and triple-competition, that uses  $N/3$  evaluation per generation, where  $N$  is the population size.

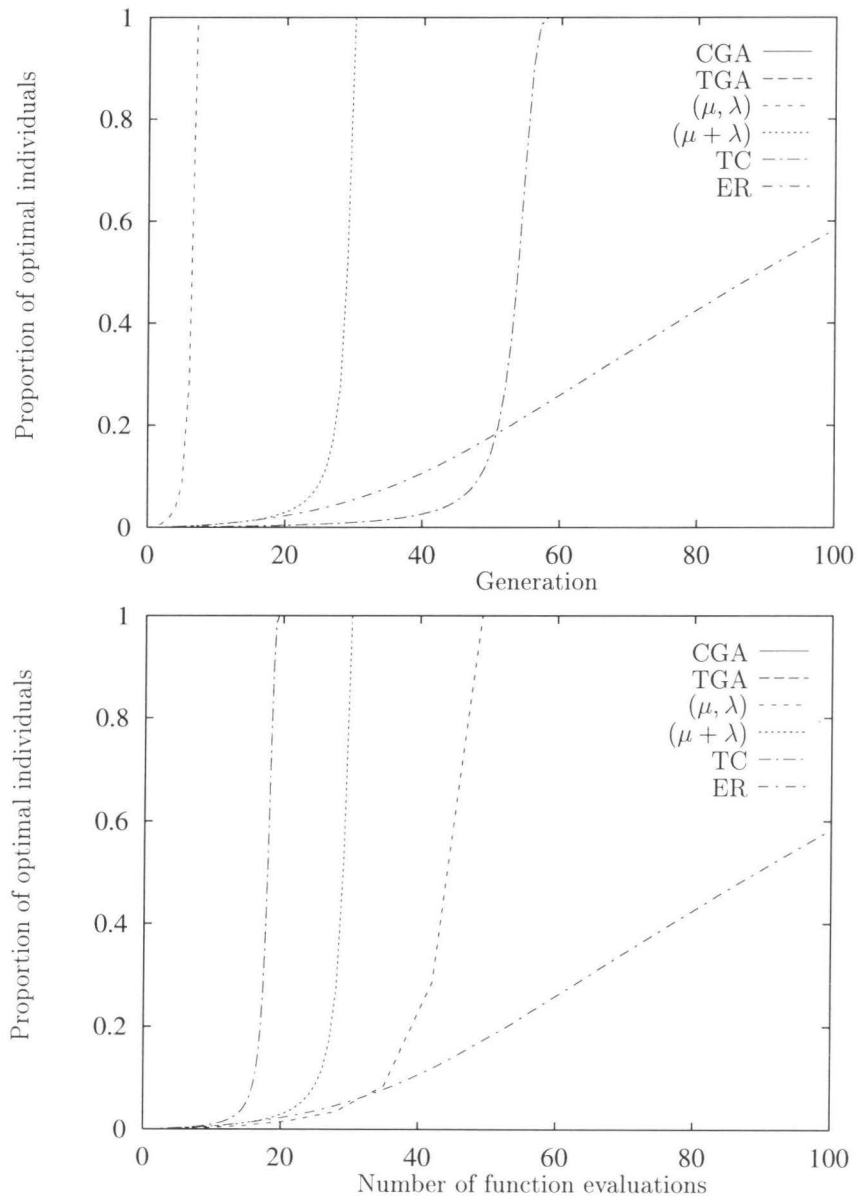


Figure 4.11: Comparison of the proportion of optimal solutions as a function of the number of generations (top), and as a function of the number of function evaluations (bottom) for the different selection schemes

The canonical genetic algorithm and the generational genetic algorithm with tournament selection are hardly visible in this plot, because these two methods do not converge to the optimal solution. In the upper plot the  $(\mu, \lambda)$ -selection converges very fast. It is followed in sequence by  $(\mu + \lambda)$ -selection, triple-competition, and elitist recombination. The bottom plot shows the equivalent number of function evaluations. Here, the triple-competition converges fastest, followed in sequence by  $(\mu + \lambda)$ -selection,  $(\mu, \lambda)$ -selection, and elitist recombination. The shapes of the curves of  $(\mu + \lambda)$ -selection,  $(\mu, \lambda)$ -selection, and triple-competition are basically the same. Initially, after a small proportion of optimal individuals is detected, a rapid convergence to a population consisting of only optimal individuals is observed. So, once an individual is detected that significantly outperforms all other individuals in the population, copies of this individual rapidly fill the complete population. The shape of the curve for elitist recombination is quite different. It moves only slowly upward. This is a result of the direct competition between parents and their offspring. As a result of this competition, it is difficult for an individual to create many duplicates, because a well-performing offspring is likely to replace the parent, and therefore does not result in an increase of the number of copies of this parent.

## 4.6 Summary

In this chapter transmission function models have been given for the canonical genetic algorithm, the generational genetic algorithm with tournament selection, the  $(\mu, \lambda)$ -selection, the  $(\mu + \lambda)$ -selection, triple-competition, and elitist recombination. We showed how these models can be used to trace the evolution of a genetic algorithm with an infinite population size. We discussed under what conditions such models can be traced more efficiently by means of a problem definition in terms of equivalence classes, and we presented a formulation in terms of equivalence classes for problems consisting of a concatenation of functions of unification. Implementations of the models for a problem consisting of two functions of unification were given. The results of tracing the models were visualized by means of population flow diagrams. Using these diagrams, it is possible to see which selection schemes are able to locate the optimum reliably. These population flow diagrams were also used to visualize the run of the infinite population genetic algorithms with the different selection schemes. For three models, i.e. the  $(\mu, \lambda)$ -selection, the  $(\mu + \lambda)$ -selection, and triple-competition, a cross-competition between the two parts of the problem was observed.



## Chapter 5

# Finite population models of genetic algorithms

In practical applications GA's with relatively small populations are used. Such GA's with a finite population can behave quite differently from the GA's with an infinite population. In this chapter an extension of the transmission function model is introduced, such that the behaviour of GA's with a fixed population size can be modelled.

In section 5.1 the differences between finite and infinite population GA's are discussed. It is shown that the path of evolution predicted by distribution based models in general is not followed exactly by a finite population GA. However, if one increases the population size, then the path can be followed more closely. Section 5.2 introduces a framework that later is used to model GA's with finite populations. This framework is a modified version of the transmission function model described in chapter 4. In section 5.3 two models for GA's with finite populations are introduced. The first model actually simulates a random run by means of sampling over distributions. By performing a large number of such runs and computing the averaged results, a model for the expected behaviour of a finite population GA is obtained. The second model splits the search space in a small number of subsets and builds a branching tree by using binomial models. This model represents all possible paths of evolution in a single tree. To validate these models we make a comparison to experimental data obtained by running GA's. The experimental setup is described in section 5.4, followed by the results of the experiments and the simulations in section 5.5. We conclude with a summary in section 5.6.

### 5.1 Finite population GA's

The population size can have a strong influence on the behaviour of a GA. Therefore, real GA's usually do not behave exactly as predicted by an infinite population model. In this section we discuss four steps during the evolution of a GA where randomness is introduced. These steps result in the non-deterministic behaviour of real GA's. Increasing the population size, usually results in a more deterministic behaviour. Next, we discuss

which of the randomized steps of a GA are most sensitive to the size of the population, and therefore result in the largest differences between the evolution of infinite and finite population GA's.

### 5.1.1 Randomness in GA's

In practice we observe that if the population size of a GA is increased, then the behaviour of the GA often becomes more predictable. For example, population sizing has been investigated by Goldberg et al. [GDC93]. In that paper conservative lower bounds on the population size are given such that sufficient copies of all building blocks are present in the population.

A random generator is used to take many decisions during a run of a genetic algorithm. If the same type of decision has to be taken sufficiently often, then the average outcome over all such experiments will approximate the expected value, due to the law of large numbers.

In case of a canonical genetic algorithm the following randomized steps are used during the GA-run:

1. Initial population generation: the random initial population is used to represent a uniform distribution over the search space. The initial population is likely to approximate the uniform distribution better when the population size increases.
2. Parent-selection: given a parent-population, fitness proportional selection gives the proportion of copies that each individual gets in the intermediate population. Given that the expected proportion of a certain type of individuals is sufficiently large the actual proportion of individuals is likely to approximate this value closely.
3. Parent-pairing: given an intermediate population, the parent pairs are selected by uniform selection. If more copies of individuals  $A$  and  $B$  are present, then the expected proportion of parent-pairs  $AB$  found in the intermediate population is approximated better.
4. Evolutionary operators: the evolutionary operators use random decisions to generate offspring. If more instances of a parent pair are present, then the combined offspring of all these pairs will approximate the expected distribution of offspring better.

For each of these randomized steps the number of times that similar decisions have to be taken is proportional to the population size. This explains why GA's with larger populations behave more deterministic.

### 5.1.2 Consequences for finite population GA's

Next, we try to identify which is the step of the evolution process that is most sensitive to the size of the population. Distribution based models, like the transmission function model

described in the chapter 4, model the behaviour of an arbitrary GA with an infinite population size. Given the population of a GA one can compute the corresponding histogram. After normalization this histogram corresponds to a distribution. A finite population can only represent a limited number of distributions over search space  $\mathcal{S}$ . This number increases rapidly when the population size  $n$  increases. Given that  $|\mathcal{S}| \gg n$ , a population of size  $n$  can represent approximately  $(|\mathcal{S}|^n/n!)$  distributions; Therefore, larger populations are better at approximating an arbitrary distribution. As a result, GA's with large populations can better approximate distribution predicted by the transmission function model, and therefore can better follow the path predicted such an infinite population model.

The following two sections give a more detailed discussion on how well a GA with a finite population can approximate the behaviour of the infinite population model. Section 5.1.2.1 discusses the conditions that determine how well a distribution can be modelled by means of a finite sample. (This section may be skipped upon first reading). The consequence of the results of this section for distribution based models of genetic algorithm are then discussed in section 5.1.2.2.

### 5.1.2.1 Approximating distributions

Next, we take a closer look at the conditions that determine how well a distribution can be modelled by means of a finite sample. Let  $\vec{d}$  denote a distribution over a discrete search space  $\mathcal{S}$  (which implies  $\sum_{s \in \mathcal{S}} d_s = 1$ ). Let  $C \subset \mathcal{S}$  denote the  $\epsilon$ -support of  $\vec{d}$  which we define to be the smallest subset of  $\mathcal{S}$  such that  $\sum_{s \in C} d_s \geq (1 - \epsilon)$ , where  $\epsilon$  is a small positive number. Furthermore, for the ease of the argument, we assume that  $|C|$  is even.

Now, we define the spread of a distribution  $\vec{d}$  by  $s(\vec{d}) = |C|$ , and we conjecture that distributions with a smaller spread can usually be represented easier by means of a finite population. To make this more precise we define the distance between two distributions  $\vec{d}$  and  $\vec{p}$  as follows

$$d(\vec{d}, \vec{p}) = \sum_{s \in C} |d_s - p_s|.$$

Now, let  $\vec{d}$  represent an arbitrary distribution over  $\mathcal{S}$ , let  $C_d$  be the  $\epsilon$ -support of  $\vec{d}$ , and let  $\vec{p}_n$  be the distribution of the best matching population of size  $n$ . First, we describe a procedure that (given a  $\vec{d}$ ) constructs the best matching  $\vec{p}_n$ . Next, we discuss a method to construct the  $\vec{d}$  vectors that are most difficult to model by a finite population, and use these to estimate the distance between  $\vec{d}$  and  $\vec{p}_n$ . If  $|C_d| > n$ , then we can construct this optimal  $\vec{p}_n$  as follows. First note that each element of the population covers a fraction  $1/n$  of the distribution given by  $\vec{p}_n$ . The maximal distance  $d(\vec{d}, \vec{p}_n)$  is two, where half of this distance is due to the fact that the elements of  $\vec{p}_n$  are not covered by  $\vec{d}$ , and the other half comes from parts of  $\vec{d}$  not covered by  $\vec{p}_n$ . The optimal  $\vec{p}_n$  covers as much from  $\vec{d}$  as possible. To construct this optimal  $\vec{p}_n$  we first select all elements of  $\vec{d}$  for which  $d_s > 1/n$  and cover these with  $\lfloor nd_s \rfloor$  elements from the population. The population always contains enough elements to perform this operation because  $\sum_{s \in \mathcal{S}} d_s = 1$ . Given that  $\alpha$  elements of the population are used during this step the maximal distance with an arbitrary placement



of the remaining elements is  $d(\vec{d}, \vec{p}_n) < 1 + \frac{n-\alpha}{n}$ . Now the remaining elements are placed such that non-covered parts of  $\vec{d}$  are covered as much as possible. Using this procedure the optimal  $\vec{p}_n$  is obtained.

Now, the maximal distance  $d(\vec{d}, \vec{p}_n)$  is determined by construction of the  $\vec{d}$  that can be approximated worst by  $\vec{p}_n$ . This worst case  $\vec{d}$  is constructed by maximally “frustrating” the optimal construction procedure of  $\vec{p}_n$ . We consider the two cases, depending on the value of  $|C|$ . First, we consider the case that  $n < |C| < 2n$ . In this case a  $\vec{d}$  is constructed, such that all elements  $d_s$  are either  $\frac{1}{2n}$  or  $\frac{3}{2n}$ . This is possible because  $|C|$  is even. Now for the optimal  $\vec{p}_n$  we have  $d(\vec{d}, \vec{p}_n) = |C|\frac{1}{2n}$ , so  $\frac{1}{2} < d(\vec{d}, \vec{p}_n) < 1$ . For  $|C| \geq 2n$  we can always construct a distribution with  $2n$  elements having  $d_s = \frac{1}{2n}$ , which results in a distance  $d(\vec{d}, \vec{p}_n) = 1$ . A distance-contribution of  $\frac{1}{2n}$  is obtained for each element of  $p_s > 0$ , so in order to get an even worse distribution we should get a penalty term larger than  $\frac{1}{2n}$  out of each element  $p_s > 0$ . This is possible if we use a  $\vec{d}$  such that  $d_s < 1/2n$  for all  $s \in \mathcal{S}$ . Now given an arbitrary distribution  $\vec{d}$  with  $d_s < 1/2n$ , the construction method of  $\vec{p}_n$  proceeds as follows. We define an ordering  $r_i$  of the elements of  $\vec{d}$  such that if  $i < j$ , then  $d_{r_i} \geq d_{r_j}$ . The optimal  $\vec{p}_n$  will now cover each of the elements in the range  $r_1$  up to  $r_n$  by an individual. Now we have

$$d(\vec{d}, \vec{p}_n) = \sum_{i=1}^n \left( \frac{1}{n} - d_{r_i} \right) + \sum_{i=n+1}^{|C|} d_{r_i}.$$

Our goal is to find a distribution  $\vec{d}$  that is difficult to model, and thus to find a  $\vec{d}$  such that this sum is maximized. The first sum is maximized by having all  $d_{r_i}$  as small as possible, and thus for all  $i < n$  we should take  $d_{r_i} = d_{r_n}$ . The second sum is maximized by taking all  $d_{r_i}$  as large as possible, and thus for all  $i > n$  we should take  $d_{r_i} = d_{r_n}$ . Because we have  $\sum_{s \in C} d_s = (1 - \epsilon)$ , the most difficult distribution to model is the uniform distribution over  $C$  given by  $d_s = (1 - \epsilon)/|C|$  for all  $s \in C$ . So given that  $|C| > 2n$  the upper bound on the distance is  $d(\vec{d}, \vec{p}_n) = 2 \left( 1 - \frac{n}{|C|} \right)$ . When combining these results we see that given values of  $|C|$  and  $n$ , the worst case distance is given by

$$d(\vec{d}, \vec{p}_n) \leq \begin{cases} |C|\frac{1}{2n} & \text{if } n < |C| < 2n \\ 2 \left( 1 - \frac{n}{|C|} \right) & |C| \geq 2n \end{cases}.$$

Thus, we see that distributions  $\vec{d}$  are more difficult to model when the spread  $s(\vec{d})$  increases. Note that the worst case distribution is an almost uniform distribution. If  $n < |C| < 2n$ , then all  $d_s$  are either  $\frac{1}{2n}$  or  $\frac{3}{2n}$ ; If  $|C| \geq 2n$ , then  $d_s = 1 - \epsilon/|C|$  for all  $s \in C$ .

### 5.1.2.2 Approximating distribution models

We discuss the consequences of these results for the match between finite and infinite population models of GA's, and the influence of population size during the different randomized

steps that GA performs. When comparing the behaviour of the infinite population model and the behaviour of a real GA with a finite population, we expect the largest differences when the  $s(\vec{d})$  is large, where  $\vec{d}$  is the distribution that is predicted by the infinite population model.

The initial population is used to represent this uniform distribution at the start of the evolution process. We have just seen that the uniform distribution is difficult to model by a finite population when  $|\mathcal{S}| > 2n$ .

The recombination steps usually increase the spread significantly. Therefore, we expect that the population size has a strong influence on how well the distribution after recombination can be approximated. For example, let us take a look at the uniform crossover with one offspring. If this crossover is applied to two binary strings that differ in  $m$  loci, then the offspring is one out of  $2^m$  possible offspring-types. Each of these offspring-types has the same probability of being generated, but only one of these offspring-types actually is generated. The evolution of the GA can be influenced strongly by which individual is actually generated; When using a less disruptive crossover operator, the spread of the distribution is increased less.

We consider the influence of the population size during the selection-step to be less important, because selection only changes the proportions of the individuals that already exist in the population. In fact, selection is likely to reduce the spread of the distribution, because it increases the proportion of the best individuals in the distribution. Therefore the distribution after selection  $Sel(\vec{d})$  often can be approximated better than the distribution before selection  $\vec{d}$ . If we want to approximate  $\vec{d}$  and  $Sel(\vec{d})$  with the same error, then the minimal population size to model  $Sel(\vec{d})$  is smaller than or equal to the population size needed to model  $\vec{d}$ .

In this section we mentioned four randomized steps used in a GA. These steps lead to the non-deterministic behaviour of GA with a small population. Due to these randomized steps the GA with finite population behaves different from the GA with an infinite population. In order to behave similar to the infinite population GA, a finite population GA has to approximate the distributions of the population given by the infinite population model. Approximation of such a distribution gets more difficult when the spread of the distribution is larger. If we look at the different randomized steps of a GA, then we see that the initial distribution has a large spread, recombination increases the spread of the distribution, and selection decreases the spread of the distribution. Thus, the initial distribution and the distributions after recombination have the largest spread, and therefore are the relatively difficult to approximate with a small population. Therefore, in the rest of the chapter, we will concentrate on models that explicitly consider the consequences of finite populations for those steps that introduce new offspring (i.e. generation of the initial population and the recombination step).

## 5.2 Modified transmission function models

We are going to extend the transmission function model. Our extension allows us to obtain both the distributions of the surviving parents and of the newly generated offspring. First, a general outline of the framework is given, followed by the implementation details for the different GA's.

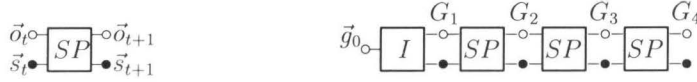
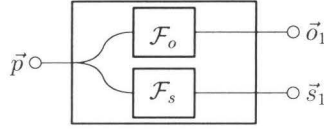
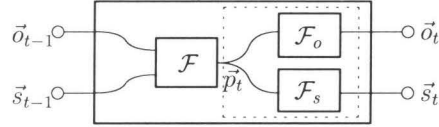


Figure 5.1: Schematic representation of single evolutionary step of the model (left) and a model for a complete evolution of a genetic algorithm (right).

In order to get a general framework evolutionary algorithms are modelled by means of *SP*-boxes, which will be described next. A *SP*-box is shown in the left part of Figure 5.1. A *SP*-box represents a single step of the evolution. The *S* stands for Selection and *P* stands for production; The inputs on the left are the distribution of the offspring  $\vec{o}_t$  of the previous evolutionary step and the distribution of the surviving parents  $\vec{s}_t$  of the previous step. The outputs are the distribution of the newly generated offspring  $\vec{o}_{t+1}$  and the distribution of the surviving parents  $\vec{s}_{t+1}$ . These surviving parents are needed to model GA's involving elitism. The generational genetic algorithm fits easily into this schema; There are no surviving parents and the distribution  $\vec{s}_t$  can be discarded during the computation of the output distributions. In the case of evolution strategies where selection is done after production of a large number of offspring one has to “reorder” the operations such that selection comes before the production step. This reordering is required because the distribution of the offspring is needed as an output.

The infinite distribution model is obtained by concatenating boxes as shown on the right hand side of Figure 5.1. Here  $G_i$  denotes the distribution of population after  $i$  steps of production. This distribution is obtained by combining the distributions  $\vec{o}_i$  and  $\vec{s}_i$  into a single distribution. Later we will describe exactly how one obtains this distribution. To start one needs a special box, called the *I*-box, which models the first step of evolution. For generational GA's the *I*-box is just an *SP*-box, except that there is only one input (recall that for generational GA's the distribution  $\vec{s}_i$  can be discarded). In case of evolution strategies the *I*-box contains a production step.

A schematic representation of a *P*-box is shown in Figure 5.2. The *P*-box takes a parent distribution as its input, and produces two outputs. The first output represents the produced offspring, while the second output represents the surviving parents. Two functions are used inside the *P*-box. The first function is  $\mathcal{F}_o : P \rightarrow P$ , which is the part of the transmission function that produces the offspring; it takes a parent distribution as its input, and it generates the distribution of the offspring. The second function is  $\mathcal{F}_s : P \rightarrow P$ , which gives the distribution of the surviving parents. These functions are specific for the different GA's; Instances of these functions for different GA's will be given below.

Figure 5.2: Implementation of a  $P$ -box.Figure 5.3: Implementation of a  $SP$ -box.

A schematic representation of a  $SP$ -box is given in Figure 5.3, the dashed box inside the  $SP$ -box is a  $P$ -box, which represents the production part of the  $SP$ -box. The selection step is performed by a function  $\mathcal{F} : P \times P \rightarrow P$ . The actual implementation of  $\mathcal{F}$  will depend on the particular GA that is modelled. The parents population is obtained by

$$\vec{p}_t = \mathcal{F}(\vec{s}_{t-1}, \vec{o}_{t-1}).$$

The distribution of the parents and the offspring is obtained by

$$\vec{o}_t = \mathcal{F}_o(\vec{p}_t) \quad \text{and} \quad \vec{s}_t = \mathcal{F}_s(\vec{p}_t).$$

### 5.2.1 Instances for the different genetic algorithms

In a number of subsections, the implementation of the functions  $\mathcal{F}$ ,  $\mathcal{F}_o$ , and  $\mathcal{F}_s$  is discussed for different evolutionary algorithms. These models are based on the transmission function models for GA's with infinite populations, which were given in section 4.1. In the case that an evolutionary algorithm does not involve elitism the parents are always discarded in the sense that one takes  $\mathcal{F}_s(\vec{p}) = \vec{u}$  where  $\vec{u}$  is the uniform distribution.

#### 5.2.1.1 Canonical genetic algorithm

In case of the canonical GA the function  $\mathcal{F}_o$  corresponds to the transmission function model. This results in

$$\mathcal{F}_o(\vec{p}) = \vec{f}(p), \quad \text{where} \quad f_i(\vec{p}) = \sum_{j,k} T_o(i \leftarrow j, k) \frac{f_j f_k}{\bar{f}^2} p_j p_k.$$

When a generational model is applied there are no surviving parents, and therefore we can set  $\mathcal{F}_s(\vec{p}) = \vec{u}$  and  $\mathcal{F}(\vec{s}, \vec{o}) = \vec{o}$ .

### 5.2.1.2 Deterministic $n$ -tournament selection

In case of the generational GA's the function  $\mathcal{F}_o$  corresponds to the transmission function model, which result in

$$\mathcal{F}_o(\vec{p}) = \vec{f}(p), \text{ where } f_i(\vec{p}) = \sum_{j,k} T_o(i \leftarrow j, k) P_{tour}^{(n)}(j, \vec{p}) P_{tour}^{(n)}(k, \vec{p}).$$

When a generational model is applied there are no surviving parents, and therefore we can set  $\mathcal{F}_s(\vec{p}) = \vec{u}$  and  $\mathcal{F}(\vec{s}, \vec{o}) = \vec{o}$ .

### 5.2.1.3 $(\mu, \lambda)$ and BGA selection

In case of  $(\mu, \lambda)$ -selection we get

$$\mathcal{F}_o(\vec{p}) = \vec{f}(p), \text{ where } f_i(\vec{p}) = \sum_{j,k} T_o(i \leftarrow j, k) p_j p_k$$

and  $\mathcal{F}_s(\vec{p}) = \vec{u}$ . The function  $\mathcal{F}$  equals

$$\mathcal{F}(\vec{s}, \vec{o}) = \text{Tr}(\vec{o}, \frac{\mu}{\lambda}),$$

so the distribution of  $\vec{s}$  is discarded during the application of  $\mathcal{F}$ . (For details, see section 4.1.3.)

### 5.2.1.4 $(\mu + \lambda)$ and CHC selection

In case of the  $(\mu + \lambda)$ -selection we get

$$\mathcal{F}_o(\vec{p}) = \vec{f}(p), \text{ where } f_i(\vec{p}) = \sum_{j,k} T_o(i \leftarrow j, k) p_j p_k.$$

and  $\mathcal{F}_s(\vec{p}) = \vec{p}$  (so it yields the parent distribution without modifications), and

$$\mathcal{F}(\vec{s}, \vec{o}) = \text{Tr} \left( \text{Bl}(\vec{s}, \vec{o}, \frac{\mu}{\mu + \lambda}), \frac{\mu}{\mu + \lambda} \right).$$

(For details, see section 4.1.4.)

### 5.2.1.5 Triple-competition selection

In case of the triple-competition selection scheme, the formula given in section 4.1.5 can be split in two parts. The first part, corresponding to the distribution of the offspring, is

$$\mathcal{F}_o(\vec{p}) = \vec{f}(p), \text{ where } f_i(\vec{p}) = \sum_{j,k} T_o(i \leftarrow j, k) P(j, k, \vec{p}),$$

and the second part, corresponding to the distribution of the parents, is

$$\mathcal{F}_o(\vec{p}) = \vec{g}(p), \quad \text{where } g_i(\vec{p}) = \sum_k P(i, k, \vec{p}).$$

These two parts are combined by

$$\mathcal{F}(\vec{s}, \vec{o}) = \text{Bl}(\vec{s}, \vec{o}, \frac{2}{3}).$$

### 5.2.1.6 Elitist recombination

The original transmission function model of elitist recombination, as given in section 4.1.6, already discriminates between newly generated offspring and surviving parents. Both are combined in  $T_{er}(i \leftarrow j, k)$ . We rewrite the transmission function model for elitist recombination such that these two types of individuals are separated:

$$T_{er,i}^{(o)}(\vec{p}) + T_{er,i}^{(s)}(\vec{p}) = \sum_{j,k} T_{er}(i \leftarrow j, k) p_j p_k,$$

where  $T_{er,i}^{(o)}(\vec{p})$  denotes the offspring labelled  $i$ , and  $T_{er,i}^{(s)}(\vec{p})$  denotes the surviving parents labelled  $i$ . Next, we define

$$\mathcal{F}_o(\vec{p}) = \frac{\overrightarrow{T_{er}^{(o)}}(\vec{p})}{|\overrightarrow{T_{er}^{(o)}}(\vec{p})|} \quad \text{and} \quad \mathcal{F}_s(\vec{p}) = \frac{\overrightarrow{T_{er}^{(s)}}(\vec{p})}{|\overrightarrow{T_{er}^{(s)}}(\vec{p})|}.$$

Now, if we introduce the additional parameter

$$\beta = |\overrightarrow{T_{er}^{(s)}}(\vec{p})|,$$

then we get

$$\mathcal{F}(\vec{s}, \vec{o}) = \text{Bl}(\vec{s}, \vec{o}, \beta).$$

The additional parameter  $\beta$  is needed because the proportion of parents that survives will vary during evolution. If the search progresses slowly, then  $\beta$  is close to one. In order to be able to sample over the offspring an integer number of offspring is needed. Given that the population size is  $n$ , we set the number of offspring equal to

$$\max\{\text{Round}(n|\overrightarrow{T_{er}^{(o)}}(\vec{p})|), 1\},$$

where this value is bounded from below by one (the reason being that the null-vector is not normalized).

### 5.3 Finite population models

In this section it is discussed how sampling in combination with the (extended) transmission function model can be used to model the influence of the generation of a finite number of offspring.

In section 5.1 four randomized steps used in GA's were discussed. New individuals are produced during the generation of the initial population, and by the application of evolutionary operators. In section 5.1 we made plausible that the influence of the population size is relatively large during these two randomized steps. We develop two hybrid models, and use these models to check this assumption. These models approximate a finite population during the generation of the initial population and the recombination, and use an infinite population model to perform the other randomized steps.

The first model is a simulation model that uses the transmission function model to evolve a distribution. After each generation the loss of information due to the finite population size is simulated by a sampling step. A single run of this simulation model mimics a single run of a GA. The second model is a branching model, which constructs a branching tree that models all possible paths of evolution. The next subsections discuss these models in detail.

#### 5.3.1 Modelling finite populations by simulation

The infinite population model is combined with simulation-steps that mimic the behaviour of the recombination step on a finite population, and with simulation-steps that mimic the influence of the finite initial population. In this way a hybrid model is obtained that follows the infinite population models during selection and mating, while behaving like a finite population GA during the generation of the initial population and during recombination. We first discuss the conditions under which we can simulate a finite sample given the distribution of the infinite sample. Next, it is discussed which steps of the GA allow such a simulation approach, and the details of the corresponding hybrid model are given.

We start with a discussion about the type of distributions for which a finite sample can be modelled. Let us assume that we know the expected distribution of a sample, denoted by  $\vec{d}$ . Now, if a sample consists of independent identically distributed sample-points, then  $\vec{d}$  can be interpreted in two different ways. First,  $\vec{d}$  represents the expected distribution of a large sample; Second,  $\vec{d}$  can be interpreted as the distribution used to draw a single sample-point. Now, given the expected distribution of the sample, one can simulate a population of size  $n$  by generating sample-points according to  $\vec{d}$ . After normalization of the histogram of this sample a new distribution is obtained, which we denote by  $\vec{p}$ . This step actually corresponds to approximating  $\vec{d}$  by a sample of size  $n$ . For a small sample this actual distribution can differ quite a lot from the expected distribution. If the size of the sample  $n$  is increased, then the difference between  $\vec{p}$  and  $\vec{d}$  is likely to decrease, and in the limit of  $n \rightarrow \infty$  the  $\vec{p}$  converges to  $\vec{d}$ . Thus,  $\vec{p}$  can be considered as a randomized simulation  $\vec{d}$ , and this simulation is likely to approximate  $\vec{d}$  better when increasing the size of the sample that  $\vec{p}$  corresponds to.

Next, we consider the use of such a simulation approach to model the influence of the population size during certain steps of the GA. Such a simulation is possible for distributions representing a sample of independent identical distributed sample points. Thus, such an approach can only be used for a population in which all individuals are generated independently of each other. This is the case during the generation of the initial population and during application of a recombination operator that produces only one offspring. In case of probabilistic selection (such as fitness proportional selection or tournament selection) the individuals in a population are also generated independently of each other (provided that selection variance reduction mechanism is used, see section 2.3.1 for details). Deterministic selection schemes do not select individuals independently of each other, and thus cannot be simulated in this manner.

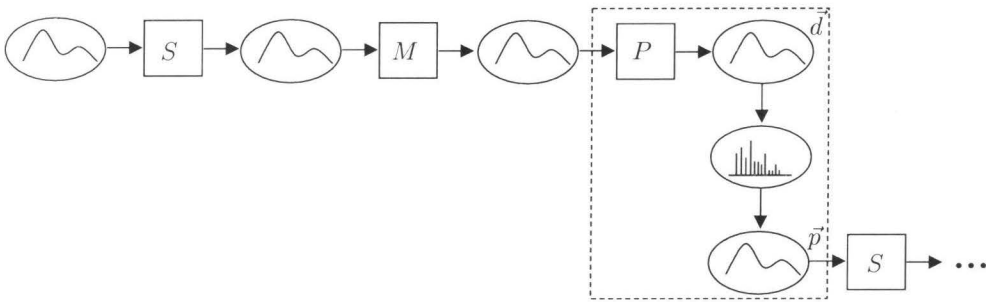


Figure 5.4: Simulate a finite recombination-step by sampling for a generational GA.

An example of simulating a finite population by sampling over a distribution is given in Figure 5.4. In the top row the evolution of a single generation is shown; Selection, mating, and recombination are performed in sequence on a distribution. These distributions represent a population of infinite size. After the recombination step a distribution  $\vec{d}$  is obtained. Next, a random sample of size  $n$  is generated according to this distribution. The distribution of this sample, denoted by  $\vec{p}$ , is computed. Next,  $\vec{p}$  is used as an input for the selection step of the infinite population model. The operations inside the dashed box mimic recombination applied to a finite population.

Figure 5.4 shows a single generation out of a run of the simulation model for a generational GA. Using the *SP*-boxes a run of an arbitrary GA can be modelled. The right-hand side of Figure 5.1 shows the model based on *SP*-boxes. In the *SP*-box a separation is made between the newly generated offspring, denoted by  $\vec{o}_t$ , and the surviving parents, denoted by  $\vec{s}_t$ . The places indicated by open circles represent the points where simulation of a sample is used to model the behaviour of a finite population. These open circles indicate all points where new individuals are generated.

Hence, we introduced in this section a simulation model that uses an infinite population during the selection step and the mating step, and a finite population during the generation of the initial population and during recombination. This simulation model is a hybrid model



in the sense that part of the evolution process is performed with an infinite population and part is performed with a finite population. This model allows us to differentiate between the influence of the population size during the different steps of evolution.

### 5.3.2 Branching over binomially distributed finite populations

Next we are going to consider a model that represents all possible paths of evolution in a single tree. This model splits the search space in a small number of subsets and builds a branching tree using binomial distributions.

Figure 5.1 denotes only a single possible path of evolution. When modelling all possible paths of evolution a tree-like structure is obtained. The input of each node is the current distribution of surviving parents and newly generated offspring, and the output is a set of branches, each one denoting a possible actual distribution of the population of size  $n$ . These branches are then fed to different *SP*-boxes.

Following all possible traces of evolution for a finite population GA requires a lot of computation. Given the cardinality of the search space  $|\mathcal{S}|$ , and the population size  $n$ , a single node has approximately  $(|\mathcal{S}|^n/n!)$  outputs, when  $|\mathcal{S}| \gg n$ . Even when a representation by means of equivalence classes is used, the amount of outputs is going to be large. The total amount of computation for the complete model is very large, because the number of nodes in the tree describing the complete evolution is approximately  $(|\mathcal{S}|^n/n!)^G$ , where  $G$  is the number of generations. To reduce the amount of computation an approximate model is introduced, where a relatively small number of branches is generated for each node. To do so, the set of all possible bit-strings  $\mathcal{S}$  is partitioned in two disjoint subsets,  $S$  and  $\bar{S}$ , such that  $S \cup \bar{S} = \mathcal{S}$ . These sets are for example chosen in such a way that elements of  $S$  contain certain parts of the optimal solution, while the elements of  $\bar{S}$  do not contain these parts. The number of elements in  $S$  can vary between 0 and  $n$ , so the number of outputs of a node is  $n + 1$ .

The same procedure can be repeated by splitting the set  $S$  in two disjoint sets  $S_1$  and  $\bar{S}_1$ . In this way it is possible to continue until the subsets exactly defines a specific population. Using only a few subsets one can already observe a good match between the model and actual GA-runs, as will be shown in the rest of this chapter.

We will focus on the case where the search space  $\mathcal{S}$  is split in two disjoint sets  $S$  and  $\bar{S}$ . The tree corresponding to this case is shown in Figure 5.5. A more detailed description of the computation is given by the dataflow graphs in Figures 5.6 and 5.7, which will be discussed later in this section. Figure 5.5 shows the computation by means of a tree. The nodes are labelled with a sequences of numbers. These numbers correspond to the number of elements in the population that belong to  $S$ . For example a node labelled 134 corresponds to a path where one individual belonging to  $S$  is present in the initial population, three such individuals are present in the first generation, and four such individuals are present in the second generation. The number of offspring produced during a single generation is  $\lambda$ . The arcs are labelled with the probability of the corresponding transition. The root of the tree corresponds to the distribution used to draw the initial population (denoted by  $\vec{g}_0$ ). The first branch corresponds to the case that no individual in

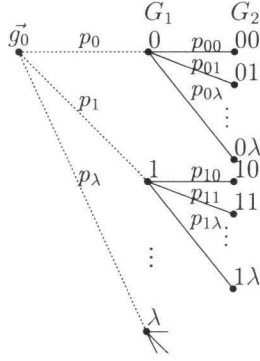


Figure 5.5: Picture of a branching tree for a finite-population model

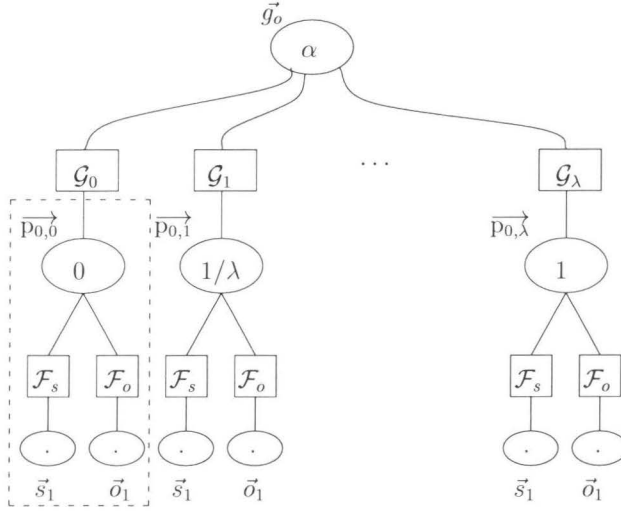
the population is contained in  $S$ , the second branch corresponds to case that exactly one individual in the population belongs to the set  $S$ , and the branch with label  $k$  corresponds to case that  $k$  individuals belong to the set  $S$ . The probability of branch  $k$  is denoted by  $p_k$ .

### 5.3.2.1 Distributions corresponding to finite populations

Assume that we have a transition to a node in which  $k$  individuals should belong to set  $S$  (out of  $\lambda$ ). If the proportion of individuals belonging to  $S$  in an distribution  $\vec{g}$  is equal to  $p$ , then the distribution corresponding to  $k$  out of  $\lambda$  individuals belonging to the set  $S$  is constructed as follows. First, the actual proportion  $p' = k/\lambda$  is computed. The proportion of all elements that belong to the set  $S$  are multiplied by factor  $p'/p$ , while the proportions of the other elements are multiplied by a factor  $(1 - p')/(1 - p)$ . So, the relative proportions of individuals that belong to the same set remain unchanged, but the relative proportions of individuals that belong to different sets change by this procedure. Let  $\mathcal{G} : \mathbb{R} \times P \rightarrow P$  be the function that performs this operation, where the real-valued parameter is the proportion of individuals that belong to the set  $S$ , and where  $P$  is the space of all possible distributions over the search space. The following identity holds

$$\sum_{k=0}^{\lambda} \text{Bin}(\lambda, k, p) \mathcal{G}\left(\frac{k}{\lambda}, \vec{g}\right) = \vec{g}.$$

Here  $\text{Bin}(n, k, p)$  is the binomial distribution that represents the probability of obtaining  $k$  successes out of  $n$  independent random events, where  $p$  is the probability of success. To prove this equality let us consider a single element  $g_i$  from distribution  $\vec{g}$ , and let us assume that element  $i$  belong to the set  $S$ . We have  $\sum_{k=0}^{\lambda} \text{Bin}(\lambda, k, p) \frac{k}{\lambda} g_i = g_i$ . After some rewriting one obtains  $\sum_{k=0}^{\lambda} \text{Bin}(\lambda, k, p) k = \lambda p$ , and hence the identity holds. The same result holds for an element of the set  $\bar{S}$ , using  $\text{Bin}(\lambda, k, p) = \text{Bin}(\lambda, \lambda - k, 1 - p)$ .

Figure 5.6: Data flow for generation  $G_0$  (for ES)

### 5.3.2.2 Transitions from generation $G_0$

The root of the branching tree is the expected distribution of the initial population. A detailed view of the computation at generation  $G_0$  is given by the dataflow graph in Figure 5.6. The ovals represent distributions, while the boxes represent operations on distributions. The dashed box marks a set of operations that correspond to a single  $P$ -box. At the top the distribution of the initial population, with label  $\vec{g}_0$ , is shown. The expected proportion of individuals in set  $S$  is denoted by  $\alpha$ . Next, the functions  $\mathcal{G}_k$  that produce the new distribution in which proportions  $\frac{k}{\lambda}$  ( $k = 0, \dots, \lambda$ ) of individuals belonging to the set  $S$  are given. The resulting distributions are labelled  $\vec{p}_{0,k}$ . Given one such a distribution, the surviving parents and offspring are computed by means of the functions  $\mathcal{F}_s$  and  $\mathcal{F}_o$ . So, after a single generation each trace is described by means of two distributions  $\vec{s}$  and  $\vec{o}$ .

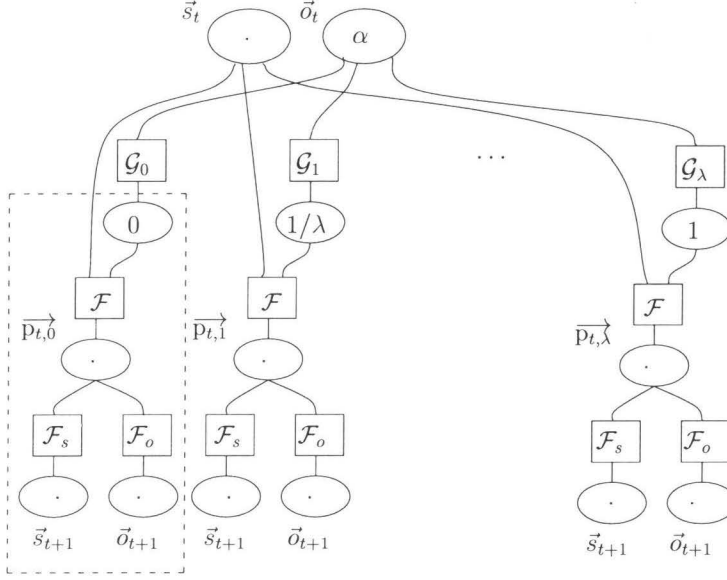
Now let us assume that the population of parents contains  $\mu$  individuals that are used to produce  $\lambda$  offspring, and the distribution of the parents in the initial population is  $\vec{g}_0$ . The distribution of the node labelled  $k$  of the first generation is

$$\vec{p}_{0,k} = \mathcal{G}\left(\frac{k}{\mu}, \vec{g}_0\right),$$

which is denoted by the symbol  $\mathcal{G}_k$  in Figure 5.6. The probability of this transition is

$$p_k = \text{Bin}(\mu, k, p),$$

where  $k$  is the proportion of individuals from  $\vec{g}_0$  that belongs to the set  $S$ .

Figure 5.7: Picture of the data flow for a single branch of generation  $G_t$  for  $t > 0$ 

### 5.3.2.3 Transitions from generation $G_t$

A detailed view of the computation of generation  $G_{t+1}$  from generation  $G_t$  is given by the dataflow graph in Figure 5.7. The dashed box denotes a set of operations that correspond to a single *SP*-box. The evolution of a single trace is shown. On top the inputs of a node consisting of the two distributions  $\vec{s}_t$  and  $\vec{o}_t$  are given. The function  $G_k$  is applied to this distribution in order to get a distribution where a proportion  $k/\lambda$  individuals are in the set  $S$ . The function  $\mathcal{F}$  is applied to this modified offspring distribution and the distribution of surviving parents to obtain the distribution  $\vec{p}_{t,k}$ :

$$\vec{p}_{t,k} = \mathcal{F}(\vec{s}_{t-1}, G_k(\frac{k}{\lambda}, \vec{o}_{t-1})).$$

Its probability is given by

$$p_{t,k} = p_x \text{Bin}(\mu, k, p),$$

where  $p_x$  is probability of reaching node  $P_{t,x}$  that is the predecessor of the node under consideration. There is a unique path from the root of the tree to the node  $P_{t,x}$  and the probability  $p_x$  is the product of the individual transition probabilities along this path.

The expected distribution  $\vec{g}_t$  is given by:

$$\vec{g}_t = \sum_{k=0}^{\lambda} p_{t,k} P_{t,k}$$

i.e. a weighted average over the distributions at this depth in the tree.

### 5.3.2.4 Merging of transitions

A complete evolution tree consists of  $(N + 1)^G$  branches, where  $N$  is the size of the population, and  $G$  is the number of generations. It is not feasible to follow all these branches. In order to obtain a feasible computation the set of followed branches is limited to those branches that have a probability above a certain threshold  $\delta$ . Branches that are not followed will be put together with the nearest branch that is being followed. The proportion of individuals in the set  $S$  is assigned a weighted average according to the formula  $k' = \sum_{k=l}^m p_{xk} p_x k$ , where  $x$  is the label of the node in the preceding generation from which the branch originates,  $l$  and  $m$  denote the range of the branches that are combined to a single branch, and  $p_x$  denotes the probability that the evolution ends in node  $x$ . The resulting value of  $k'$  does not have to be integer anymore in case branches are merged. This probability of the joined branch is set to  $\sum_{k=l}^m p_{xk}$ . If none of the branches has a probability above the threshold, then the net effect is that the infinite population model is traced.

Given the lower bound on the probability of a traced branch, the total number of followed branches of a single generation is smaller than  $\lfloor \frac{1}{\delta} \rfloor$ . The maximal width of the tree at generation  $G_i$  is  $n^i$ , so the upper bound on total number of branches followed is given by

$$n_t \leq \sum_{i=0}^G \min\{\lfloor \frac{1}{\delta} \rfloor, n^i\} \leq 1 + \lfloor \frac{1}{\delta} \rfloor G.$$

A tighter upper bound is obtained by observing that the  $n^i \leq \lfloor \frac{1}{\delta} \rfloor$  for generations  $i$  such that  $i < \frac{n}{\log(\lfloor \frac{1}{\delta} \rfloor)}$ . Using this bound one obtains

$$n_t \leq n^\alpha / \ln(n) + \beta \lfloor \frac{1}{\delta} \rfloor,$$

where  $\alpha = \max\{g + 1, G\}$  and  $\beta = \min\{0, G - g\}$ .

Each node of the tree given in Figure 5.5 corresponds to a constant amount of computation because each node corresponds to a single evaluation of the functions  $\mathcal{G}$ ,  $\mathcal{F}$ ,  $\mathcal{F}_s$ , and  $\mathcal{F}_o$ . Therefore these bounds are also the bounds on the total amount of computation.

## 5.4 Experiment setup

To test the models we implemented the models for different GA's, implemented the corresponding real GA's, and applied these to the cross-competition problem. This problem (which is defined in section 4.4) consists of two parts, the first part corresponds to a oneMax problem, and the second part correspond to a deceptive trap-function. The individuals are coded as bit-strings of length 12 for the real GA's, and are coded by means of 49 equivalence classes (see section 4.3 for details) for the models.

When coding individuals in linear strings of bits, genetic drift influences the performance of finite population genetic algorithms. The cross-competition problem is susceptible to genetic drift. Within a partition corresponding to a single function of unitation all

loci have similar importance, so there is no reason for some loci to converge faster than other loci. But due to genetic drift some loci might converge faster than others within one specific run of the GA. This type of genetic drift will be ruled out because a formulation in terms of equivalence classes will be used. The underlying assumption is that all loci belonging to the same partition are governed by the same probability of containing a one-bit. In order to get rid of the genetic drift in the real genetic algorithm a modified version of the uniform crossover operator is applied. This crossover accepts two parent individuals. For both individuals an intermediate individual is created in which all bits within the same partition are shuffled. Such an intermediate individual has the same fitness as the original individual because the fitness of a part is given by a function of unitation. Next, uniform crossover is applied to these intermediate individuals. Due to this shuffling, the probability of observing a one-bit in each of the loci of a partition becomes equal again, so genetic drift of individual loci is removed. The removal of genetic drift during the experiments allows us to focus on the influence of the other assumptions.

Next, the details about the implemented GA's are presented. For the generational genetic algorithms, and  $(\mu \nmid \lambda)$ -selection schemes, we have two implementations. The first implementation is a straightforward implementation of the algorithm, while the second implementation contains a selection method that reduces the selection variance (see sections 2.3.1, and 2.9 for details).

In case of triple-competition and elitist recombination the algorithms are defined in such a way that the selection variance is small already.

We consider the population size  $n$  of the  $(\mu \nmid \lambda)$ -selection to be equal to  $\mu$ , because this is the size of the smallest population that is used to pass information to subsequent generations; Therefore, this is considered as an appropriate measure for the maximal amount of information that can be propagated to subsequent generations. Given this definition of  $n$  the amount of computation required to evaluate a generation  $G_t$ , where  $t > 0$ , differs per algorithm. The generational GA's, the  $(\mu + \lambda)$ -selection, and the elitist recombination use  $n$  function evaluations per generation, the  $(\mu, \lambda)$ -selection uses  $n\lambda/\mu$  function evaluations per generation ( $\lambda = 7\mu$ ), and the triple-competition uses  $n/3$  function evaluations per generation.

## 5.5 Experiments

In this section the results of the experiments are discussed.

The simulation model can be applied directly to the problem. In case of the branching model the set  $S$  has to be chosen. Here, we define the set  $S$  to correspond to exactly those elements of  $\mathcal{S}$  that contain an optimal building block in the deceptive part. Because elements of the set  $S$  are relatively likely to result in convergence to the global optimum, it is of interest to know which proportion of the population is part of this set. This proportion might differ between different runs, but it is possible to estimate the probabilities that a certain proportion of the population is contained in the set  $S$  in generation  $t$ .

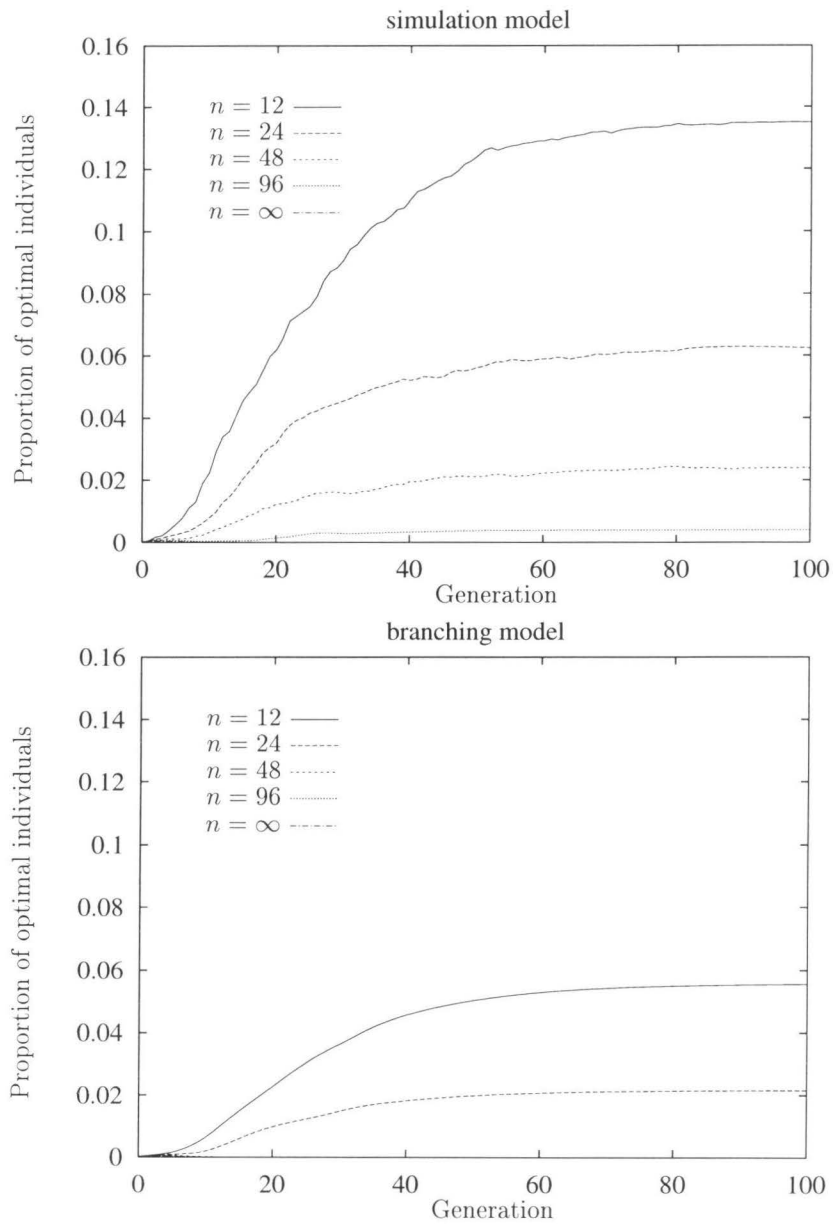


Figure 5.8: Proportion of optimal solutions for the canonical genetic algorithm both for the simulation model (top) and the branching model (bottom).

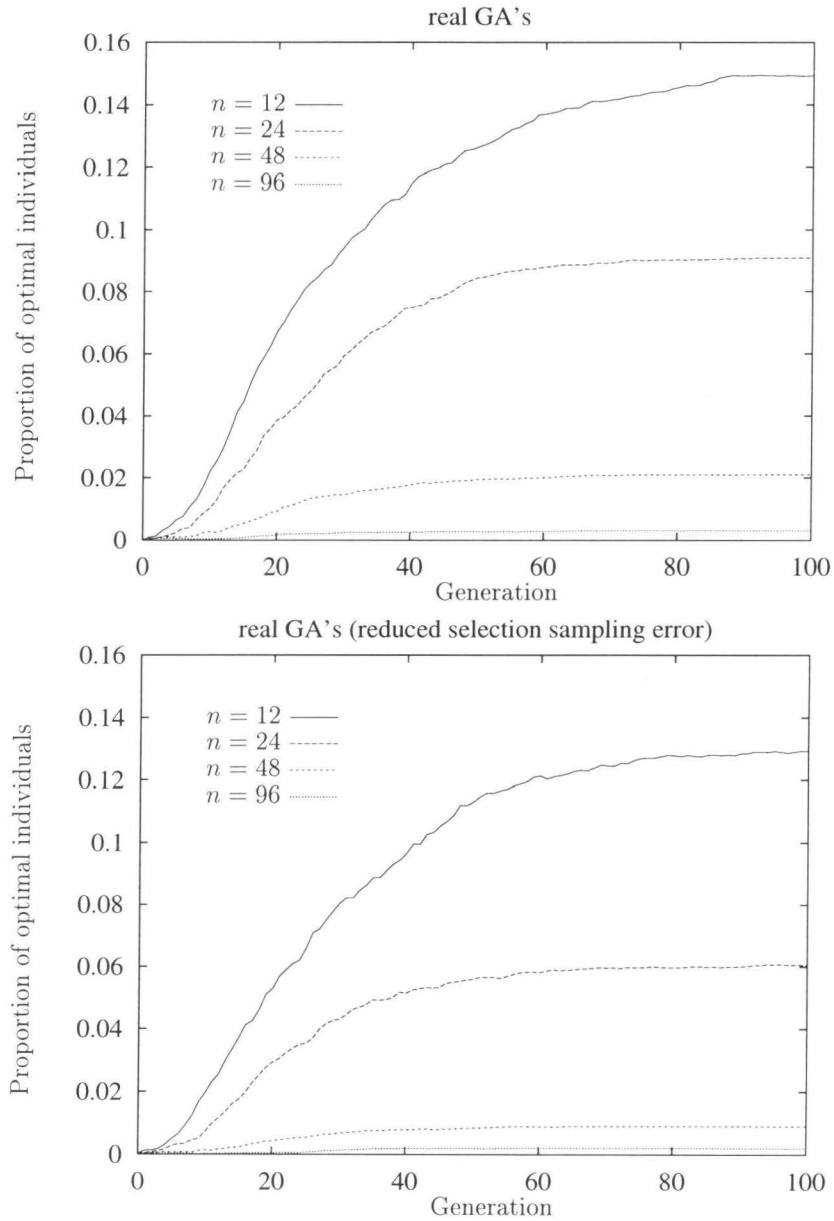


Figure 5.9: Proportion of optimal solutions for the canonical genetic algorithm both real GA's both without (top) and with (bottom) reduction of selection variance.



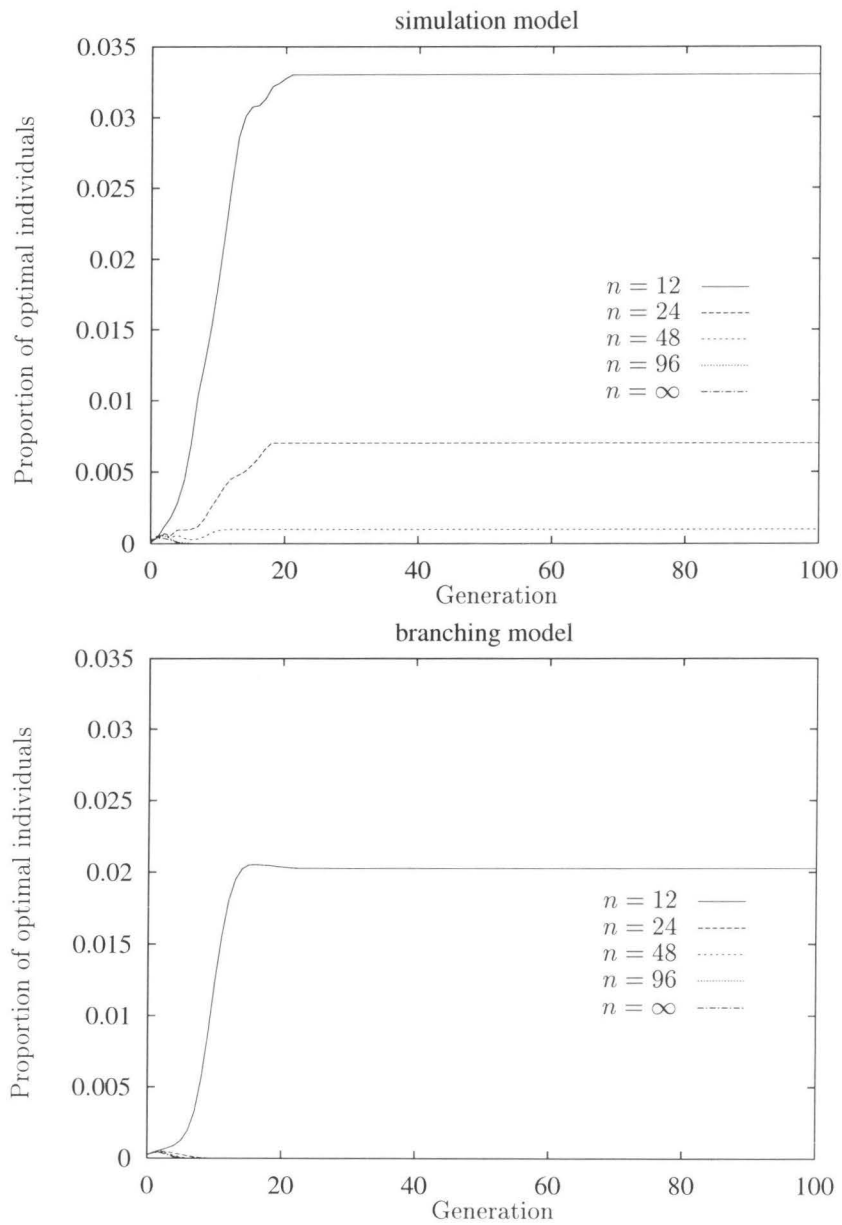


Figure 5.10: Proportion of optimal solutions for the generational genetic algorithm with 2-tournament selection both for the simulation model (top) and the branching model (bottom).

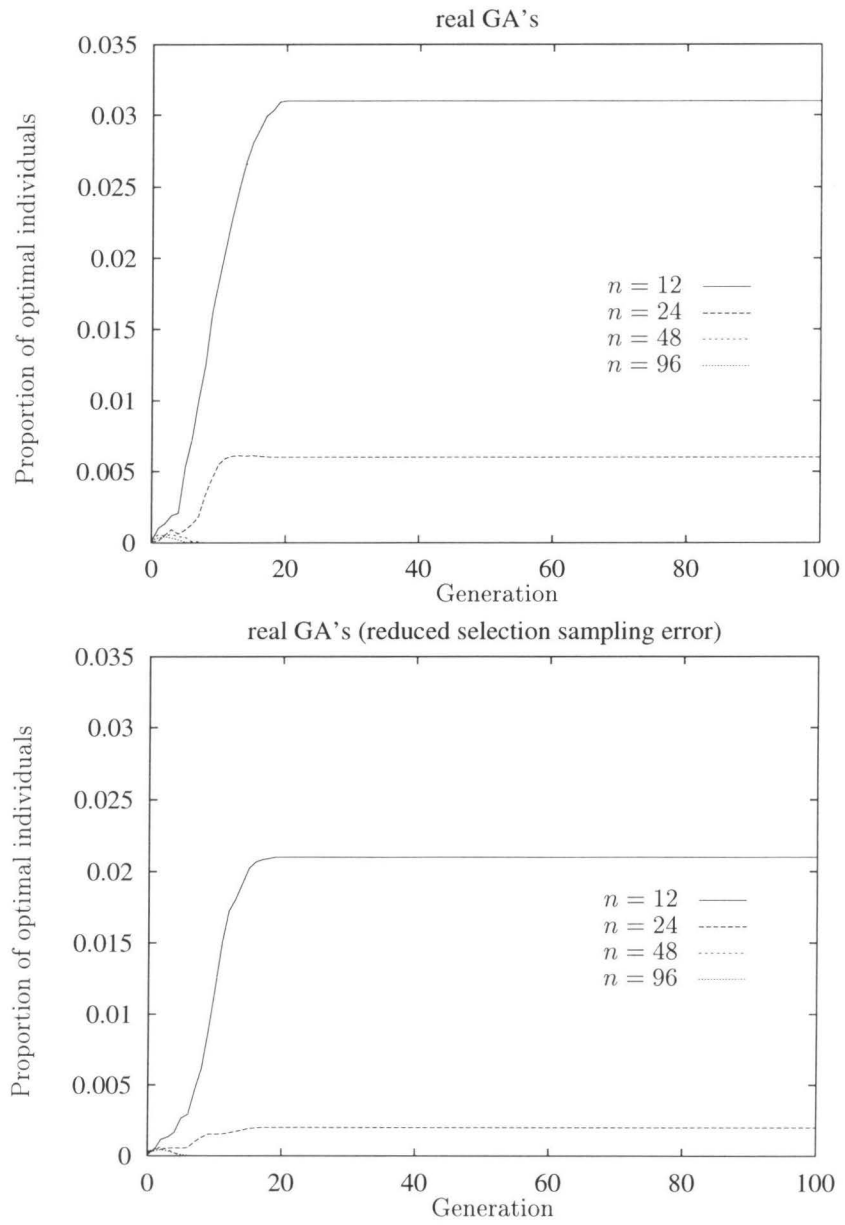


Figure 5.11: Proportion of optimal solutions for the generational genetic algorithm with 2-tournament selection both without (top) and with (bottom) reduction of selection variance.

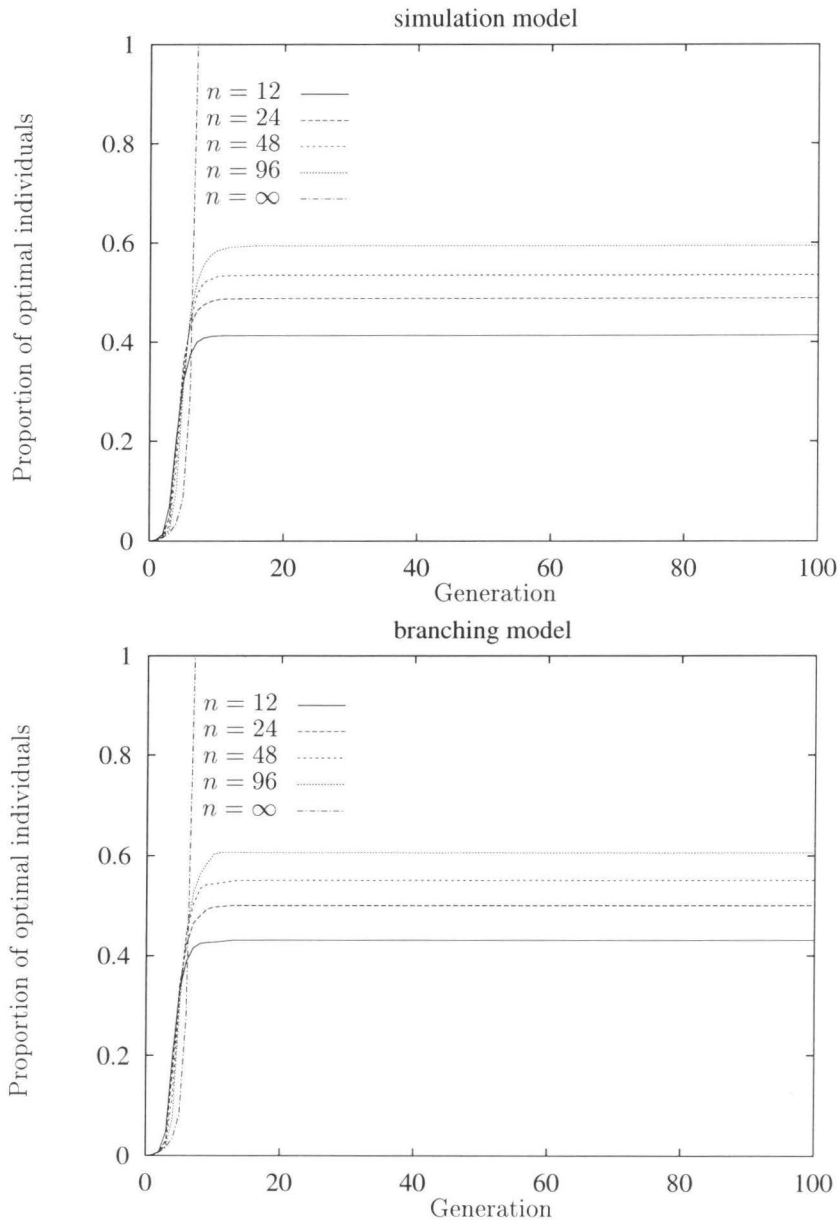


Figure 5.12: Proportion of optimal solutions for the  $(\mu, \lambda)$  selection (BGA) both for the simulation model (top) and the branching model (bottom).

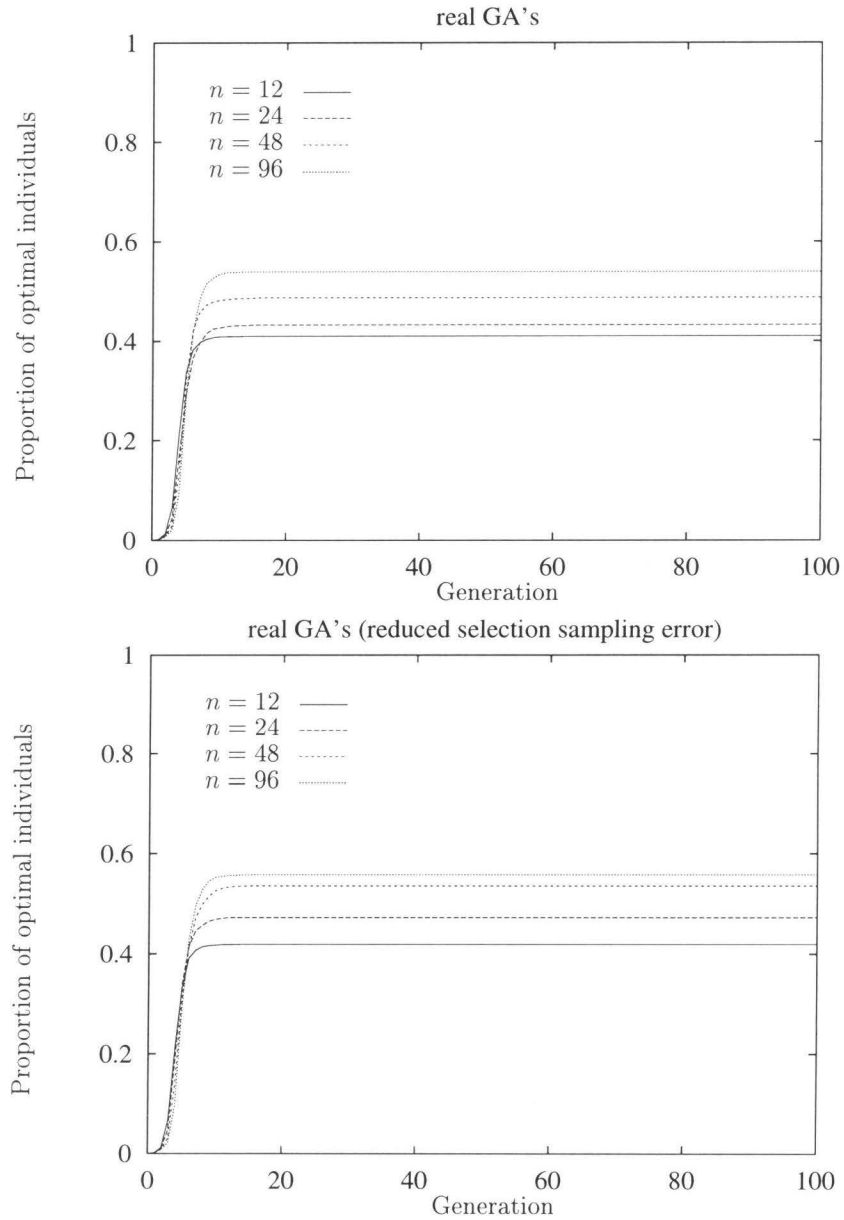


Figure 5.13: Proportion of optimal solutions for the  $(\mu, \lambda)$  selection (BGA) both without (top) and with (bottom) reduction of selection variance.

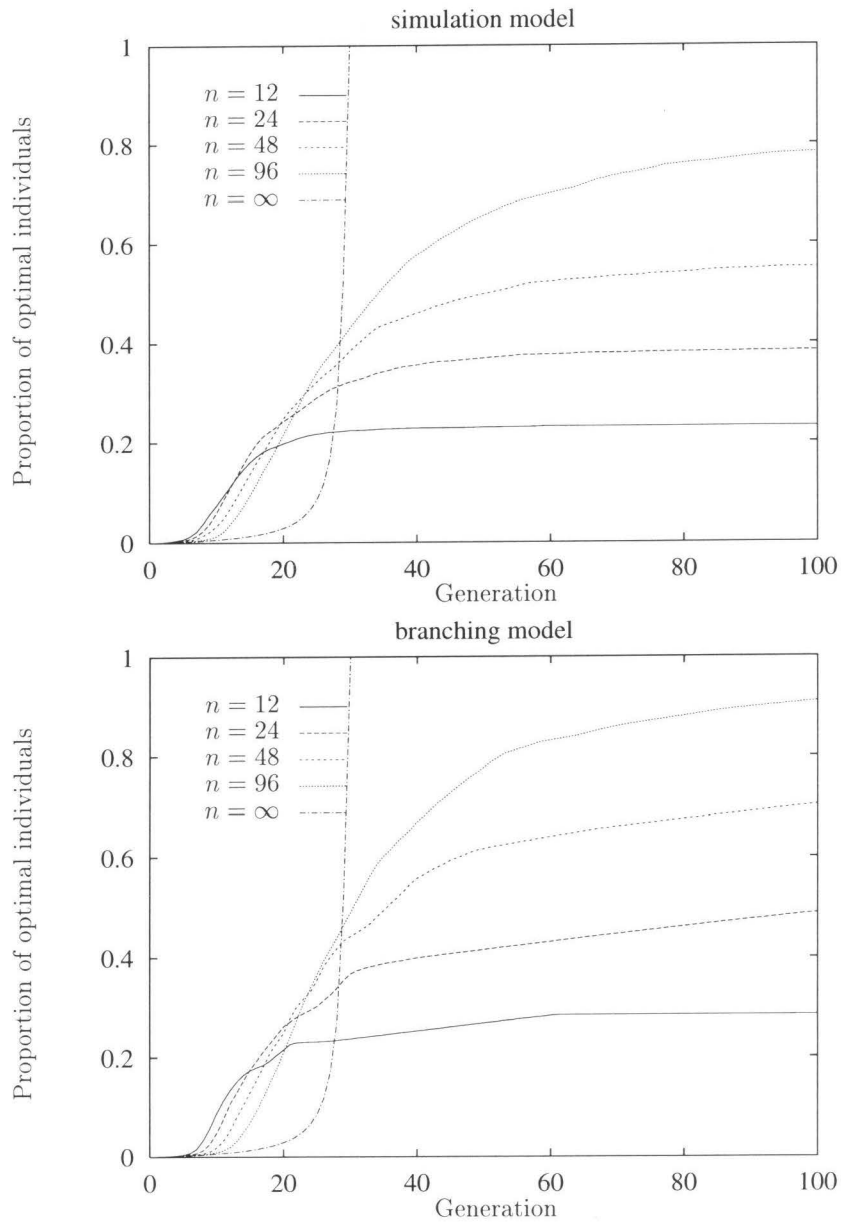


Figure 5.14: Proportion of optimal solutions for the  $(\mu + \lambda)$  selection (CHC) both for the simulation model (top) and the branching model (bottom).

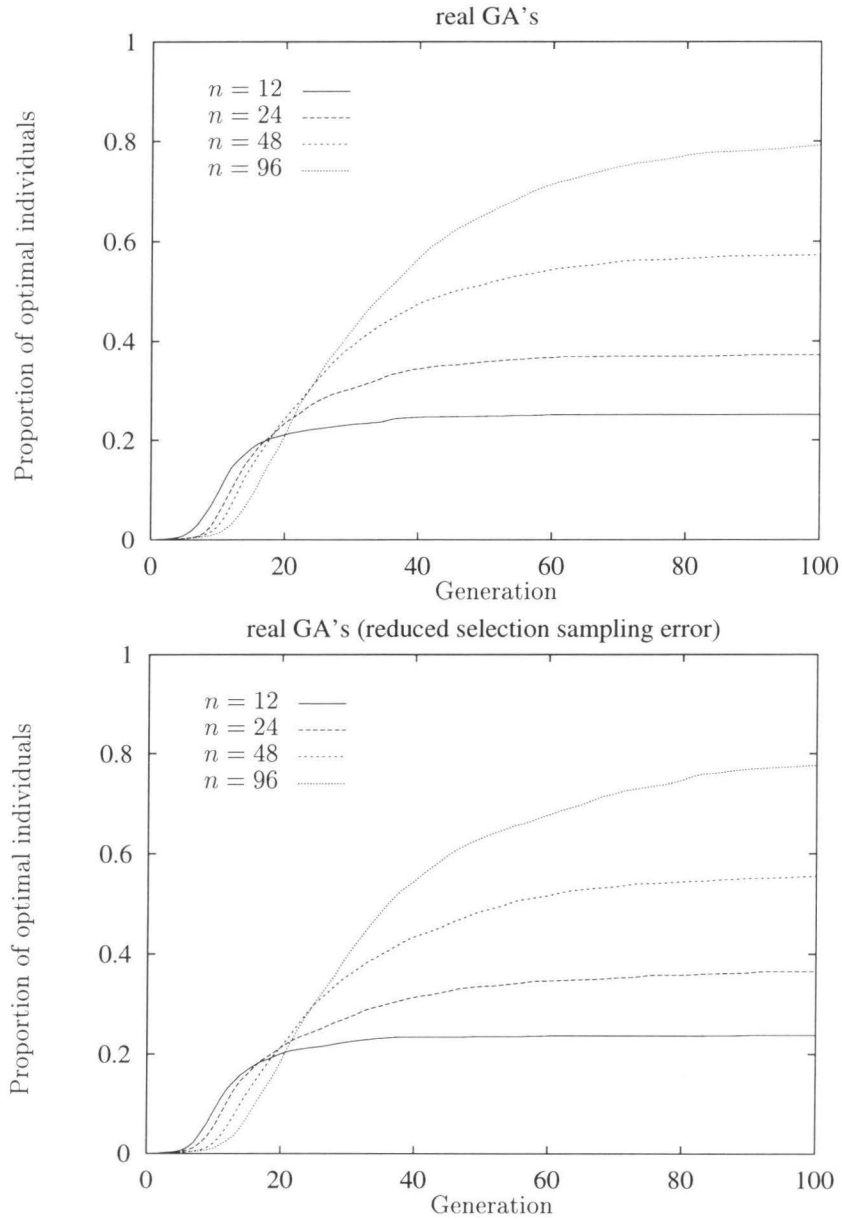


Figure 5.15: Proportion of optimal solutions for the  $(\mu + \lambda)$  selection (CHC) both without (top) and with (bottom) reduction of selection variance.

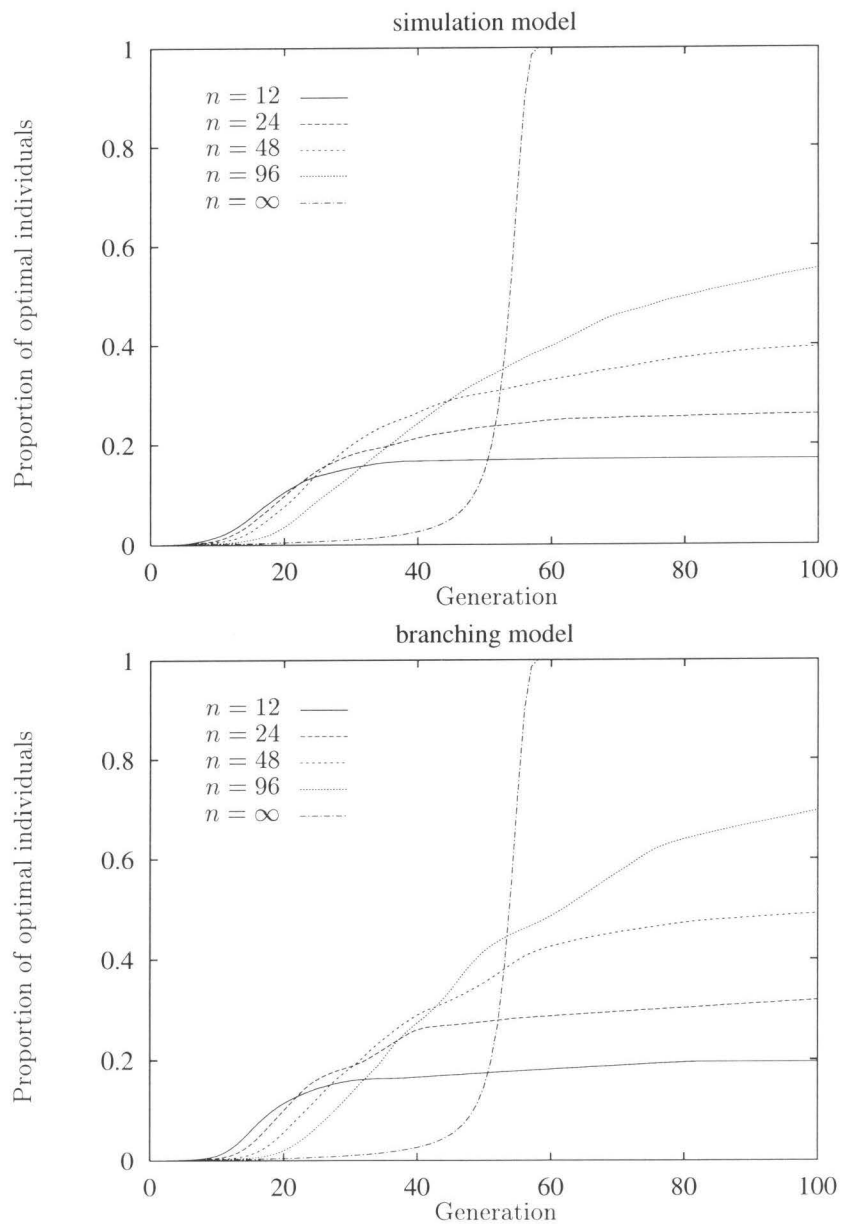


Figure 5.16: Proportion of optimal solutions for triple-competition both for the simulation model (top) and the branching model (bottom).

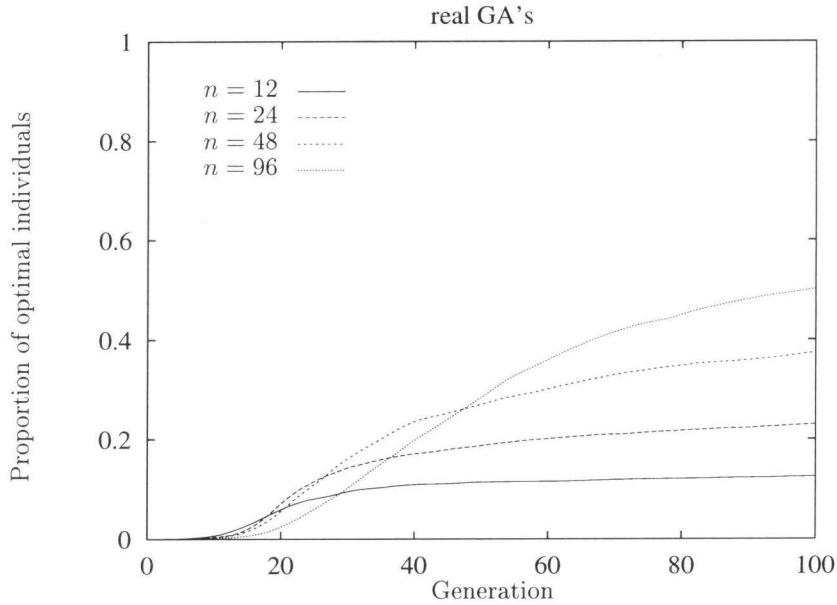


Figure 5.17: Proportion of optimal solutions for triple-competition.

The probability that  $k$  individuals out of a set of  $\lambda$  offspring belong to the set  $S$  is given by the binomial distribution  $\text{Bin}(\lambda, k, p)$  where  $p$  denotes the expected proportion of elements that belong to the set  $S$ . Given that  $k$  out of  $\lambda$  offspring belong to the set  $S$  the actual proportion is  $p' = k/\lambda$ . For large populations  $p'$  will be close to  $p$ , but for small population sizes the difference between these two might be rather large.

Both tracing of the models and real experiments have been performed for all the selection schemes discussed in previous sections. The modelling is done by tracing the behaviour of both the finite population model and the infinite population model. Plots are shown for population sizes  $n = 12, 24, 48, 96$ , and  $\infty$ . (Recall that  $n$  refers to the number of parent individuals that are used to create the set of offspring.) The results for the simulation model and the real GA's are obtained by computing the averages over 1,000 independent runs.

For all selection schemes a number of plots are shown. The first plot shows the results for the simulation model, the second plot the results for the branching model, the third plot shows results of the real GA, and the fourth plot shows the results for a real GA with reduction of selection variance (if applicable). In these plots the horizontal axis shows the index of the generation, and the vertical axis shows the proportion of optimal individuals.

Figures 5.8 and 5.9 show the results when using a generational genetic algorithm with fitness proportional selection.



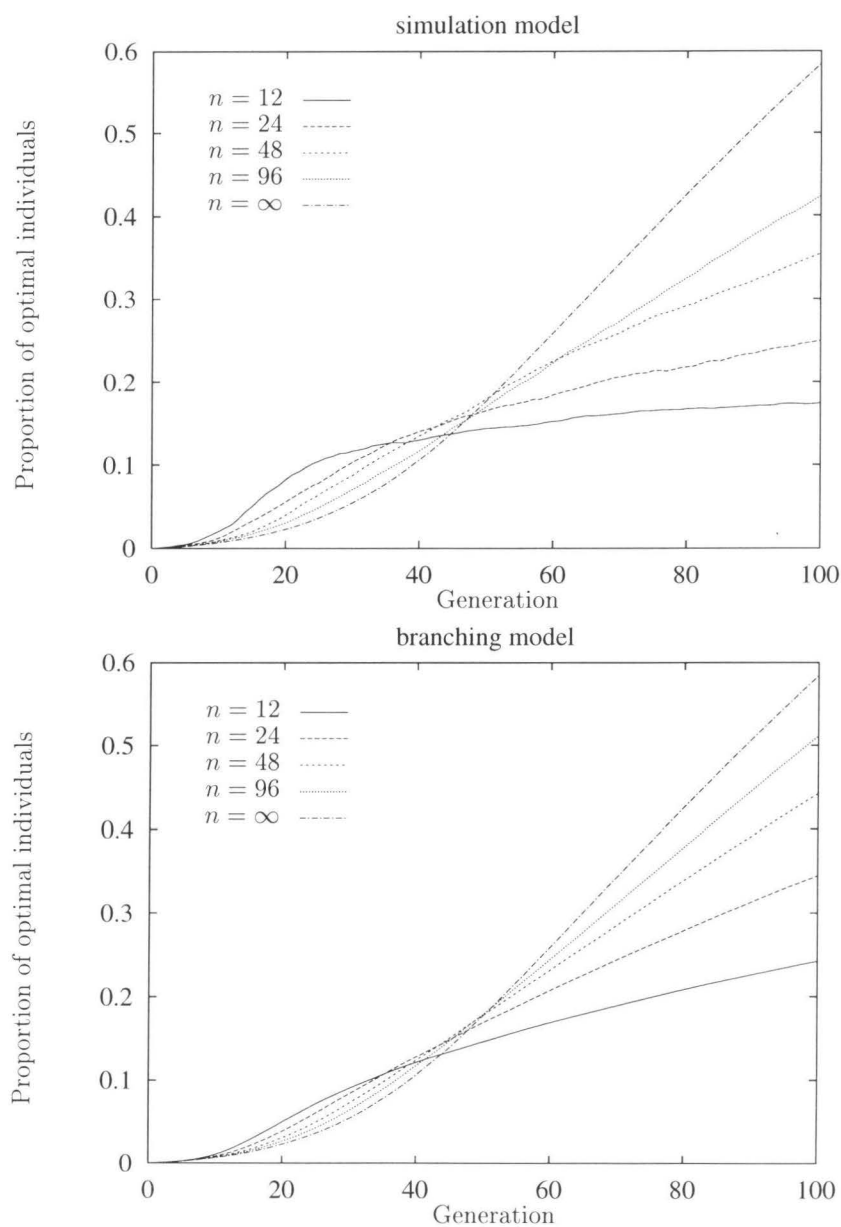


Figure 5.18: Proportion of optimal solutions for elitist recombination both for the simulation model (top) and the branching model (bottom).

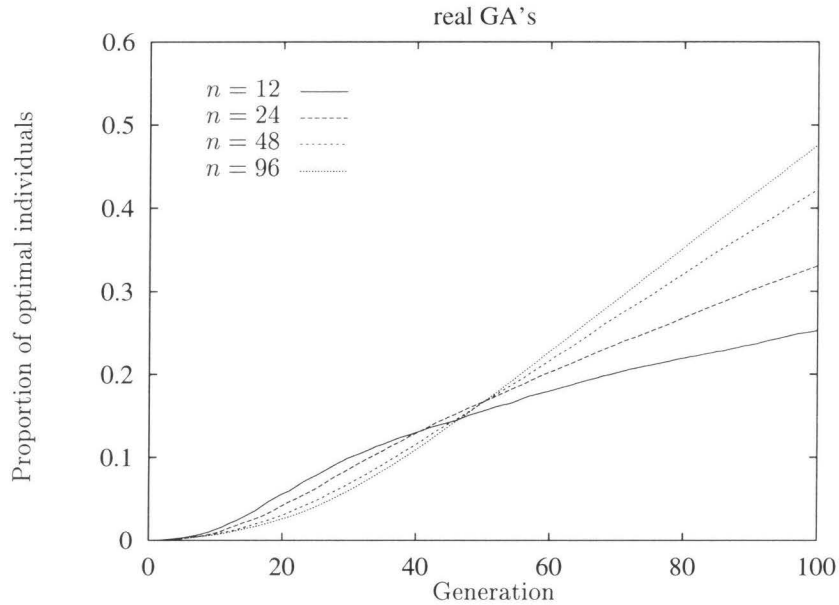


Figure 5.19: Proportion of optimal solutions for elitist recombination.

This genetic algorithm is not likely to find the global optimal solution. The results of the simulation model are similar to the results obtained with the real GA's. The simulation corresponds best to the implementation with reduction of selection variance. In the branching model the performance of the GA deteriorates when the population size is increased. The same behaviour is observed for the real GA's. The branching model underestimates the probability that the optimum is located.

Figures 5.10 and 5.10 show the results when applying tournament selection. Tournament selection results in a very fast convergence. The optimal solution should be found fast, otherwise it will not be found at all. We see roughly the same qualitative behaviour, but the actual probabilities differ between the simulation and the experiments.

Figures 5.12 and 5.13 show the results for the  $(\mu, \lambda)$ -selection. The qualitative behaviour of the simulated and experimental results match well. The results for the real GA's show that selection variance is small for this selection scheme. This is to be expected, because the parents are selected more often than in case of the generational GA's. As a result, the ratio between the expected and the actual number of copies of a parent is likely to be relatively close to one.

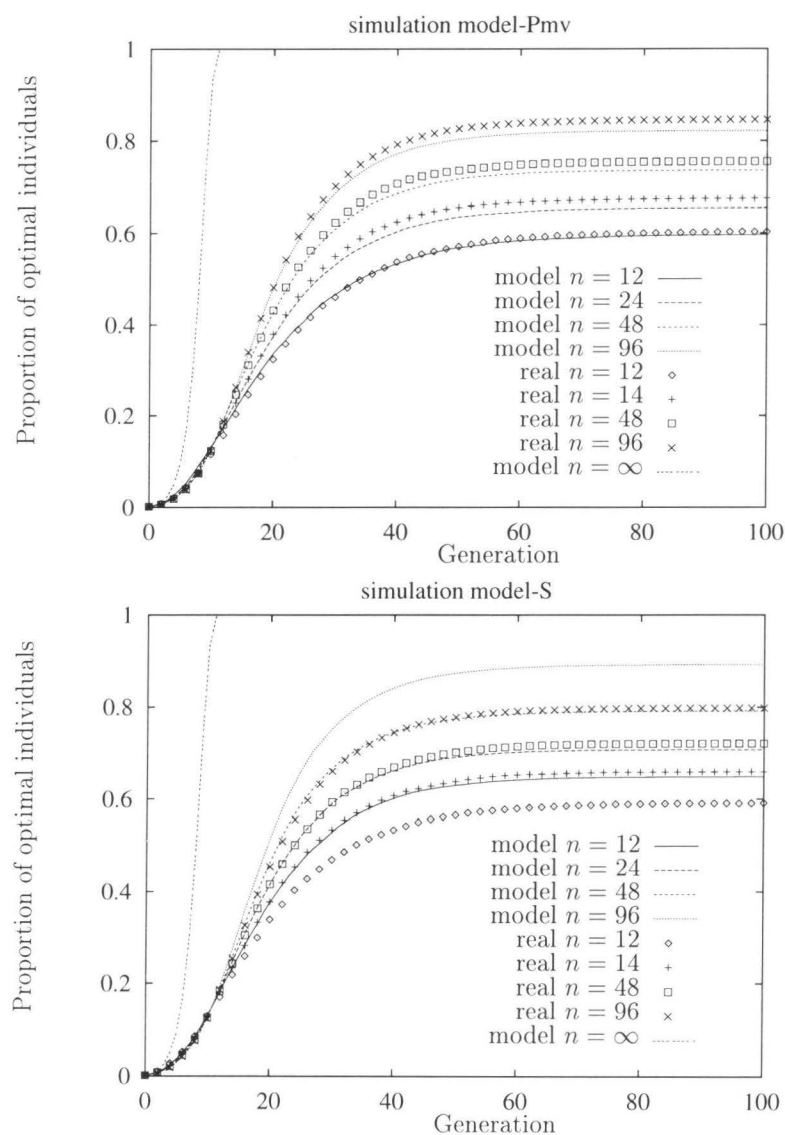


Figure 5.20: Problem 2: proportion of optimal solutions for the generational genetic algorithm with fitness proportional selection for real GA with reduction of sampling errors during selection and the a model simulating finite recombination (top) and for a real GA without reduction of sampling errors during selection and a model simulating finite selection (bottom).

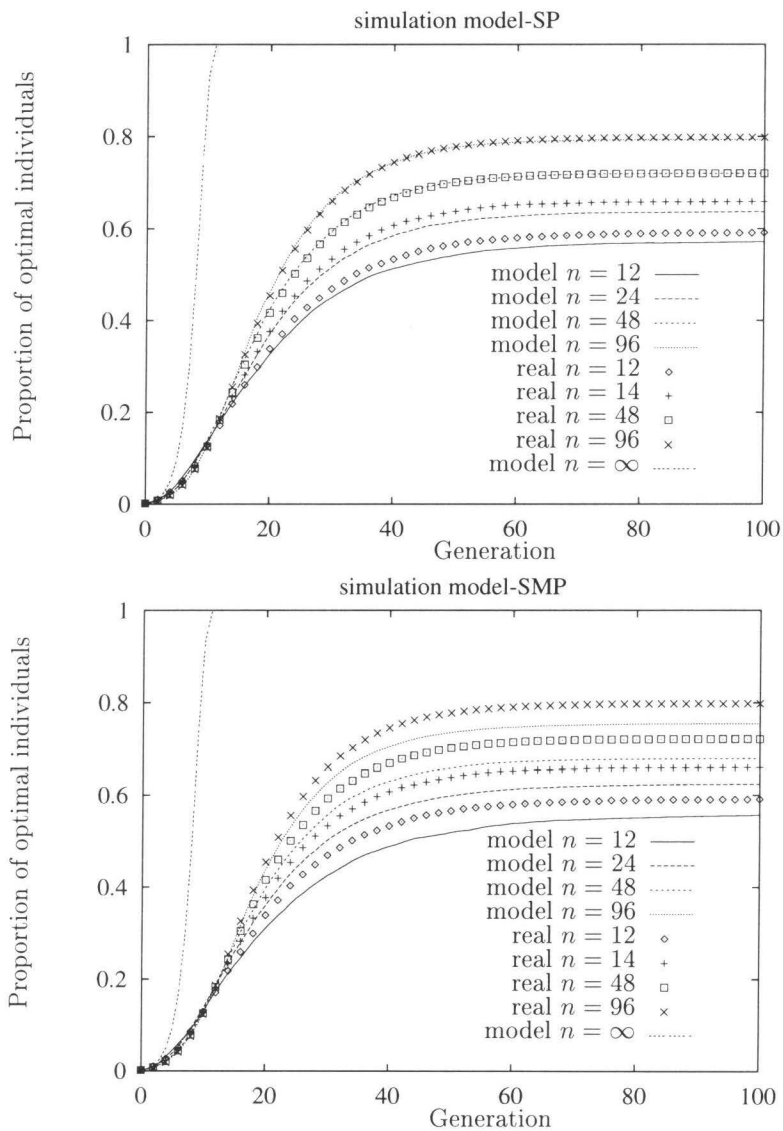


Figure 5.21: Problem 2: proportion of optimal solutions for the generational genetic algorithm with fitness proportional selection for real GA without reduction of sampling errors during selection and the simulating finite selection and recombination (top) and when simulating finite selection, mating, and recombination (bottom).

Figures 5.14 and 5.15 show the results for a  $(\mu + \lambda)$ -selection. The simulation model predicts the results of the runs of the real GA's very well. The branching model overestimates the proportions of optimal solutions for all population sizes. If we compare the real GA with and without reduction of selection variance, then the differences are small. This is again due to the fact that well-performing individuals produce many copies. A well-performing individual might get no chance to reproduce once in a while, but if this individual survives during several generations, then this event does not have a strong influence on the progress of the evolution process. For this reason we expect that minimization of sampling errors during selection is not very important for the  $(\mu + \lambda)$ -selection.

Figures 5.16 and 5.17 show the results for the triple-competition. Both models show approximately the same results as the real GA. Again the branching model overestimates the proportions during all runs.

Figures 5.18 and 5.19 show the results for the elitist recombination algorithm. Elitist recombination converges quite slowly, when compared to the other GA's. The results of the models match the results of the real GA runs reasonably. In the model we only sample over the offspring. However, in the elitist recombination parents and offspring are in direct competition. Whether the parents survive depends on the fitness of their direct offspring. Therefore, computing both distributions independently is not correct. This discrepancy between the models and the real elitist recombination is probably the reason for the difference between the results predicted by the models and the results obtained from the real GA runs. If we would like to model this interaction correctly, then we should generate the distribution of the parents dependent upon the distribution of the offspring after sampling. In order to compute this parent distribution we should keep track of the which parents produced which offspring.

The branching model assumes a building block processing approach to optimization (recall that set  $S$  corresponds to those individuals that contain the optimal building block in the deceptive part). If we compare the results of the branching model to the results of the real runs, then a good match is observed for all GA's, except for the generational GA's. When using the simulation model these generational GA's are also modelled quite well, so the *SP*-boxes seem to work correctly. An explanation for the difference is that the underlying assumption of the branching model is not true for the two generational genetic algorithms: the generational GA's do not process the order-six deceptive building block. This explanation is further strengthened by the fact that the generational GA's are not well able to optimize this problem.

A second smaller problem instance is introduced in which the oneMax part consists of six bits and the deceptive part consists of three bits. The generational GA with fitness proportional selection is applied to this problem. The results are shown in Figures 5.20 and 5.21. The upper graph of Figure 5.20 shows the results for the model (given by the lines) and the results for the real GA with reduction of sampling errors during selection (given by the markers). The match between the model and the real GA is good, especially for small population sizes. The lower graph shows the results when using sampling of the distribution of initial population and sampling of the distributions after the selection step by means of a sampling step, while recombination is done by means of the transmission

function model. The sampling errors by selection are here modelled explicitly, and hence we added the results of a GA without reduction of the sampling error during selection to this plot. The model in this case results in too optimistic predictions. This effect can be explained by means of the following example. Let  $A$  be a highly fit string, while  $B$  is a string with a very low fitness. Assume that the recombination produces a distribution containing an equal proportion of strings  $A$  and  $B$  and that this proportion is smaller than  $1/n$ . After the selection the proportion of  $A$  can easily be larger than  $1/n$ , while the proportion of  $B$  is close to zero. Hence, if we only sample after selection, then it is not surprising that the resulting model results in too optimistic predictions.

Figure 5.21 shows the results for the real GA without reduction of the sampling error during selection. The upper graph shows the results when simulating a finite selection and a finite recombination by means of sampling. The results predicted by the model match well with the results obtained with the real GA's. The lower graph shows the results when simulating a finite selection, mating and recombination. In this case the match between the model and the results of the actual GA is not so good. We expect this to be due to the fact that our model assumption, which is that the individuals are generated independently, does not hold here.

The best performance is obtained when using finite recombination, as was already suggested by the theoretical analysis. The distribution after recombination has quite a large spread and therefore the discrepancies are quite large at this point during evolution. Furthermore we observe a good match for the simulation-P with a GA with reduction of sampling errors during selection, and for the simulation-SP with a real GA without this reduction of sampling errors. This indicates that we can make a separation between these two sampling errors by means of our model.

## 5.6 Summary

Infinite population models can behave quite differently from their finite population counterparts. We conjecture that the influence of population size is strongest when the expected distribution of the population has the largest spread.

We introduced two models of finite population GA's, applied these models to a test-problem, and compared the behaviour of the models to the behaviour shown by real GA's. The simulation model predicts the behaviour of all selection schemes well. The branching model only fails on the generational GA's, which suggests that generational GA's are not able to process order-six building blocks efficiently.

The experiments show that modelling the consequences of finite populations during recombination gives a good match between models and real GA's, and that modelling a finite population after the selection step results in a less good match.

In case of generational GA with fitness proportional selection we have shown how to model the consequences of the finite population during selection and during recombination (both together and separately). The results further strengthened our suspicion that finite population sizes have a strong influence during the recombination phase.



## Chapter 6

# Mixing Evolutionary Algorithm

When a genetic algorithm solves a problem involving building blocks, the GA has to grow the building blocks, and then the GA has to mix these building blocks in order to obtain the (optimal) solution. Finding a good balance between the growing and the mixing process is a prerequisite to get a reliable evolutionary algorithm [GDT93, TG93]. Different building blocks can have different probabilities of being mixed. Such differences in mixing probability can easily lead to a loss of the building blocks that are difficult to mix, and as a result lead to premature convergence. By allocating relatively many trials to individuals that contain building blocks with a low probability of being mixed, such effects can be prevented. In this chapter we introduce the mixing evolutionary algorithm (mixEA) in which the allocation of trials is a more explicit procedure than in the standard evolutionary algorithms. Experiments indicate that the mixEA is a reliable optimizer on a set of building block problems that are difficult to handle with more traditional genetic algorithms. In the case that the global optimum is not found, the mixEA creates a small population containing a high concentration of building blocks.

This chapter is organized as follows. In section 6.1 mixing of building blocks and allocation of trials is discussed. Section 6.2 introduces the mixEA. The results of some experiments are given in section 6.3. This is followed by a summary in section 6.4.

### 6.1 Mixing of building blocks

In chapter 3 a simple building block oriented problem was studied. The problem involved two compatible building blocks, which were identical. Here, we discuss problems having building blocks with different fitness contributions and different probabilities of being transferred. In section 6.1.1 the consequences of such differences between building blocks for the mixing process and for the reliability of evolutionary optimizers are discussed. In section 6.1.2 a different method of allocation of recombinative trials is proposed, which can help to prevent premature convergence for such problems.



### 6.1.1 Transfer and mixing of building blocks

Mixing of building blocks is a process that involves preserving the individuals that contain building blocks, selecting pairs of individuals with different but compatible building blocks, and the allocation of enough opportunities to mix these building blocks for such pairs of individuals. If two individuals, containing compatible building blocks, are used as parents during recombination, then these building blocks are mixed with a certain probability. If enough tries are allocated to this pair of parents, then the building blocks will get mixed eventually. Let the probability that a building block is being transferred be denoted by  $P_{tr}$ . In sections 3.1 and 3.3, it was shown that when using uniform crossover this probability depends upon the order of the building block, and upon the probability that optimal bit-values are present within the partition corresponding to this building block. In section 3.1 an explicit formula for  $P_{tr}$  is given. For a simple problem involving two building blocks that both have the same fitness contribution, and approximately the same  $P_{tr}$ , the problem of mixing these blocks can already be quite difficult. Cross-competition between building blocks can lead to the loss of building blocks, and thereby lower the probability of finding the global optimum. If the fitness contributions of the building blocks differ strongly, or if the  $P_{tr}$ 's of building blocks differ strongly, then the mixing of building blocks becomes even more difficult. These two factors are first discussed independently of each other.

If all building blocks have approximately the same  $P_{tr}$  and one building block has a much larger fitness contribution, then the evolutionary search focuses on this (high-fitness) building block until all individuals in the population contain this building block. Meanwhile, the partitions of the other building blocks are susceptible to drift, which might lead to the loss of diversity of alleles in those partitions. If all building blocks have approximately similar fitness contributions, but the values of  $P_{tr}$  differ a lot, then the search tends to focus on those building blocks that have the highest value of  $P_{tr}$ . Again the partitions of the other building blocks (those with lower values of  $P_{tr}$ ) are susceptible to genetic drift.

In the case of two opposing effects, such as when the building blocks with the lowest values of  $P_{tr}$  have the highest fitness contribution, it is difficult to predict whether the building block will be retained. Let us call such a building block a high-order building block, because a high-order is likely to result in a low value of  $P_{tr}$ . In case of a canonical GA the following condition is required to retain a building block:  $(f_{bb}/\bar{f})P_{tr} \geq 1$ , where  $f_{bb}$  denotes the average fitness of the individuals that contain the building block. This condition has to be satisfied, such that the proportion of individuals containing the building block does not decrease. So, in the case that  $P_{tr}$  of a building block is low there has to be a large fitness advantage of the individuals to compensate for the low value of  $P_{tr}$ .

Even when the separate fitness contributions of the building blocks with high  $P_{tr}$  are small, these building blocks can easily combine to generate highly fit individuals containing many of such building blocks. The resulting individuals can push high performance individuals containing building blocks with low  $P_{tr}$  out of the population.

This effect is even enlarged by the fact that the value  $P_{tr}$  of a building block tends to increase when the proportion of individuals that contain this building block increases (cf. section 3.3). So, if the proportion of the low-order building blocks grows, then this rate

of growth will tend to increase, resulting in an even more rapid growth of these low-order building blocks.

In elitist GA's the retaining of well-performing individuals is guaranteed. However, even when using such a GA, it is not certain that the high-order building block is retained. Low-order building blocks can be combined, resulting in individuals that outperform the individuals that only contain a few high-order building blocks.

Due to these effects most GA's that allocate recombinative trials based on fitness only, have a (strong) preference for building blocks with a high value of  $P_{tr}$ . Originally, GA's were developed as adaptive systems. Good on-line performance in a dynamically changing environment (context) was considered to be important. Allocation of trials based on fitness is appropriate in that case. Building blocks with a low value  $P_{tr}$  are less interesting because these building blocks do not allow for the fast adaptation that is required for such an on-line system. Nowadays evolutionary computation methods are often used for optimization problems. The problem definition (context) does not change, and the best solution should be found independent of the  $P_{tr}$  of the building blocks contained in this optimal solution. So in case of an optimization problem, adjustment of the allocation of trials to compensate for the bias towards building blocks with a high value of  $P_{tr}$  can help to produce a more reliable function optimizer.

### 6.1.2 Allocation of recombinative trials

In most GA's the fitness is taken to be equal to the objective value of the function to be optimized. So, the selection of parents is usually biased towards individuals having a relatively high objective value. Because the number of trials (recombination operations) assigned to a selected parent is fixed, there is a strong relation between the fitness and the assigned number of recombinative trials. Fitness undoubtedly is one of the important factors to be considered when determining allocation of trials. However, the probability  $P_{tr}$  that building blocks are transferred should be considered too when optimization is the goal. When thinking in terms of building block mixing, it seems logical that more trials should be assigned to those individuals that contain important building blocks with a low value of  $P_{tr}$ .

In our view a reliable selection scheme for optimization has two tasks to perform. The first task is fitness-based selection of the individuals, which is used to drive the population towards regions of relatively high fitness. The second task is the allocation of recombinative trials in such a way that an efficient exploitation of the building blocks, present in a certain parent, is obtained. Relatively many recombination trials should be allocated to those individuals that contain building blocks with a high fitness contribution, but a low value of  $P_{tr}$ . Given an individual, it is usually not known what building blocks are present within this individual, and what the  $P_{tr}$  values of these blocks are. Therefore, an allocation of trials directly based upon  $P_{tr}$  seems to be impossible. So, the properties of the separate building blocks cannot be measured, but it is possible to measure the properties of a complete individual. One such property is the probability that this individual recombines successfully. Therefore, let us define  $P_{tr}^{(ind)}$  as the probability that a recombination of the

current individual with a randomly selected other individual of the population results in an offspring that outperforms both of its parents. It is easy to estimate this probability by actually performing a set of recombination operations. If recombination produces offspring that outperforms both parents, then a large number of the building blocks of the parents is probably transferred to the offspring. If we have an individual that contains a few building blocks, then  $P_{tr}^{(ind)}$  depends among other things on the  $P_{tr}$  values of the building blocks with the highest fitness contribution present within the individual. The value  $P_{tr}^{(ind)}$  can be measured, and gives us at least partial information about the  $P_{tr}$  values of the building blocks present in an individual; Therefore, the  $P_{tr}^{(ind)}$  value might be used to control the allocation of trials.

## 6.2 Mixing evolutionary algorithm

In this section we introduce the mixing evolutionary algorithm. The mixEA is an evolutionary algorithm that actually separates the two tasks of the selection scheme, and that uses the  $P_{tr}^{(ind)}$  in order to allocate an appropriate number of recombinative trials to an individual.

The mixEA is designed to handle building block oriented problems. The primary goal is to locate the optimum. If this goal is too difficult to attain, the algorithm is meant to locate a (small) set of relatively good solutions that do contain as many building blocks as possible. In order to reach this goal the mixEA

1. separates fitness-based selection and allocation of recombinative trials, and
2. prevents (rapid) duplication of building blocks with high values of  $P_{tr}$ .

First, an outline of the mixEA is presented in section 6.2.1. Next, a more detailed description of the mixEA is given section 6.2.2 together with an explanation of how our goals are attained.

### 6.2.1 Outline mixEA

During the evolution the mixEA allocates an exponentially increasing number of trials to the individuals in the population. Simultaneously, the population size is decreased exponentially, such that the total number of function evaluations per generations remains approximately the same. The number of trials is increased, because during the later phases of evolution one expects individuals to contain a large number of building blocks, and therefore to have a low value of  $P_{tr}^{(ind)}$ . The decrease of the population size is allowed because the information about the search space is available in a more condensed form when evolution proceeds because a single individual probably contains multiple building blocks. Thus, one does not have to lose information by decreasing the population size during later phases of evolution, because a smaller population suffices to cover all the information.

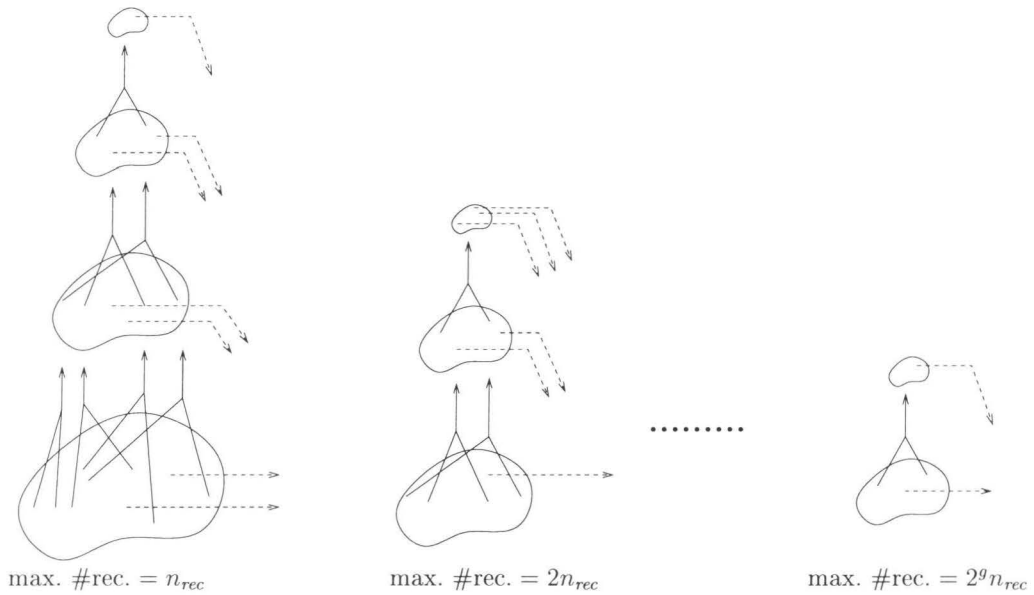


Figure 6.1: A schematic outline of the mixing evolutionary algorithm.

A visual representation of this process is given in the leftmost part of Figure 6.1. Each bean-like shape represents a population. On the left-bottom the random initial population is shown. A successful recombination operation is denoted by two solid lines starting at the parents merged in an upward arrow that denotes the propagation of the offspring to the population. If an offspring is produced, both of its parents are discarded. The subsequent populations in a stack decrease in size. The maximal number of recombinations for each individual is given by the number below this stack of populations. The stack terminates when the next population contains less than two individuals. At termination of a stack, all individuals that did not recombine successfully are collected. This is denoted by the dashed arrows. The collected individuals are put in the bottom population of the next stack. Duplicates of complete individuals are removed from this population. The maximal number of recombination operations is increased by a factor two and the evaluation of this new stack is started.

### 6.2.2 Detailed description

The selection scheme of the mixEA differentiates between two tasks: selection of parents and allocation of trials. The mixEA starts with a large population and it tries to condense all building blocks in a set of individuals that decreases in size. The ideal final situation corresponds to all building blocks being condensed in a single individual representing, the optimal solution. In order to attain this goal the mixEA uses a (rapidly) decreasing

population size.

Given a certain generation, the next generation is created according to the following rules. Two parents are selected uniformly at random. The recombination operator is used to create a single offspring, which is only accepted when it outperforms both of its parents in terms of fitness. If the offspring is accepted, then it is put in the population of the next generation (one level higher in the stack), and both parents are discarded. If the offspring is not accepted, then the parents are put back in their population such that each of these individuals gets a new opportunity to recombine. This process is continued until the current population contains less than two individuals, or the maximal number of recombination per individual is exceeded. Evolution now continues at the next generation, one level up in the current evolutionary stack. For each offspring that is accepted its two parents have been discarded, so the pool of offspring is at least twice as small as the parent pool.

The evolution of a stack continues until the current population contains less than two individuals. Now all remaining individuals over all generations in the stack are collected in the first population of the next evolutionary stack, duplicates of individuals are removed, and the maximal number of recombination events is increased by a factor two for this new stack. Given that the first population of the previous stack contained  $N$  individuals, the first population of the current stack contains at most  $N - S_{rec}$  individuals, where  $S_{rec}$  is the number of successful recombination event in the previous evolutionary stack.

The allocation of trials to an individual is influenced by its  $P_{tr}^{(ind)}$  value. An individual with a high value  $P_{tr}^{(ind)}$  recombines easily. Therefore, it uses only few recombinative trials in order to produce offspring. Furthermore, due to the fact that it recombined successfully the information contained in this individual is likely to be propagated upwards in the evolutionary stack. This process continues until the offspring population contains less than two individuals. This happens also when  $P_{tr}^{(ind)}$  has decreased such that  $P_{tr}^{(ind)} < 1/N_{rec,g}$ , where  $N_{rec,g}$  is the maximal number of recombination operations per individual in the stack, which is given by  $N_{rec,g} = 2^g n_{rec}$ . The information present in the individual is likely to be propagated upwards in the stack. The value of  $P_{tr}^{(ind)}$  decreases during evolution because individuals are likely to contain more building blocks, and therefore more recombinative trials are required in order to preserve enough of these building blocks, such that an offspring with superior fitness is obtained. Individuals that fail to recombine are likely to contain a few building blocks with low values of  $P_{tr}$ , or a large number of building blocks with a relatively high value of  $P_{tr}$ , but the combined effect of these building blocks again leads to a low value of  $P_{tr}^{(ind)}$ .

Prevention of the rapid duplication of building blocks with high values of  $P_{tr}$  is achieved by allowing each parent to produce only a single offspring, by removing duplicates at the first population of each stack. Furthermore, duplicates of building blocks are removed because individuals that have some building blocks in common are likely to recombine, thereby removing one set of the common building blocks. Building blocks with high values of  $P_{tr}$  are likely to move far up in the first few evolutionary stacks, and thereby these building blocks rapidly decrease in number without getting lost.

The mixEA requires only three parameters, i.e. the size of the initial population, the maximal number of recombinations operations per individual in the first evolutionary stack  $n_{rec}$ , and the maximal number of subsequent evolution stacks. The algorithm is not very sensitive with respect to the actual values of these parameters and hence parameter tuning is easy.

An “opposite” approach is the reproductive evaluation [Whi87]. During reproductive evaluation additional copies are assigned to individuals that have created relatively many good offspring. We think reproductive evaluation indeed might help to obtain rapid convergence, but when one is interested in actual optimization such a system is likely to fail because it enlarges the preference of the GA for building blocks with a large value of  $P_{tr}$ .

Next we present the pseudo-code of the mixEA. The following tuples are used:

*resultGen* = [*parents*, *offspring*];  
*pair* = [*first*, *second*];

Here, *resultGen* is used to contain the results of a generation, where *parents* represents a population containing the parents that did not produce an offspring, and *offspring* represents a population containing the offspring, and *pair* is used to represent a pair of parents, where *first* is the first parent and *second* is the second parent.

```
MixEA(initPopSize, maxStack, nrec)
  S0 = initialPopulation(initPopSize);
  for t = 1 to maxStack do
    St = EvolveStack(St-1, nrec2t-1);
  od;
end
```

```
EvolveStack(P0, maxRec)
  failIndiv = {};
  t = 0;
  while (|Pt > 2) do
    resultGen : res;
    res = EvolveGen(Pt, maxRec);
    add res.parents to failIndiv;
    Pt+1 = res.offspring;
    t = t + 1;
  od;
  add Pt to failIndiv;
  EvolveStack = failIndiv;
end
```

```
EvolveGen(P, maxRec)
  resultGen : res;
```

```

res = {};
for g = 1 to maxRec do
  pairs = randomPartitionSet(P);
  for each p ∈ pairs do
    offs = evolve(p.first, p.second);
    if (Fitness(offs) > Fitness(p.first)) and (Fitness(offs) > Fitness(p.first)) then
      add offs to res.offspring;
      remove p.first from P;
      remove p.second from P;
    fi;
  od;
od;
res.parents = P;
EvolveGen = res;
end

```

Here, the function *initialPopulation*(*initPopSize*) generates a random initial population, the function *randomPartitionSet*(*P*) randomly partitions a set *P* in a set of pairs (if  $|P|$  is odd, then one element remains), the operation *add* puts an element in a set, and the operation *remove* removes an element from a set.

### 6.3 Experiments

During our experiments we used the fully deceptive trap function (see section 2.6). The fitness contribution for a set of  $k$  bits that together can compose a building block is calculated according to the following formula,

$$f(x) = \begin{cases} 1 & \text{if } u(x) = k \\ r^{\frac{k-1-u(x)}{k-1}} & \text{otherwise} \end{cases}$$

where  $u(x)$  counts the number of one-bits in string  $x$ , and  $r < 1$  denotes the fitness ratio between the optimal and the suboptimal solution. This type of building blocks is difficult to detect because all lower order schemata direct the search towards the local optimum containing just 0-bits. A fitness ratio of  $r = 0.7$  is used, and no assumptions regarding the linkage of loci are made. The fitness of an individual is taken to be the sum of the partial fitnesses of the different subfunctions.

During our experiments the mixEA is compared to two other evolutionary algorithms. The first is a generational genetic algorithm (GGA) using tournament selection (tournament size = 2), in which crossover is applied with probability 0.7 and the mutation rate is set to  $1/l$ , where  $l$  is the length of the bit-string. The second is a steady-state genetic algorithm (SSGA) applying uniform selection and truncation reduction, crossover is always applied and the mutation rate is set to  $1/l$ . Both GA's terminate when the optimal solution

| order<br>bb | num<br>bb | mixEA                      |                            |                 | GGA                        |                            |                 | SSGA                       |                            |                 |
|-------------|-----------|----------------------------|----------------------------|-----------------|----------------------------|----------------------------|-----------------|----------------------------|----------------------------|-----------------|
|             |           | frac <sub>bb</sub><br>best | frac <sub>bb</sub><br>pop. | % succ.<br>runs | frac <sub>bb</sub><br>best | frac <sub>bb</sub><br>pop. | % succ.<br>runs | frac <sub>bb</sub><br>best | frac <sub>bb</sub><br>pop. | % succ.<br>runs |
| 3           | 13        | 1.0                        | 1.0                        | 100             | 0.83                       | 1.0                        | 10              | 1.0                        | 1.0                        | 100             |
| 4           | 10        | 1.0                        | 1.0                        | 100             | 0.33                       | 0.26                       | 0               | 0.71                       | 0.97                       | 0               |
| 5           | 8         | 0.86                       | 1.0                        | 10              | 0.18                       | 0.03                       | 0               | 0.19                       | 0.48                       | 0               |
| 6           | 7         | 0.76                       | 1.0                        | 10              | 0.12                       | 0                          | 0               | 0.12                       | 0.3                        | 0               |
| 7           | 6         | 0.68                       | 1.0                        | 0               | 0.07                       | 0                          | 0               | 0.08                       | 0.1                        | 0               |
| 8           | 5         | 0.6                        | 1.0                        | 0               | 0.02                       | 0                          | 0               | 0.04                       | 0.03                       | 0               |
| 10          | 4         | 0.48                       | 0.99                       | 0               | 0                          | 0                          | 0               | 0                          | 0                          | 0               |

Table 6.1: Results for different order of the building blocks (first set of problems), where  $\text{frac}_{bb}$  best/pop represents the fraction of building blocks present within the Best solution/final Population, and % succ. shows the percentage of successful runs.

| order<br>bb | mixEA                        |                             | GGA                          |                             | SSGA                         |                             |
|-------------|------------------------------|-----------------------------|------------------------------|-----------------------------|------------------------------|-----------------------------|
|             | $\mathbb{P}_{k,\text{best}}$ | $\mathbb{P}_{k,\text{pop}}$ | $\mathbb{P}_{k,\text{best}}$ | $\mathbb{P}_{k,\text{pop}}$ | $\mathbb{P}_{k,\text{best}}$ | $\mathbb{P}_{k,\text{pop}}$ |
| 3           | 0.95                         | 1.0                         | 1.0                          | 1.0                         | 1.0                          | 1.0                         |
| 4           | 0.95                         | 1.0                         | 0.25                         | 0                           | 1.0                          | 1.0                         |
| 5           | 0.95                         | 1.0                         | 0                            | 0                           | 0                            | 0.1                         |
| 6           | 0.85                         | 1.0                         | 0                            | 0                           | 0                            | 0                           |
| 7           | 0.75                         | 0.9                         | 0                            | 0                           | 0                            | 0                           |
| 8           | 0.30                         | 0.9                         | 0                            | 0                           | 0                            | 0                           |
| 9           | 0.25                         | 0.85                        | 0                            | 0                           | 0                            | 0                           |
| 10          | 0                            | 0.55                        | 0                            | 0                           | 0                            | 0                           |

Table 6.2: Results for a fully deceptive problem involving building blocks of different order (second problem), where  $\mathbb{P}_{k,\text{best}}/\mathbb{P}_{k,\text{pop}}$  denote the probabilities that the building block of order  $k$  is present in the Best solution/final Population, respectively.

is obtained, or when the variance in fitness is zero, or when the number of function evaluations exceeds 500,000. We use  $n_{\text{rec}} = 32$ , so the mixEA allows 32 recombination-trials per individual during its first evolution process. An upper bound of 16,384 is used for the maximal number of recombinations per individual, which results in at most ten subsequent evolution processes to take place. No mutation is applied. All three selection schemes have an (initial) population containing 4096 individuals and use uniform crossover. All results are obtained by taking an average over twenty independent runs.

Two sets of experiments are conducted. First, it is investigated how the different selection schemes behave when all building blocks have the same  $P_{tr}$ . Next, it is investigated what happens when different building blocks have different values of  $P_{tr}$ . The first set of experiments is used to study the behaviour of the different selection schemes in relation to



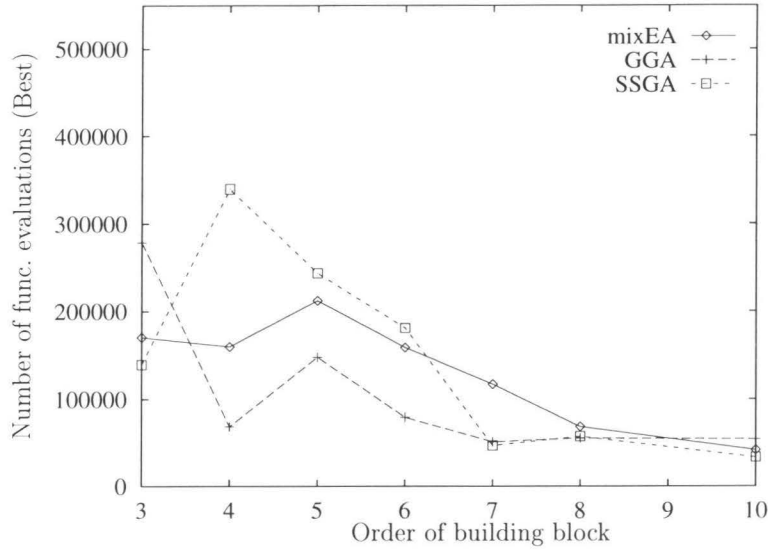


Figure 6.2: Number of function evaluations before obtaining the best solution for the first set of problems.

the order of the building blocks  $k$ . The number of building blocks is adjusted such that the length of the bit-string is approximately 40 bits. The results are shown in Table 6.1. For each selection scheme the first column in this table, denoted by  $\text{frac}_{bb} \text{ best}$ , shows the fraction of all building blocks present in the best obtained solution. Here, a value of one means that all building blocks are present. The mixEA performs significantly better than the other two selection schemes when using this measure. The second column, denoted by  $\text{frac}_{bb} \text{ pop}$ , shows the fraction of all building blocks that are present in the final population. For the mixEA this fraction is close to 1.0 during this experiment, which means that almost all building blocks are still present within the final population. Usually this population consists of approximately ten individuals for the mixEA. In case of the GGA the fraction of building blocks drops rapidly when the order of the building blocks increases. An interesting observation is that this fraction is zero for all problems having building blocks containing more than 5 bit positions even when the best solution does contain a small fraction of the building blocks. This means that the best solution is not present in the final population, and apparently the GGA lost track of the most important parts of the search space. The SSGA performs better, but it also is not able to preserve large building blocks in its population. The third column shows the percentage of successful runs (i.e. runs that found the global optimum). Figure 6.2 and 6.3 show the number of function evaluations performed before the best solution is found (left) and the total number of evaluations performed before termination (right). Comparing the two graphs for the mixEA we see

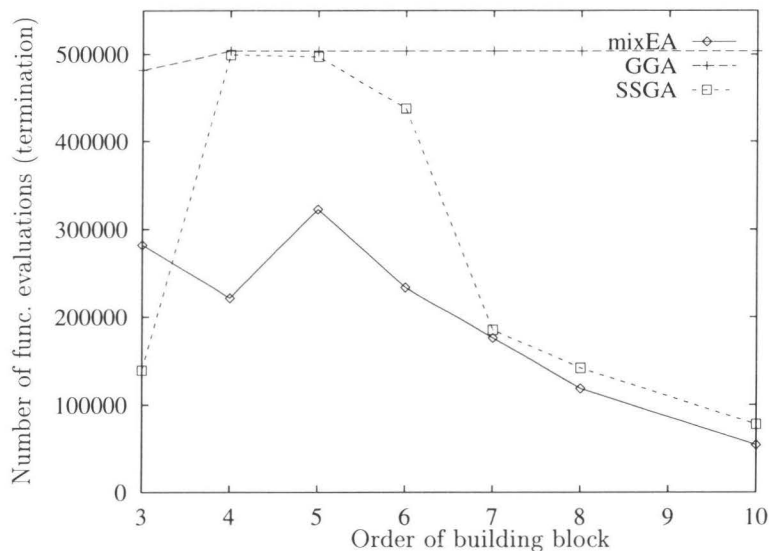


Figure 6.3: Number of function evaluations before termination of the algorithms for the first set of problems.

that this selection scheme terminates relatively fast after it has found its best solution. This in contrast to the GGA that almost always uses the maximal number of function evaluations before it terminates.

From the first set of experiments it is clear that the mixEA can handle problems involving relatively large building blocks when all building blocks have the same order (same  $P_{tr}$ ). A second series of experiments is conducted to study the behaviour of the selection schemes when applied to a problem instance involving a set of building blocks of different order. The actual problem instance used, contains eight building blocks having order ranging from three to ten bits, resulting in a problem having a length of 52 bits. The results are shown in Table 6.2. Each row in this table summarizes the result for a building block of a certain order. For each selection scheme the first column ( $\mathbb{P}_{k,best}$ ) shows the probability that the building block of order  $k$  is present within the best obtained solution, and the second column ( $\mathbb{P}_{k,pop}$ ) shows the probability that the building block of order  $k$  is present in the population. All three selection schemes easily find the order-three building block, but the order-four building block is already difficult to find for the GGA and the SSGA fails on the order-five building-block. The mixEA is able to find and mix the larger building blocks with a reasonable probability. When studying the probability that a building block is present in the final population roughly the same results are obtained. Again the mixEA significantly outperforms the other two methods. It is also interesting to note that the probability of the largest building block to be present in the final population

is still 55%, which means that the final population of the mixEA is still very diverse and is likely to contain all building blocks. These building blocks are concentrated in a small population of approximately ten individuals. Results on a larger test-suite are shown in chapter 8 in which we compare a number of different algorithms with each other.

## 6.4 Summary

A selection scheme basically has two tasks: the selection of parents to drive the population towards more promising parts of the search space and the allocation of recombinative trials in order to get an effective mixing of building blocks. Most traditional GA's do not differentiate between these two tasks and both are steered by fitness only. As a result the search is usually biased towards finding building blocks with a low value of  $P_{tr}$ , because these building blocks are recombined easily. The mixEA is a simple selection scheme that uses a sequence of evolution stacks. In the mixEA selection of parents is also steered by fitness, but the second task, allocation of trials is varied during the evolution process. Subsequent evolutionary stacks use an increasing number of trials because more complex building blocks are likely to be present.

During our experiments the mixEA significantly outperformed two traditional GA selection schemes on a number of test problems, especially when large building blocks need to be processed. Also when the problem instance contains many building blocks having different values of  $P_{tr}$  the mixEA performs well. When the optimum is not found, the mixEA usually terminates with a small final population that still contains most of the building blocks. Analysis of this population can reveal more information regarding the actual structure of the search space.

## Chapter 7

# Building Block Filtering and Mixing

Problems involving high-order building blocks with unknown linkage are difficult to solve. Neither  $n$ -point crossover nor uniform crossover can mix high-order building blocks efficiently. Linkage learning methods, as briefly discussed in section 1.6, might help in generating more efficient recombination operators. Even when having an efficient crossover, it might still be difficult to strike the balance between exploration and exploitation. In this chapter the bbf-GA is developed. This is a hybrid GA that handles building blocks effectively and efficiently. A three-stage approach is used. During the first stage a large number of rapidly converging GA's is used to explore the search-space. During the second stage the best individual of each GA is filtered to locate the (potential) building blocks present in this individual. The third stage consists of a GA that exploits these masked individuals by mixing them to obtain the global optimal solution. The bbf-GA performs well on a set of test-problems, and is able to locate and mix more building blocks than the competitor GA's.

Section 7.1 presents the basic outline of the bbf-GA. In section 7.2 the building block filtering method is described. This algorithm takes a bit-string as its input, and produces a corresponding mask that marks the most important parts of the bit-string. The produced pairs of bit-strings and masks can be processed by means of the masked crossover introduced in section 7.3. In section 7.4 the bbf-GA is introduced. The test-problems are given in section 7.5 followed by the experimental results in section 7.6. Some enhancements for practical applications are suggested in section 7.7, and a summary is given in section 7.8.

### 7.1 Outline of bbf-GA

The GA is often assumed to be an efficient method for solving building block oriented problems, because it is able to find building blocks independently of each other, and mix these building blocks afterwards. In chapter 2 and 3 some reasons were given why GA's might fail to locate the optimum. An important problem is to find an appropriate balance between exploration and exploitation. A GA with emphasis on exploration (searching the complete search-space) is likely to be slow, a GA with emphasis on exploitation (preserva-

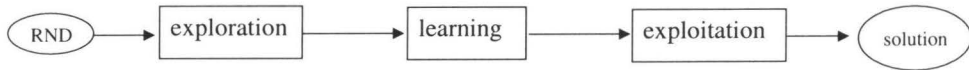


Figure 7.1: Basic scheme of the algorithm

tion and duplication of the currently best individuals) is likely to converge too rapidly to suboptimal solutions.

The hybrid bbf-GA separates the exploration and exploitation task, and handles building blocks effectively and efficiently. In a traditional GA there is no explicit separation between exploration and exploitation. Both processes are done simultaneously. During the first few generations emphasis is on exploration, and during the final steps the GA is likely to focus on exploitation. There are no GA parameters to balance these two processes explicitly. Most GA parameters influence this balance indirectly, but it is difficult to predict how the balance will turn out. Consequently a lot of hand-tuning of parameters might be required for each new problem. In the bbf-GA the separation between exploration and exploitation is made more explicit. A basic outline of the method we are aiming at is given in Figure 7.1. Here one differentiates between three subsequent phases:

**exploration:** find a set of individuals that each contain a few building blocks,

**learning:** locate the most important bits (and hope that these bits will cover a part/the core of the building block), and

**exploitation:** mix the individuals (treating the masked parts as a single piece).

The hybrid bbf-GA follows this schematic approach. Exploration and exploitation are performed by means of GA's. Learning is done by a linkage learning algorithm which is called building block filtering.

## 7.2 Filtering of building blocks

Standard crossover operators are not always efficient. Problem-specific crossover operators or problem-specific choices of the operators can make the building block processing more efficient. Another approach lies in the usage of an evolutionary algorithm that really is able to learn something about the problem it solves, and that uses this information to steer the recombination process. In this chapter an evolutionary system that aims at this goal is investigated. To learn the linkage of loci, bit-strings are analysed by measuring the changes in fitness when changing each of the bits of the string, one at a time. The first GA that uses this approach is the GEMGA [Kar96c, Kar96a, BKW98]. Simultaneously, but independently, the building block filtering [vK96c] was developed. Both methods estimate marginal fitness contributions of the separate bits. The GEMGA basically uses this information to guide the evolution process by computing a global decomposition of

the search space (during the so-called preRecombinationExpression phase). The building block filtering method uses the information to make a decomposition of a bit-string in order to extract the loci that belong to the same building block and their optimal values. So, the building block filtering method aims at extracting local information. In this chapter a hybrid GA, the bbf-GA, that incorporates the building block filtering method is introduced.

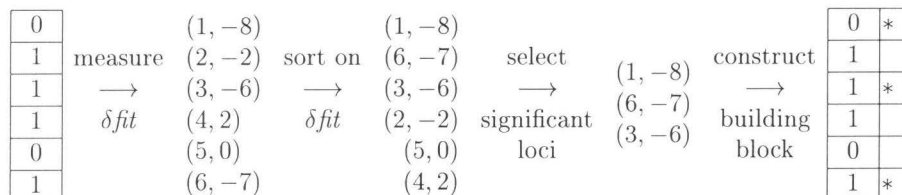


Figure 7.2: Example of one filtering step

If it is known which bits within a given individual belong to the same building block, then one can process the corresponding building block easily. If this type of information is not available, then it would be interesting to be able to extract such information. For this purpose the building block filtering method was developed. Given a specific individual this filtering method locates a (small) set of bits that have a relatively large influence on the fitness of the individual. Next, one assumes that this set of bits corresponds to an optimal building block, or at least that a schema corresponding to these bits is a schema that almost covers a single building block. These bits are marked and processed as a single unit when applying crossover. In this way the bias introduced by the choice of representation can be reduced. The defining length and the order of the schema that represents the optimal building block is not that important anymore, because the schema is processed in one piece. A detailed description of such a crossover operator is given in section 7.3.

The building block filtering method is applied in Figure 7.2. On the left a single individual of length six is given. This individual contains the bit-string 011101. The method uses four steps. During the first step the marginal fitness contribution,  $\delta fit$ , of each of the bits is measured, resulting in a set of tuples. A single tuple contains the index of a bit and the  $\delta fit$  observed when changing this bit. Next, these tuples are ordered on  $\delta fit$ . In the third step a truncation rule is used to select a subset of tuples, and in the fourth step a masked individual is created. The bit-string of this masked individual is exactly the same as the bit-string of the original individual, but a mask is added that indicates the most significant bits.

In the first step the change of fitness,  $\delta fit$ , of each of the loci is measured. The measurement for a single locus is performed by changing the value to its complement. So a 1-bit is changed to a 0-bit and a 0-bit to a 1-bit. The  $\delta fit$  is the change in fitness due to changing the value of the locus. It is used as a measure for the marginal fitness contribution of the corresponding locus with the context of the complete bit-string. In the example in Figure 7.2 a bit-string of length 6 is used. Let us assume that the main fitness contribution

within this string is coming from a building block containing loci 1, 3 and 6, resulting in a fitness increase of +7 when the schema 0#1##1 is present. The  $\delta fit$  of each locus is measured by flipping its value, and observing the change in fitness. A set of tuples of type (position,  $\delta fit$ ) is created. Flipping the bit in loci 1, 3 or 6 breaks schema 0#1##1. As a result the positive fitness contribution +7 gets lost. In our example the  $\delta fit$ -values of loci 1, 3, and 6 are respectively -8, -6, and -7.

The pseudo-code of this filtering step uses the following tuple:

*locus* = [*index*,  $\delta fit$ ];

Here, the *index* denote the position of the locus, and  $\delta fit$  denotes a change of fitness. Now, the filtering step is given by the following pseudo-code:

```

FilteringStep(x)
  loci : list of locus;
  loci = {};
  ## Step 1: compute the marginal fitness of all loci
  for j = 1 to l do
    x' = x;
    x'j = 1 - x'j;
     $\delta fit = Fitness(x') - Fitness(x)$ ;
    add [j,  $\delta fit$ ] to list loci;
  od;
  ## Step 2: sort the list of loci on increasing  $\delta fit$ 
  sort $\delta fit$ (loci);
  ## Step 3: select the most important loci
  loci = Truncate(loci);
  ## Step 4: construct the masked individual
  ConstructMask(x, loci);
end

```

Here *x* is the bit-string that has to be filtered, {} denotes an empty list, *Fitness*(*x*) computes the fitness of individual *x*, and the *add* operation adds a tuple to a list.

During the third step one has to truncate the ordered sequence of tuples in order to select a set of the most influential loci. The simplest approach is to mask a fixed number of tuples. A more complex approach would be to try to estimate the optimal number of bits to select. We estimate this truncation bound by taking a random test-individual, transferring the bit-values of the loci from the filtered individual to the test-individual one by one in the order given by the filtering step, and tracking the change of the fitness of the test-individual. At the moment the building block is transferred completely, a significant increase of the fitness of the test-individual is to be expected. So, in case of the example

shown in Figure 7.2 the bit-values of loci 1, 6, 3, 2, 5, and 4 are transferred in sequence. The bit whose transfer increases the fitness of the test-individual most, and simultaneously results in a new fitness that is larger than the initial fitness of the test-individual, is used as an estimate of the truncation position. Assume that the bit at ranked position  $r$  results in the largest increase in fitness. Now, if the filtered individual and the test-individual have exactly the same bit-values from ranked position  $r + 1$  to  $r + s$ , then the truncation point is chosen in the middle of this range at position  $r + \frac{s}{2}$ . An upper bound on  $r$  is assumed as one expects to find building blocks of limited size only. Masking half of a bit-string would not make much sense. Application of this procedure with a single test-individual gives only a rough estimate of the optimal truncation point. Therefore, this procedure is repeated a number of times using different random test-individuals; The median of all obtained truncation points is used to select the bits that are masked.

```

Truncate( $x$ ,  $loci$ )
   $truncPoints$  = list of number;
  for  $i = 1$  to  $numTest$  do
     $fitBlock$  = list of locus;
     $fitBlock$  = {};
    ## generate a random test individual
     $y$  = RandomIndiv();
     $fit$  = Fitness( $y$ );
    ## determine the number of loci to transfer
    for  $r = 1$  to  $l'$  do
       $k = loci_r.i$ ;
      if ( $x_k \neq y_k$ ) then
         $y_k = x_k$ ;
         $\delta fit$  = Fitness( $y$ ) -  $fit$ ;
         $s$  = NumSimilarValues( $x$ ,  $y$ ,  $loci$ ,  $r + 1$ );
        add [ $r + \frac{s}{2}$ ,  $\delta fit$ ] to list  $fitBlock$ ;
      fi;
    od;
     $t$  = MaximalLocus( $fitBlock$ );
    if ( $t.\delta fit > 0$ ) then
      add  $t.index$  to list  $truncPoints$ ;
    fi;
  od;
  ## generate the truncated list of loci
   $truncationPoint$  = Median( $truncPoints$ );
   $Truncate$  = TruncateList( $loci$ ,  $truncationPoints$ );
end

```

Here *RandomIndiv()* generates a random test-individual, *NumSimilarValues*( $x$ ,  $y$ ,  $loci$ ,



|   |   |   |   |   |
|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 1 |
| * |   | * | * |   |

×

|   |   |   |   |   |
|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 0 |
|   |   | * | * | * |

↓

|   |   |   |   |   |
|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 |
| * |   |   | * | * |

Figure 7.3: An example of the masked crossover

$r$ ) determines the number of subsequent ranked loci in  $x$  and  $y$  that have the same value starting from locus  $r$ ,  $MaximalLocus(fitBlock)$  extracts the locus of list  $fitBlock$  that has the highest value of  $\delta fit$ ,  $Median(m)$  extracts the median from a list of numbers, and  $TruncateList(list, pos)$  truncates a list at location  $pos$ .

### 7.3 Mixing with masked uniform crossover

The filtering method described in section 7.2 produces a masked individual, where the masked bits are considered to be a potential building block. Next, different building blocks have to be combined in order to find a globally optimal solution to the problem. Masks do not have to represent exactly a building block. It is possible that a mask includes loci that do not belong to the same building block, or it might miss some of the loci that do belong to the building block. Therefore, a genetic algorithm is used with a special crossover operator to perform the mixing task. Next, the usage of the mask during the crossover is described, and it is shown how to transfer the information present in the mask to the offspring.

In the case that the parent individuals have disjunct masks, the crossover is relatively simple. The masked loci of the first parent are transferred to the offspring, next the masked loci of the second parent are added, and then uniform crossover is applied for the remaining loci. If there is an overlap between the masks of the two parents at a certain locus, then two cases have to be distinguished. If both parents have the same bit-value at this locus, then the masked parts are compatible and the same procedure can be applied. If both parents define different values for the locus, then a parent is chosen at random to provide the value for this locus.

The next step involves the creation of a mask for the offspring based on the two masks of the parents. A locus is masked in the offspring when it was masked by one of the parents, or when it was masked by both of the parents and the bit-values of the both parents at the corresponding locus are equal. If both parents masked a locus but define a different value, then the locus is not masked in the offspring. As a result a pruning of the masks can happen.

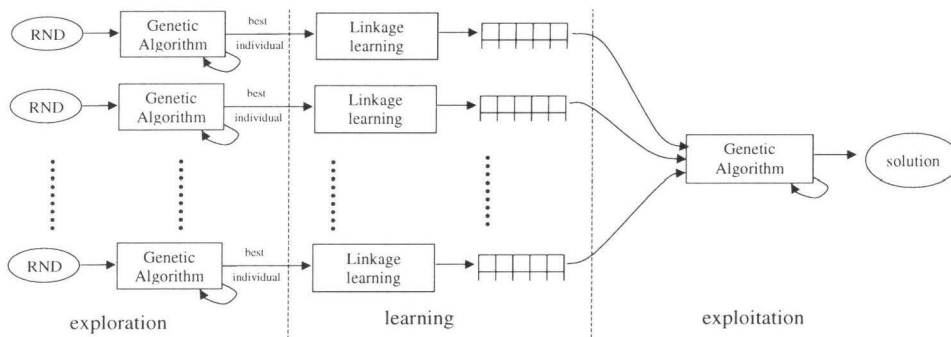


Figure 7.4: Overview of the algorithm

Figure 7.3 shows an example of the application of the masked crossover. The first locus and the last locus are masked by exactly one parent, so the offspring inherits the bit-value from the corresponding parents, and the corresponding mask-bits of the offspring are set for these loci. The third locus is masked by both parents, but the parents have non-matching bit-values, so an arbitrary parent is selected, and the mask-bit is not propagated. The fourth locus is also masked by both parents, but this time the bit-values match, so it does not matter from which parent the value of this locus is taken. In this case the corresponding mask-bit in the offspring is set.

## 7.4 The hybrid bbf-GA

The bbf-GA uses a three-stage approach, that follows the schematic overview given in Figure 7.1. During the first stage a large number of rapidly converging GA's is used to explore the search-space. The best individual of each GA is passed to the next stage. In the second stage each individual is filtered in order to mask the building blocks present in this individual. The third stage consists of a GA that exploits these masked individuals by mixing them to obtain the global optimal solution. Figure 7.4 shows a schematic representation of this algorithm.

**Exploration:** we use a large set of small-population GA's running independently to explore the search space. This way cross-competition between building blocks is prevented (as different building blocks can be discovered by different GA's). The initial populations of these GA's are chosen at random. It is important to use small-population GA's during this phase because:

1. GA's with small populations converge fast (i.e. reduction of the computational overhead),

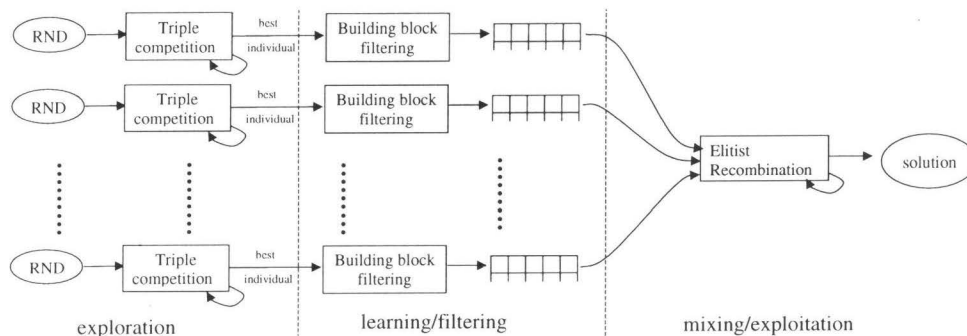


Figure 7.5: The bbf-GA

- GA's with small populations are not very reliable, so different GA's are likely to explore different parts of the search space. When using a reliable GA for exploration the most important building block is discovered repeatedly, and the GA is likely to be too slow.

**Learning:** we want, given an individual (suboptimal solution), to locate the most important part of it. In fact one tries to learn the linkage of bits within the context of this specific individual (bit-string)

**Exploitation:** search for the optimal solution by means of a single reliable GA. The initial population of this GA consists of all the masked individuals created during the previous phase. This GA should exploit all information of the different GA's used during the exploration phase and should make use of the linkage learned during the learning phase.

During the exploration phase a GA is needed that converges rapidly, and is likely to converge to different solutions, such that different parts of the search space are explored. Although this GA does not have to be reliable, it is important that the GA really does an exploration of part of the search space and therefore a GA that prevents too much duplication is preferred. The triple-competition selection, described in section 2.3.2 meets these requirements. Nothing can be assumed about the linkage of bits during exploration; Therefore, the uniform crossover is used.

The linkage learning can be performed by means of the building block filtering described in section 7.2. The output of this second phase is a set of masked individuals.

During the exploitation phase a reliable GA is required that performs well on a mixing task, even in the case that the population is quite small. Therefore elitist recombination, described in section 2.3.3 is chosen. The information provided by the building block filtering is exploited by using the masked crossover operator. A detailed schematic representation of the bbf-GA is given in Figure 7.5

## 7.5 Test-problems

To study the effectiveness of the filtering and the mixing, two scalable test-problems are used. The number of building blocks  $m$  can be adjusted and the size of the optimal building blocks  $d$  is adjustable. For both test-problems the global optimum can be partitioned in a set of  $m$  order  $d$  building blocks.

### 7.5.1 The fully deceptive problem

The first test-problem is based on the fully deceptive trap-functions that have been discussed in section 2.6. The following formula is used to compute the fitness contribution of a single part:

$$f_D(b) = \begin{cases} \alpha d & \text{if } u(b) = d \\ (d - u(b))/d & \text{otherwise} \end{cases}.$$

Here  $\alpha > 1$ . The global optimum of the deceptive part contains only 1-bits, which results in a fitness contribution of  $\alpha$ .

Based on this building block a scalable test-problem can be constructed by concatenating  $m$  of these order  $d$  building blocks whose fitness contribution is determined by  $f_D(b)$ . The fitness of an complete individual is computed as the sum of the contributions of all its parts divided by  $m\alpha$ , so the fitness ranges from 0 to 1. A solution to this problem can be coded in a straightforward manner in a bit-string of length  $l = m \times d$ . The actual linkage of the bits belonging to the same part is assumed to be unknown, so we have a problem with loose linkage.

### 7.5.2 The NK-bb problem

Typical properties of the fully deceptive problem are that the first-order statistics for the fitness values of each locus are deceptive, and that the different parts can be optimized completely independent of each other. Therefore a second test-problem, the NK-bb problem, is introduced that does not have these properties, but does have building blocks. As a basis for the problem we use NK-landscapes described in section 2.11.

In order to build a test-problem with  $m$  building blocks of size  $d$  we take an NK-landscape length  $l = m \times d$  with a random neighbourhood, and we select a random partition of the bit-strings in  $m$  parts. The fitness contribution of a bit is set to one if the part that bit belongs to is completely filled with 1-bits, otherwise the contribution of the bit is determined by the underlying NK-landscape. The fitness of a complete bit-string is computed as the average fitness-contribution of all the bits. The optimal solution is again a bit-string consisting of only 1-bits, each given a fitness-contribution of one. The NK-bb problem does have a set of independent building blocks of order  $d$ . The optimal building blocks are completely independent of each other, but if an optimal building block is not present within a certain part, then we have nonlinear interactions that cross the boundaries of the partitions, so this problem is not separable.

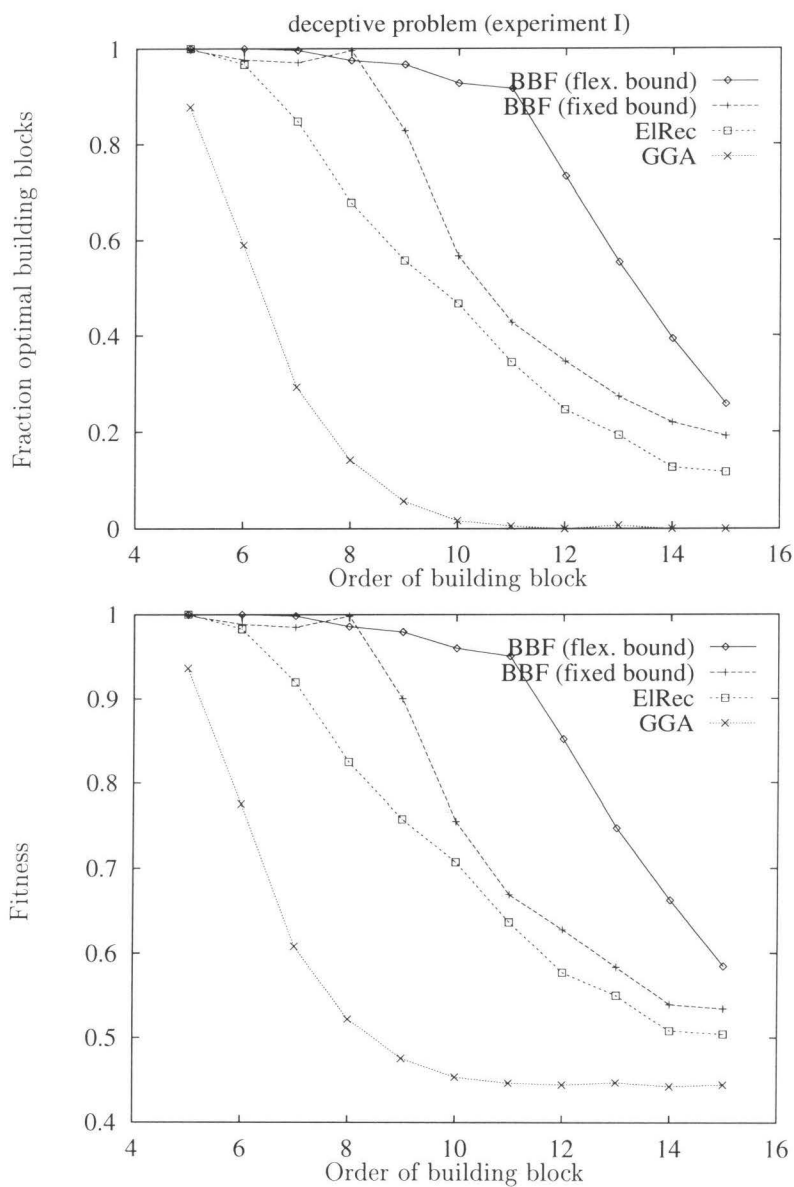


Figure 7.6: The average number of optimal building blocks in the best solution (top) and the fitness (bottom) when using different optimization methods for the deceptive problem.

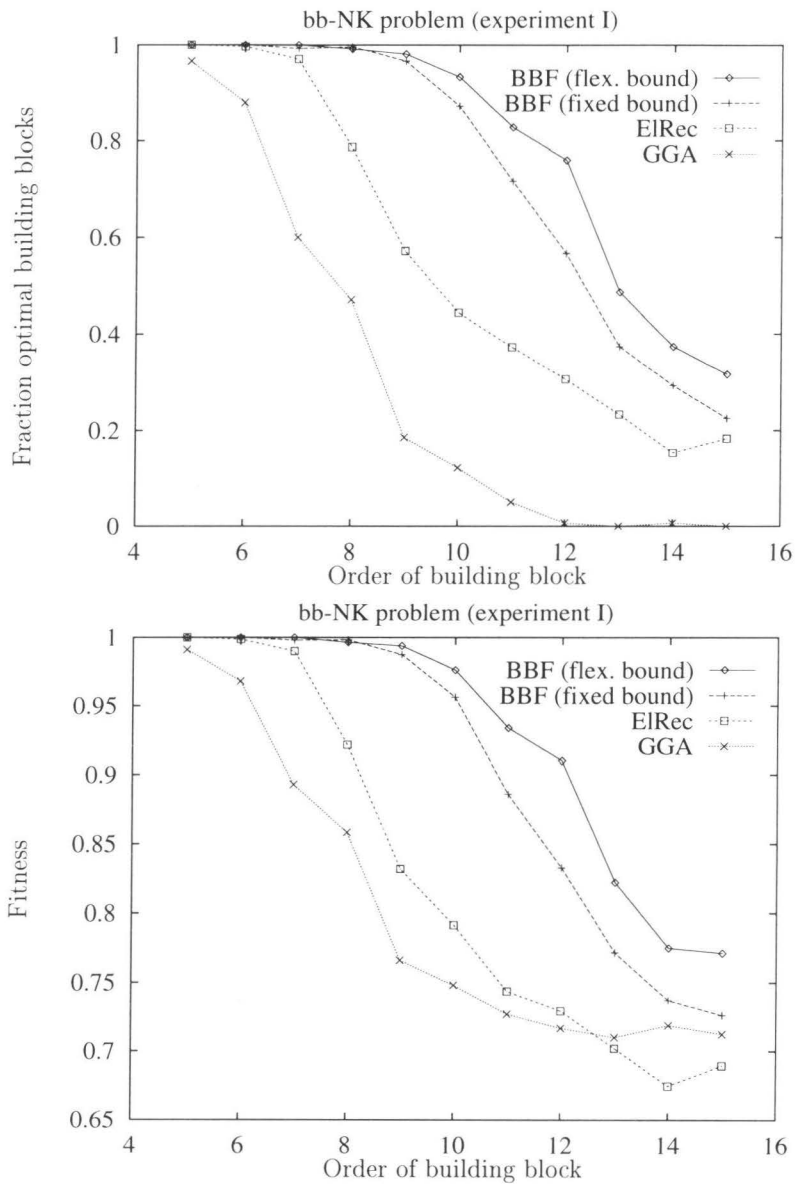


Figure 7.7: The average number of optimal building blocks in the best solution (top) and the fitness (bottom) when using different optimization methods for the NK-bb problem.

## 7.6 Experiments

For the exploratory phase of the bbF-GA a population size of 24 is used, which is evolved for 12 generations. Two variants of building block filtering have been investigated. The first variant uses a fixed bound for the number of bits that are going to be selected in the mask. The number of selected bits is set to eight. The second variant uses a flexible number of bits that is determined by the procedure given at the end of section 7.2. During the third stage elitist recombination with masked crossover is applied for 100 generations on a population of 100 masked individuals.

For the purpose of comparison we also conducted tests using the elitist recombination, the generational GA with tournament selection, and a steady-state GA. Elitist recombination using uniform crossover, is also applied directly to the problem with a population of 300 individuals that is allowed to converge for 100 generations. Furthermore a generational GA with tournament selection is applied with tournament-size 2, population size 300, 100 generations, and crossover is always applied.

For the deceptive problem  $\alpha = 1.5$  is used, and for the NK-bb problem a random neighbourhood of size  $k = 10$  is taken. All results shown are averaged over 30 independent runs. The bbF-GA with fixed bound for the number of bits to mask, uses 28,000 fitness evaluations per experiment, while the bbF-GA with flexible bounds uses approximately 10,000 additional evaluations to determine the bound during the filtering stage. The generational GA and elitist recombination use 30,000 fitness evaluations per experiment.

A set of 44 different experiments (experiment I) was conducted for varying orders of the building blocks  $d$ , and varying numbers of building blocks  $m$ . Each experiment was repeated 30 times. The upper graphs of Figures 7.6 and 7.7 show the average properties of the overall best individual. In case of the bbF-GA and elitist recombination this individual is present in the last population, but in case of the generational GA this individual might be lost, as the off-line performance is shown. For  $m$  the smallest value such that  $l = m \times d \geq 60$  is taken, so for  $d = 9$  one has  $m = 7$  such that the size of an individual becomes 63 bits. The upper graph of Figure 7.6 shows the average fraction of building blocks present in the best solution for the deceptive problem. The lower graphs shows the off-line best fitness. The upper graph of Figure 7.7 shows the average fraction of building blocks present in the best solution for the NK-bb problem, and the lower graph shows the corresponding fitness. On both problems the bbF-GA using a flexible bound performs best followed by the bbF-GA with fixed bound, the elitist recombination, and the generational GA in sequence. The flexible bound selection for the bbF-GA performs well on the deceptive problem while on the NK-bb problem the results are only slightly better than for the bbF-GA with a fixed bound. The lower graph of Figure 7.7 shows the average fitness of the best solution as a function of the order of the optimal building block for the NK-bb problem. It is interesting to see that the generational GA outperforms elitist recombination for  $d > 12$  even though it does not find any of the optimal building-blocks. Note that these graphs show the properties of the overall best individual, so it might be the case that the individual is not present in the final population of the generational GA and that the generational GA mainly does a random search.

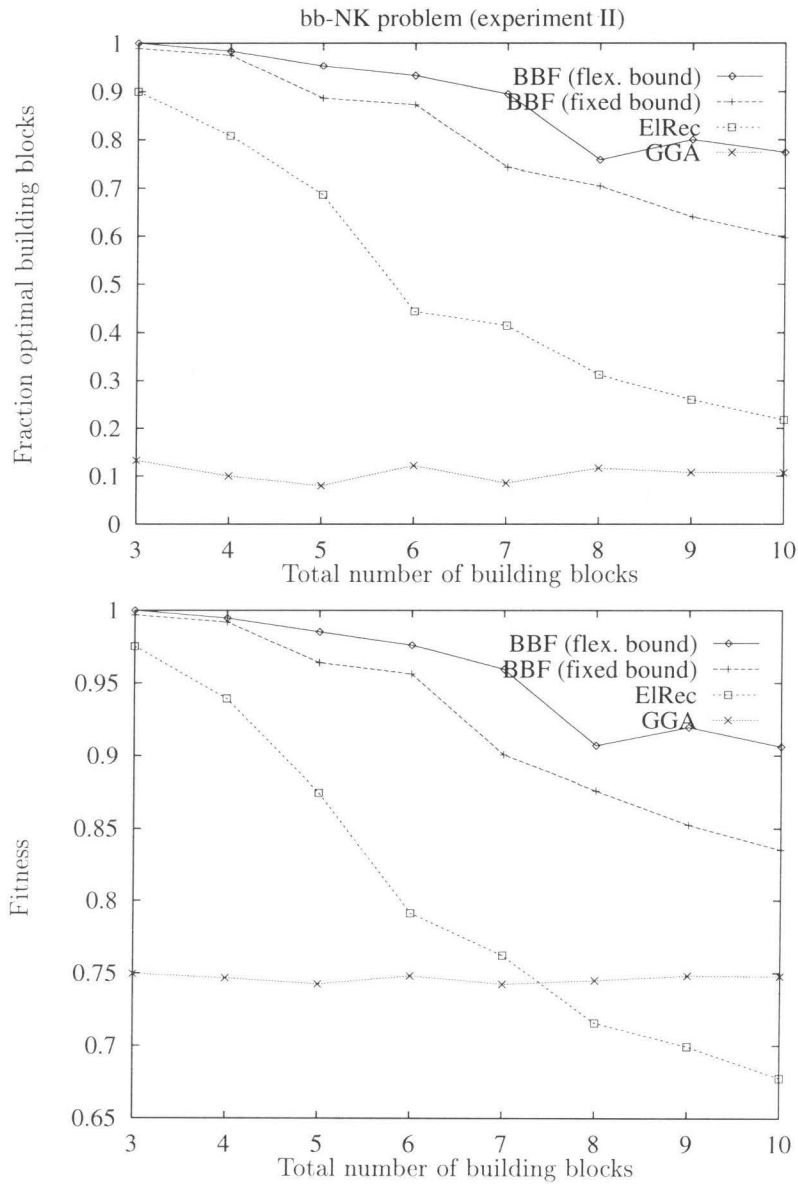


Figure 7.8: The average number of optimal building blocks in the best solution (top) and the fitness (bottom) when using different optimization methods for the NK-bb problem (Experiment II).



A second set of 32 experiments (experiment II) was conducted. During these experiments the size of the building blocks was fixed at  $d = 10$  while the number of building blocks was varied. Again each experiment was repeated 30 times. Figure 7.8 shows the fraction of optimal building blocks (top) and the fitness (bottom) of the best solution for the NK-bb problem. Again the bbf-GA with flexible bound performs best while the generational GA performs worst. The performance of the BFF-GA's degrades less when increasing the number of building blocks than the elitist recombination. The fraction of building blocks in the best solution is low and nearly constant for the generational GA.

## 7.7 Enhancements to the bbf-GA

The test-problems are difficult and the bbf-GA gives good results. However, more extensive tests are required to properly assess the performance of the bbf-GA, and compare it to other algorithms. Results on a larger test-suite are shown in chapter 8. The test-problems shown in the current chapter are basically concatenations of functions of unitation. In the NK-bb problem interactions that cross the boundaries of building blocks are present, but these interactions are likely to be relatively small. A number of improvements have been added to the bbf-GA in order to prepare it for broader application. These improvements are discussed in the next subsections. The results shown in section 8 are obtained with the bbf-GA containing the enhancements described here.

### 7.7.1 Filtering step

Two changes have been applied to the filtering step. The first change involves the correction for the average marginal fitness of a locus. This average marginal fitness is determined over the set of all individuals that are used during the learning phase. During the filtering step the marginal fitness of locus  $i$  in function  $\text{FilterStep}(x)$  is now computed by the formula

$$\delta fit'_i = \text{Fitness}(x') - \text{Fitness}(x) - \text{Av}(\delta fit_i, x_i);$$

where  $x'$  and  $x$  only differ at locus  $i$ ,  $\delta fit_i$  is given in the first definition of the filtering step, and  $\text{Av}_i(\delta fit_i, x_i)$  denotes the average change in fitness when changing the value of locus  $i$  from  $x_i$  to the complementary value. This average change in fitness is computed over the set of all individuals that are filtered. This adjustment helps in getting an appropriate ordering of the loci. The original rule performed well on functions of unitation, but did not always perform well in case that building blocks overlap.

The function Truncation has been reformulated based on the definition of a building block, as given in section 2.6. There, a building block is defined as a schema such that schema fitness of the complete schema is strictly larger than the sum of the schema fitnesses of an arbitrary partition. This reflects the fact that a building block has a larger fitness contribution than one would expect, when accumulating the fitnesses of its parts. Given this definition of a building block one can locate the truncation point by comparing the fitness after transferring the values of  $k$  loci, to the fitness after transferring  $(k - 1)$  loci

plus the fitness of transferring only ranked locus  $k$ . If the transfer of  $k$  loci results in a larger fitness contribution, and results in a positive overall fitness contribution, then this locus is marked as a possible truncation point. Furthermore all subsequent loci where the two individuals match are marked as possible truncation points. After applying this procedure to all test-individuals, the locus with the maximal number of marks is selected. Furthermore a minimal value is imposed on the number of loci transferred.

### 7.7.2 Masked crossover

In the masked crossover operator defined in section 7.3 a single mask is generated for the offspring. When building blocks can overlap, it is less convenient to merge the masks of both parents in a single mask for the offspring. Therefore a representation is chosen where each individual carries a list of masks. During crossover the parents are allowed to take turns in propagating a masked set of bit-values to the offspring. The masked bits are transferred in a random order, and a set of masked bits is only transferred when these bits are compatible with the bits already present in the offspring, and the mask defines at least one bit that is not present in the offspring. After all masks of both parents have been processed, the remaining loci in the offspring are determined by a uniform crossover of both parents. No pruning of masks is performed, and the offspring contains a list of all masks that were transferred successfully from one of the parents to this offspring. Exact duplicates of masks are removed. A further advantage of this approach is that the individuals in the final population also contain information on the decomposition of these individuals by means of the mask-list these individuals carry with them.

### 7.7.3 Deterministic crowding

The elitist recombination has been replaced by deterministic crowding, which is described in section 2.3.3. The only difference between elitist recombination and deterministic crowding is that in deterministic crowding each parent competes with only one offspring; The parents and offspring are paired such that the similarity between the parents and the offspring is maximized. Due to the masked crossover, the same parts of an individual are transferred time after time. This can lead to a rapid duplication of the best few masked individuals. Deterministic crowding performs better at preventing this type of duplication.

### 7.7.4 Population size during exploitation

In the explorative phase some loci might always converge to the same value. If that case, such a locus can never get the opposite value during the exploitation phase. To prevent this the population size during exploitation is doubled, and the second half of the population is filled with random individuals. These individuals provide the bit-values that might have been lost. A further advantage is that all masked individuals on average get one opportunity to duplicate their masked part before encountering another masked individual.

## 7.8 Summary

The proposed building block filtering method is able to discover parts of high-order building blocks even in quasi-random landscapes. Using this filtering method we create masked individuals which are processed with a special crossover operator. This crossover operator uses the mask to process a set of bits, assumed to be relatively important, in one piece while the non-masked bits are processed by a uniform crossover. The masked crossover also computes a new mask for the offspring based on the masks and the bit-values of both of the parents.

The bbf-GA is a hybrid three-stage GA. During the first stage an exploration of the search space is performed. The second stage locates a set of potential building blocks, and the third stage tries to exploit the the building blocks by mixing them in order to generate the optimal solution. The bbf-GA outperformed its competitors and scales better when increasing either the size, or the number of building blocks on our test-problems.

# Chapter 8

## Comparison of genetic algorithms

In this chapter a diverse test-suite of binary optimization problems is given. For all problems it is assumed that the linkage of the bits is not known. This test-suite is used to make a comparison between a set of eight different optimization methods introduced or discussed in this dissertation: the random-mutation hill-climber, the generational genetic algorithm with fitness proportional selection, the generational genetic algorithm with tournament selection, the elitist recombination, the triple-competition, the mixing evolutionary algorithm, and the building block filtering genetic algorithm. These methods are described more in detail in section 8.2. The results on the test-suite are shown and discussed in section 8.3, followed by a summary in section 8.4.

### 8.1 Test-suite

The focus of this dissertation is recombinative evolutionary search, and most research has been directed towards obtaining a better understanding of the processes of building block discovery and mixing. Hereby, we have restricted ourselves to a limited class of problems. However, by studying the behaviour of GA's on these problems thoroughly and by enhancing the building block processing capabilities of GA's, we hope to have extended the range of problems where GA can outperform other (randomized) search methods. Therefore we developed a test-suite of problems in order to show how the different methods perform on binary problems with unknown linkage. We have explicitly chosen to focus on problems with unknown linkage, because the challenging problems are those where linkage is not known. If linkage is known, then efficient traditional methods are likely to exist. For example, suppose we have a binary coded problem that can be decomposed in  $m$  independent subproblems each encoded by a bit-string of order  $k$ . Solving all subproblems by means of a complete enumeration of the search-space of these subproblems and combining the results requires  $m2^k$  function evaluations. If the linkage information and thus the decomposition of the problem is not known, then solving this problem by means of enumeration requires  $2^{mk}$  function evaluations. Given that a decomposition of the search space is possible, it is really valuable to have methods that are able to discover and exploit knowledge about

|   | schema    | fitness |   | schema   | fitness |
|---|-----------|---------|---|----------|---------|
| 1 | #####1    | 0.0080  | 5 | ###1#### | 0.0080  |
| 2 | #####1#   | 0.0080  | 6 | ##1##### | 0.0080  |
| 3 | #####1##  | 0.0080  | 7 | #1#####  | 0.0080  |
| 4 | #####1### | 0.0080  | 8 | 1#####   | 0.0080  |

Table 8.1: Building blocks of problem BT1 (Royal road).

linkage within a problem-space.

For all problems in this test-suite we assume the linkage to be unknown. A binary problem with known linkage can easily be transformed to a problem with unknown linkage. To do so one generates a random permutation  $\pi$  of length  $l$ , where  $l$  is the length of the bit-string. If the original problem is defined as  $f(b)$ , then a problem with unknown linkage can be defined by  $g(b) = (f \circ \pi)(b)$ .

In some of the problems below one can not optimize the subfunctions independently of each other due to the fact that the subfunctions use overlapping partitions. If the subfunctions require opposite values for the overlapping loci, then the subfunctions are not compatible. This effect is called “frustration”.

Multimodality means that there is more than one local optimum. Multimodal problems might be difficult to optimize because the GA might be tracking different local optima simultaneously. If the local optima differ in many bits, then the recombination of near optimal strings close to different optima is likely to result in inferior offspring.

Most problems involve some kind of deception, and therefore are at least partially deceptive. Basically, when a partition of order  $k$  is fully deceptive, then processing of only schema of order  $m < k$  guides the search to the bit-wise complement of the optimal schema.

In the rest of this section, let  $\vec{b}$  denote a binary vector of length  $l$ , let  $b_i$ ,  $1 \leq i \leq l$  denote element  $i$  of this vector, and let  $b_{i,j}$  denote the substring consisting of bits  $b_i b_{i+1} \cdots b_j$  of  $\vec{b}$ .

### 8.1.1 BT1: Royal road

The Royal-road function by Mitchel is a simple function, involving building blocks of order 8 [Mit96]. The function does not involve deception. An individual consists of a string of 64 bits, divided in 8 blocks of 8 bits. Each substring of order 8 is evaluated by the function

$$f_{\text{mitchel}}(x) = \begin{cases} 1.0 & \text{for } x = 11111111 \\ 0.0 & \text{otherwise} \end{cases},$$

so, this function is defined as

$$F_{BT1}(\vec{b}) = \sum_{i=0}^7 f_{\text{mitchel}}(b_{8i,8i+7}).$$

This problem involves no deception and it is relatively easy to find the local optimum by doing hill-climbing. The only structure defined by this function are the completed building

|   | schema     | fitness |    | schema    | fitness |
|---|------------|---------|----|-----------|---------|
| 1 | #####1     | 7.5     | 6  | ####1#### | 7.5     |
| 2 | #####1#    | 7.5     | 7  | ###1##### | 7.5     |
| 3 | #####1##   | 7.5     | 8  | ##1#####  | 7.5     |
| 4 | #####1###  | 7.5     | 9  | #1#####   | 7.5     |
| 5 | #####1#### | 7.5     | 10 | 1#####    | 7.5     |

Table 8.2: Building blocks of problem BT2 (massively multimodal).

blocks. Therefore if a locus changes value, this will only influence fitness if the number of building blocks in the string changes. Because the building blocks are relatively large it can take a while before a building block is detected in a certain partition. Genetic drift can then easily result in the loss of alleles in this partition.

Figure 8.1 shows the building blocks of a royal road function of eight bits according to the definition of the building block given in section 2.6. The block 11111111 is not included, because this block can be composed from the lower order building blocks.

### 8.1.2 BT2: Massively multimodal function

A single subfunction of the so-called massively multimodal function is defined by

$$f_{multi}(x) = u(x) + g(x) \quad \text{where} \quad g(x) = \begin{cases} 2 & \text{if } odd(u(x)) \\ 0 & \text{otherwise} \end{cases},$$

where  $x$  corresponds to a bit-string of order 5. The complete function consists of a concatenation 20 of such blocks, so an individual is represented by a bit-string of 100 bits, so

$$F_{BT2}(\vec{b}) = \sum_{i=0}^{19} f_{multi}(b_{5i,5i+4}).$$

This problem is highly multimodal. We use the standard binary coding for this problem. In that case the problem is relatively simple because the first order schemata are not deceptive.

Figure 8.2 shows the building blocks for the massively multimodal function of ten bits.

### 8.1.3 BT3/4: Deceptive trap function

Deceptive trap functions were introduced by Ackley [Ack87]. We use the deceptive function as defined by Deb and Goldberg [DG93]. A difficult instance of this class can be created by using the parametrized set of fully deceptive trap functions [Gol89a]. A fully deceptive

|   | schema      | fitness |    | schema     | fitness |
|---|-------------|---------|----|------------|---------|
| 1 | #####0      | 3.688   | 7  | ###0#####  | 3.688   |
| 2 | #####0#     | 3.688   | 8  | ##0#####   | 3.688   |
| 3 | #####0##    | 3.688   | 9  | #0#####    | 3.688   |
| 4 | #####0###   | 3.688   | 10 | 0#####     | 3.688   |
| 5 | #####0####  | 3.688   | 11 | #####11111 | 6.688   |
| 6 | #####0##### | 3.688   | 12 | 11111##### | 6.688   |

Table 8.3: Building blocks of problem BT3 (deceptive function)

trap (sub)function of order  $k$  has value [Kar95]

$$f_{decept}(x) = \begin{cases} k & \text{if } u(x) = k \\ k - u(x) - 1 & \text{otherwise} \end{cases}$$

where  $u(x)$  is a function that counts the number of 1-bits in  $x$ . The global optimum of this function is the string consisting of  $k$  1-bits resulting in the maximal fitness contribution  $k$ . The second best solution is a string consisting of  $k$  0-bits having value  $(k - 1)$ . Because decreasing the number of one bits usually increases fitness, except for the optimal string, hill-climbing algorithms will be strongly attracted by the second best optimum.

By concatenating  $m$  of these order  $k$  subfunctions a building block problem is created, that has a solution which can be represented by a bit-string of length  $l = m \times k$ .

We use two variants of this function, the first function is BT3 with  $k = 5$ ,  $m = 40$ , and  $l = 200$ , so

$$F_{BT3}(\vec{b}) = \sum_{i=0}^{39} f_{decept}(b_{5i, 5i+4}).$$

The second function is BT4 with  $k = 8$ ,  $m = 25$ , and  $l = 200$ , so

$$F_{BT4}(\vec{b}) = \sum_{i=0}^{24} f_{decept}(b_{8i, 8i+7}).$$

These problems are difficult due to the deception present in these problems and due to the fact that a single problem instance contains many building blocks. The BT4 is more difficult than the BT3 because high-order building blocks are difficult to process when linkage is unknown.

Figure 8.3 shows the building blocks for an order-five deceptive problem of ten bits.

#### 8.1.4 BT5: function with frustration

Mühlenbein [MR98] defined a few test-functions where the building blocks are conflicting. Such “frustrated” problems are difficult to solve due to these conflicts. Even if the EA

|   | schema | fitness |   | schema | fitness |
|---|--------|---------|---|--------|---------|
| 1 | ####0  | 0.57    | 5 | 0####  | 0.373   |
| 2 | ###0#  | 0.461   | 6 | 1##0#  | 0.525   |
| 3 | ##0##  | 0.57    | 7 | 10###  | 0.525   |
| 4 | #0###  | 0.461   |   |        |         |

Table 8.4: Building blocks of problem  $F_{cubanI}^5$ .

|   | schema     | fitness |    | schema    | fitness |
|---|------------|---------|----|-----------|---------|
| 1 | #####0     | 0.918   | 10 | 1##0##### | 0.873   |
| 2 | #####0#    | 0.809   | 11 | 10#####   | 0.873   |
| 3 | #####0##   | 0.918   | 12 | ####1#000 | 2.125   |
| 4 | #####0###  | 0.809   | 13 | ####10#00 | 2.125   |
| 5 | #####0#### | 0.943   | 14 | ####100#0 | 2.125   |
| 6 | ###0#####  | 0.809   | 15 | ####1000# | 2.125   |
| 7 | ##0#####   | 0.918   | 16 | ##0#01##0 | 1.735   |
| 8 | #0#####    | 0.809   | 17 | ##0#0#010 | 2.33    |
| 9 | 0#####     | 0.72    |    |           |         |

Table 8.5: Building blocks of problem BT5 (function with “frustration”).

finds all building blocks, then it might still be difficult to combine them. The functions  $F_{cubanI}^3$  and  $F_{cubanI}^5$  are given by

$$F_{cubanI}^3(x) = \begin{cases} 0.595 & \text{if } x = 000 \\ 0.200 & \text{if } x = 001 \\ 0.595 & \text{if } x = 010 \\ 0.100 & \text{if } x = 011 \\ 1.00 & \text{if } x = 100 \\ 0.05 & \text{if } x = 101 \\ 0.09 & \text{if } x = 110 \\ 0.15 & \text{if } x = 111 \end{cases},$$

$$F_{cubanI}^5(x_1x_2x_3x_4x_5) = \begin{cases} 4F_{cubanI}^3(x_1x_2x_3) & \text{if } x_4 = x_2 \text{ and } x_5 = x_3 \\ 0 & \text{otherwise} \end{cases}.$$

Given the auxiliary function  $F_{cubanI}^5$  the test-problem BT5 is defined as

$$F_{BT5}(\vec{b}) = \sum_{j=0}^{L-1} F_{cubanI}^5(b_{4j,4(j+1)}),$$

where the subsequent substrings overlap in one locus. For odd values of  $L$  this function has a single global optimum. The optimum consist of alternating substrings 10000 and 00101.



The first substring is the optimum of subfunctions  $F_{cubanI}^5$ , having value 4.0. The second substring corresponds to the third-best solution having value 0.8. The optimal solution for  $L = 15$  is

$$10000010100000101000001010000010100000101000001010000010100000,$$

and consists of 61 bits.

Adjacent parts can not both contain an optimal building block due to “frustration”. The optimal solution contains an optimal building block in all odd parts. This way an individual containing eight optimal blocks can be constructed. Individuals with optimal blocks in the even parts can contain at most seven optimal blocks. Due to the conflicts all the bits of the string interact.

Figure 8.4 shows the building blocks of the function  $F_{cubanI}^5$ , and Figure 8.5 shows the building blocks of function BT5 of nine bits. Note that the building blocks of the first five bits correspond to the building blocks of  $F_{cubanI}^5$ . The range of bits from locus five to locus nine have different building blocks due to the overlap in locus five.

### 8.1.5 BT6: function with long blocks

This is a modified version of the BT5 problem with a larger size of the blocks. A block consists of ten bits. The fitness of a block is defined by

$$F_{long}^5(x) = \min\{F_{cubanI}^5(x_{0-4}), F_{cubanI}^5(x_{5-9})\}.$$

The test-problem BT6 is given by

$$f_{BT6}(\vec{b}) = \sum_{j=0}^{L-1} F_{long}^5(b_{9j,9(j+1)}),$$

where subsequent substrings overlap in one locus. The optimal value for  $L = 7$  is

$$1000010000010100101000010000010100101000010000010100101000010000,$$

which consists of 64 bits.

### 8.1.6 BT7: function with frustration

Mühlenbein [MR98] defined the following function that involves “frustration”, and where the optimum is composed of an alternation of different building blocks.

$$F_{cuban2}^5(x) = \begin{cases} u(x) & \text{for } x = 0***0 \\ 0 & \text{for } x = 0***1 \\ u(x) & \text{for } x = 1***0 \\ u(x) - 2 & \text{for } x = 1***1 \end{cases}.$$

|   | schema | fitness |   | schema | fitness |
|---|--------|---------|---|--------|---------|
| 1 | ####0  | 2.0     | 4 | #1###  | 1.75    |
| 2 | ###1#  | 1.75    | 5 | 1####  | 2.0     |
| 3 | ##1##  | 1.75    |   |        |         |

Table 8.6: Building blocks of problem  $F_{cuban2}^5$  (subfunction of BT7).

|    | schema      | fitness |    | schema         | fitness |
|----|-------------|---------|----|----------------|---------|
| 1  | #####0      | 4.07    | 18 | #####11####    | 3.848   |
| 2  | #####1#     | 3.82    | 19 | #####1#1####   | 3.973   |
| 3  | #####1##    | 3.82    | 20 | #####1##1####  | 3.848   |
| 4  | #####1###   | 3.82    | 21 | ###11#####     | 3.848   |
| 5  | #####1####  | 3.848   | 22 | ##1#1#####     | 3.973   |
| 6  | #####0##### | 3.559   | 23 | #1##1#####     | 3.848   |
| 7  | #####0##### | 3.668   | 24 | 1##0#####      | 3.622   |
| 8  | #####0##### | 3.559   | 25 | 10#####        | 3.622   |
| 9  | #####0##### | 4.095   | 26 | #####000####   | 4.067   |
| 10 | #####1##### | 3.82    | 27 | #####0#00####  | 4.067   |
| 11 | #####1##### | 3.82    | 28 | #####00#0####  | 4.067   |
| 12 | #####1##### | 3.82    | 29 | #####0#01####  | 4.317   |
| 13 | #####1##### | 3.848   | 30 | #####010####   | 4.317   |
| 14 | ###0#####   | 3.559   | 31 | ##000#####     | 4.067   |
| 15 | #0#####     | 3.667   | 32 | #0#00#####     | 4.067   |
| 16 | #0#####     | 3.559   | 33 | #00#0#####     | 4.067   |
| 17 | 0#####      | 3.47    | 34 | #####10000#### | 5.847   |

Table 8.7: Building blocks of problem BT7 (problem with “frustration”).

The test-problem BT7 is defined as

$$F_{BT7}(\vec{b}) = \sum_{i=0}^{L-1} f_{\text{aux}}(\vec{b}, i),$$

where the auxiliary function is defined as

$$f_{\text{aux}}(\vec{b}, i) = \begin{cases} F_{cuban1}^5(b_{5i, 5(i+1)}) & \text{if } i \text{ is odd} \\ F_{cuban2}^5(b_{5i, 5(i+1)}) & \text{otherwise} \end{cases}.$$

Subsequent substrings overlap in one bit, and  $L = 4m + 1$  for an integer  $m$ . The definition of  $F_{cuban1}^5(x)$  is given in section 8.1.4. The optimum string consists of a concatenation of the four substrings 10000, 01110, 00101, and 11111. The first and the third substring are evaluated by  $F_{cuban1}^5$  while the second and the fourth substring are evaluated by  $F_{cuban2}^5$ .

The fitness-contribution of these substrings is 1.0, 3, 0.8, and 3 respectively. The first substring is the only local optimum out of these four substrings. The global optimum consists of a repetition of these four strings, starting with substring one. For  $L=17$  the global optimum is,

100001110010111110000111001011111000011100101111100001110010111110000

which has a value of 32.2.

Figure 8.6 shows the building blocks for  $F_{cuban2}^5$ , and Figure 8.7 shows the building blocks for a function BT5 of 17 bits.

### 8.1.7 BT8: NK-landscape

The NK-landscapes [Kau93] are quasi-random fitness landscapes that were already discussed in section 2.11<sup>1</sup>. We take  $F_{BT8}(\vec{b}) = f_{NK}(\vec{b})$ . So,

$$f_{BT8}(\vec{b}) = \sum_{i=1}^l f_i(b_i \cdot N_i),$$

where  $f_i$  is a random function,  $x_i$  denotes the value of bit  $i$  of individual  $x$ ,  $N_i$  is a bit-string representing the values of the loci that form the neighbourhood of bit  $i$ ,  $\cdot$  is the concatenation operator, and  $k$  is a parameter of the landscape that takes a value in the range 0 to  $l-1$ . Here, an NK-landscape with a length of 80 bits, with random linkage, and a neighbourhood of size  $k=7$  is used.

### 8.1.8 BT9: NK-bb landscape

We introduce a new type of NK-landscape, the so-called NK-bb landscape. In order to build a test-problem with  $m$  building blocks of size  $d$  we take an NK-landscape length  $l = m \times d$  with a random neighbourhood, and we select a random partitioning of the bit-string in  $m$  parts. The fitness contribution of a bit is 1.0 if the part that bit belongs to is completely filled with 1-bits, otherwise the contribution of the bit is determined by the underlying NK-landscape. The fitness of a complete bit-string is the average fitness-contribution of all the bits. This corresponds to

$$F_{BT9}(\vec{b}) = \frac{1}{l} \sum_{i=0}^{m-1} f_{aux}(\vec{b}, i),$$

where

$$f_{aux}(\vec{b}, i) = \begin{cases} d & \text{if } u(b_{di, di+d-1}) = d \\ \sum_{j=d \cdot i}^{di+d-1} f_j(b_j \cdot N_j) & \text{otherwise} \end{cases}$$

<sup>1</sup>We used an implementation of NK-landscapes by Terry Jones from the Santa Fe Institute

and where the neighbourhoods  $N_j$  are random neighbourhoods over the complete string  $\vec{b}$ . The optimal solution is again a bit-string consisting of only 1-bits having fitness 1.0. The bb-NK problem does have a set of independent building blocks of order  $d$ . The optimal building blocks are independent of each other, but if an optimal building block is not present within a certain part then we have nonlinear interactions that cross the boundaries of the partitions, so this problem is not separable. We consider a problem with a length of 80 bits, with random linkage, and a neighbourhood of size  $k = 7$ . The size of a single block was set to 8 bits, and there is no relation between the neighbourhood structure and the location of these 8 bits.

## 8.2 Compared methods

The following algorithms will be compared on this set of test-functions.

1. RMHC: Random-mutation hill-climber with  $p_m = 1/l$ ,
2. CGA: canonical GA as described in section 2.3.1 with  $n = 300$ ,  $p_c = 1.0$  and  $p_m = 1/l$ ,
3. TGA: generational GA with tournament selection as described in section 2.3.1 with  $n = 300$ ,  $p_c = 0.6$  and  $p_m = 1/l$ ,
4. TGAc: generational GA with tournament selection with  $n = 300$ ,  $p_c = 1.0$  and  $p_m = 1/l$ ,
5. ER: Elitist recombination described in section 2.3.3 with  $n = 300$ ,
6. TC: Triple-competition described in section 2.3.2 with  $n = 300$ ,
7. mixEA: Mixing evolutionary algorithm as introduced in chapter 6, with  $n_{init} = 3000$ ,  $n_{rec} = 32$ , maximal number of stacks = 10 , and
8. bbf-GA: Building block filtering GA as introduced in section 7.7 where during exploration we use 100 populations of size 24 that are evaluated for 12 generations, during the filtering step the number of test-individuals is nine, the minimal mask length is five, the maximal length is twelve, and the maximal number of failures to increase fitness is six out of nine, and during the exploitation step deterministic crowding with population size 200 is used, where the second half of the population is filled with randomly generated individuals.

If the mutation operator is applied to a string, then for each locus a change of value is performed with probability  $p_m = 1/l$ , where  $l$  is the length of the bit-string. The recombination operator is the uniform crossover. The maximal number of function evaluations is set to 100,000. For the CGA and the TGAc we always use recombination ( $p_c = 1$ ). Always using recombination helps in preventing to rapid convergence, and thereby helps to overcome premature convergence. For comparison we added the TGA with  $p_c = 0.6$ .

If a problem can be solved by means of hill-climbing, then it is likely that such a hill-climbing approach is more efficient than an evolutionary approach. Hill-climbing methods can preserve information efficiently, while EA's always carry the overhead of evolving a population of individuals. In our suite of optimization methods we have included a randomized hill-climbing algorithm. This randomized hill-climber is similar to the random-mutation hill climbing [FM93, Mit96], except for the applied mutation operator. In the original implementation exactly one bit was changed by the mutation operator. We consider this to be too restrictive. Therefore we apply the mutation operator, that is used in the evolutionary algorithms. On average the value of one locus is changed, but it is possible that multiple loci change. This can help in preventing that the hill-climber becomes trapped. The pseudo-code for the random-mutation hill-climber is as follows.

```

RMHC()
  best = RandomString();
  numEval = 1;
  while (numEval < maxNumEval) do
    mut = mutate(best);
    if (Fitness(mut) ≥ Fitness(best)) then
      best = mut;
    fi;
  od;
  RMHC = best;
end

```

Here, the *mutate*(*x*) creates a copy of the string *x*, and changes each bit to the opposite value with a probability  $1/l$ .

In all cases the fitness is set equal to the value of the objective function. All test-functions have to be maximized, and for all test-functions we have  $f(x) \geq 0$ .

### 8.3 Results

To compare these eight methods a number of performance measures are used:

1. proportion of runs that found the optimum,
2. distance to the optimum,
3. number of function evaluations until the best individual is observed,
4. number of function evaluations before termination,
5. proportion of building blocks in the best solution,
6. proportion of building blocks in the final population, and

7. average number of building blocks in each individual in the final population.

All results have been computed by running 50 independent experiments and computing the median values over all these runs. The median value of a set of values  $x_0, x_1, \dots, x_n$  is a value  $x_m$  that exists in this set such that half of the values is smaller than or equal to  $x_m$ . In case of symmetric distributions the median value is close to the average value. In case of a skewed distribution this does not have to be the case. We use the median values instead of the average values, because the median values are less sensitive to outliers in the data-set.

Next, we will show the results in a number of tables. In these tables a row contains the results for a single test-problem, and each column represents an optimization method. For each test-problem the best results are shown in boldface. In case of a tie, multiple entries are shown in boldface.

|     | RMHC        | CGA  | TGA         | TGAc        | ER          | TC          | mixEA | bbf-GA      |
|-----|-------------|------|-------------|-------------|-------------|-------------|-------|-------------|
| BT1 | <b>1.00</b> | 0.00 | 0.28        | 0.24        | <b>1.00</b> | 0.54        | 0.00  | <b>1.00</b> |
| BT2 | 0.72        | 0.00 | <b>1.00</b> | <b>1.00</b> | <b>1.00</b> | <b>1.00</b> | 0.00  | 0.02        |
| BT3 | 0.00        | 0.00 | 0.00        | 0.00        | 0.00        | 0.00        | 0.00  | <b>0.74</b> |
| BT4 | 0.00        | 0.00 | 0.00        | 0.00        | 0.00        | 0.00        | 0.00  | 0.00        |
| BT5 | 0.00        | 0.00 | 0.00        | 0.00        | <b>0.04</b> | 0.00        | 0.00  | 0.00        |
| BT6 | 0.00        | 0.00 | 0.00        | 0.00        | <b>0.08</b> | 0.02        | 0.00  | 0.06        |
| BT7 | 0.00        | 0.00 | 0.04        | 0.00        | <b>0.04</b> | 0.00        | 0.00  | 0.00        |
| BT8 | —           | —    | —           | —           | —           | —           | —     | —           |
| BT9 | 0.00        | 0.00 | 0.00        | 0.00        | 0.24        | 0.02        | 0.00  | <b>0.90</b> |

Table 8.8: Proportion of runs that find the optimum.

Table 8.8 shows the proportion of runs that find the global optimal solution. So a value 0.8 means that during 80% of the runs the global optimum is found. The elitist recombination performs best on five out of nine problems and the bbf-GA performs best on four out of nine problems when using this performance measure. The bbf-GA outperforms the elitist recombination on problems where uniform crossover is expected to be inefficient, because large building blocks have to be transferred in one piece, such as BT3, BT4, and BT9. The random-mutation hill-climbing always finds the optimum for the problems BT1. This indicates that this problem is relatively easy to solve, and therefore does not require an evolutionary algorithm.

Table 8.9 shows the median distance to the optimum. On six out of nine problems the bbf-GA approaches the optimal fitness value the closest. Especially on those problems where uniform crossover is not very efficient, the bbf-GA performs very well; However, it is outperformed by elitist recombination on problems involving “frustration” (BT5 and BT7) and problems that do not have building blocks (BT2 and BT8).

Table 8.10 shows the median number of function evaluations performed before the best individual is obtained. This value is used as an indicator for the speed of a method.

|     | RMHC        | CGA   | TGA         | TGA <sub>c</sub> | ER          | TC          | mixEA | bbf-GA      |
|-----|-------------|-------|-------------|------------------|-------------|-------------|-------|-------------|
| BT1 | <b>0.00</b> | 2.00  | 1.00        | 1.00             | 0.00        | 0.00        | 4.00  | <b>0.00</b> |
| BT2 | <b>0.00</b> | 29.00 | <b>0.00</b> | <b>0.00</b>      | <b>0.00</b> | <b>0.00</b> | 10.00 | 10.00       |
| BT3 | 37.00       | 76.00 | 40.00       | 40.00            | 40.00       | 40.00       | 77.00 | <b>0.00</b> |
| BT4 | 25.00       | 63.00 | 25.00       | 25.00            | 25.00       | 25.00       | 55.00 | <b>9.00</b> |
| BT5 | <b>0.08</b> | 1.34  | <b>0.08</b> | <b>0.08</b>      | <b>0.08</b> | 0.16        | 0.64  | 0.20        |
| BT6 | 1.86        | 1.64  | 0.44        | 0.86             | <b>0.04</b> | 0.16        | 4.50  | <b>0.04</b> |
| BT7 | <b>0.08</b> | 7.26  | <b>0.08</b> | 1.20             | 0.08        | 0.12        | 2.99  | 0.12        |
| BT8 | 0.26        | 0.36  | <b>0.24</b> | <b>0.24</b>      | 0.26        | 0.26        | 0.34  | 0.33        |
| BT9 | 0.24        | 0.32  | 0.12        | 0.14             | 0.03        | 0.10        | 0.26  | <b>0.00</b> |

Table 8.9: Median of the minimum distance to optimum over all runs.

|     | RMHC        | CGA   | TGA   | TGA <sub>c</sub> | ER    | TC           | mixEA | bbf-GA |
|-----|-------------|-------|-------|------------------|-------|--------------|-------|--------|
| BT1 | 10373       | 46424 | 56297 | 60254            | 19121 | <b>8599</b>  | 49000 | 30841  |
| BT2 | 66764       | 56644 | 19339 | 14765            | 12844 | <b>5649</b>  | 75063 | 43133  |
| BT3 | <b>2192</b> | 76849 | 21793 | 16807            | 18279 | 7728         | 89639 | 45435  |
| BT4 | <b>2241</b> | 67551 | 19366 | 15642            | 14869 | 6685         | 42000 | 48944  |
| BT5 | 61620       | 55254 | 68898 | 65680            | 37481 | <b>7507</b>  | 74186 | 34075  |
| BT6 | 27805       | 62284 | 65161 | 76313            | 28189 | <b>8580</b>  | 89573 | 35287  |
| BT7 | 25981       | 58035 | 50286 | 49228            | 23802 | <b>6760</b>  | 41960 | 37052  |
| BT8 | 23129       | 59333 | 81465 | 78859            | 92933 | <b>14223</b> | 75644 | 19720  |
| BT9 | 14004       | 43946 | 87936 | 83657            | 46986 | <b>12173</b> | 85360 | 34054  |

Table 8.10: Median of the number of function evaluations before finding best over all runs.

The triple-competition finds the best solution very fast. If we also consider the median distance to the optimal fitness-value, which is shown in Figure 8.9, then we see that triple-competition is not only fast, but it also finds relatively good solutions. Elitist recombination and TGA<sub>c</sub> are also relatively fast.

Table 8.11 shows the number of iterations before termination. A run is terminated when the optimum is reached, when the maximal number of function evaluations is exceeded, or when the method can not continue (in case of the mixEA and bbf-GA). The random-mutation hill-climber performs best on problems BT1, where it always finds the optimal solution. In all other cases the bbf-GA performs best.

Table 8.12 shows the fraction of optimal building blocks present in the best solution. The fraction of optimal building blocks in the optimal solution is 1 for BT1-4 and BT9, 0.47 for BT5, 0.57 for BT6, and 0.29 for BT7. The bbf-GA performs best on five out of the eight problems for which this measure is defined.

Table 8.12 shows the fraction of optimal building blocks present in the final population. A building block is present when at least one individual contains this building block. The

|     | RMHC         | mixEA  | bbf-GA       |
|-----|--------------|--------|--------------|
| BT1 | <b>10373</b> | 100409 | 31031        |
| BT2 | 75780        | 99948  | <b>53849</b> |
| BT3 | 100001       | 100001 | <b>45782</b> |
| BT4 | 100001       | 85310  | <b>63802</b> |
| BT5 | 100001       | 98819  | <b>49906</b> |
| BT6 | 100001       | 100010 | <b>50179</b> |
| BT7 | 100001       | 65108  | <b>50727</b> |
| BT8 | 100001       | 100021 | <b>51822</b> |
| BT9 | 100001       | 100016 | <b>35769</b> |

Table 8.11: Median of the number of iterations before termination over all runs.

|     | RMHC        | CGA         | TGA         | TGAc        | ER          | TC          | mixEA | bbf-GA      |
|-----|-------------|-------------|-------------|-------------|-------------|-------------|-------|-------------|
| BT1 | <b>1.00</b> | 0.75        | 0.88        | 0.88        | <b>1.00</b> | <b>1.00</b> | 0.50  | <b>1.00</b> |
| BT2 | <b>1.00</b> | 0.40        | <b>1.00</b> | <b>1.00</b> | <b>1.00</b> | <b>1.00</b> | 0.80  | 0.75        |
| BT3 | 0.07        | 0.00        | 0.00        | 0.00        | 0.00        | 0.00        | 0.11  | <b>1.00</b> |
| BT4 | 0.00        | 0.00        | 0.00        | 0.00        | 0.00        | 0.00        | 0.00  | <b>0.64</b> |
| BT5 | <b>0.47</b> | 0.27        | 0.40        | 0.40        | 0.40        | 0.27        | 0.20  | 0.20        |
| BT6 | 0.29        | <b>0.43</b> | 0.29        | 0.29        | <b>0.43</b> | 0.29        | 0.29  | <b>0.43</b> |
| BT7 | 0.18        | 0.24        | 0.24        | <b>0.29</b> | 0.18        | 0.18        | 0.18  | 0.12        |
| BT8 | —           | —           | —           | —           | —           | —           | —     | —           |
| BT9 | 0.10        | 0.20        | 0.55        | 0.50        | 0.90        | 0.60        | 0.40  | <b>1.00</b> |

Table 8.12: Median proportion of optimal building blocks in best.

fraction is a value between zero and one. The bbf-GA performs best on four out of the eight problems for which this measure is defined. The canonical genetic algorithm performs well on three problems. This is mainly due to the fact that the CGA does not converge, and hence all building blocks are still present in the population. This does not mean that the CGA is effective at optimizing these problems as we have seen in Tables 8.8 and 8.9.

Table 8.14 shows the average number of building blocks per individual. The random-mutation hill-climber uses a population of size one, therefore this figure corresponds to the number of building blocks in the final solution. The mixEA and the bbf-GA perform well on the deceptive trap functions. Even though the final population mixEA on most problems contains a large fraction of all building blocks, as shown in Table 8.13, it performs not well in getting these building blocks mixed. This is probably due to the inefficiency of the uniform crossover at mixing building blocks. In a traditional GA this is remedied by the fact that many duplicates of building blocks are generated. If two individuals are recombined, then the building blocks present in both individuals are likely to be present in the offspring (if no mutation is applied, then these building blocks are always present in the



|     | RMHC        | CGA         | TGA         | TGAc        | ER          | TC          | mixEA       | bbf-GA      |
|-----|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|
| BT1 | <b>1.00</b> | 0.88        | 0.88        | 0.88        | <b>1.00</b> | <b>1.00</b> | <b>1.00</b> | <b>1.00</b> |
| BT2 | <b>1.00</b> | <b>1.00</b> | <b>1.00</b> | <b>1.00</b> | <b>1.00</b> | <b>1.00</b> | 0.80        | 0.90        |
| BT3 | 0.07        | 0.47        | 0.00        | 0.00        | 0.00        | 0.00        | 0.95        | <b>1.00</b> |
| BT4 | 0.00        | 0.00        | 0.00        | 0.00        | 0.00        | 0.00        | 0.20        | <b>0.64</b> |
| BT5 | 0.47        | <b>1.00</b> | 0.73        | 0.80        | 0.53        | 0.27        | 0.43        | 0.30        |
| BT6 | 0.29        | 0.43        | 0.57        | 0.43        | 0.43        | 0.29        | <b>1.00</b> | 0.57        |
| BT7 | 0.18        | <b>0.94</b> | 0.76        | 0.71        | 0.24        | 0.18        | 0.24        | 0.29        |
| BT8 | —           | —           | —           | —           | —           | —           | —           | —           |
| BT9 | 0.10        | 0.70        | 0.60        | 0.50        | 0.90        | 0.60        | <b>1.00</b> | <b>1.00</b> |

Table 8.13: Median proportion of optimal building blocks in population.

|     | RMHC         | CGA  | TGA   | TGAc  | ER           | TC           | mixEA | bbf-GA       |
|-----|--------------|------|-------|-------|--------------|--------------|-------|--------------|
| BT1 | <b>8.00</b>  | 2.59 | 4.16  | 4.09  | 7.97         | 7.97         | 0.54  | 6.61         |
| BT2 | <b>20.00</b> | 1.98 | 16.05 | 16.37 | <b>20.00</b> | <b>20.00</b> | 8.25  | 14.52        |
| BT3 | 3.00         | 0.13 | 0.00  | 0.00  | 0.00         | 0.00         | 5.90  | <b>37.66</b> |
| BT4 | 0.00         | 0.00 | 0.00  | 0.00  | 0.00         | 0.00         | 1.92  | <b>15.92</b> |
| BT5 | <b>7.00</b>  | 2.44 | 4.75  | 4.45  | 5.97         | 3.99         | 3.00  | 2.00         |
| BT6 | 2.00         | 1.48 | 1.88  | 1.44  | <b>2.99</b>  | 1.99         | 0.85  | 2.13         |
| BT7 | <b>3.00</b>  | 1.92 | 3.71  | 4.81  | 2.99         | 2.99         | 2.50  | 2.05         |
| BT8 | —            | —    | —     | —     | —            | —            | —     | —            |
| BT9 | 1.00         | 0.05 | 3.81  | 3.31  | <b>8.89</b>  | 5.98         | 1.06  | 8.02         |

Table 8.14: Median of the average number of building blocks per individual.

offspring). MixEA prevents duplication of good individuals, and therefore has difficulties when it has to combine a very large number of building blocks.

A set of graphs is included that show the distance to the optimum of the best individual as a function of the number of function evaluations for all problems, and all optimization methods. This best individual is determined off-line, which means that this individual does not have to be present in the final population (this is possible for the generational GA's); In these plots both axes use a logarithmic scale. The diamonds in these graphs denoted the median distance to the optimum, computed over all runs, and the length of the error-bars is chosen such that for 90% of the runs the best individuals fall in the corresponding region.

Figures 8.1 and 8.2 show the results for the problem BT1 (the royal road). On this problem only eight different fitness values are possible. The corresponding distances to the optimum are clearly visible in all graphs as a set of discrete steps that the median takes when converging towards the optimum. The RMHC converges fast to the optimum on this problem, indicating that this is a relatively simple problem. The CGA, TGA, and TGAc converge slowly. The ER converges fast and reliable. The TC also converges fast, but it

does not converge to the optimum in all cases, which is clear from the large confidence interval. The mixEA converges slowly. When using the mixEA different individuals are likely to contain many different building blocks. Therefore the probability of a successful recombination becomes very small during later phases of evolution. The bbF-GA converges slowly during the exploration and filtering phase. However, given the masked individuals the optimum is found almost instantly during the exploitation phase.

Figures 8.3 and 8.4 show the curves for problem BT2 (the massively multimodal problem). The CGA fails on this problem. During the later stages of evolution the average fitness is close to 140, which means that the fitness proportional selection induces almost no selective pressure. Tournament selection does not have this problem, and both the TGA and the TGA<sub>c</sub> find the optimum fast. ER and TC both converge rapidly and reliably. The graph of the mixEA shows a jump close to 20,000. This corresponds to the transition from the first to the second evolutionary stack. Apparently, the mixEA is often able to recombine individuals in the first stack (using at most 32 recombination per pair of parents), but these individuals do not combine easily.

Figures 8.5 and 8.6 show the results on problem BT3 (the deceptive problem of order 5). The RMHC, TGA, TGA<sub>c</sub>, ER, and TC converge fast to the deceptive attractor, which results in a distance of 40 with respect to the optimum. The CGA converges much slower towards the deceptive attractor. This is again due to the low selective pressure when the average fitness becomes large. The mixEA converges slowly, but as we have seen from table 8.14 its population contains a high density of optimal building blocks. The bbF-GA finds the optimal solution easily.

Figures 8.7 and 8.8 show the results for problem BT4 (the deceptive problem of order 8). On this problem RMHC, CGA, TGA, TGA<sub>c</sub>, ER, and TC converge to the deceptive attractor, resulting in a distance of 25 with respect to the optimum. The bbF-GA performs much better, and combines approximately 15 out of 25 optimal building blocks in a single individual.

Figures 8.9 and 8.10 show the curves for problem BT5 (function with “frustration”). The RMHC converges slowly, indicating that this function can not be optimized easily by hill-climbing. None of the methods finds the optimum on this problem. A closer inspection showed that the individuals that are filtered contain the building blocks for 2.7 functions  $F_{cubant}^5$  on average. As a result of this, the building block filtering marked part of these building blocks. As a result suboptimal building blocks were processed during the exploitation phase.

Figures 8.12 and 8.13 show the curves for problem BT6 (function with long blocks and “frustration”). Recall that for this problem each building block was composed of two building blocks of the kind used in problem BT5. All methods show roughly the same convergence characteristics as on problem BT5, except for the bbF-GA which is able to locate the optimum in a number of runs.

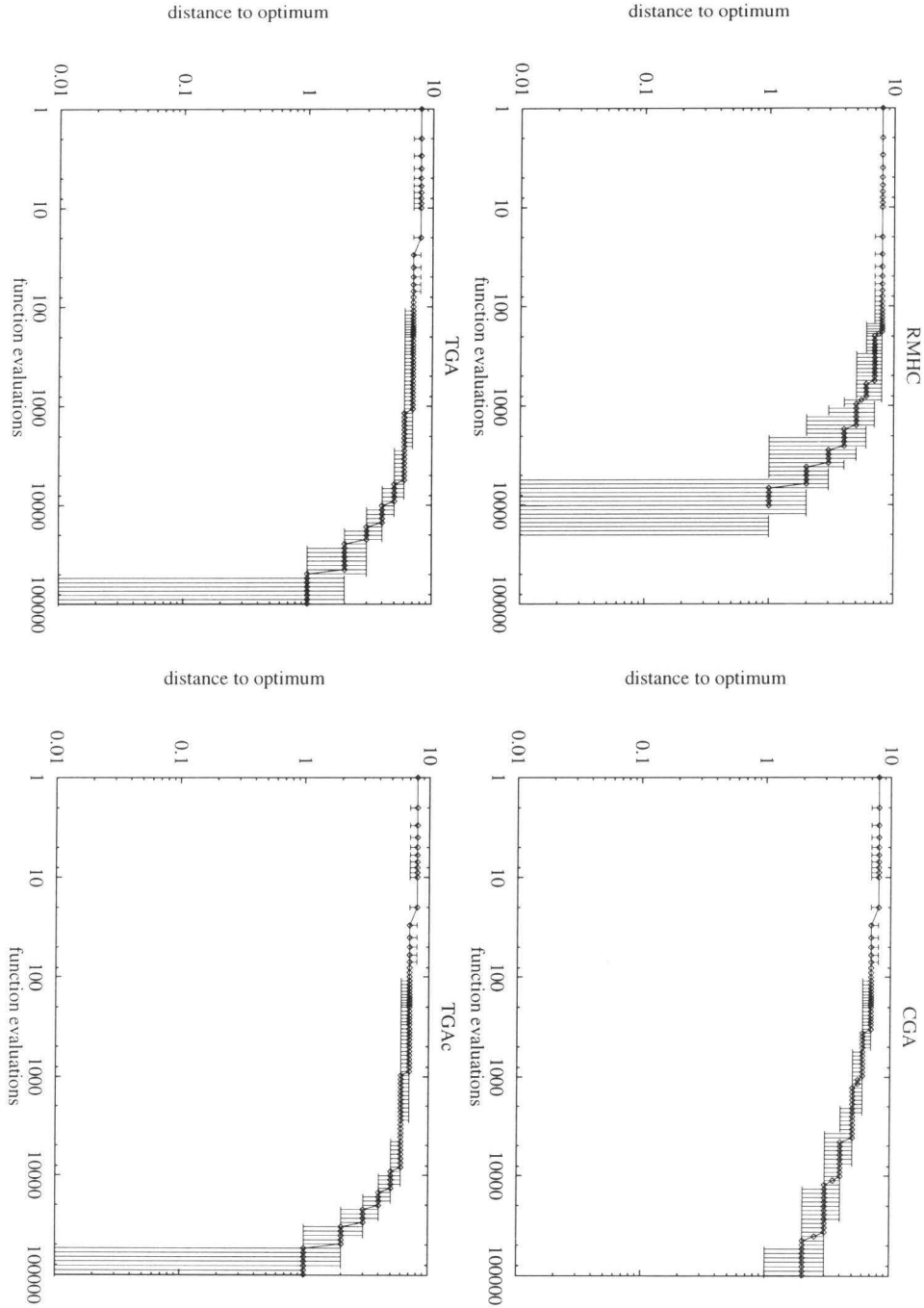


Figure 8.1: Results on the BT1 (Royal-Road function).

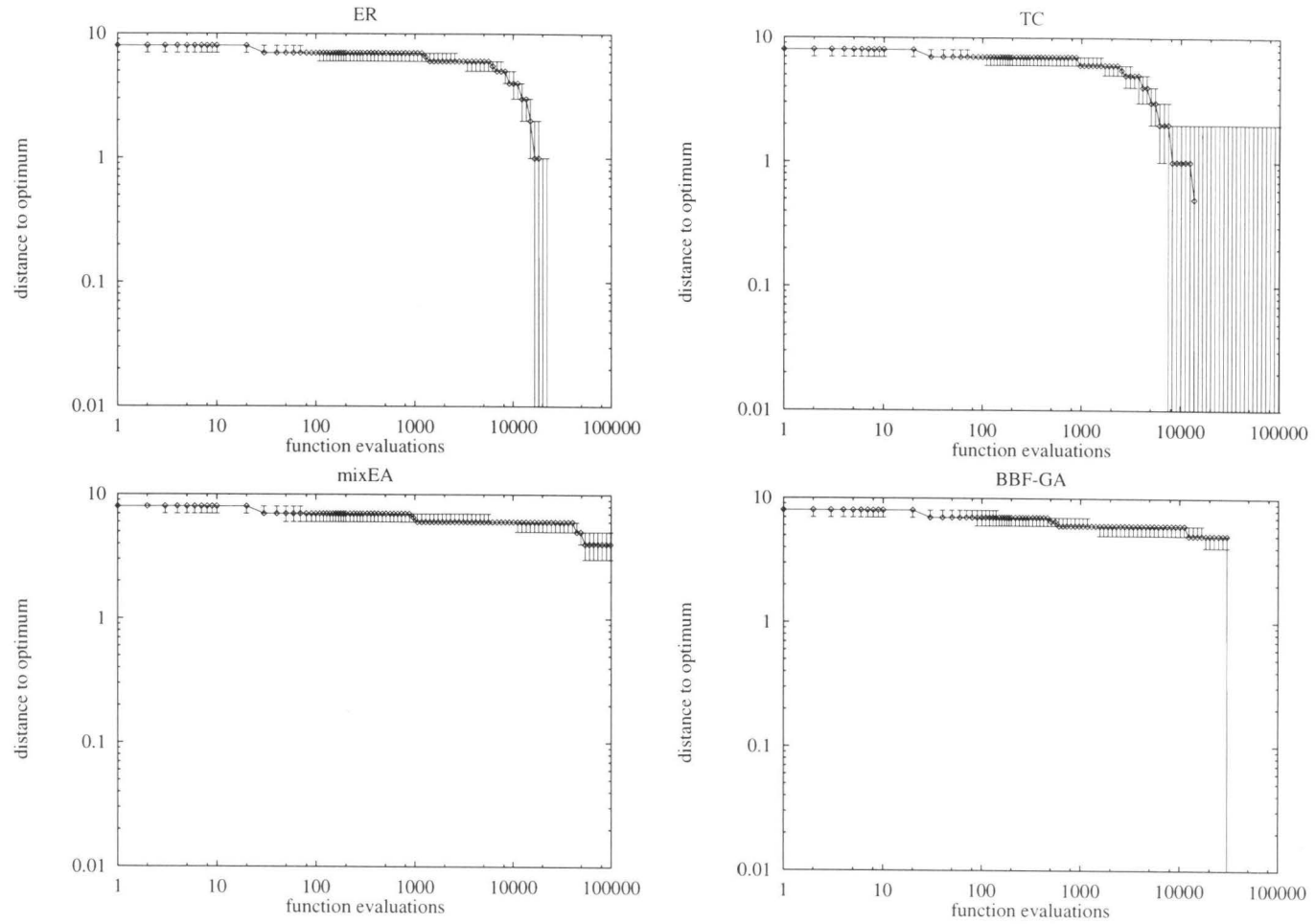


Figure 8.2: Results on BT1 (the Royal-Road function).

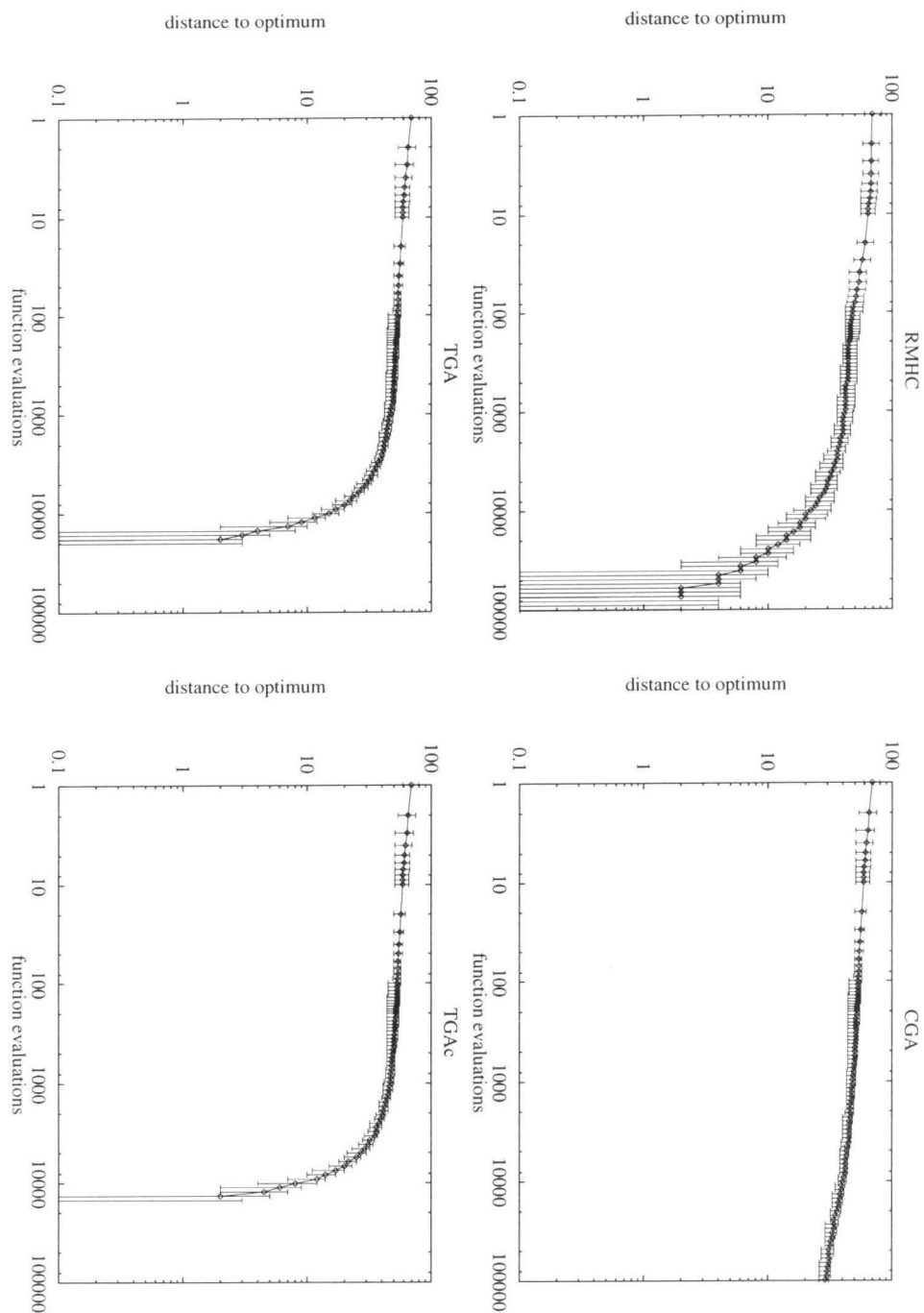


Figure 8.3: Results on BT2 (the massively multimodal).

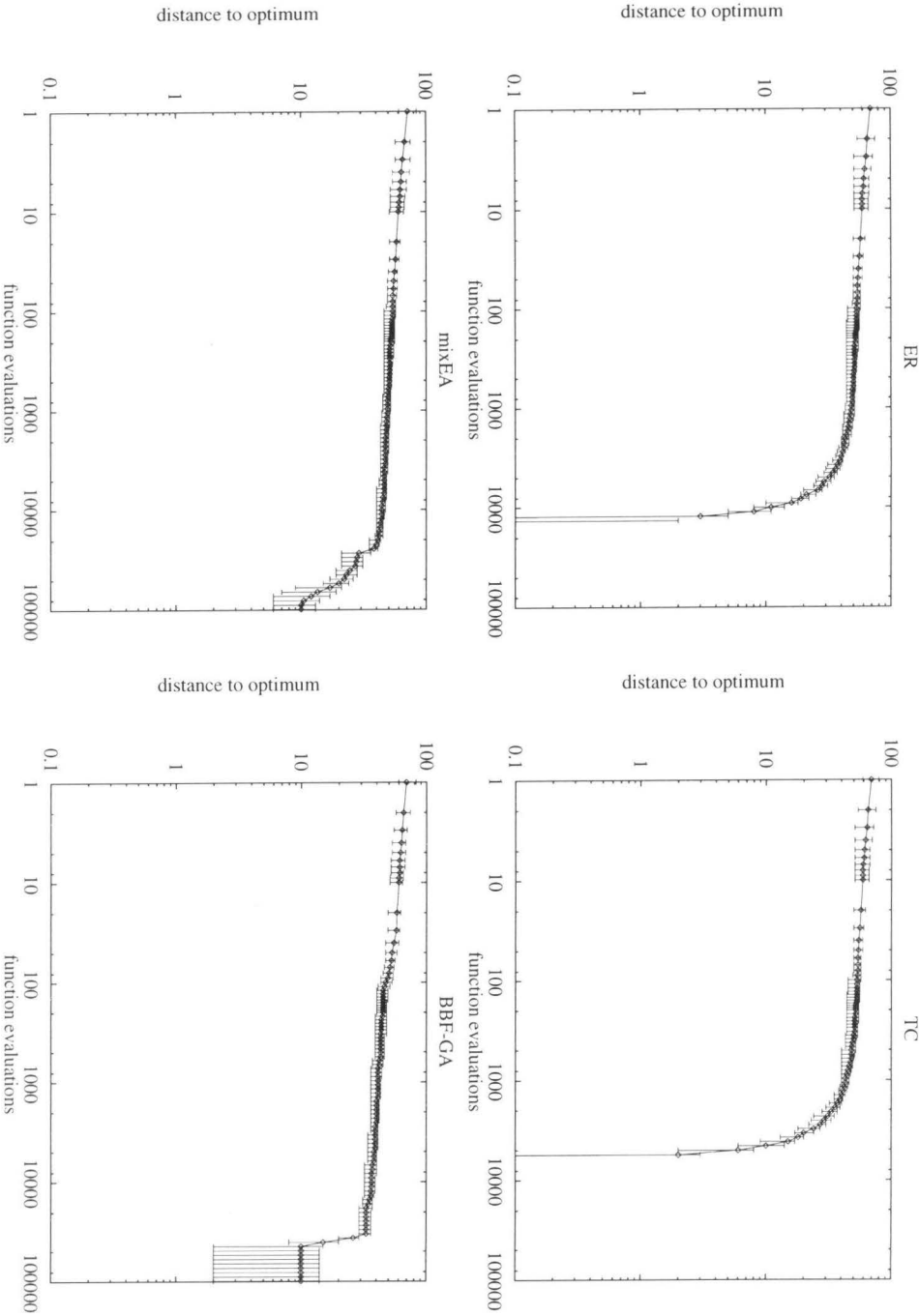


Figure 8.4: Results on BT2 (the massively multimodal).

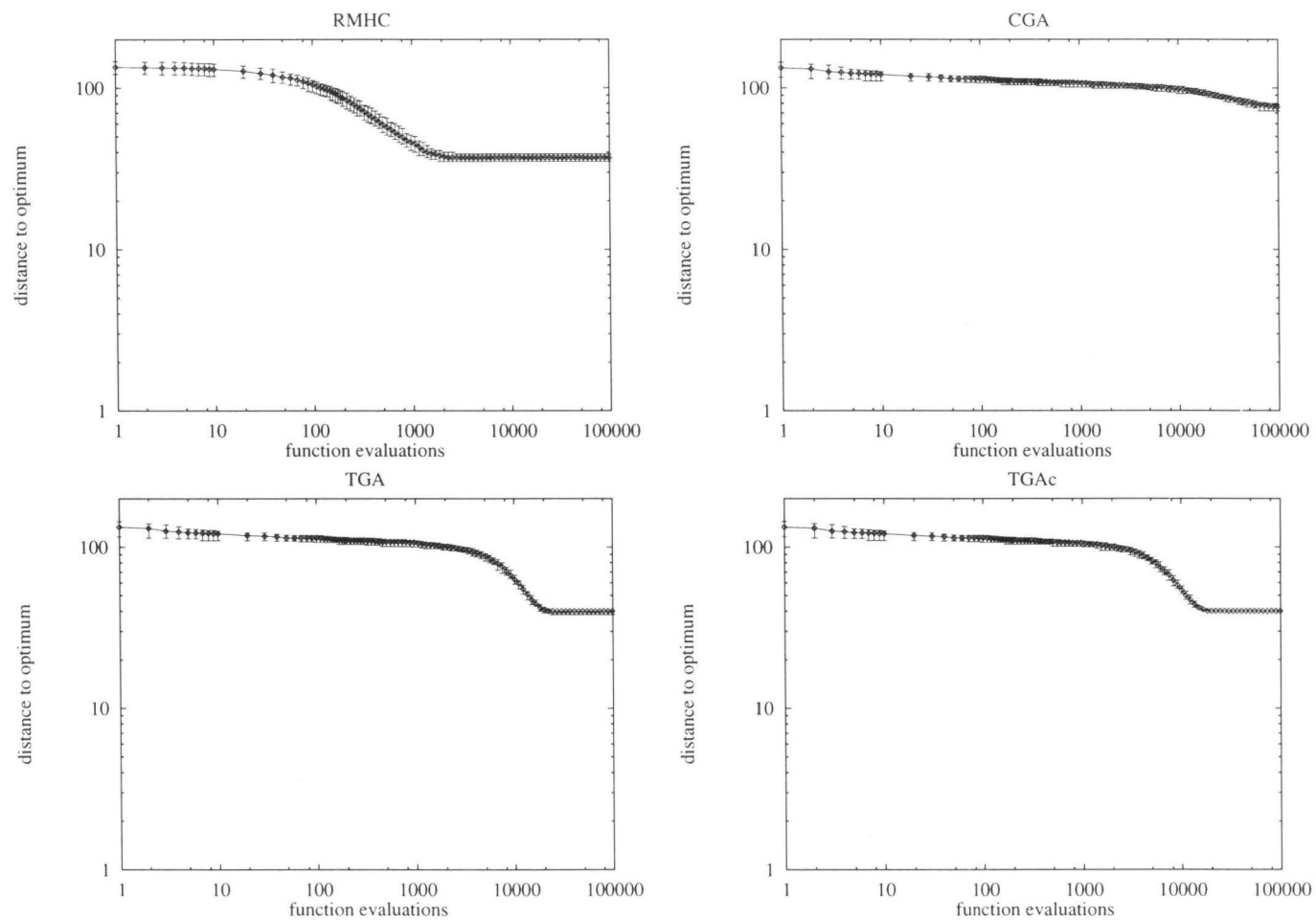


Figure 8.5: Results on BT3 (the deceptive trap function of order 5).

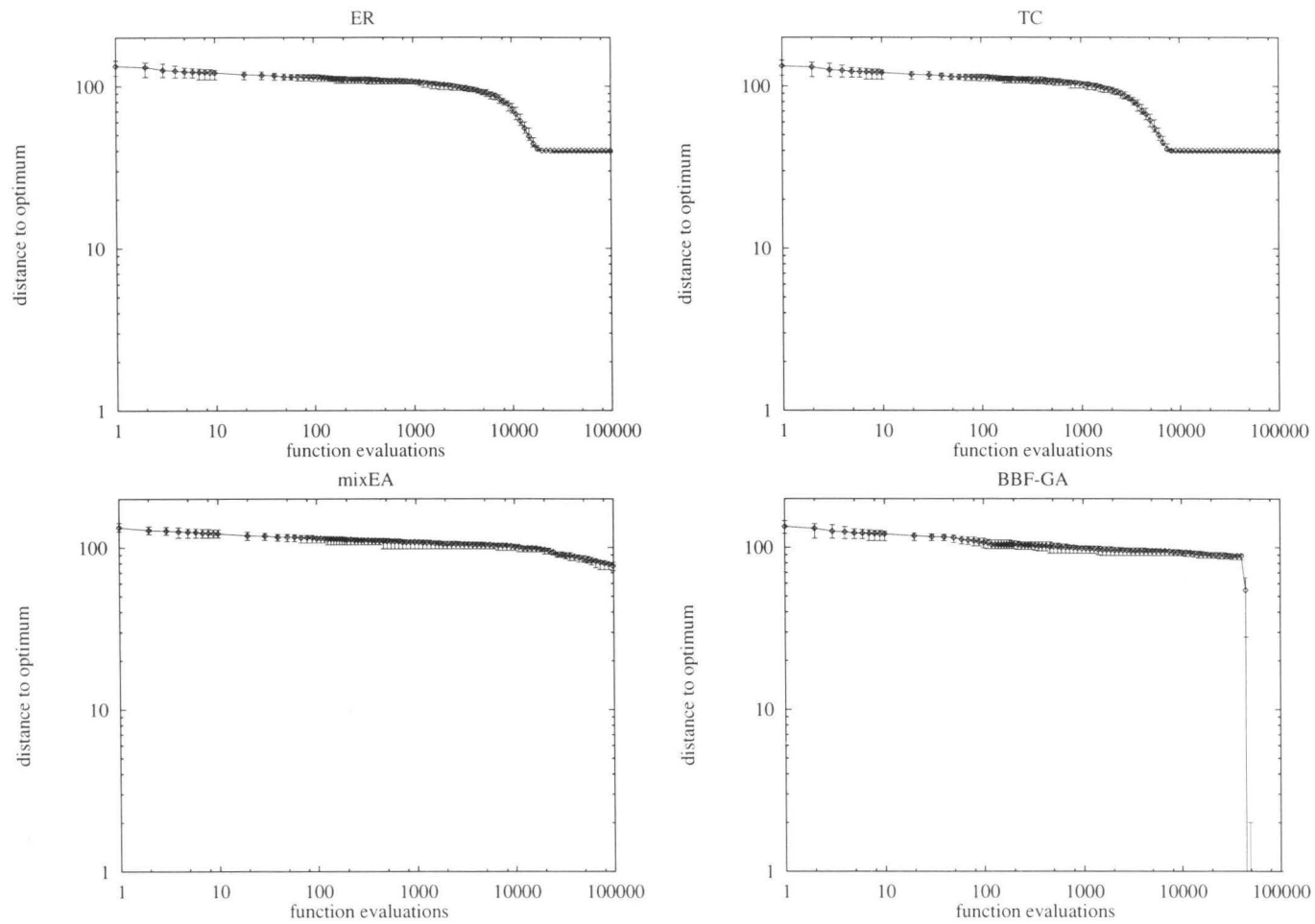


Figure 8.6: Results on BT3 (the deceptive trap function of order 5).



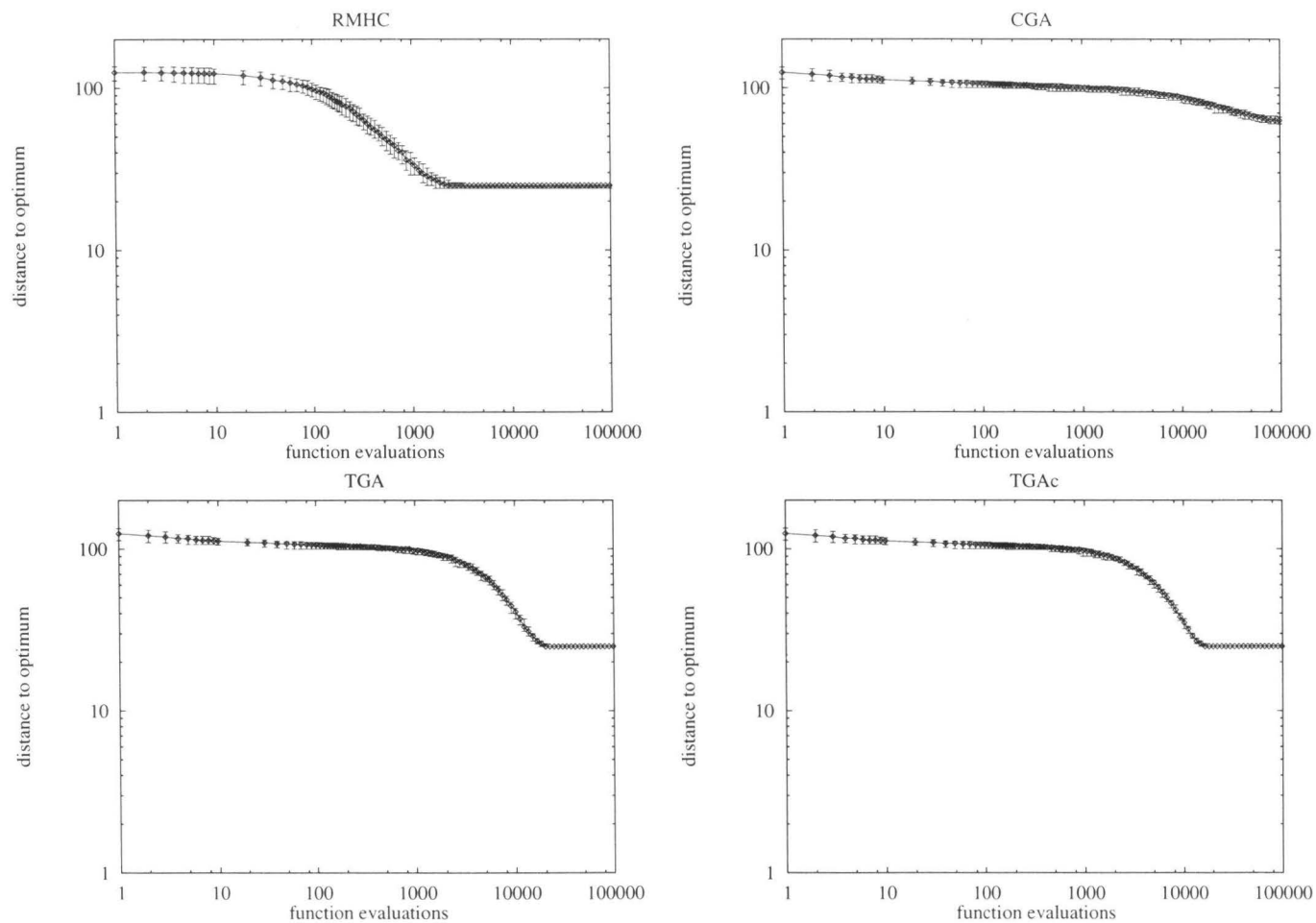


Figure 8.7: Results on BT4 (the deceptive trap function of order 8).

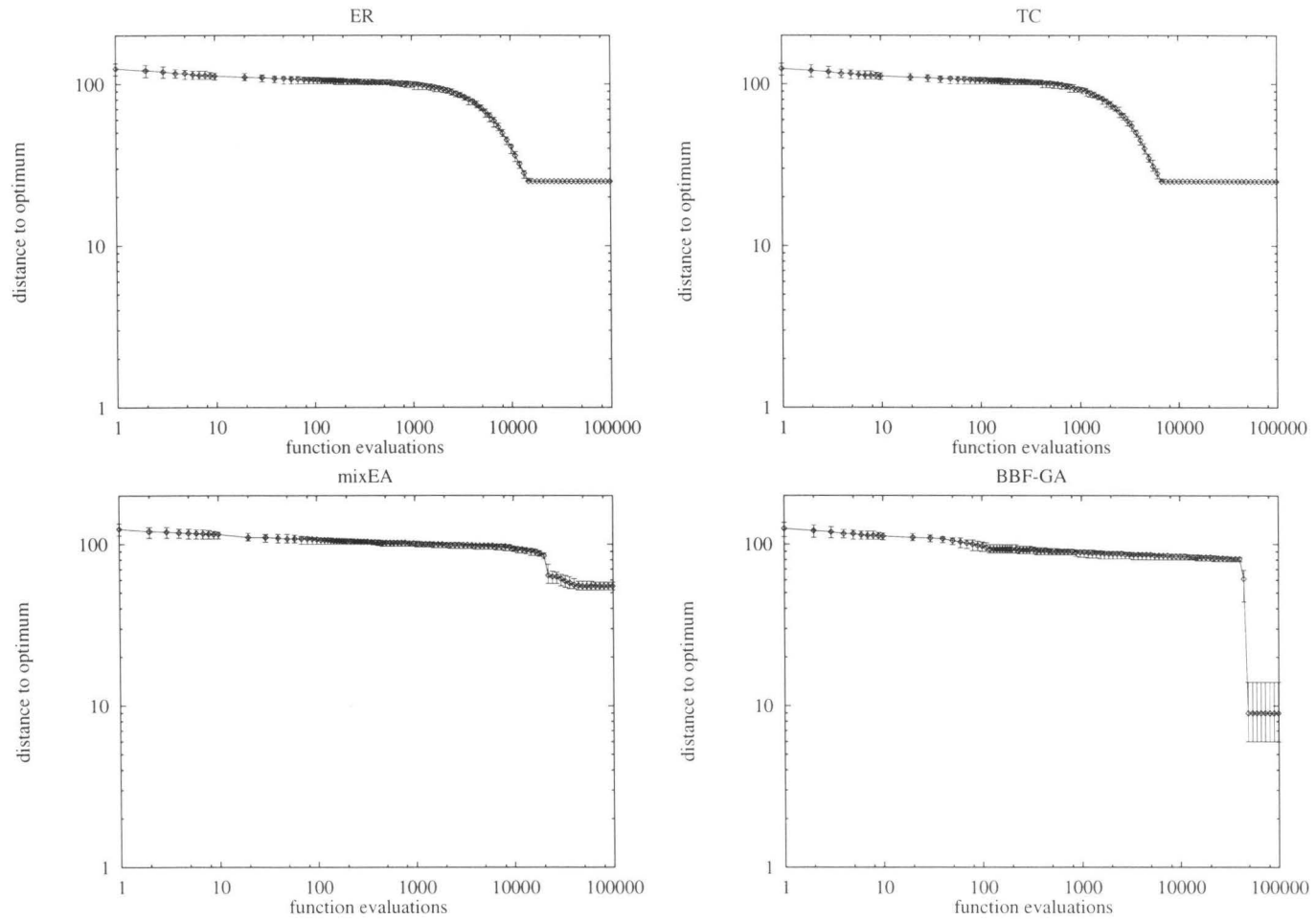


Figure 8.8: Results on BT4 (the deceptive trap function of order 8).

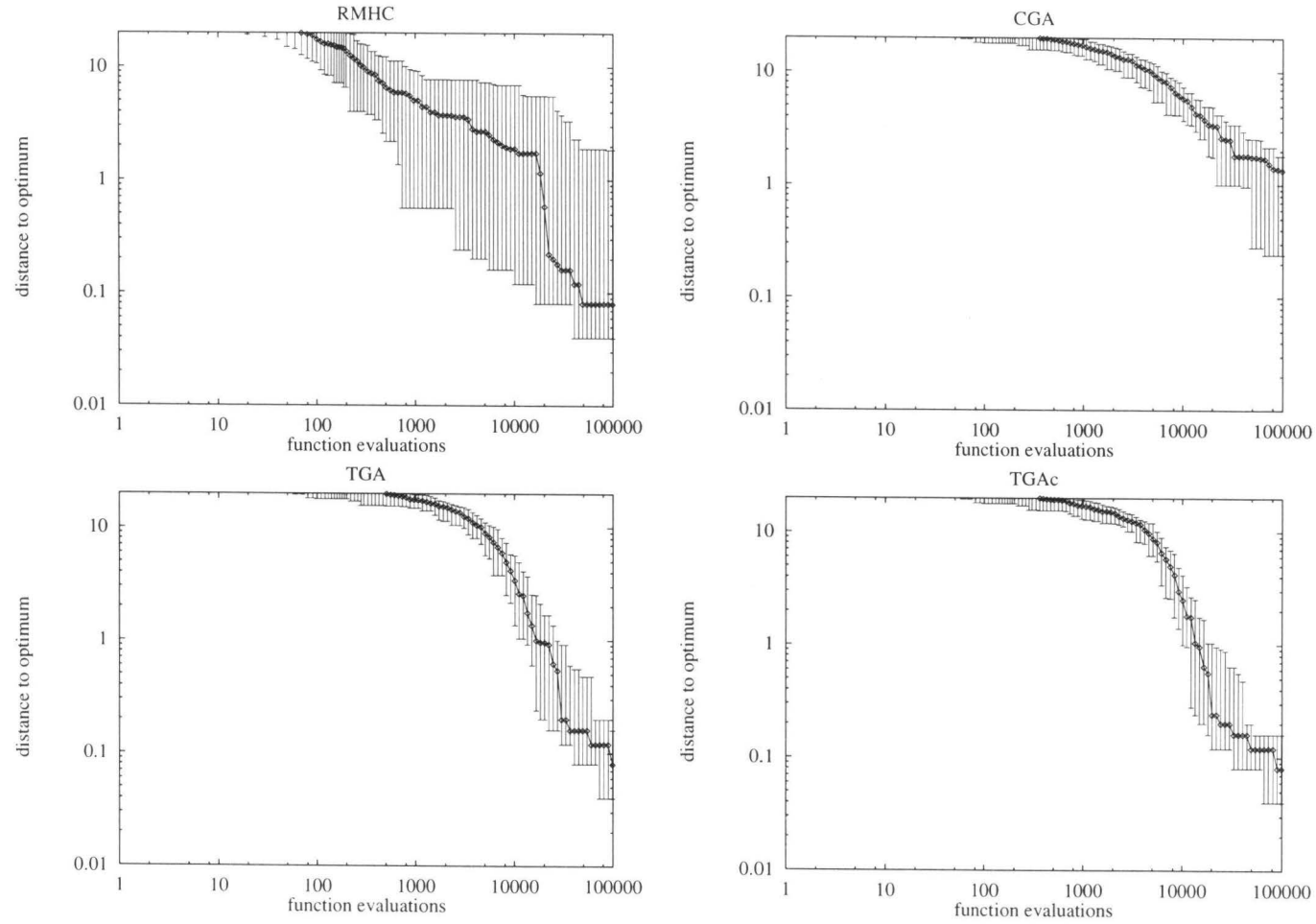


Figure 8.9: Results on BT5 (function with “frustration”).

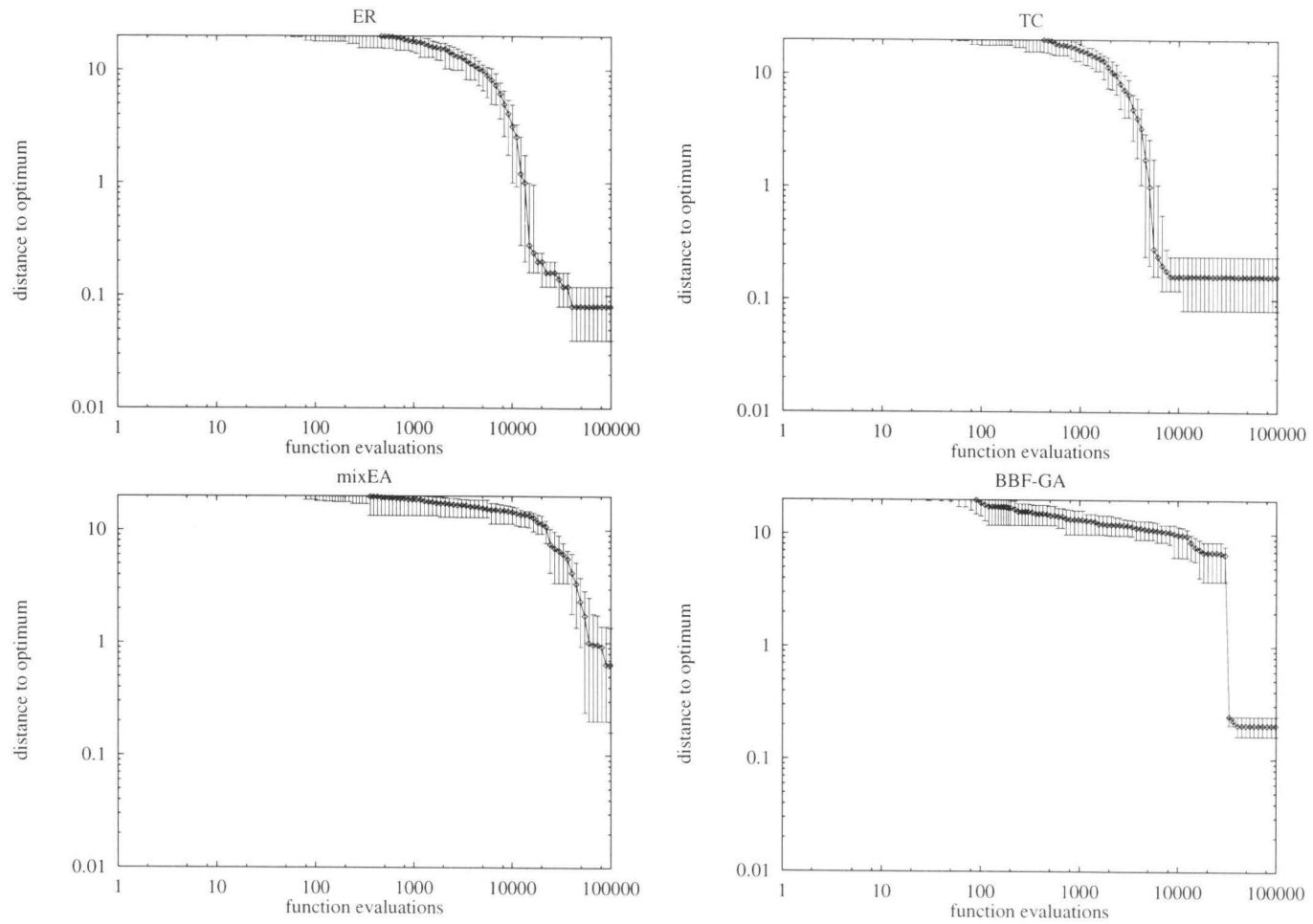


Figure 8.10: Results on BT5 (function with “frustration”).

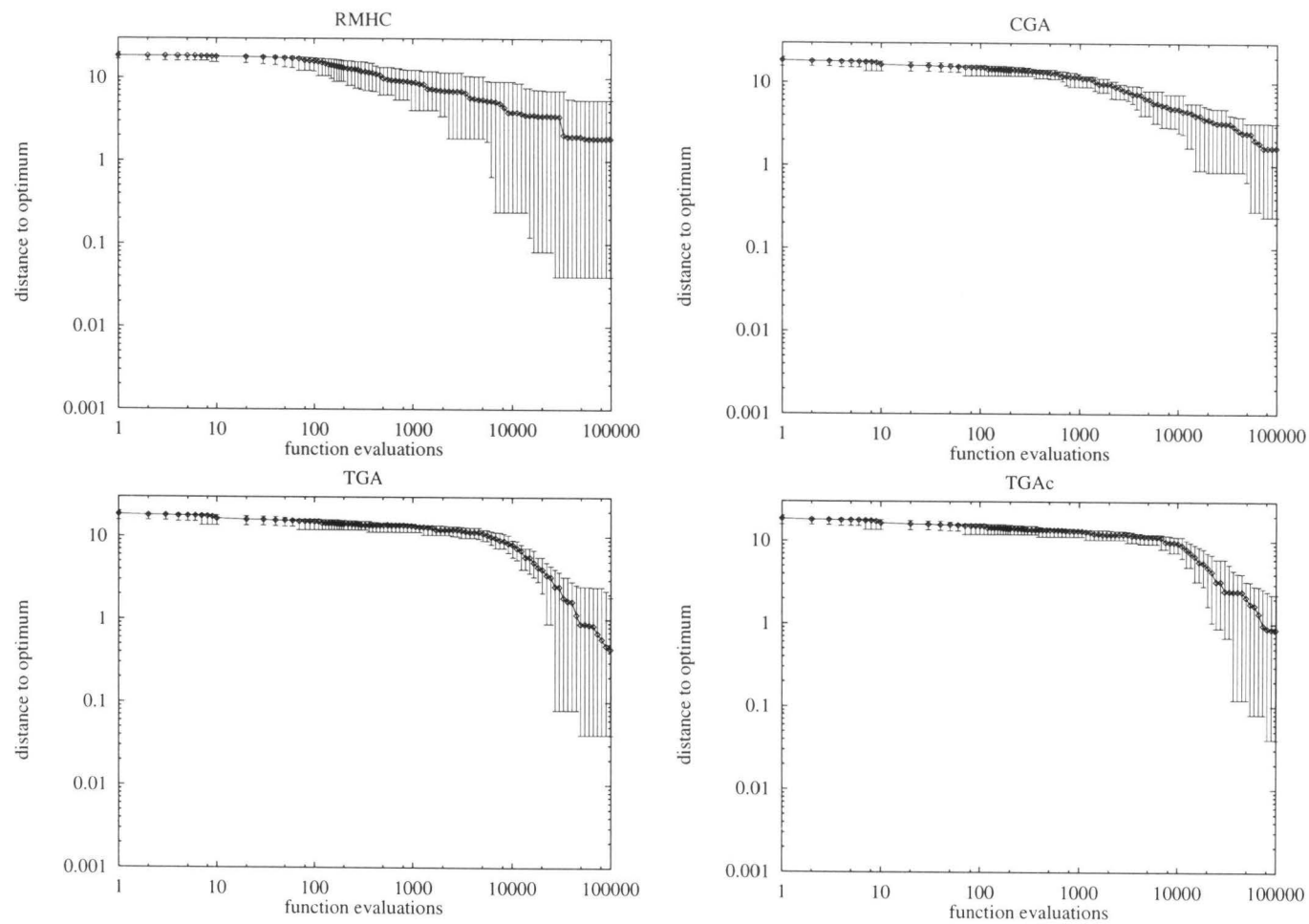


Figure 8.11: Results on BT6 (function with long blocks and frustration).

Figure 8.12: Results on the Fchm4

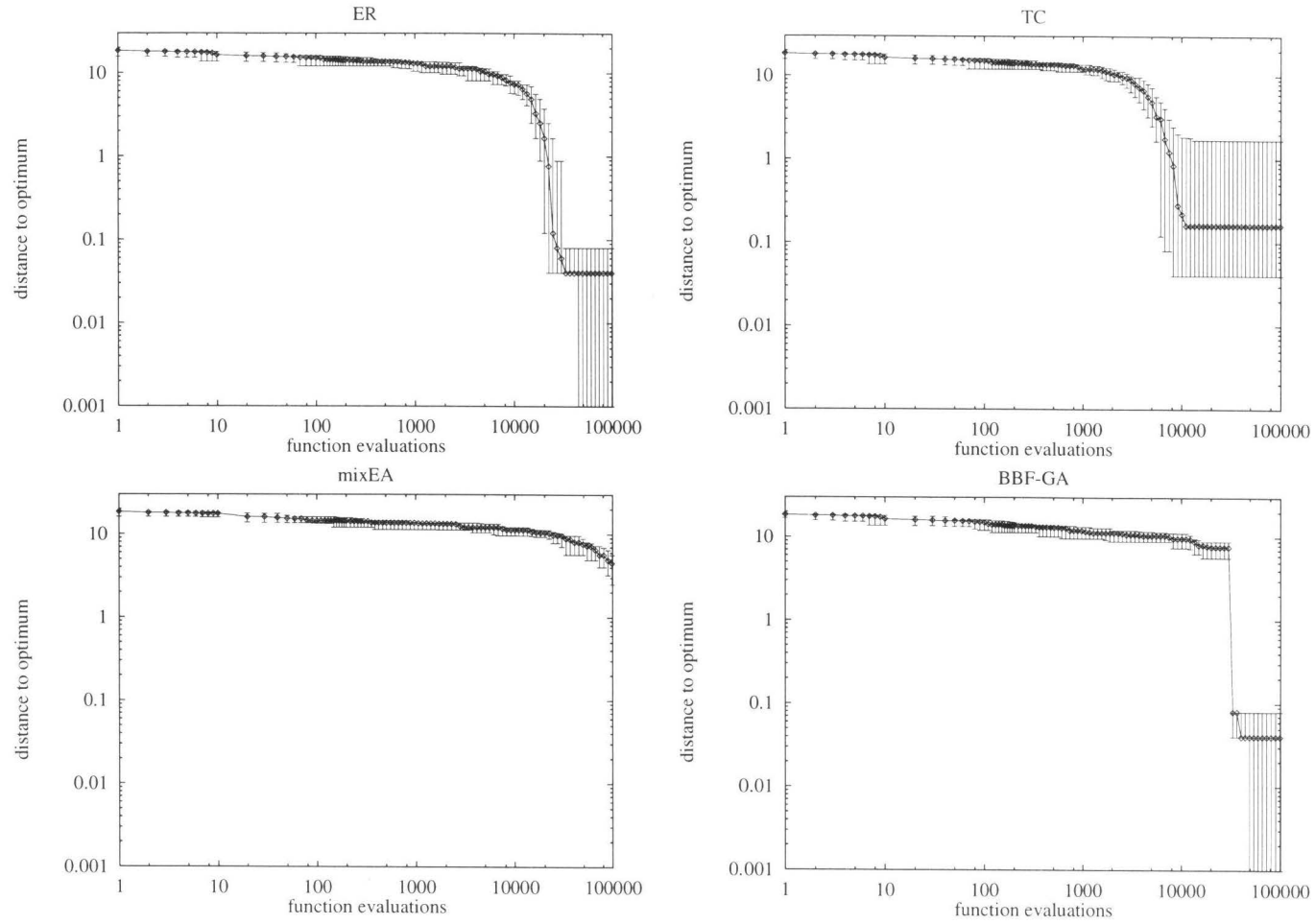


Figure 8.13: Results on BT6 (function with long blocks and frustration).

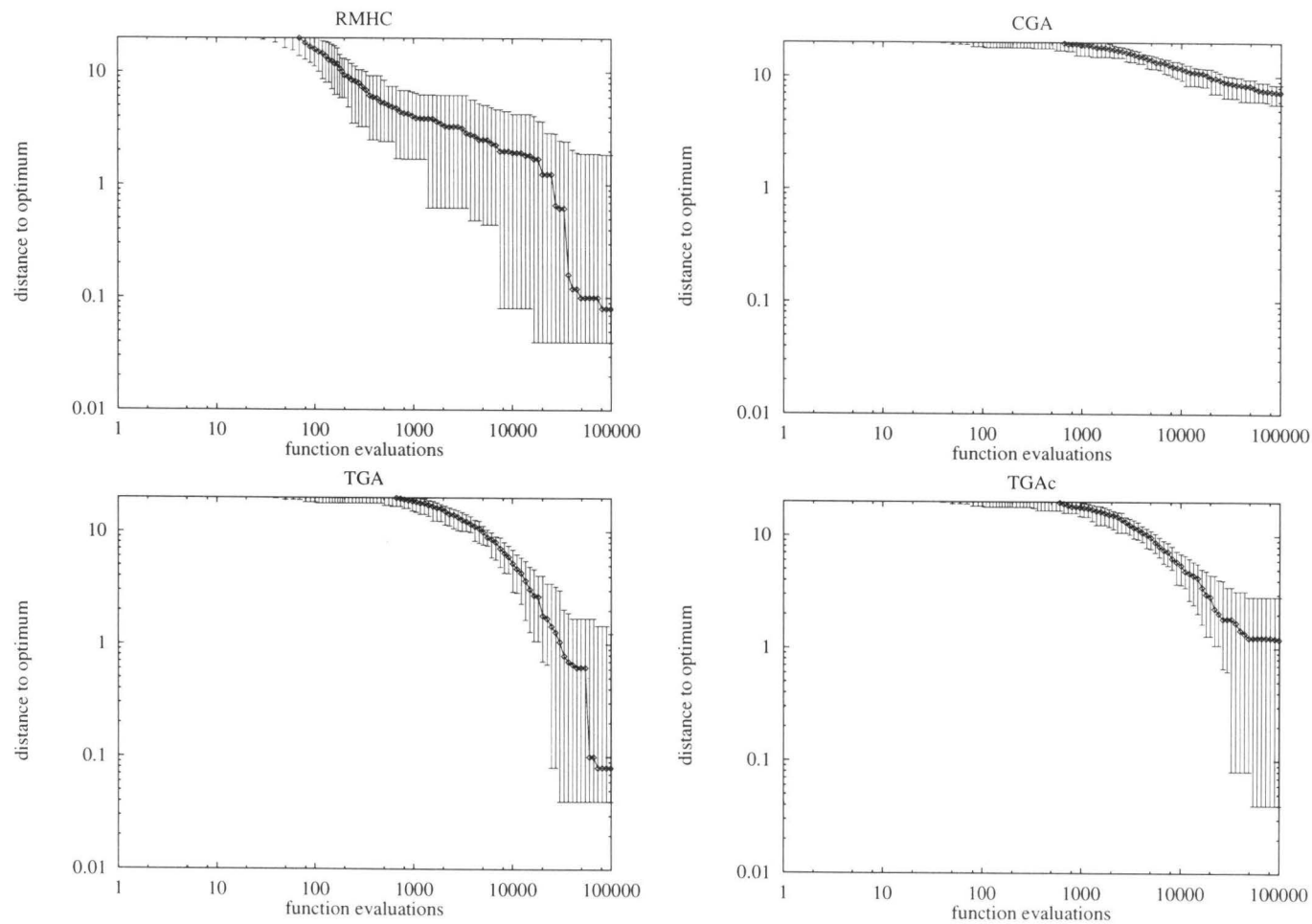


Figure 8.14: Results on BT7 (function with and frustration).

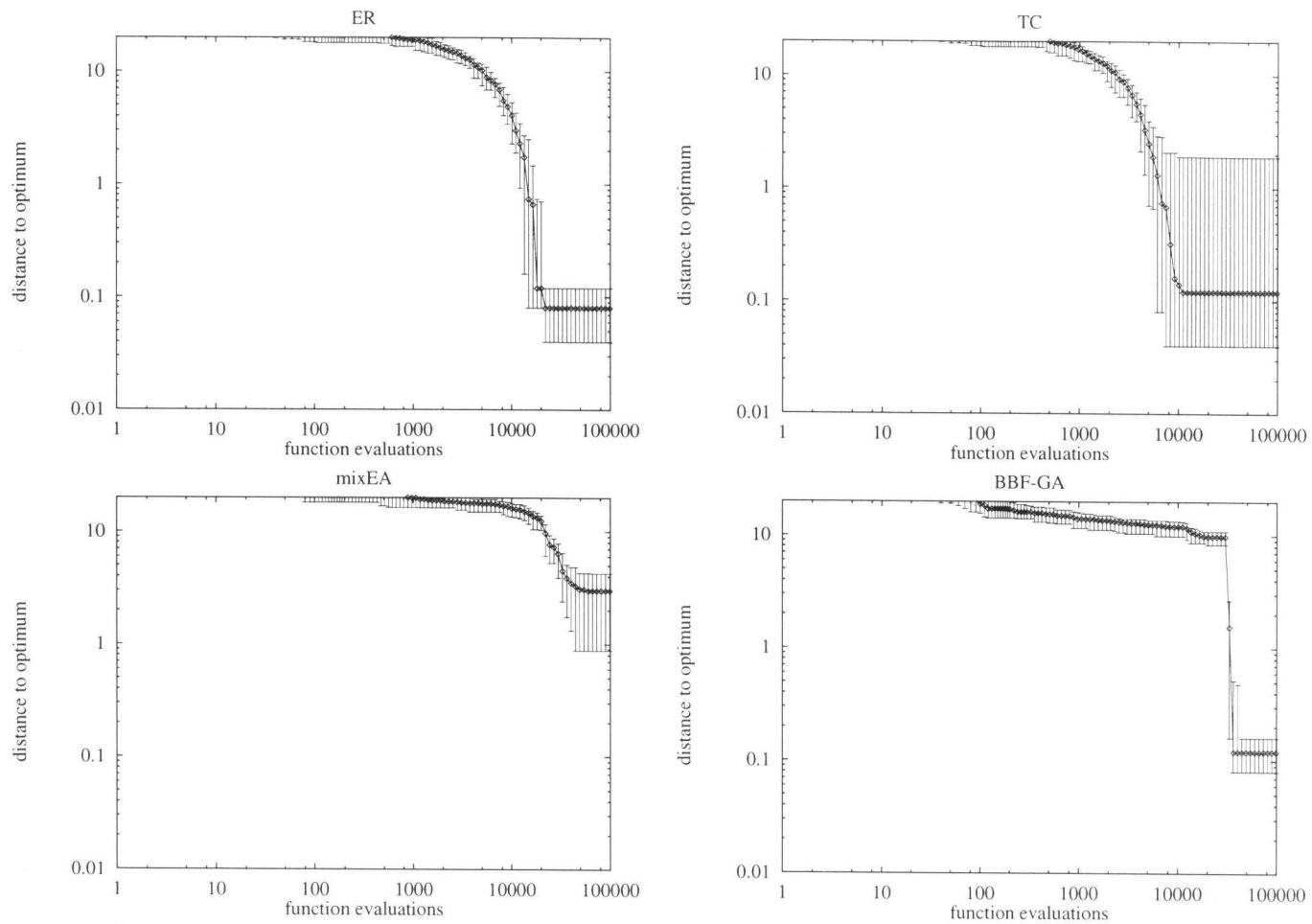


Figure 8.15: Results on BT7 (function with and frustration).



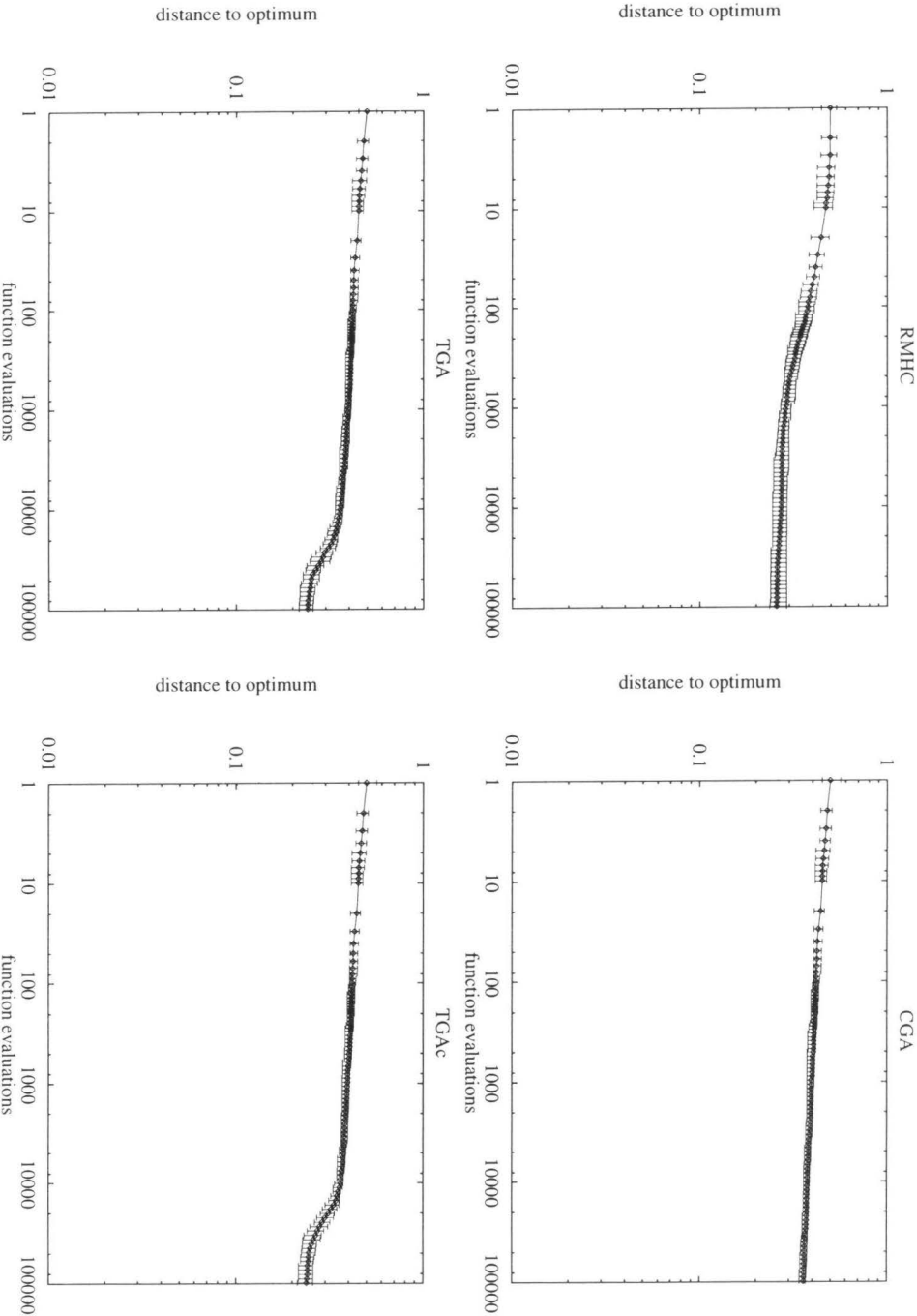


Figure 8.16: Results on BT8 (the NK-landscape).

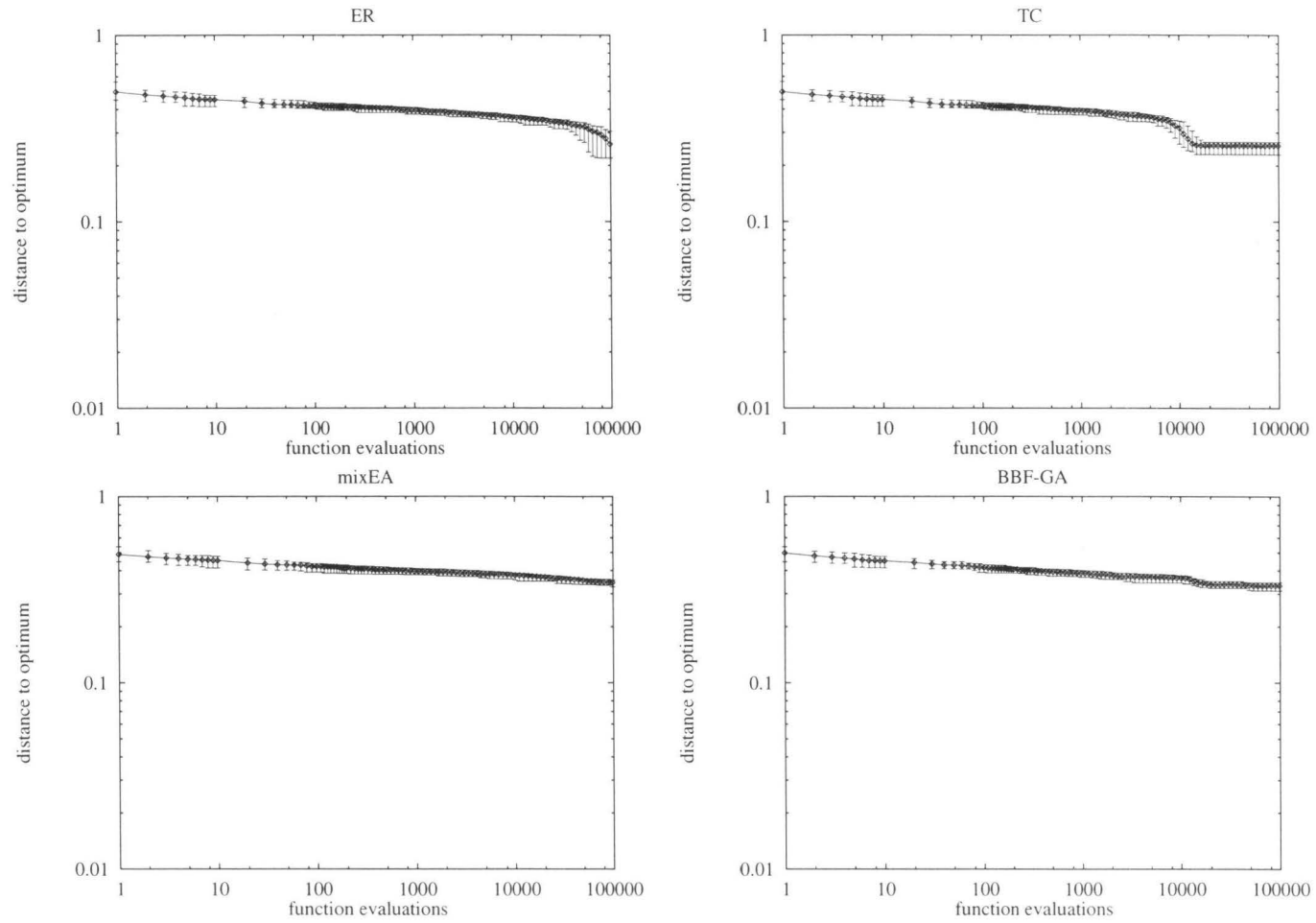


Figure 8.17: Results on BT8 (the NK-landscape).

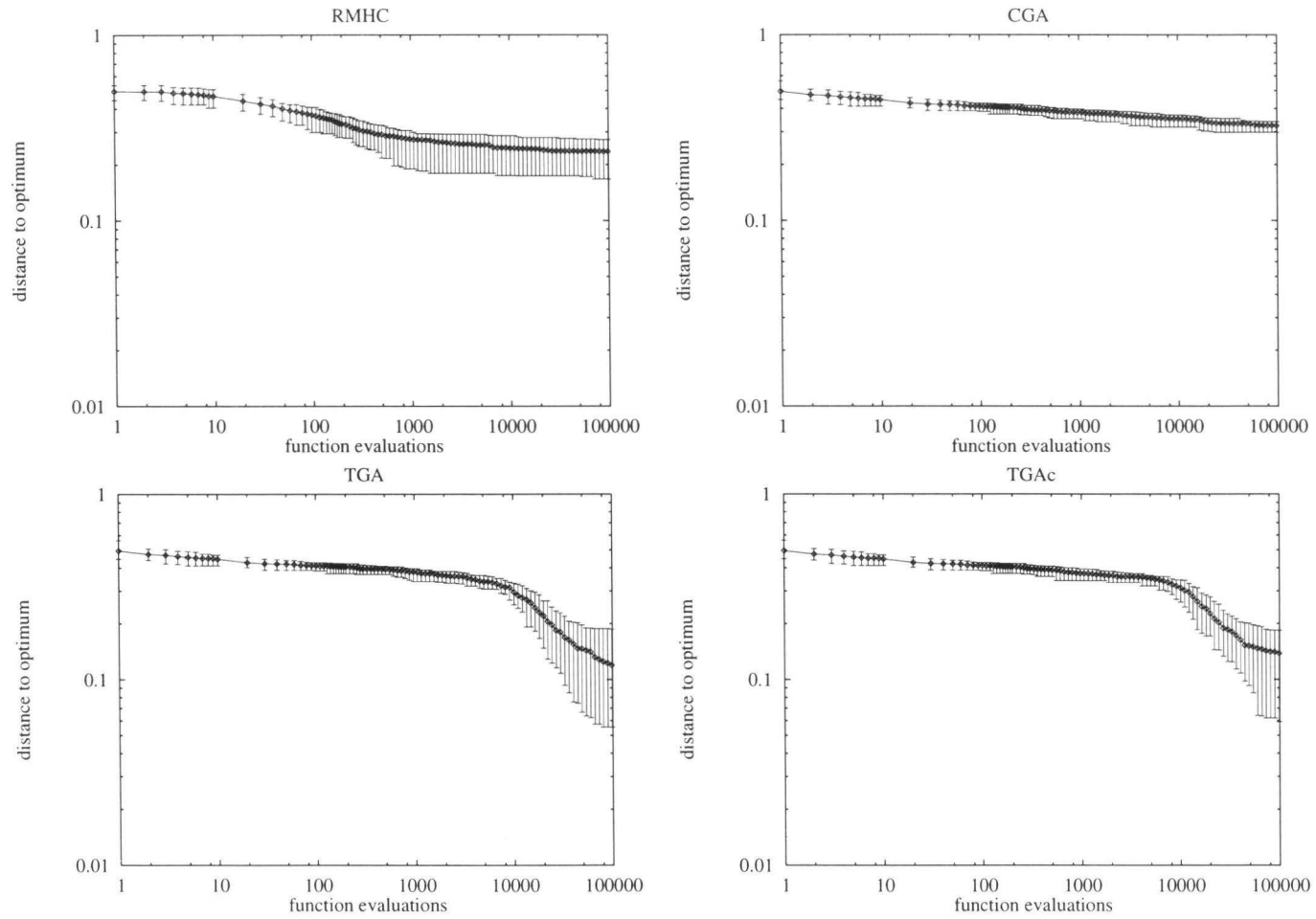


Figure 8.18: Results on BT9 (the NK-bb landscapes).

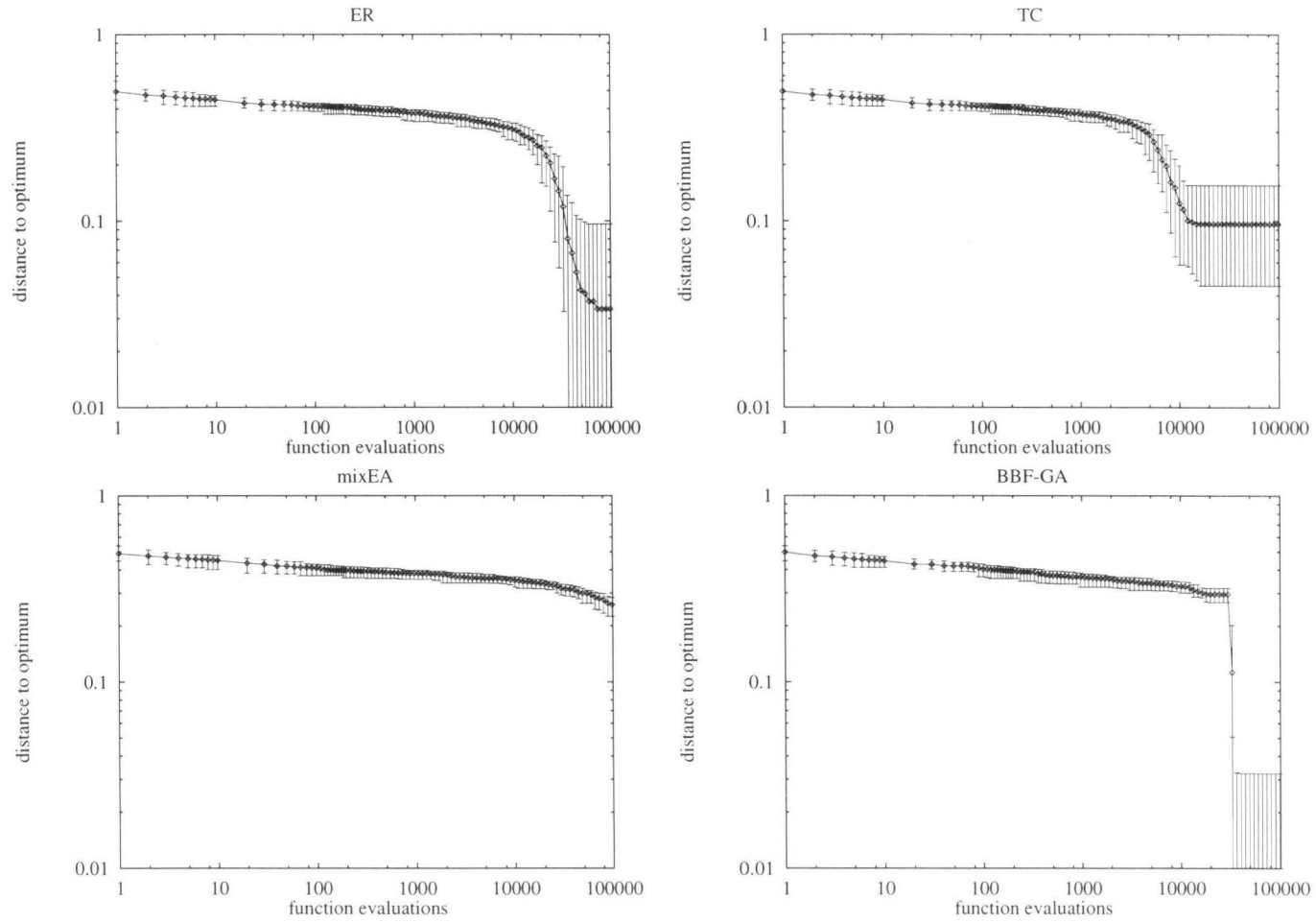


Figure 8.19: Results on BT9 (the NK-bb landscapes).

Figures 8.14 and 8.15 show the curves for problem BT5 (function with “frustration”). On this problem the TGA outperforms the TGAc. This function also uses the subfunction  $F_{cuban1}^5$ , so again the building block filtering step marked suboptimal building blocks for this subfunction.

Figures 8.16 and 8.17 show the curves for problem BT8 (the NK-landscape with random linkage). The optimal solution is not known for this problem. Therefore it is assumed that the optimal fitness is 1. All methods converge slowly in the direction of the optimum.

Figures 8.18 and 8.19 show the curves for problem BT9 (the NK-bb landscapes). Only the bbf-GA is likely to be able to locate the optimal solution.

|     | mixEA |
|-----|-------|
| BT1 | 1911  |
| BT2 | 2     |
| BT3 | 17    |
| BT4 | 3     |
| BT5 | 3     |
| BT6 | 75    |
| BT7 | 2     |
| BT8 | 161   |
| BT9 | 95    |

Table 8.15: Size final population.

Table 8.15 shows the size of the final population for the mixEA. The final population size of the other methods is 300, except for the RMHC that has population size 1, and bbf-GA that has population size 200 during the exploitation phase. Only in case of the mixEA the final population size is variable. On most problems the final population is quite small. In case of problem BT1 the mixEA has a large final population size. This is due to the fact that this problem does not have any structure except for the building blocks. As a result, the mixEA spends all its function evaluations during the building of the first evolutionary stack.

Table 8.16 shows the results for the building block filtering step that is used for exploration of the search space. The first column shows the proportion of masks that is accepted. The second and third column count the number of building blocks present in the set of 100 individuals obtained after the exploration step. Here we count the number of optimal building blocks of order-five that maximize a subfunction. The second column shows the number of optimal (order-five) building blocks that have been masked correctly. A building block is said to be masked correctly by the building block filtering step when all its bits are masked, and no other bits in the string are masked (not even if these bits also correspond to a building block). The maximal value for this column is 100. The third column shows the number of optimal building blocks present in the individuals after exploration. The fourth column shows the average fraction of bits of all building blocks that are masked.

|     | proportion<br>masks<br>accepted | number of<br>building<br>blocks<br>masked | number of<br>building<br>blocks<br>present | fraction<br>masked<br>bits |
|-----|---------------------------------|---|--|----------------------------|
| BT1 | 0.89                            | 81.50                                     | 109.50                                     | 0.83                       |
| BT2 | 0.07                            | 0.00                                      | 317.50                                     | 0.01                       |
| BT3 | 0.96                            | 29.00                                     | 326.50                                     | 0.48                       |
| BT4 | 0.23                            | 21.00                                     | 24.50                                      | 0.91                       |
| BT5 | 0.88                            | 0.00                                      | 265.50                                     | 0.54                       |
| BT6 | 0.86                            | 25.50                                     | 41.00                                      | 0.90                       |
| BT7 | 0.80                            | 1.00                                      | 245.50                                     | 0.33                       |
| BT8 | 0.18                            | —   | —  | —                          |
| BT9 | 0.57                            | 12.00                                     | 67.00                                      | 0.72                       |

Table 8.16: Performance of building block filtering.

When comparing the values of the second and the third column we see that the building block filtering step is likely to fail in case that the average number of building blocks per individual is relatively high. In that case the filtering procedure is not effective, because it is likely to mask parts of different building blocks. Therefore the structure of the underlying problem-space is not detected correctly. Problems where the building blocks are found easily can be solved with the traditional GA's without a linkage-learning procedure like the building block filter. The building block filtering proves its power on problems with deceptive building blocks, and problems with high-order building blocks. In these cases only a small number of building blocks is detected, and these are masked correctly.

Some of these test-functions are also used by other researchers. Here we will briefly discuss their results. Mühlenbein uses the FDA, an algorithm that is based on factorizing a probability distribution into marginal distributions. He applies this algorithm to a test-suite that includes BT2, BT5, and BT7. A major difference is that he assumes the linkage to be known, and he explicitly uses this linkage information to decompose the problem in a set of subproblems. For example in case of BT7 the complete search space contains  $2^{69}$  strings. When using the information of the decomposition all subfunctions can be computed using only 544 ( $= 17 \cdot 2^5$ ) function evaluations. Composing the optimal solution when having this information is easy. The FDA solves this problem in 6800 function evaluations. On the other problems the FDA also converges faster than the methods presented here, but it is important to note that the effective search space is orders of magnitude larger without the linkage-information.

Problems with unknown linkage are used to test the performance (significantly modified version) of GEMGA [BKW98]. The original GEMGA was developed by Kargupta [Kar96a]. This algorithm measures the marginal fitness contributions of bits, and uses this information to discover the linkage information. The problems BT2 and BT3 are used among others to assess the performance of the GEMGA. The optimum is located in 88,000 func-

tion evaluations in for BT2, and 55,000 function evaluations for BT3. GEMGA was not tested on functions containing higher-order building blocks or having conflicting building blocks, and hence we can not make a further comparison.

### 8.3.1 bbf-GA

The bbf-GA performs well on problems involving building blocks. For problems containing building blocks that are difficult to find, the building block filtering step performs well. On the current test-suite it often masks the building block correctly. On such problems the bbf-GA performs well. For problems where the building blocks are easy to find, the building block filter is likely to mark parts of different building blocks. Because masks are likely to correspond to parts of building blocks, the exploitation phase of the bbf-GA is not able to mix these building blocks effectively. The bbf-GA seems to extend the range of applicability of GA's because some problems involving building blocks that are difficult to process by means of traditional GA's can be handled with the bbf-GA.

### 8.3.2 Mixing EA

The mixEA converges slowly. It is able to find a large number of optimal building blocks, but it often fails to combine and thus to find the global optimum on the current problem. One of the reasons for these problems is the fact that the mixEA really works hard to prevent any duplication of building blocks. The goal of the mixEA is to find all building blocks needed to find the optimum. In order to make the probability to find these building blocks as large as possible, duplication of individuals is prevented. Unfortunately, this also makes the mixing of building blocks more difficult. In the later stages of evolution large parts of each of the individuals have to be retained during recombination in order to produce superior offspring. Retaining large parts of individuals is quite difficult because most individuals differ in many loci from each other. Therefore the probability of a successful recombination step becomes low. For example, suppose that we have a problem involving three compatible deceptive building blocks of order  $k$ , labelled  $A$ ,  $B$ , and  $C$ . In that case it is easier to first create the individuals  $AB$  and  $AC$  (where  $AB$  represents the individual containing building blocks  $A$  and  $B$ ); Given individual  $AB$  and  $AC$  the optimal individual  $ABC$  is created with probability  $2^{-2k}$  by uniform crossover. The mixEA is more likely to end up in a state where for example  $AB$  and  $C$  are the only available individuals. For this pair the probability of a successful recombination is only  $2^{-3k}$ .

The mixEA performs well on finding high-order building blocks, and collecting these in a small population. However, it fails to mix these building blocks. Hybrid methods that use the mixEA for exploration, and next apply an elitist GA on the final population of the mixEA might provide possibilities to solve these problems.

## 8.4 Summary

The purpose of this chapter was to compare the introduced evolutionary algorithms on a large test-suite. The results show that the building block filtering step (contained in the bbf-GA) is very effective in masking building block when an individual contains only a single building block. Therefore the bbf-GA is able to outperform traditional GA's on problem for which building blocks are relatively difficult to find and process. The mixEA is also good at finding and preserving building blocks, but it has problems to mix these building blocks. The elitist recombination performs reliable on most problems, but does not handle problems involving high-order deception very well. The triple-competition converges fast and in many cases finds near-optimal solutions rapidly.





## Chapter 9

# Cluster Evolution Strategies

Many randomized search methods can be regarded as random sampling methods with a (non-uniform) sampling density function. Differences between the methods are then reflected in different shapes of the sampling density function and in different adaptation mechanisms that update this density function based on the observed sample-points. A description of such algorithms in terms of sampling density functions shows similarities to the transmission function models that have been used in sections 4 and 5. An evolutionary algorithm is proposed, that uses an enhanced selection mechanism. It uses not only fitness values of the individuals but also considers the distribution of individuals over the search-space. After a fitness-based selection, the individuals are clustered, and a representative is selected for each cluster. The next generation is created using only these representatives. Usually the set of representatives is relatively small, and the efficient incorporation of local search techniques is possible.

The outline of this chapter is as follows. In section 9.1 a number of optimization methods are described as biased random sampling methods. Section 9.2 discusses some of the problems that evolutionary optimizers can encounter when optimizing a real-valued function. In section 9.3 the cluster evolution strategy (CLES) is introduced, and it is discussed how this method can avoid some of the problems mentioned in section 9.2. Section 9.4 presents the details of the clustering method used in CLES, section 9.5 describes the evolutionary operators, and section 9.6 presents the local search methods that are incorporated. A test-suite and performance-measures are presented in section 9.7, followed by the results in section 9.8. This chapter is concluded by a summary in section 9.9.

### 9.1 Randomized sampling methods

Evolutionary algorithms belong to the large class of randomized search methods, containing among others, Monte-Carlo methods, simulated annealing [AK89, KGV83], clustering methods with random sampling [TŽ89], evolutionary programming [BS93, Fog94], and evolution strategies [BS93, Rec73, Sch95]. Randomized sampling methods can be modelled by means of a sampling density function over the search space  $\mathcal{S}$ . In case of a discrete search

space such a density function assigns a value to each point in the search space. This value denotes the probability that the point is obtained during the next step that the algorithm makes. All these probabilities have to sum to one, so if all elements of the search space are enumerated, then

$$\sum_{i=1}^{|\mathcal{S}|} d_i = 1,$$

where  $d_i$  is the probability assigned to element  $i$  of search space  $\mathcal{S}$ . Modelling by means of sampling density is also possible for real-valued spaces, in which case the sum is replaced by an integral over the complete search space.

The simplest random sampling method is the unbiased random search. This method generates a random point in the search space, computes the fitness of this point, and keeps track of the best point observed. The sampling density function of this method is a uniform density function over the complete search space. This density function is not changed. If one can expect that the search space has a certain amount of smoothness, then faster randomized sampling methods are the biased random searches, such as a hill-climber with a Gaussian distributed kernel. Such a method uses a Gaussian distributed sampling density function that is centred around the current best individual, so the sampling density function changes each time that a better individual is discovered. A slightly more complex method is the  $(1+1)$ -ES. This method is similar to the previous method, except for the fact that the  $\sigma$ -parameter, which determines the width of the Gaussian, is a self-adaptive parameter. The adaptation of the parameters is done by means of an evolution process. A more detailed description of the  $(1+1)$ -ES will be given in section 9.6.2. In case of simulated annealing the sampling density function is determined by the current best point, the neighbourhood structure and a temperature based cooling schedule. A more detailed description of this algorithm can be obtained from Aarts et al. [AK89]. In case of a generational evolutionary algorithm the sampling density function can be determined as follows. Given a  $n$ -ary evolutionary operator and a set of  $n$  parent individuals, one computes the probability that certain offspring is obtained. This information defines a sampling density function corresponding to an evolutionary operator with a fixed set of inputs. Now given the current population one can take all possible sets of  $n$  parents, and the corresponding probabilities of these sets. The sampling density function of the evolutionary algorithm is obtained by a weighted superposition of the sampling density function of the  $n$ -ary evolutionary operator over these parent-sets, where the weights are the probability that the set of parents is obtained. The sampling density function of the evolutionary algorithm remains constant during a single generation. It is the selection scheme that modifies the sampling density function by generating a new parent population, and thereby changing the probabilities of obtaining the different sets of parent individuals. So, in a generational genetic algorithm the update of the sampling density function is obtained when one moves to the next generation, and the offspring of the current generation are considered as parents. It is quite straightforward to generalize this model for the case that more than one evolutionary operator is used.

Most randomized sampling methods try to improve the unbiased random search by focusing the search process on a subset of the search space. The robustness of such algo-

space such a density function assigns a value to each point in the search space. This value denotes the probability that the point is obtained during the next step that the algorithm makes. All these probabilities have to sum to one, so if all elements of the search space are enumerated, then

$$\sum_{i=1}^{|\mathcal{S}|} d_i = 1,$$

where  $d_i$  is the probability assigned to element  $i$  of search space  $\mathcal{S}$ . Modelling by means of sampling density is also possible for real-valued spaces, in which case the sum is replaced by an integral over the complete search space.

The simplest random sampling method is the unbiased random search. This method generates a random point in the search space, computes the fitness of this point, and keeps track of the best point observed. The sampling density function of this method is a uniform density function over the complete search space. This density function is not changed. If one can expect that the search space has a certain amount of smoothness, then faster randomized sampling methods are the biased random searches, such as a hill-climber with a Gaussian distributed kernel. Such a method uses a Gaussian distributed sampling density function that is centred around the current best individual, so the sampling density function changes each time that a better individual is discovered. A slightly more complex method is the  $(1+1)$ -ES. This method is similar to the previous method, except for the fact that the  $\sigma$ -parameter, which determines the width of the Gaussian, is a self-adaptive parameter. The adaptation of the parameters is done by means of an evolution process. A more detailed description of the  $(1+1)$ -ES will be given in section 9.6.2. In case of simulated annealing the sampling density function is determined by the current best point, the neighbourhood structure and a temperature based cooling schedule. A more detailed description of this algorithm can be obtained from Aarts et al. [AK89]. In case of a generational evolutionary algorithm the sampling density function can be determined as follows. Given a  $n$ -ary evolutionary operator and a set of  $n$  parent individuals, one computes the probability that certain offspring is obtained. This information defines a sampling density function corresponding to an evolutionary operator with a fixed set of inputs. Now given the current population one can take all possible sets of  $n$  parents, and the corresponding probabilities of these sets. The sampling density function of the evolutionary algorithm is obtained by a weighted superposition of the sampling density function of the  $n$ -ary evolutionary operator over these parent-sets, where the weights are the probability that the set of parents is obtained. The sampling density function of the evolutionary algorithm remains constant during a single generation. It is the selection scheme that modifies the sampling density function by generating a new parent population, and thereby changing the probabilities of obtaining the different sets of parent individuals. So, in a generational genetic algorithm the update of the sampling density function is obtained when one moves to the next generation, and the offspring of the current generation are considered as parents. It is quite straightforward to generalize this model for the case that more than one evolutionary operator is used.

Most randomized sampling methods try to improve the unbiased random search by focusing the search process on a subset of the search space. The robustness of such algo-

rithms is related to the probability that the global optimum is located in such a subset. If it is not, then one can only hope to find a suboptimal solution of reasonable quality. There are at least two approaches to circumvent this problem. The first approach is to avoid the exclusion of certain parts of the search space. Simulated annealing takes this approach. Instead of restricting the search to a subset, the algorithm modifies the sampling density function. As the search proceeds this density function gets increasingly peaked around the current best solution. Even though it is never impossible to find the location of the global optimum during the next iteration of the simulated annealing algorithm, the probability that this will happen can become arbitrarily small. Hence, the proof of convergence does not really provide us with a practical guarantee that there is convergence within a finite amount of time. The second approach is to control the subset selection process by requiring a certain amount of evidence before a restriction of the search-space is permitted. This second approach is implicitly used in many population-based evolutionary algorithms. Here the selection mechanism focuses the search on a subset of the search-space by controlling the probability of the different sets of parent individuals.

The main difference between the two approaches is that the first approach samples over the search space according to a density function that is non-zero everywhere, while the second approach does allow the density function to become zero at some points. For methods that embrace the first approach, there are often proofs that these methods eventually will find the global optimum. Sometimes one can even prove convergence to the optimum with probability one when allowing for an infinite amount of computation. Note that this does not tell anything about the speed of convergence, the probability that the optimum is found within a certain number of steps, or about the probability that these methods outperform unbiased random search.

Each method that uses the second approach can easily be modified such that it corresponds to the first approach too. To do so, one only has to generate a complete random individual once in a while, and check whether this individual outperforms the current best individual. Because all points in the search space can be generated, the sampling density function is non-zero everywhere. Given a sufficient amount of steps the optimum is likely to be found. Even though such an approach might be of theoretical interest, it does not make much sense from a practical point of view. On those problems where the randomly generated points are accepted, the original method apparently failed, and the modified method basically corresponds to a random search with a large and complicated overhead. In case of evolutionary algorithms it has been shown that the optimum is always found provided that the mutation operator is disruptive enough. It would be more interesting to compare the two approaches when only a limited amount of computation is allowed. However, such questions might only be answered when one restricts the class of problems.

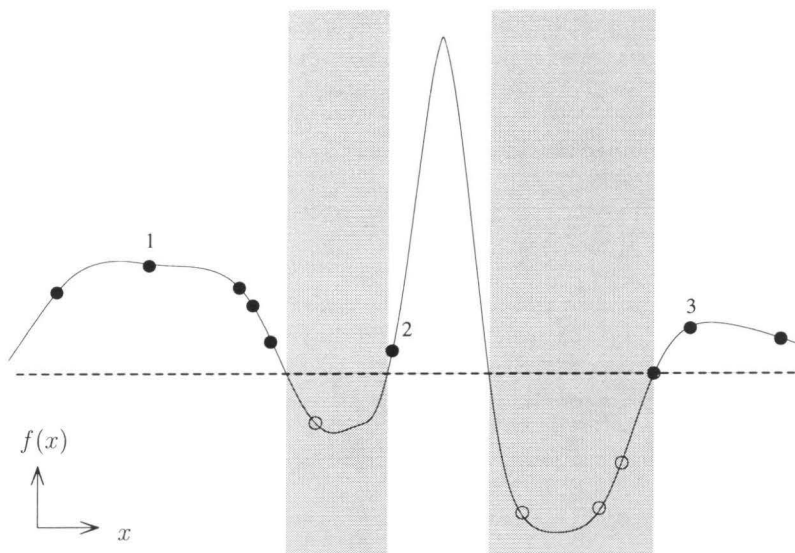


Figure 9.1: Example of the distribution of a population over a search-space.

## 9.2 Evolutionary numerical optimization

Some evolutionary algorithms are known to have problems with general function optimization tasks [DeJ93, Bäck96]. A possible reason for these difficulties is that most evolutionary algorithms have a strong bias towards large regions with a high overall fitness [MV95]. When the global optimum is present within this region, this behaviour does not cause a problem, however if the region of attraction of the global optimum does not coincide with this region, then the evolutionary algorithm is likely to converge to a suboptimal solution. At first sight this might seem to be a problem that is inherent to evolutionary algorithms. The fast convergence to regions having a high average fitness is one of the important reasons why an evolutionary algorithm is able to outperform an unbiased random search, in many cases. As we shall try to show, a careful design of an evolutionary algorithm can circumvent this type of problem, resulting in a more reliable function optimizer. In a given evolutionary algorithm the sampling density function is mainly determined by the distribution of the population over the search-space and the definition of the genetic operators. Large regions having a high average fitness are likely to be discovered early, and many individuals in the population belong to such regions. The large number of individuals in such a region can result in a high probability of introducing new individuals in the same region. A simple way to lower the probability of such over-sampling is to use disruptive evolutionary operators, and simultaneously to assign a limited lifetime to individuals. The disruptive operators prevent that many (almost identical) copies of an individual are gen-

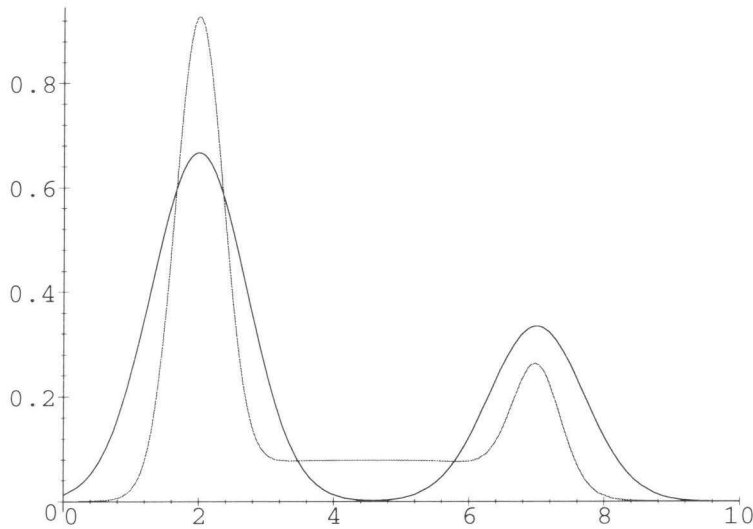


Figure 9.2: Density of parents and offspring when applying intermediate crossover

erated; However, even when the operators are disruptive, an individual that survives for many generations can generate quite a lot of copies. A limited lifetime can prevent such duplication. Limiting the lifetime of individuals has to be done with care. The population size should be chosen large enough to allow well-performing individuals to propagate their information before its lifetime expires. The information of individuals residing in a region with a low density (for example a small region corresponding to a narrow peak) is more likely to get lost than information contained in individuals residing in a region with a high density (for example a broad peak with relatively high average fitness).

A simple example of a search-space is shown in Figure 9.1. The circles and bullets show the locations of the individuals of the current population in the search space. Along the horizontal direction we have the value of the individuals, while the vertical position gives a measure for the fitness of the individual. The global optimum is located in a narrow peak. The individual that is closest to the optimum, is not the best performing individual. If a high selective pressure is applied, then this individual (that is close to the global optimum) is likely to be thrown out of the population. However, many individuals in the broad suboptimal peaks will be retained. Application of recombination can even make the situation worse, as we will show next. It was already observed in section 3.3 that the recombination can easily amplify an unbalanced distribution of the parent population among different parts of the search space. Here, we will show that a similar effect is likely to be present for recombination in real-valued space. Figure 9.2 shows an artificial example of the evolution of the density function for a one-dimensional numerical optimization problem. The black line represents density function of parents in the search-space. Apparently the

evolutionary algorithm has discovered two regions of interest, centred around the values 2 and 7. The left-hand region contains  $2/3$  of the sample-points, while the right-hand region only attracted  $1/3$  of the sample-points. The sample-points belonging to a single region are normally distributed. The grey line represents the density function of offspring when using uniform selection of parents and an intermediate recombination operator [BS93]. The intermediate recombination generates an individual uniform at random between the two parent individuals. Now, three cases can be distinguished, depending on what regions the parents belong to. The first case is that both parents are from the left-hand region, which has probability  $4/9$ . In this case the offspring is given by a Gaussian distribution, which is more narrow than the original distribution in the left-hand region. The second case corresponds to both parents being taken from the right-hand region, which happens with probability  $1/9$ , and results in a Gaussian distribution around the centre of the right-hand region. In the remaining cases the two parents are taken from different regions, and the resulting offspring is distributed uniformly over the interval between the two parents. Note that the density decreases when the distance between the parents increases. These three contributions have been approximated, and the resulting distribution is shown by the grey line in Figure 9.2. Now, let us assume that selection is applied and that the individuals between the two peaks are discarded. Two observations can be made. First, the density function of offspring after selection is more strongly peaked than the density function of the parents. Second, it can be observed that the application of this recombination operator results in an enlargement of the existing differences in number of individuals in each of the two regions.

In order to prevent these effects one can use a more sophisticated selection mechanism. Such a mechanism should not only be biased towards the fittest individuals, but also take the distribution of the population over the search-space into consideration. In the next section we propose the Cluster Evolution Strategy.

### 9.3 Outline of CLES

The Cluster Evolution Strategy (CLES) uses a two-stage selection process. The first stage is a fitness-based selection, while the second phase of selection is based on the distribution of individuals over the search space. During the first stage a subset of the complete population, containing the best individuals, is selected. This step results in the required selective pressure, which guides the search towards regions of high average fitness. Next, a clustering process is applied to the individuals selected during the first phase, and the best individual of each cluster is selected as a representative of that cluster. A new population is created by applying the evolutionary operators to these representatives only. Using only representatives as parents helps in preventing premature convergence. As we have seen in section 9.2, if a region extracts relatively many individuals, then this unbalance is likely to be enlarged by the recombination operator. This problem is resolved as each cluster of individuals is represented by only one representative, independent of the size of this cluster. Nearly identical individuals are likely to belong to the same cluster, and thus only one of



these individuals needs to be present in the set of representatives. Because the presented two-stage selection schedule is not very sensitive to premature convergence, it is relatively safe to preserve the selected individuals. This results in population-elitism, which helps in preserving information and propagating obtained information to subsequent generations. Let us return to the example shown in Figure 9.1. The circles and bullets represent individuals in the current population. These individuals are distributed uniformly over the search space. The dashed line represents the selection boundary, so that the individuals in the grey region are discarded. Given the distribution of the surviving individuals, three clusters can be found: the left-hand cluster contains five individuals, the middle cluster one individual, and the right-hand cluster three individuals. The representatives of the three clusters are numbered. Observe that the second representative is closest to the global optimum. This individual does not perform so well, however due to the fact that it performs reasonably well, and that it is far away from all other individuals it becomes important to retain this individual until the potential of the region around this point is investigated a little better.

Within many randomized clustering methods local optimization is applied for two reasons [TŽ89]. First, if the search space around an individual is relatively smooth, there is no reason to use a slow global optimization method. Second, by applying local optimization in an early phase the potential of different sample-points can be compared better. For these reasons we have incorporated local search in our algorithm too. Local optimization is applied to the representatives only, as these representatives are assumed to be typical for the current population. Applying local optimization to the (small) set of representatives reduces the amount of computation required and decreases the probability of locating the same local optimum multiple times. Because high quality representatives are likely survive for several generations due to the population-elitism, the local optimization can be done in stages. During each generation only a limited number of local optimization steps are spend on each representative, so the number of local optimization steps spend on an individual is proportional to the number of generations it is able to survive. This multi-stage local optimization process prevents that a lot of local optimization steps are spend on an individual that is discarded immediately afterwards.

Next, the pseudo-code of the algorithm for the main loop of the cluster evolution strategy is given.

```

ClusterEvolutionStrategy()
   $i = 0$ ;
   $N = (\lambda + 1)N_{repres}$ ;
   $P_0 = randomPopulation(N)$ ;
  repeat
     $i = i + 1$ ;
    ## two-stage selection
     $P_i = selectBest(N_{repres}, P_{i-1})$ ;
     $R_i = selectClusterRepresentatives(P_i, \tau, r_{min})$ ;

```

```

if  $\neg$  ready then
  ## local optimization of representatives
  for all  $x \in R_i$  do
    localOptimize( $x, s_{loc}$ );
  od;
  ## production of new offspring
  while  $|P_i| < N$  do
    if  $|R_i| > 1$  then
      select  $p_1$  and  $p_2 \in R_i$ , ( $p_1 \neq p_2$ );
       $P_i = P_i \cup \text{recombine}(p_1, p_2)$ ;
    else
       $P_i = P_i \cup \text{mutate}(p)$ ,  $p \in R_i$ ;
    fi;
  od;
fi;
until ready;
end

```

Here,  $P_i$  denotes the population created during the  $i^{th}$  generation,  $N$  is the maximal population size,  $N_{repres}$  is the maximal number of representatives,  $\lambda$  is the minimal number of offspring per parent,  $|R|$  is the cardinality of set  $R$ ,  $R_i \subseteq P_i$  is the actual set of representatives. The function *randomPopulation*( $N$ ) generates a random population of size  $N$ , *selectBest*( $N_{repres}, P_i$ ) selects the  $N_{repres}$  best individuals from population  $P_i$ , *selectClusterRepresentatives*( $P_i, \tau, r_{min}$ ) denotes the selection of the representatives of all clusters, *localOptimize*( $x, s_{loc}$ ) performs the local optimization step on individual  $x$ , *recombine*( $p_1, p_2$ ) denotes an application of the recombination operator to parents  $p_1$  and  $p_2$ , and *mutate*( $p$ ) denotes an application of the mutation operator to parent  $p$ . The ready predicate becomes true when the desired objective value is obtained, or when the total allowed number of function evaluations  $max_{eval}$  is reached, or when all clusters have a size below a certain minimal threshold. Note that individuals that survive selection are retained for a complete generation, even if the individual is not selected as a representative. Retaining well performing individuals is important, because this helps in accumulating more information about the shape and size of the cluster the individual belongs to. In the next subsections detailed descriptions of the different parts of the algorithm are presented.

## 9.4 Clustering method

The clustering problem can be stated as follows: Given a space  $\mathcal{S}$  and a set  $X$  of sample-points  $\vec{x}_i \in \mathcal{S}$ , try to discover a number of subsets  $A_i \subseteq \mathcal{S}$  under the restrictions  $\cup_i A_i = \mathcal{S}$  and  $A_i \cap A_j = \emptyset$  if ( $i \neq j$ ), that minimizes a certain measure  $F(A)$ . The exact definition of  $F(A)$ , and the (maximal) number of clusters to be discovered differs per application.

Clustering problems tend to be difficult, especially if the number of clusters is not known in advance [KR90].

For the problems we are handling the search space is  $\mathcal{S} = [a, b]^n$ , where  $a, b \in \mathbb{R}$ . It is inherent to these problems that the density of the sample is very low. When allowing a high density of the sample that covers the complete search-space there would be little need for complex optimization methods such as an evolutionary algorithm. In order to get a good balance between speed and quality of the clustering a specialized clustering heuristic was developed. We did not use the existing clustering methods because many of these methods require the number of clusters to be known beforehand. Furthermore these methods aim at the best possible clustering, while for our purposes an approximation to the optimal clustering is sufficient. Furthermore the clustering method has to be fast, because the clustering is applied during each generation.

In order to avoid the curse of dimensionality [Sco93], and to increase the density of the sample a set of  $n$  mappings of  $\mathcal{S}$  to a one-dimensional axis have been defined by

$$g_i : \mathbb{R}^n \rightarrow \mathbb{R} = \vec{x} \cdot \vec{e}_i,$$

where  $\vec{e}_i$  is the unit vector along the  $i^{th}$  dimension.

For each dimension  $i$  the following procedure is applied. The set  $\mathcal{S}$  is sorted on the value  $g_i(\vec{x})$ , and  $D(\vec{x})$  is defined as the distance between  $\vec{x}$  and its predecessor in this ordered set. The expected value of  $D(\vec{x})$  is

$$E[D(\cdot)] = (\max_x \{g_i(\vec{x})\} - \min_x \{g_i(\vec{x})\}) / (N_{repres} - 1).$$

A cluster boundary is assumed in front of each sample-point  $\vec{x}$  satisfying the condition,

$$D(\vec{x}) \geq \tau \max\{E[D(\cdot)], r_{min}\},$$

where  $r_{min}$  is a lower bound on the resolution. For each cluster a representative is chosen: the individual with the highest fitness in this cluster. By repeating this process for each dimension a complete set of representatives is obtained.

After the set of representatives is determined, a second pass is made over all dimensions and all data to determine the sizes of the clusters that the different representatives belong to. Given a representative  $r$ , the size of the corresponding clusters is given by the vector  $\vec{C}^{(r)}$ . To determine  $C_i^{(r)}$  the sample-points are ordered on the value  $g_i(\vec{x})$ , and the cluster boundaries are determined again. Now the size along dimension  $i$  is given by

$$C_i^{(r)} = \max\{g_i(\vec{x}_{last}^{(r)}) - g_i(\vec{x}_{first}^{(r)}), r_{min}\},$$

where  $\vec{x}_{first}^{(r)}$  and  $\vec{x}_{last}^{(r)}$  are the first and the last individual of the cluster that is represented by  $r$ .

Next, the pseudo-code for the clustering step is given. The following tuple is used by this code:

$$point = [index, map];$$

Here, the *index* denote the position of the point, and *map* denotes the mapping of this point on a projection axis. Now, the clustering step is given by the following pseudo-code:

```

Clustering( $\{\vec{x}\}$ )
  for  $k = 1$  to  $d$  do
     $gx$  : list of point;
     $gx = \{\}$ ;
    ## Create mapping on axis  $\vec{e}_k$ 
    for  $i = 1$  to  $N$  do
      add  $[i, \vec{x}_i \cdot \vec{e}_k]$  to  $gx$ ;
    od;
     $sort_{map}(gx)$ ;
    ## compute the distance-threshold used to determine cluster boundaries
     $threshold = \min \left\{ \tau \frac{(gx_N.map - gx_1.map)}{N}, \tau r_{min} \right\}$ ;
    ## Create first set for dimension  $k$  and add first point
     $createNewSet(k)$ ;
     $addToCurrentSet(gx_1.map)$ ;
    for  $i = 2$  to  $N$  do
      if  $(gx_i.map - gx_{i-1}.map) > threshold$  then
        ## Boundary detected: create a new set for dimension  $k$ 
         $createNewSet(k)$ ;
      fi;
       $addToCurrentSet(gx_i.map)$ ;
    od;
  od;
end

```

Here,  $\{\vec{x}\}$  denotes the set of all data-points,  $gx$  contains a one-dimensional projection all data-points, the *add* operation adds a tuple to a set,  $createNewSet(k)$  creates a new set for dimension  $k$  and makes this the current set, and  $addToCurrentSet(point)$  adds a point to the current set. The procedure creates a partition of the data-points over a number of sets for each dimension. Given a dimension each data-point is assigned to exactly one set for this dimension. Each set delivers one representative, and the size of the cluster of this representative  $\vec{C}$  is determined by looking up the set of this representative in each of the partitions.

## 9.5 Evolutionary operators

The evolutionary operators also strongly influence the shape of the sampling density function, and hence the design of these operators can have a strong influence on the reliability

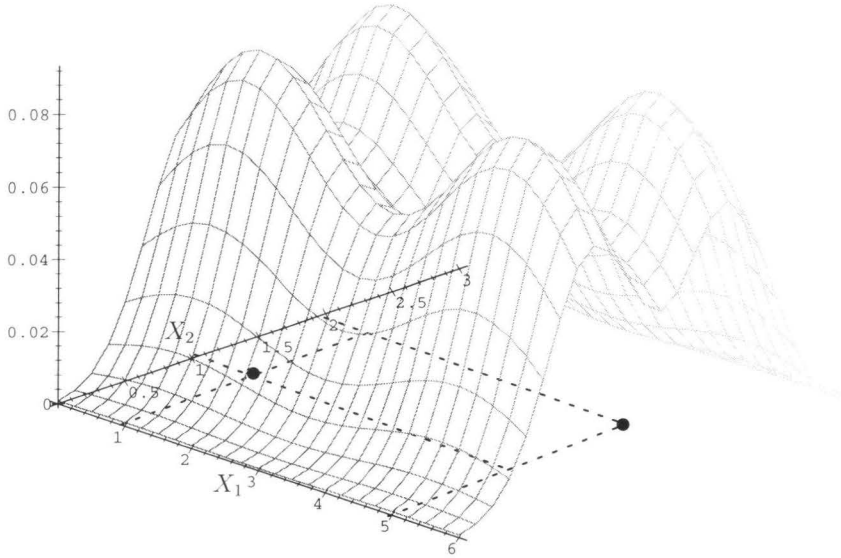


Figure 9.3: Density function used by main recombination operator for in two-dimensional search space

of the evolutionary optimizer. Many recombination operators, such as for example intermediate recombination, generate offspring by means of a density function that is focussed on the centre of the current population. This effect is especially strong in high-dimensional spaces. Such a bias towards the centre is undesirable, because such a bias corresponds to the assumption that the global optimum is not located at the boundary of the search space, and that the local optima that initially attract the individuals are scattered evenly around this global optimum. Even though many artificial test-functions adhere to these assumptions, there is no reason to suspect that an arbitrary real function optimization problem adheres also to these assumptions.

Our main operator is a (disruptive) recombination operator. For each dimension it chooses the value of either of its parents with equal probability, and adds some Gaussian noise to it. Given two different parents  $\vec{x}^{(p1)}$  and  $\vec{x}^{(p2)}$ , an offspring is created according to

$$x_i^{(o)} = (x_i^{(p1)} \text{ or } x_i^{(p2)}) + N(0, \sigma^2)$$

where  $N(0, \sigma^2)$  is a Gaussian distributed random variable with mean zero and standard deviation  $\sigma = |x_i^{(p1)} - x_i^{(p2)}|/3$ . Figure 9.3 shows the density function for a two-dimensional search-space. The parents are represented by the bullets at locations (1, 1) and (2, 5). This operator enforces a quite broad distribution of the individuals, where the width of this distribution is proportional to the distance between the parents. Only if all parents are

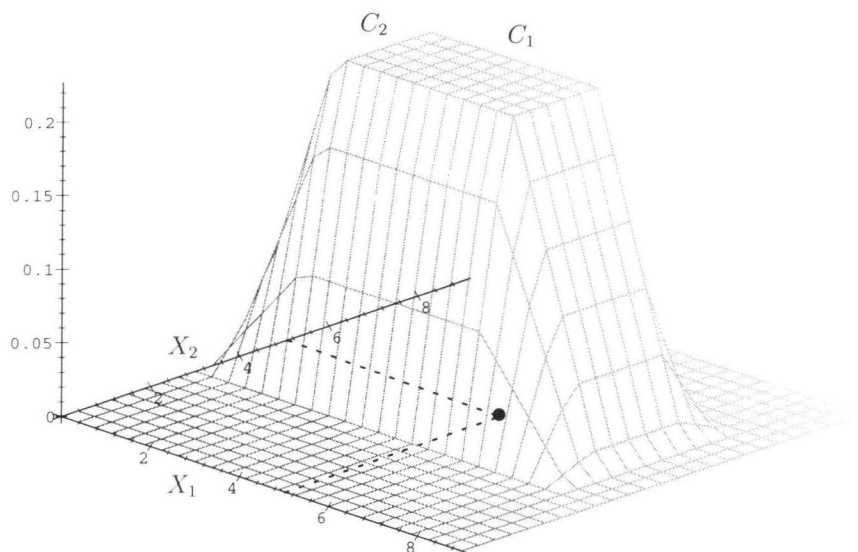


Figure 9.4: Density function used by mutation operator

located in a small part of the search space, then the offspring are located in the same part. Furthermore, an over-sampling of the centre of the current distribution is prevented, as the region mid-way between the two parents gets a relatively low density. If this operator is applied for a large population then the offspring are scattered over roughly the same region as the parent individuals. Only after selection of the fittest individual a shrinking of the region under investigation takes place. By taking different alleles from different parents, this operator can create new combinations of existing alleles. (This operator shows some resemblance to the fuzzy recombination operator [VMC95].)

The operator prevents that the sampling density function narrows too rapidly. However, when all well-performing values for different dimensions have been collected, the operator might have problems in combining different alleles in order to find the optimal solution. Because the Gaussian noise is proportional to the distance between the two parents along that dimension, values that are far apart cannot be exchanged easily. In order to enlarge the probability of this kind of long-distance exchanges, a discrete recombination operator is applied with a low probability  $P_{discrete}$ . The discrete recombination operator creates a value for the offspring using the formula

$$x_i^{(o)} = x_i^{(p1)} \text{ or } x_i^{(p2)}.$$

If there is only a single representative left, then it is not possible to apply recombination anymore. Only under these circumstances an unary mutation operator is applied.

Figure 9.4 shows the density function used by this mutation operator for a two-dimensional search-space. The parent is represented by the bullet at location  $(5, 5)$ , and  $\tilde{C}$  is the size of the cluster this parent belongs to. The sampling density is constant over the region that covers the cluster, and it decreases linearly outside this region. A constant density sampling over the cluster is chosen, such that the search does not become biased towards the centre of the cluster, and only selection can decrease the size of the cluster. Only if the best individuals are located in a part of the cluster, then the search will focus on this part due to selection. The sides, that are located outside the cluster, are added such that some of the newly generated sample-points are located outside the cluster, and the cluster can grow.

## 9.6 Local search strategy

In order to get a better insight in the relative quality of the different clusters, some kind of local search is needed. We have chosen to use a simple and fast local optimizer because the global search is provided by the evolutionary algorithm, and the evolutionary algorithm as a whole should be fast. The local optimizer is only allowed to use a limited number of function evaluations, given by  $s_{loc}$ . Well-performing individuals are preserved due to the population-elitism. Such individuals are passed to the local optimizer during the subsequent generations, so a single pass through the local optimizer just has to perform a partial optimization. Two different local optimizers have been implemented. The first optimizer is based on Lagrange polynomials, and the second optimizer applies a  $(1 + 1)$ -Evolution Strategy.

### 9.6.1 Lagrange optimizer

A simple approach to create a local optimizer is to draw two random sample-points in a neighbourhood of the point  $\vec{x}$  to be optimized, build a one-dimensional quadratic model by means of elementary calculus and then use this model to estimate the location  $\vec{x}'$  of the optimum. This procedure is implemented as follows. Given the original sample-point  $\vec{x}$ , a base vector  $\vec{e}_i$  is chosen at random. Next, a step size  $h \in [0, C_i/2]$  is determined uniform at random, where  $C_i$  denotes the size of the cluster of  $\vec{x}$  along dimension  $\vec{e}_i$ . A second-order Lagrange polynomial is determined by means of the triple of points  $\vec{x}$ ,  $\vec{x} - h \cdot \vec{e}_i$  and  $\vec{x} + h \cdot \vec{e}_i$ , and the corresponding fitness values. The point  $\vec{x}'$  is set to the position of the maximum of this polynomial. If  $\vec{x}'$  is unbounded, then it is replaced with  $\vec{x} \pm 3 \cdot h \cdot \vec{e}_i$  where the sign is determined by which of the two sample-points  $\vec{x} \pm h \cdot \vec{e}_i$  has the highest fitness. The fittest of the four evaluated points  $\vec{x}$ ,  $\vec{x} - h \cdot \vec{e}_i$ ,  $\vec{x} + h \cdot \vec{e}_i$ , and  $\vec{x}'$  replaces the original individual  $\vec{x}$ . A selected base vector  $\vec{e}_i$  is not used for a second time until all other base vectors  $\vec{e}_j (j \neq i)$  have been used at least once.

The Lagrange optimizer locally builds a one-dimensional quadratic approximation to the function that has to be optimized, and exploits this approximation to guess where the optimum is located. If such a quadratic model matches the local structure, then this

optimizer is likely to perform well. If this assumption is not met, then this local optimizer might fail.

### 9.6.2 (1 + 1)-ES optimizer

The second local optimizer is based on a two-membered (1+1)-Evolution Strategy [Rec73, Rec94, Sch81, Sch95]. This local optimizer is more general than the Lagrange optimizer, because it does not assume a quadratic model for the function under consideration. When applying this (1 + 1)-ES a single parent is used. A single offspring is created by applying the following formula for each dimension  $i$ ,

$$x_i^{(o)} = x_i^{(p)} + N(0, \sigma^2),$$

where  $N(0, \sigma^2)$  is a normally distributed random variable with mean zero and variance  $\sigma^2$ . The offspring replaces the parent if it outperforms the parent on the function  $f$  to be optimized. The value of  $\sigma$  is adjusted by means of the 1/5-success rule [Sch95]:

*determine the ratio of the number of successful mutations to the total number of trials. If the ratio is greater than 1/5 the variance should be increased, if it is less than 1/5 then decrease the variance.*

Within our algorithm each point  $\vec{x}$  is associated with a cluster of size  $\vec{C}$ . It is easy to incorporate this information into the optimization process by setting  $\sigma = \sigma' \cdot C_i$ , where  $\sigma'$  is a scalar parameter with initial value 0.5 that will be updated according to the 1/5-success rule. So, the rule for constructing offspring becomes

$$x'_i = x_i + N(0, (\sigma' C_i)^2).$$

The 1/5-success rule is implemented by means of a multiplication. After a failed optimization step the  $\sigma'$  is multiplied by the parameter  $m_{fail} < 1$ , and when a successful step occurred  $\sigma'$  is multiplied by  $(1/m_{fail})^5$ .

## 9.7 Test-suite

For our experiments we used the test-problems and performance measures defined for the first International Contest on Evolutionary Optimization, taking place during the Third IEEE conference on Evolutionary Computation (Japan 1996).

A test-suite containing five test-problems is used. All test-problems have a scalable dimension, and for each problem both a five-dimensional and a ten-dimensional problem instance are tested. The test-problems are the sphere model, the Griewangk's function, the Shekel's foxholes, the Michalewicz function, and the Langerman's function. The sphere model is defined by:

$$f(\vec{x}) = \sum_{i=1}^N (x_i - 1)^2,$$



where  $x_i \in [-5, 5]$ . The goal is to find a value  $\vec{x}$  such that  $f(\vec{x}) \leq 1 \cdot 10^{-6}$ . The Griewangk's function is defined by:

$$f(\vec{x}) = \frac{1}{d} \sum_{i=1}^N (x_i - 100)^2 - \prod_{i=1}^N \cos\left(\frac{x_i - 100}{\sqrt{i}}\right) + 1,$$

where  $d = 4000$ , and  $x_i \in [-600, 600]$ . The goal is to find a value  $\vec{x}$  such that  $f(\vec{x}) \leq 1 \cdot 10^{-4}$ . The Shekel's foxholes function is defined by:

$$f(\vec{x}) = - \sum_{i=1}^m \frac{1}{|\vec{x} - A(i)|^2 + c_i},$$

where  $m = 30$ ,  $x_i \in [0, 10]$ ,  $A$  is a  $n \times m$  matrix,  $c$  is a  $m$ -vector and  $A(i)$  denotes the  $i^{th}$  column of  $A$ . The goal is to find a value  $\vec{x}$  such that  $f(\vec{x}) \leq -9$ .

The Michalewicz's function is defined by:

$$f(\vec{x}) = - \sum_{i=1}^N \sin(x_i) \sin^{2m}\left(\frac{ix_i^2}{\pi}\right),$$

where  $m = 10$  and  $x_i \in [0, \pi]$ . The goal is to find a value  $\vec{x}$  such that  $f(\vec{x}) \leq -4.687$  for the five dimensional versions and  $f(\vec{x}) \leq -9.66$  for the ten dimensional version.

The Langerman's function is defined by:

$$f(\vec{x}) = - \sum_{i=1}^m c_i \left( \exp^{-\frac{1}{\pi} |\vec{x} - A(i)|^2} \cos(\pi |\vec{x} - A(i)|^2) \right),$$

where  $m = 5$ ,  $x_i \in [0, 10]$ ,  $A$  is a  $n \times m$  matrix,  $c$  is a  $m$ -vector and  $A(i)$  denotes the  $i^{th}$  column of  $A$ . The goal is to find a value  $\vec{x}$  such that  $f(\vec{x}) \leq -1.4$ .

Three performance measures were defined. These performance measures should be computed over twenty independent runs. The first measure is the Expected Number of Evaluations per Success (ENES), which is defined as the total number of evaluations divided by the number of successful runs. The second measure is the best Value reached during all runs (BV), and The third measure is the Relative Time (RT) measures the computational overhead introduced by the evolutionary algorithm. It is defined as the ratio between the amount of computation spend in the evolutionary algorithm and amount of computation spend on pure function evaluations. This value is computed by measuring the CPU-time for a run of the evolutionary algorithm for 10,000 iterations  $CT$  and the CPU-time for 10,000 function evaluations  $ET$ . Now the relative time is given by the formula  $RT = (CT - ET)/ET$ .

## 9.8 Results

During our experiments the following parameter settings are used. The number of function evaluations  $s_{loc}$  during a single application of the local optimizer is set to nine, the minimal

| Test problem    | ENES         |              | RT         |          |
|-----------------|--------------|--------------|------------|----------|
|                 | (1 + 1)-ES   | Lagrange     | (1 + 1)-ES | Lagrange |
| Sphere 5D       | 14459        | bf 1452      | 31         | 75       |
| Sphere 10D      | 35275        | <b>3462</b>  | 27         | 32       |
| Griewangk 5D    | 46212        | <b>22039</b> | 2.2        | 2.0      |
| Griewangk 10D   | 65590        | <b>19125</b> | 1.8        | 1.1      |
| Shekel 5D       | <b>43067</b> | 51845        | 1.17       | 0.96     |
| Shekel 10D      | <b>39151</b> | 363685       | 0.88       | 0.53     |
| Michalewicz 5D  | 15600        | <b>10661</b> | 4.7        | 3.9      |
| Michalewicz 10D | 104993       | <b>41765</b> | 3.8        | 2.6      |
| Langerman 5D    | 12543        | <b>11343</b> | 2.7        | 1.8      |
| Langerman 10D   | 75477        | <b>61729</b> | 1.8        | 0.8      |

Table 9.1: Performance measures ENES and RT for the cluster evolution strategy with the (1+1)-ES and the Lagrange local optimizers.

|                 | Induct | DGL   | DE     | Simplex | Morphic  | GEMGA    | Scatter  |
|-----------------|--------|-------|--------|---------|----------|----------|----------|
| Sphere 5D       | 20     | 243   | 736    | 326     | 1278.1   | 1522.6   | 12218    |
| Sphere 10D      | 40     | 243   | 1892   | 1099    | 2934.7   | 6392.2   | 85692    |
| Griewangk 5D    | 41     | 21141 | 5765   | 35639   | 89741    | 511797   | 2977996  |
| Griewangk 10D   | 79     | 20898 | 13508  | 6446    | 2230597  | 890683   | 2110889  |
| Shekel 5D       | 74     | 6318  | 76210  | 13836   | 190134   | 451992   | 29449    |
| Shekel 10D      | 120    | 6075  | 744250 | 259477  | 4440948  | 14900000 | 879409   |
| Michalewicz 5D  | 120    | 6804  | 1877   | 9925    | 1534.1   | 60219    | 33468    |
| Michalewicz 10D | 501    | 14823 | 10083  | 236348  | 26277    | 234698   | 20233341 |
| Langerman 5D    | 176    | 4131  | 5308   | 74720   | 232496   | 4578.2   | 6149     |
| Langerman 10D   | 372    | 26973 | 44733  | 1032627 | 15727653 | 443436   | 1071086  |

Table 9.2: ENES performance measure

number of offspring per representative  $\lambda$  is set to two, the decision boundary for the clustering algorithm  $\tau$  is set to 1.5, the minimal allowed resolution  $r_{min}$  is set to  $10^{-4}$ , the probability that the discrete recombination is applied  $P_{discrete}$  is set to  $\frac{1}{10}$ , the multiplier  $m_{fail}$  is used for updating the  $\sigma$  value of the (1 + 1)-ES local optimizer is set to 0.95, the maximal number of representatives  $N_{repres}$  is set to 100, and the maximal number of function evaluations during a single run  $max_{eval}$  is set to 50,000 for the 5-dimensional and to 100,000 for the 10-dimensional test problems. All parameters have been set by using only a few experiments, or their value is determined by means of an educated guess. So, no problem-specific parameter tuning has been performed. We have chosen not to do any problem-specific parameter tuning, because we prefer to have a single parameter setting that performs robustly over a large range of different problems.

Two sets of experiments are conducted. During the first set a (1+1)-ES is used as a local

optimization method. The second set of experiments uses a Lagrange interpolation during the local optimization step. A single Lagrange interpolation will require three function evaluations. The results are shown in table 9.1, where the number on the left-hand side of the  $/$ -symbol corresponds to the (1+1)-ES method and the right-hand side of the  $/$ -symbol to the Lagrange method.

When using the ENES-measure we see that all test-problems can be solved within a reasonable number of function evaluations. The best result is shown in a bold font. The Lagrange method outperforms the (1+1)-ES method on eight out of the ten test-problems. Only in case of the Shekel function the (1+1)-ES method performs better. Probably the quadratic model used in the Lagrange method does not give a good local approximation in case of the Shekel test-problems. When comparing the results for the corresponding 5-dimensional and 10-dimensional test-problems we see that the increase in the ENES-measure is moderate in most cases. In case of the Shekel test-problems the 10-dimensional version seems to be even easier to optimize than the 5-dimensional version, when using the (1+1)-ES. We expect that this effect is a result of a larger number of representatives being selected in the 10-dimensional case, resulting in a better exploration of the complete search-space.

The BV measure is of no interest in our case because the defined acceptance threshold is reached for all test problems. As a result this measure only reflects the location of the acceptance threshold.

The RT measure shows that the overhead of the algorithm is reasonable on the presented test problems. Only in case of the sphere function the overhead is large, however this mainly due to the fact that the sphere function is relatively easy.

When comparing both cluster evolution strategies, we see that the Lagrange method performs best in most cases. Nevertheless we prefer the (1+1)-ES methods. This method makes less assumptions regarding the actual problem being optimized, and is therefore expected to result in a more reliable evolutionary algorithm.

The cluster evolution strategies have been compared to other optimization methods on this test-suite during the first International Contest on Evolutionary Optimization [BDL<sup>+</sup>96]. The competitors were Inductive search (uses an oracle to optimize sub-functions) [BP96], DGL-optimization (Latin square sampling with function domain contraction techniques) [LS96], Differential evolution (uses three-parent recombination where third parent is updated by difference between first and second parent) [SP96], simplex GA and hybrid methods (uses simplex method and local hill-climbing) [SB96], Morphic search strategies (hill-climbing on randomly remapped search spaces) [KD96], GEMGA (linkage learning method) [Kar96b], and Scatter search (population-based tabu-search) [FGMV96]. The results for the ENES-measure are shown in Figure 9.2. It was not possible to compare the different optimization methods fairly because the functions in the test-suite were not diverse enough. The large differences in performance are mainly due to the different amount of prior information used by the different competitors. Some solved the test-functions as binary optimization problems while others included advanced numerical optimization techniques and performed problem-specific parameter tuning. We have chosen not to do problem-specific parameter tuning because we are mainly interested in robust optimization

methods that do not require such problem-specific tuning.

## 9.9 Summary

In this chapter we introduced the Cluster Evolution Strategies. Viewing an evolution process as a self-adaptive sampling density function helps in understanding some of the problems that can occur during a function optimization task. A large region, having a high overall fitness acts like an attractor to the population. If an unbalanced distribution of the population over the search-space is obtained, then this unbalance can easily be enlarged by recombination. A two-phase selection schedule is proposed. A cluster analysis is used to reduce the population to a set of representative individuals. The evolutionary operators are applied to these representatives only. This approach helps to overcome the problems noted in this chapter, and enhances the sampling density function, thereby resulting in a more reliable function optimizer.

The search process is decomposed in a local and a global search strategy. The global search is performed by an evolutionary algorithm. This algorithm is tailored towards reliable global search, which is reflected in the type of evolutionary operators used. These operators are not likely to perform well on a local optimization task unless the whole population is converging towards a single point in the search space. The local search is performed by means of a separate local optimizer, that is applied during every generation. The local optimizers are tailored for speed, and assume that the search-space is not too complex in the local neighbourhood, where they operate.

A nice additional advantage of the clustering approach is that it allows the efficient use of local optimization techniques. By applying local optimization to representatives only, the probability of obtaining the same optimum several times becomes smaller. Elitism is needed within our algorithm because the local optimization is done incrementally. The longer the lifetime of a representative, the more local optimization is performed on the corresponding point.

## Chapter 10

# Evolutionary Constrained Numerical Optimization

A promising domain of application for evolutionary algorithm is constrained optimization. In this chapter we investigate the performance of different evolutionary algorithms on a scalable constrained optimization problem, and on test-suite of eight constrained optimization problems.

First we introduce so-called stepping-stones problems: two simple constrained optimization problems with scalable dimension, adjustable complexity, and a known optimal solution. The stepping-stones problems are simple enough to be analysed; However, the constraints in the problem correspond to the type of constraints used in real nonlinear numerical constrained optimization problems. We study these problems to get a better understanding of the effects that take place in the domain of numerical constrained optimization. A set of evolutionary algorithms, all using different selection schemes, is applied to this problem. The performance of the evolutionary algorithms differs strongly on the stepping-stones problems; Selection schemes that only use a limited number of offspring as parents for the next generation consistently outperform the schemes that accept all offspring as parents and adjust their fertility based on (relative) fitness during the experiments. One of the selection-schemes corresponds to the selection scheme of the cluster evolution strategy (CLES). To get a fair comparison the CLES is used without the local optimizer.

Next, the performance of a set of evolutionary algorithms is assessed on a test-suite containing eight constrained optimization problems. Some of these evolutionary algorithms incorporate local optimization of individuals.

A brief introduction to numerical constrained optimization problems is given in section 10.1. Section 10.2 describes a specific problem for evolutionary optimizers and introduces the stepping-stones constrained optimization problem. During our experiments we studied the effectiveness of selection schemes used in genetic algorithms and in evolution strategies. The different selection schemes are discussed in section 10.3. The results of experiments are presented in section 10.4. Section 10.5 introduces the larger test-suite and reports the results of experiments conducted on this test-suite. A comparison to the results reported

in literature is given in section 10.6, followed by a discussion of our results in section 10.7. This chapter is concluded by a summary in section 10.8.

## 10.1 Numerical constrained optimization

Many numerical constraint optimization problems (NCOP) can be written in the form:

$$\begin{array}{ll} \text{objective:} & \text{MAXIMIZE } f(\vec{x}) \\ \text{constraints:} & x_{i,low} \leq x_i \leq x_{i,high} \quad (\text{box}) \\ & G(\vec{x}) \leq 0 \quad (\text{inequality}) \\ & H(\vec{x}) = 0 \quad (\text{equality}) \end{array}$$

where  $\vec{x} \in \mathbb{R}^d$  is called the objective vector,  $f(\vec{x})$  is the objective function,  $G(\vec{x}) \leq 0$  denotes the set of inequality constraints of type  $g(\vec{x}) \leq 0$  and  $H(\vec{x}) = 0$  denotes the set of equality constraints of type  $h(\vec{x}) = 0$ . The goal is to find the objective vector  $\vec{x}^*$  that maximizes the objective function and simultaneously satisfies all the constraints. Note that function optimization problems can be written as a NCOP with only box-constraints, so the class of function optimization problems is a subset of the class of NCOP's.

The complexity of the constraints can have a strong influence on the difficulty of NCOP's. Linear constraints (such as box-constraints) are relatively easy to process, because such constraints give a feasible region consisting of a single convex hull. Nonlinear constraints can result in one or more irregularly shaped feasible regions.

When applying evolutionary algorithms to a NCOP, one has to cope with the constraints. These constraints can be handled by means of penalty functions, decoders, or repair operators [MJ91]. Penalty functions can be used to recast the original problem to a new problem having only box-constraints and using a modified objective:

$$f^{(1)}(\vec{x}) = f(\vec{x}) - \alpha \cdot P(\vec{x}, G, H).$$

The function  $P(\vec{x}, G, H)$  is assumed to be a measure for the number of constraints in the sets  $G$  and  $H$  that are violated for objective vector  $\vec{x}$ , and  $\alpha$  is a scalar that balances the relative strength of the objective  $f(\vec{x})$  and the penalty  $P(\vec{x}, G, H)$ . When handling a black-box NCOP, it is not known whether the feasible regions intersect with regions of relatively high fitness. If this is not the case, then small values of  $\alpha$  focus the search on the infeasible region when using  $f^{(1)}(\vec{x})$ ; Thus reducing the probability that the optimum is found.

To prevent the necessity of choosing a value  $\alpha$  the following objective can be used:

$$f^{(2)}(\vec{x}) = \begin{cases} f(\vec{x}) & \text{if } G \text{ and } H \text{ satisfied} \\ -P(\vec{x}, G, H) & \text{otherwise.} \end{cases}$$

When optimizing this function the original objective,  $f(\vec{x})$  is only calculated if none of the constraints are violated. For some NCOP's this might even be a requirement, because the original objective function can be undefined when constraints are violated.

It is possible to introduce gradient information in  $P(\vec{x}, G, H)$  by also incorporating an additional term indicating how strongly the constraints are violated. The search process can benefit from such gradient information when comparing two individuals violating the same number of constraints.

It is important that this gradient term cannot get too large, because this might result in a competition between constraints. Therefore we propose:

$$P(\vec{x}, G, H) = \sum_{g \in G} p(g(\vec{x})) + \sum_{h \in H} p(-|h(\vec{x})|)$$

where

$$p(y) = \begin{cases} 1 + (1 - e^{-y/\gamma}) & \text{if } y < 0 \\ 0 & \text{otherwise} \end{cases}$$

where  $\gamma$  is a scaling factor. When using this  $P(\vec{x}, G, H)$  each violated constraint results in a contribution to  $P(\vec{x}, G, H)$  in the range  $[1, 2]$ . Hence, if two different constraints are violated, then their contribution to the  $P(\vec{x}, G, H)$  differs by at most a factor two, and thus it is prevented that different constraints start to compete and that the evolutionary search focusses on the satisfaction of a subset of all constraints. Figure 10.1 shows a simple inequality constraint  $g(x)$ , and its corresponding penalty  $P(x, g)$ .

In the rest of this chapter we use this penalty function combined with the objective function  $f^{(2)}(\vec{x})$ .

Much more information on constrained optimization problems and the way such problems are handled by evolutionary algorithms can be found in books and papers of Michalewicz [MJ91, Mic92, Mic94, Mic96].

## 10.2 Regions of attraction

The region of attraction of a local optimum  $\vec{z}$  is defined as the largest set of points  $attr(\vec{z}) \subseteq \mathbb{R}^d$ , such that for any starting point  $\vec{y} \in attr(\vec{z})$  the infinitely small step steepest ascent algorithm will converge to this local optimum  $\vec{z}$  [TŽ89]. The relative sizes of the regions of attraction of different local optima can strongly influence the behaviour of EA's.

In section 9.2 it was discussed that EA's often converge rapidly on relatively large regions with a high average fitness. If such a region attracts a reasonable fraction of individuals, then recombination is effective at producing new relatively fit individuals in the same region, thereby increasing the fraction of individuals in these regions even further. Within constrained numerical optimization problems it can easily happen that the optimum is located in a narrow peak, even when the objective function is a smooth function. In the rest of this chapter we study the performance of different evolutionary algorithms on the stepping-stones problem (SSP-a), which is defined by:

$$\begin{aligned} \text{objective:} & \quad \text{MAXIMIZE } \sum_{i=1}^d (x_i/\pi + 1) \\ \text{constraints:} & \quad -\pi \leq x_i \leq \pi \\ & \quad \exp x_i/\pi + \cos(2 \cdot x_i) - 1 \leq 0. \end{aligned}$$

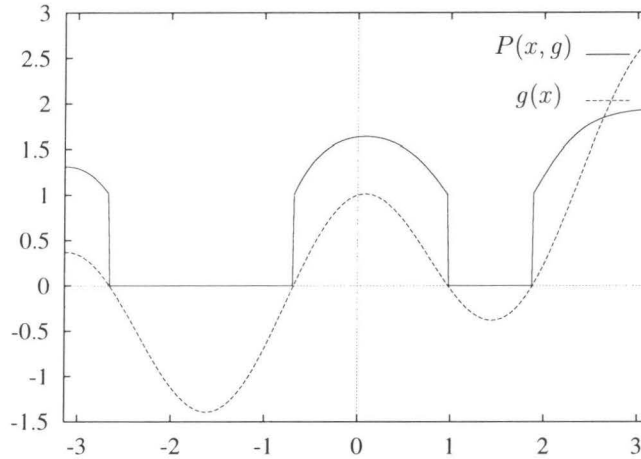


Figure 10.1: The constraint  $g(x) = \exp(x_i/\pi) + \cos(2 \cdot \pi \cdot x_i) - 1 \leq 0$  and an associated penalty function.

The objective function is linear, and without the constraints this would be a very simple optimization problem. The dimension of the problem, denoted by  $d$ , is variable. By increasing this parameter the problem gets more difficult. Figure 10.2 shows a two-dimensional version of the SSP, where the objective function is assumed to be zero if any of the constraints is violated. The feasible region of the SSP is split in  $2^d$  parts, which we call the stepping-stones. The relative size of a stone containing the optimal value along  $n$  dimensions, when compared to the size of the search-space, is approximately,

$$\frac{0.906^n \cdot 1.973^{d-n}}{(2 \cdot \pi)^d} \approx \left( \frac{1.973}{2 \cdot \pi} \right)^d \left( \frac{1}{2} \right)^n.$$

So for each additional  $x_i$  having the correct sign, the area of the corresponding stone is reduced by a factor two.

The optimal solution is located in a narrow peak. The region of attraction of this peak can be extended a little bit by using an appropriate penalty function; however if the dimension of the problem  $d$  becomes larger, then the region of attraction of the optimal solution becomes very small compared to the volume of search space.

The SSP is designed to show a certain weak point of evolutionary algorithms, i.e. their preference for broad peaks. However, it also shows a possible strong point of population based optimization techniques as we will explain next. The SSP has a disconnected feasible



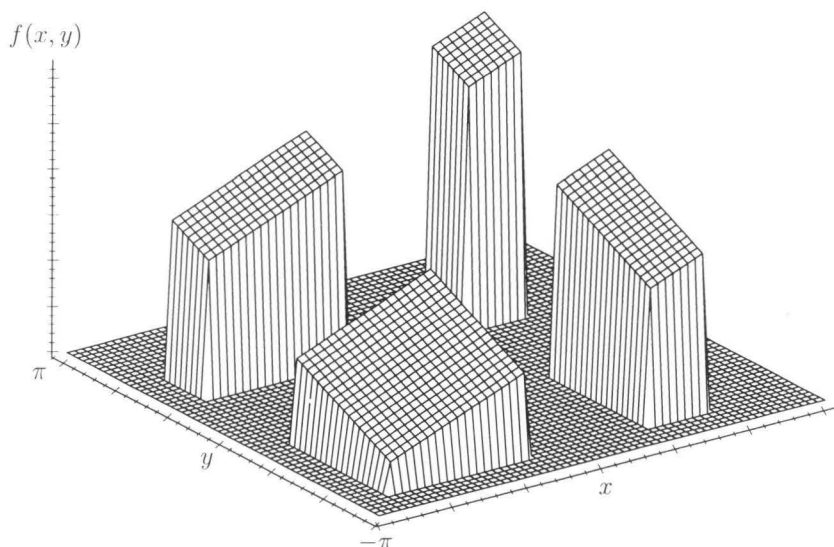


Figure 10.2: The stepping-stones problem SSP-a in two dimensions.

region. Therefore, one cannot assume the existence of a complete feasible path from a current feasible solution to the global optimum. This causes problems to some path-oriented optimization techniques, because these methods have difficulties to follow a path through the infeasible region. An additional advantage of evolutionary techniques using recombination is the implicit parallelism that occurs when several dimensions can be optimized independently of each other, as is the case for the SSP.

A second stepping-stones problem (SSP-b) is introduced. This problem has the same set of constraints as the SSP-a, but a different objective function has to be maximized, i.e.

$$f(\vec{x}) = \begin{cases} \sum_{i=1}^d (x_i/\pi + 1) & \text{if all } x_i \geq 0 \\ 1 & \text{otherwise.} \end{cases}$$

Figure 10.3 shows a two-dimensional version of this problem. The SSP-b is assumed to be more difficult than the SSP-a, because its objective function gives less information regarding the location of the optimal solution. All points  $\vec{x}$  that are at suboptimal peaks (stones) have the same fitness, independent of the number of near-optimal values  $x_i$  of these points.

## 10.3 Selection schemes

In order to get a proper comparison between different selection schemes all EA's use the same evolutionary operators, and the same parameter settings.

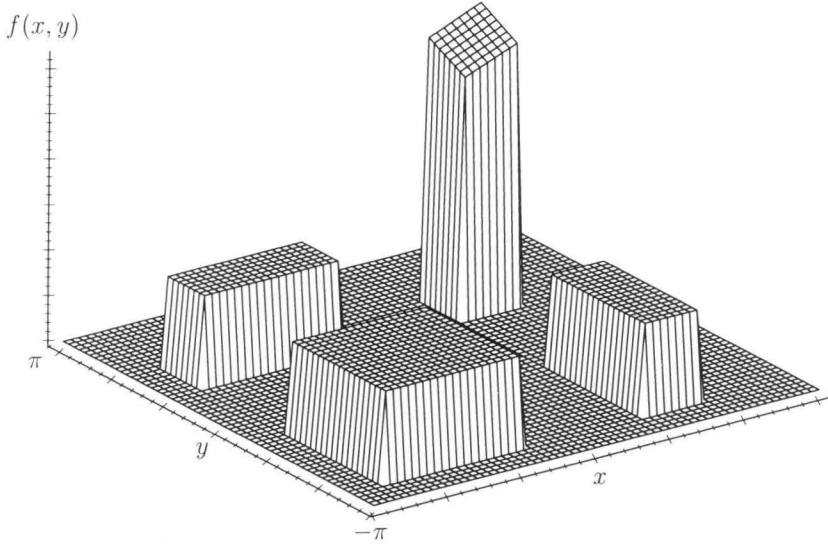


Figure 10.3: The stepping-stones problem SSP-b in two dimensions.

The Gaussian and the discrete recombination operator defined in section 9.5 are used. The discrete recombination operator is applied with a low probability  $P_{discrete}$ . All algorithms use a generational approach and a population size  $P_{size}$ . The following selection schemes are compared:

**tournament-selection:** A generational genetic algorithm with tournament selection (see section 2.3.1),

**ES-selection:** A  $(\mu, \lambda)$  selection with  $\mu = P_{size}$ , and  $\lambda = 7P_{size}$  (see section 2.3.1),

**trunc-selection:** A  $(\mu, \lambda)$  selection with  $\mu = \lambda = P_{size}$  (see section 2.3.2),

**cluster-selection:** Two stage selection scheme of CLES (see section 9.3).

## 10.4 Experiments

During all experiments we use the following parameter settings. The scaling factor  $\gamma$  for the penalty-function is one, the population size  $P_{size}$  is 100, the probability  $P_{discrete}$  that discrete recombination is applied is set to 0.1, the tournament-selection uses a tournament size  $\tau$  of 2 or 3, and the maximal number of function evaluations during a single run is set to 100,000. During the experiments the dimension of the problem is varied between one and twenty. The results are all averaged over seventy independent runs.

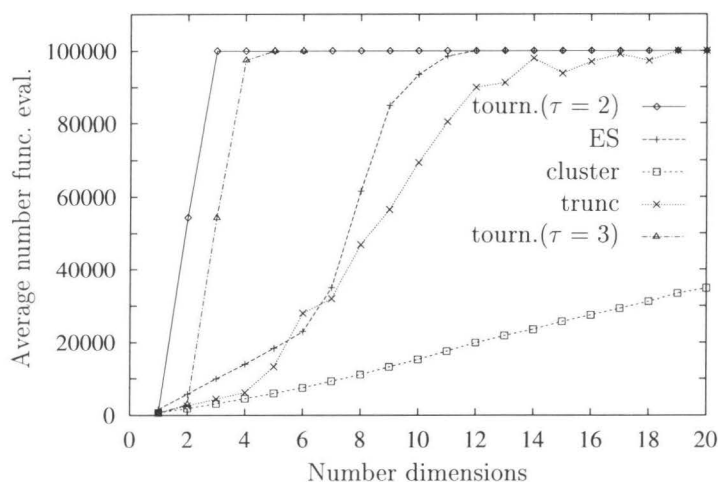


Figure 10.4: The average number of evaluations used to located the optimum on the SSP-a.

Figure 10.4 shows the average number of function evaluations. The tournament-selection performs poor, and cluster-selection performs best. Even when it does not converge any more the cluster-selection is usually able to terminate before it reaches the maximal number of function evaluations. This is due to the fact that it can detect an unsuccessful run because all clusters get a diminishing size. In case of a successful run the cluster-selection is also efficient in terms of the number of function evaluations needed.

Figures 10.5 and 10.6 show the percentage of runs that located the optimum as a function of the dimensionality  $d$  of the problem. A run is said to be successful if  $|f(\vec{x}^*) - f(\vec{x}')| \leq 10^{-3}$  for the solution  $\vec{x}'$ , where  $\vec{x}^*$  is the location of the global optimum. Tournament selection performs poorly. The other three selection schemes perform consistently better.

Figure 10.5 shows the results when applying the different selection schemes to the SSP-a. The tournament of size three performs better than a tournament size of two. We expect that the low performance of the tournament selection is due to the fact that all offspring, also the infeasible ones, are accepted and become the parents of the subsequent generation. A larger tournament size corresponds to a higher selective pressure, and therefore focusses the search more on the best individuals in the population; Therefore, an increase of tournament size leads to better performance. The ES-selection gets its selective pressure due to small fraction of well-performing offspring that are allowed to become parents. The trunc-selection and the cluster-selection get their selective pressure from the population-elitism.

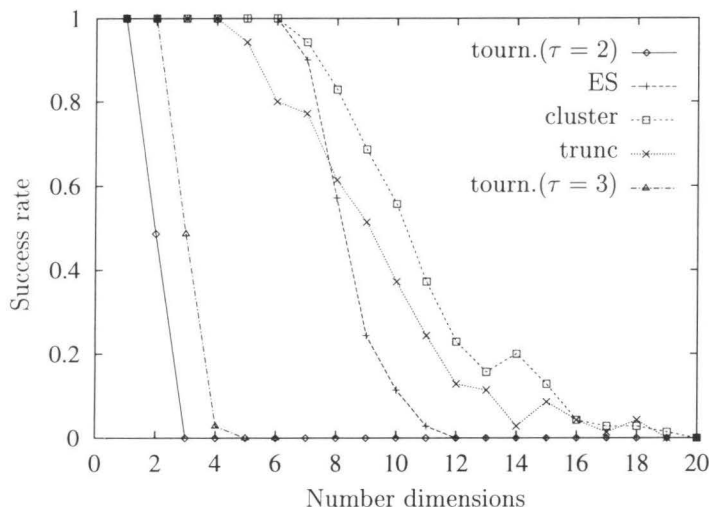


Figure 10.5: Percentage of runs that located the optimum on the SSP-a.

Figure 10.6 shows the results when applying the different selection schemes to the SSP-b. Please note that the range of the horizontal axis differs between two graphs. In problem SSP-b the objective function gives less information. As soon as one of the variables  $x_i$  is smaller than zero the objective function returns a constant value. Therefore, it can be expected that the SSP-b is more difficult than the SSP-a. The best results are still observed for the ES-selection, cluster-selection and trunc-selection. It is interesting to observe that the EA with tournament selection has a higher success-rate on the SSP-b than on the SSP-a. On the SSP-a the tournament selection performs relatively badly. Usually it converges rapidly to one of the suboptimal solutions. In the SSP-b all suboptima have the same fitness. Therefore, the tournament selection converges less rapidly to a single peak, and more time is available to find the actual global optimum. The ES-selection, cluster-selection, and trunc-selection are less susceptible to this type of premature convergence because these selection schemes enforce selective pressure by only using a subset of all individuals in the population as parents. Once the individuals that are allowed to reproduce are selected, the selection of the actual parents is performed in an unbiased manner. Therefore, these selection schemes are able to enforce a high selective pressure, without focusing only on the best few individuals in the population. As a result these selection schemes are less sensitive to premature convergence [vKKE95].

The cluster-selection uses a clustering step during selection in order to get a better sampling of the search-space, and to decrease the preference of the EA for broad peaks. This is an important advantage because it is not only important to find the different parts

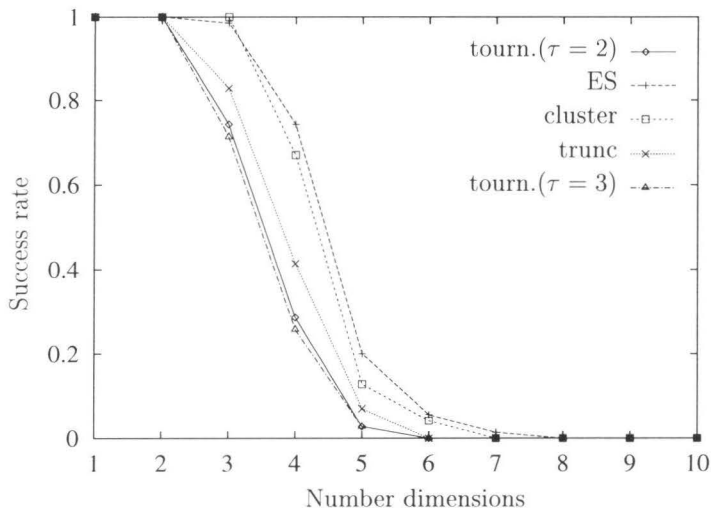


Figure 10.6: Percentage of runs that located the optimum on the SSP-b.

that constitute the optimal solution, but the algorithm should also get the opportunity to mix these parts to construct the optimal solution. A high selective pressure can easily result in dense clusters of well-performing individuals that recombine easily, and such clusters can give rise to loss of information that is needed to find the optimum, but that is not typical for these clusters. This corresponds to a kind of cross-competition between alleles, which seems to be difficult to prevent if one does not consider the distribution of the population during the selection.

## 10.5 Larger test-suite

In this section results are reported of experiments that are conducted on a larger test-suite of numerical constrained optimization problems to assess the performance of a number of different evolutionary algorithms. The test-problems are taken from a paper by Michalewicz and Schoenauer [MS96]. The test-suite involves problem with linear and non-linear objective functions, linear and nonlinear constraints, and convex and nonconvex feasible regions.

Michalewicz and Schoenauer [MS96] give an overview of many different ways in which constraints can be processed. A set of different penalty based methods for handling constraints is compared by means of the Genocop system. Here a different perspective is taken. Emphasis is put on a proper tradeoff between exploration and exploitation. Therefore a

single penalty-method is used and our attention is directed towards the comparison of the different selection schemes. A comparison is made between:

- M1: cluster evolution strategy with  $N = 2n$  and  $\tau = 1.5$  with local optimization,
- M2: cluster evolution strategy with  $N = 2n$ ,  $\tau = 1.5$  without local optimization,
- M3: cluster evolution strategy with  $N = 2n$  and  $\tau = 10$  with local optimization,
- M4:  $(\mu, \lambda)$  selection with  $\mu = n$  and  $\lambda = 7n$ , and
- M5: a set of independent local optimizers of size  $N = n$ .

Local optimization is performed by the  $(1+1)$ -ES as described in section 9.6.2. The GA M5 uses a population of such local optimizers that are evolved in parallel. During the experiments a value  $n = 100$  is used. The population sizes for the different evolutionary algorithms are set in such a manner that the size of the smallest population (after a fitness-based selection) is the same for all algorithms. This means that different algorithms are tested with different population sizes. We consider this to be a fair way of comparing the different algorithms because this minimal population size determines the bottle-neck for passing information to subsequent generations of the evolution process. For the cluster evolution strategy we set this size equal to the size of the population after fitness-based selection (which is equal to the maximal number of representatives taken). For the  $(\mu, \lambda)$  selection we take  $\mu$  (the size of the parent population) equal to  $n$ .

The penalty approach described in section 10.1 assumes that the objective function has to be maximized and that the objective values are always larger than one. This test-suite contains both maximization and minimization problems, and the range of objective values differs for the different functions. Therefore the following fitness function is used:

$$f^{(3)}(\vec{x}) = \begin{cases} \kappa \cdot f(\vec{x}) & \text{if G and H are satisfied} \\ \omega - P(\vec{x}, G, H) & \text{otherwise.} \end{cases}$$

Here  $\kappa$  is set to 1 for an maximization problem and to -1 for a minimization problem,  $\omega$  is set to a large negative value such that the feasible solutions are preferred over infeasible solutions, and  $-P(\vec{x}, G, H)$  is given in section 10.1. The function  $f^{(3)}(\vec{x})$  is the fitness function that is maximized by means of the different GA's.

### 10.5.1 Test-problems

The test-suite described in this section has been taken from Michalewicz and Schoenauer [MS96]. The penalty method we apply is strict in the sense that very small violations of a constraint already result in a large penalty. Equality constraints are often not satisfied exactly and therefore result in a (small) violation. Hence, the penalty approach we use here is not directly applicable to problems involving equality constraints. However, it will be relatively easy to adjust the method to handle such constraints too by relaxing

the requirements. One can achieve this goal by the introduction of a small threshold value and allowing constraint violations that are smaller than this threshold. Of course this threshold value has to be chosen carefully. We decided to exclude the test-problems that involve equality constraints, and hence we use only eight out of the eleven test-problems contained in this test-suite. Next, the definitions of the different test-problems are given.

#### 10.5.1.1 Test-problem $G_1$

The first problem is a rather simple problem involving the minimization of a quadratic function under linear constraints. The problem is to minimize the function

$$G_1(\vec{x}) = 5 \sum_{i=1}^4 x_i - 5 \sum_{i=1}^4 x_i^2 - \sum_{i=5}^{13} x_i,$$

under the constraints

$$\begin{aligned} 2x_1 + 2x_2 + x_{10} + x_{11} &\leq 10, \\ 2x_1 + 2x_3 + x_{10} + x_{12} &\leq 10, \\ 2x_2 + 2x_3 + x_{11} + x_{12} &\leq 10, \\ -8x_1 + x_{10} &\leq 0, \\ -8x_2 + x_{11} &\leq 0, \\ -8x_3 + x_{12} &\leq 0, \\ -2x_4 - x_5 + x_{10} &\leq 0, \\ -2x_6 - x_7 + x_{11} &\leq 0, \\ -2x_8 - x_9 + x_{12} &\leq 0, \end{aligned}$$

and box constraints

$$\begin{aligned} 0 &\leq x_i \leq 1 && \text{for } i = 1, \dots, 9, \\ 0 &\leq x_i \leq 100 && \text{for } i = 10, \dots, 12, \\ 0 &\leq x_{13} \leq 1, \end{aligned}$$

with a minimum at

$$x^* = (1, 1, 1, 1, 1, 1, 1, 1, 1, 3, 3, 3, 1),$$

having value  $G_1(x^*) = -15$ .

#### 10.5.1.2 Test-problem $G_2$

This problem uses a nonlinear objective function, a linear constraint, and a nonlinear constraint. The dimension of the problem is scaleable. During our experiments a 20-dimensional problem instance was used. The problem is to maximize the function

$$G_2(\vec{x}) = \left| \frac{\sum_{i=1}^n \cos^4(x_i) - 2 \prod_{i=1}^n \cos^2(x_i)}{\sqrt{(\sum_{i=1}^n i x_i^2)}} \right|,$$

under the constraints

$$\begin{aligned}\prod_{i=1}^n x_i &\geq 0.75, \\ \sum_{i=1}^n x_i &\leq 7.5n,\end{aligned}$$

and box constraints

$$0 \leq x_i \leq 10 \quad \text{for } i = 1, \dots, n.$$

The best solution reported in literature has value  $G_2(\vec{x}) = 0.803553$  [MS96]. This solution was obtained by means of a GA that restricts the search to the boundary defined by the non-linear constraint. We found a better solution, which is not located at this boundary, i.e.

$$\begin{aligned}\vec{x}^b = & (3.1636147009232, 3.1285736318293633, 3.0958686047204558, 3.0560136161512346, \\ & 3.024386226254852, 2.994476533654338, 2.955710630137849, 2.924596337930907, \\ & 0.49515224295065546, 0.4873251273288, 0.47996310267256376, 0.4743587517049861, \\ & 0.4679452481653243, 0.471375117272565, 0.4631084808116345, 0.4620033707110338, \\ & 0.4534465260045891, 0.4490982581421707, 0.44331571099667527, 0.4370865239459931),\end{aligned}$$

having value  $G_4(\vec{x}^b) = 0.8036028872409808$ .

#### 10.5.1.3 Test-problem $G_4$

This problem involves a quadratic objective function and six nonlinear constraints. The problem definition is also given in Himmelblau [Him72] (problem 11) and in Homaifar, Qi and Lai [HQL94]. The problem is to minimize the function

$$G_4(\vec{x}) = 5.3578547x_3^2 + 0.8356891x_1x_5 + 37.293239x_1 - 40792.141,$$

under the constraints

$$\begin{aligned}0 &\leq 85.334407 + 0.0056858x_2x_5 + 0.00026x_1x_4 - 0.0022053x_3x_5 \leq 92, \\ 90 &\leq 80.51249 + 0.0071317x_2x_5 + 0.0029955x_1x_2 + 0.0021813x_3^2 \leq 110, \\ 20 &\leq 9.300961 + 0.0047026x_3x_5 + 0.0012547x_1x_3 + 0.0019085x_3x_4 \leq 25,\end{aligned}$$

and box constraints

$$\begin{aligned}78 &\leq x_1 \leq 102, \\ 33 &\leq x_2 \leq 45, \\ 27 &\leq x_i \leq 45 \quad \text{for } i = 3, \dots, 5,\end{aligned}$$

with the best known solution being [MS96, HQL94]

$$\vec{x}^* = (78, 33, 29.995, 45, 36.776),$$

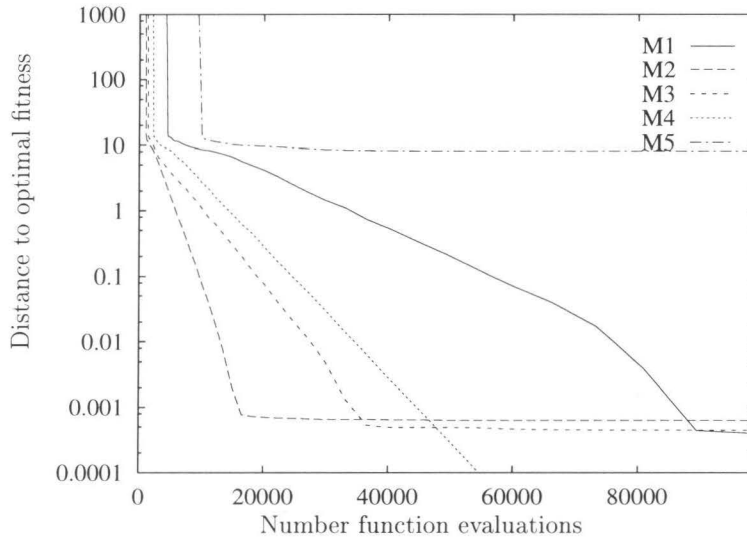
having value  $G_4(\vec{x}^*) = -30665.5$ .

We found a better solution i.e.

$$\begin{aligned}\vec{x}^b = & (78.00000353186088, 33.000011837015656, 27.071004256261414, \\ & 44.999998875082134, 44.96921990751058),\end{aligned}$$

having value  $G_4(\vec{x}^b) = -31025.559$ .



Figure 10.7: Convergence curves for problem  $G_1$ 

#### 10.5.1.4 Test-problem $G_6$

This is a minimization problem with a cubic objective function, and two nonlinear constraints. These two constraints define two circular boundaries that have a slightly different radius and are displaced a little. This results in a feasible region consisting of a narrow curved valley. This valley gets more narrow as one approaches one of the end points of the valley. The optimal solution is located at one such an end point of the feasible region. The task is to minimize the function

$$G_6(\vec{x}) = (x_1 - 10)^3 + (x_2 - 20)^3,$$

under the constraints

$$\begin{aligned} (x_1 - 5)^2 + (x_2 - 5)^2 &\geq 100, \\ (x_1 - 6)^2 + (x_2 - 5)^2 &\leq 82.81, \end{aligned}$$

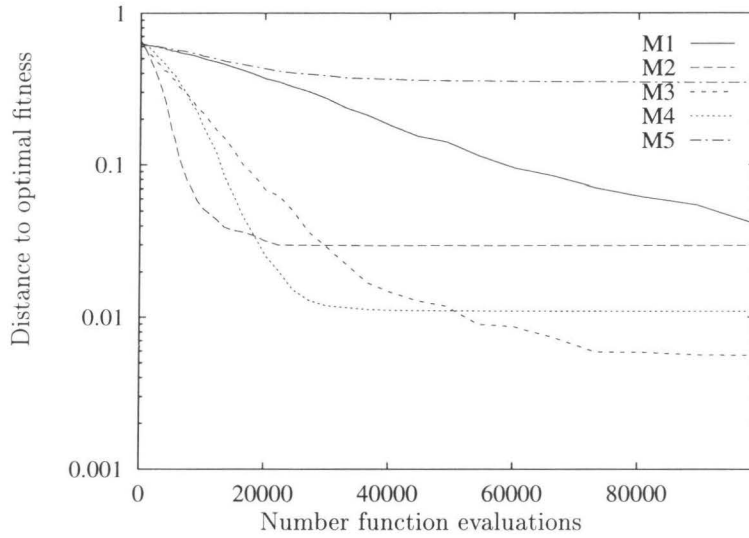
and box constraints

$$\begin{aligned} 13 &\leq x_1 \leq 100, \\ 0 &\leq x_2 \leq 100, \end{aligned}$$

with the optimum at

$$\vec{x}^* = (14.095, 0.84296),$$

having value  $G_6(\vec{x}^*) = -6961.81381$ .

Figure 10.8: Convergence curves for problem  $G_2$ 

#### 10.5.1.5 Test-problem $G_7$

This is a problem with a quadratic objective function, three linear constraints, and five nonlinear constraints. It is also presented in Hock et. al. [HS81] as problem 113. The task is to minimize the function

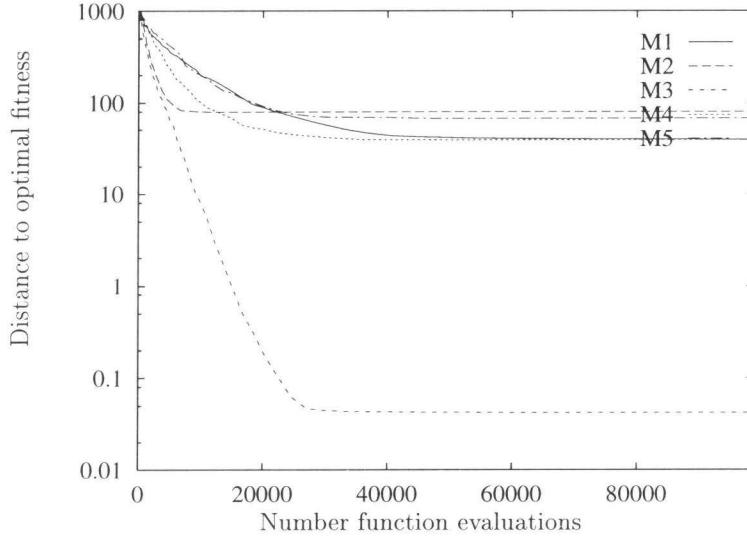
$$G_7(\vec{x}) = x_1^2 + x_2^2 + x_1x_2 - 14x_1 - 16x_2 + (x_3 - 10)^2 + 4(x_4 - 5)^2 + (x_5 - 3)^2 \\ + 2(x_6 - 1)^2 + 5x_7^2 + 7(x_8 - 11)^2 + 2(x_9 - 10)^2 + (x_{10} - 7)^2 + 45,$$

under the constraints

$$\begin{aligned} 105 - 4x_1 - 5x_2 + 3x_7 - 9x_8 &\geq 0, \\ -3(x_1 - 2)^2 - 4(x_2 - 3)^2 - 2x_3^2 + 7x_4 + 120 &\geq 0, \\ -10x_1 + 8x_2 + 17x_7 - 2x_8 &\geq 0, \\ -x_1^2 - 2(x_2 - 2)^2 + 2x_1x_2 - 14x_5 + 6x_6 &\geq 0, \\ 8x_1 - 2x_2 - 5x_9 + 2x_{10} + 12 &\geq 0, \\ -5x_1^2 - 8x_2 - (x_3 - 6)^2 + 2x_4 + 40 &\geq 0, \\ 3x_1 - 6x_2 - 12(x_9 - 8)^2 + 7x_{10} &\geq 0, \\ -0.5(x_1 - 8)^2 - 2(x_2 - 4)^2 - 3x_5^2 + x_6 + 30 &\geq 0, \end{aligned}$$

and box constraints

$$-10 \leq x_i \leq 10 \quad \text{for } i = 1, \dots, n,$$

Figure 10.9: Convergence curves for problem  $G_4$ 

having an optimum in the vicinity of

$$\vec{x}^* = (2.171996, 2.363683, 8.773926, 5.095984, 0.9906548, \\ 1.430574, 1.321644, 9.828726, 8.280092, 8.375927),$$

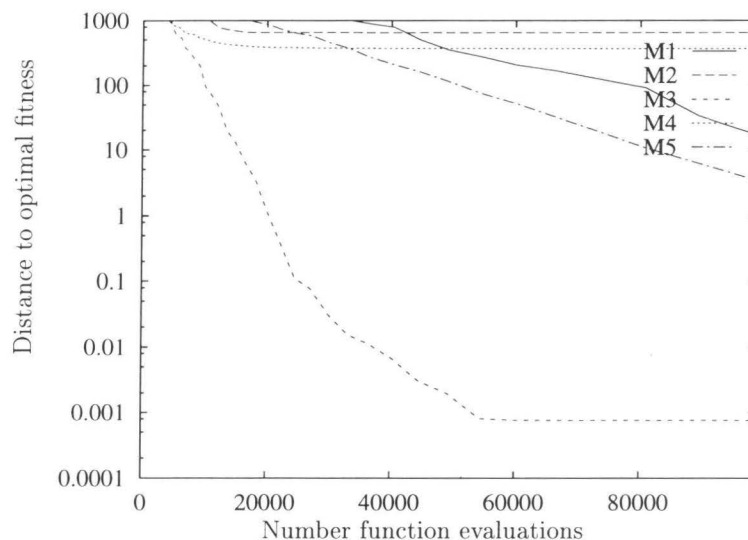
having value  $G_7(\vec{x}^*) = 24.3062091$ . This vector violates the constraints (probably due to the fact that the fields  $x_i^*$  are rounded at six decimal digits). We did a local search to find a good feasible solution. The best solution we found is

$$\vec{x}^b = (2.1719964442208672, 2.3636826986308432, 8.773925396935452, 5.095983794272395, \\ 0.9906550582295296, 1.430574792390953, 1.321644402301921, 9.828725992985344, \\ 8.280091841045932, 8.375926524435313),$$

having value  $G_7(\vec{x}^b) = 24.306209537412375$ .

#### 10.5.1.6 Test-problem $G_8$

This problem is a maximization problem with a nonlinear objective function and two nonlinear constraints. This problem was used by Schoenauer et al. [SX93]. The definitions given in [SX93] and [MS96] differ, and the (qualitative) problem description that is given does not correspond to either of the definitions, probably due to small typographic errors.

Figure 10.10: Convergence curves for problem  $G_6$ 

Here the following definition is used. Maximize the function

$$G_8(\vec{x}) = \frac{\sin^3(2\pi x_1) \sin(2\pi x_2)}{x_1^3(x_1 + x_2)},$$

under the constraints

$$\begin{aligned} x_1^2 - x_2 + 1 &\leq 0, \\ 1 - x_1 + (x_2 - 4)^2 &\leq 0, \end{aligned}$$

and box constraints

$$0 \leq x_i \leq 10 \text{ for } i = 1, 2.$$

The best solution is obtained with

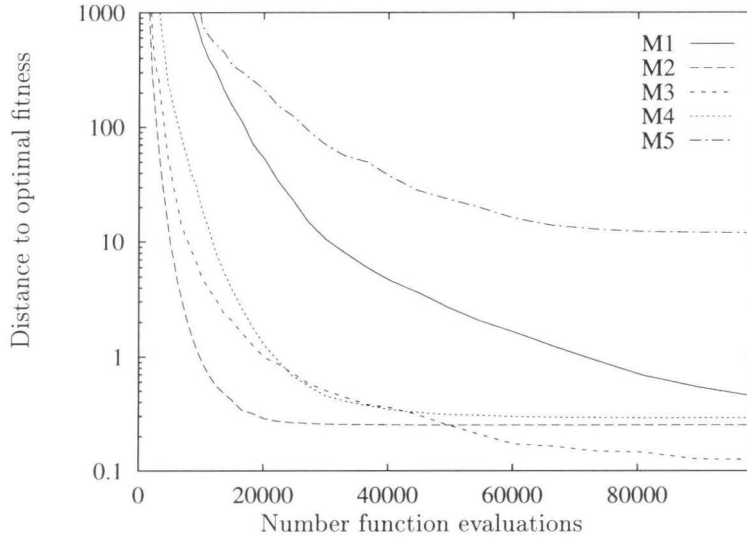
$$\vec{x}^b = (1.22797130420132, 4.245373062532398),$$

having fitness  $G_8(\vec{x}^b) = 0.095825$ .

#### 10.5.1.7 Test-problem $G_9$

This problem has a polynomial objective function and four nonlinear constraints. It is also presented in Hock et. al. [HS81] as problem 100. Minimize the function

$$\begin{aligned} G_9(\vec{x}) = & (x_1 - 10)^2 + 5(x_2 - 12)^2 + x_3^4 + 3(x_4 - 11)^2 + \\ & 10x_5^6 + 7x_6^2 + x_7^4 - 4x_6x_7 - 10x_6 - 8x_7, \end{aligned}$$

Figure 10.11: Convergence curves for problem  $G_7$ 

under the constraints

$$\begin{aligned}
 127 - 2x_1^2 - 3x_2^4 - x_3 - 4x_4^2 - 5x_5 &\geq 0, \\
 282 - 7x_1 - 3x_2 - 10x_3^2 - x_4 + x_5 &\geq 0, \\
 196 - 23x_1 - x_2^2 - 6x_6^2 + 8x_7 &\geq 0, \\
 -4x_1^2 - x_2^2 + 3x_1x_2 - 2x_3^2 - 5x_6 + 11x_7 &\geq 0,
 \end{aligned}$$

and box constraints

$$-10 \leq x_i \leq 10 \quad \text{for } i = 1, \dots, 7,$$

has an optimum at

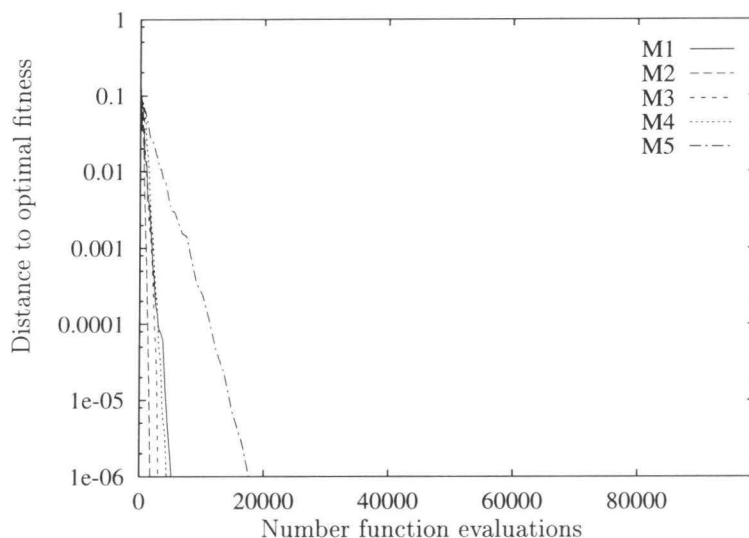
$$\vec{x}^* = (2.330499, 1.951372, -0.4775414, 4.365726, -0.6244870, 1.038131, 1.594227),$$

with value  $G_9(\vec{x}^*) = 680.6301112407559$ .

#### 10.5.1.8 Test-problem $G_{10}$

The problem has a linear objective function, three linear constraints, and three nonlinear constraints. It is also presented in Hock et. al. [HS81] as a heat exchanger design (problem 106). The task is to minimize the function

$$G_{10}(\vec{x}) = x_1 + x_2 + x_3,$$

Figure 10.12: Convergence curves for problem  $G_8$ 

under the constraints

$$\begin{aligned}
 1 - 0.0025(x_4 + x_6) &\geq 0, \\
 1 - 0.0025(x_5 + x_7 - x_4) &\geq 0, \\
 1 - 0.01(x_8 - x_5) &\geq 0, \\
 x_1x_6 - 833.33252x_4 - 100x_1 + 83333.333 &\geq 0, \\
 x_2(x_7 - x_4) + 1250(-x_5 + x_4) &\geq 0, \\
 x_3x_8 - 1250000 - x_3x_5 + 2500x_5 &\geq 0,
 \end{aligned}$$

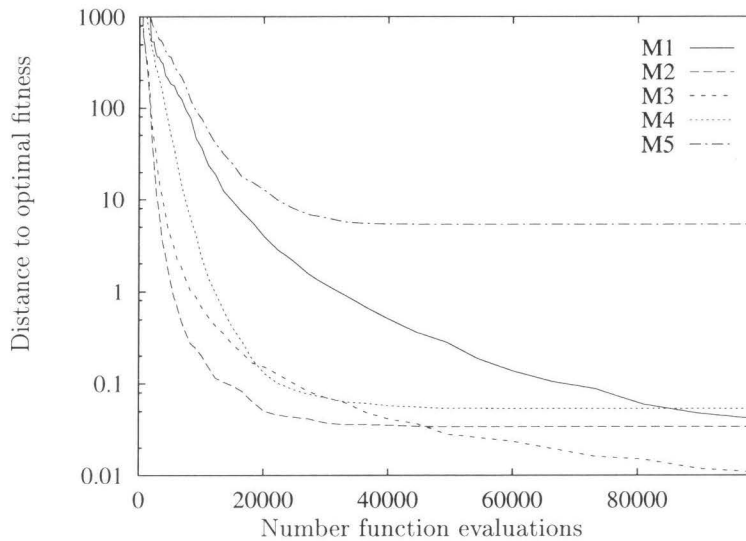
and box constraints

$$\begin{aligned}
 100 &\leq x_1 \leq 10,000, \\
 1,000 &\leq x_i \leq 10,000 \quad \text{for } i = 2, 3, \\
 10 &\leq x_i \leq 1,000 \quad \text{for } i = 4, \dots, 8,
 \end{aligned}$$

$\vec{x}^* = (579.3167, 1359.943, 5110.071, 182.0174, 295.5985, 217.9799, 286.4162, 395.5979)$ ,  
resulting in a value  $G_{10} = 7049.3307$ .

## 10.5.2 Results

In this section the results obtained when applying the different GA's to the problems in the test-suite are presented, and a brief comparison to results obtained by others is given

Figure 10.13: Convergence curves for problem  $G_9$ 

in the next section. The characteristics of the different test-problems are summarized in Table 10.1.

All experiments have been repeated sixty times. The results are presented by means of a set of plots showing the convergence curves. These convergence curves are determined by tracking the difference in fitness between the best performing individual and the optimal fitness as a function of the number of fitness evaluations during each run. Next a median curve is computed by taking the (point-wise) median over all these curves.

Figure 10.7 shows the results for problem  $G_1$ . Note that the y-axis uses a logarithmic scale. For problem  $G_1$  the GA M5 (that only uses a population of local optimizers and no crossover) fails to locate the optimum. All other GA's approach the optimum quite well. The GA M4 locates the optimum closest. Furthermore notice that M2 (which does not use a local optimizer) converges fastest.

Figure 10.8 shows the results for problem  $G_2$ . Again GA M5 fails to come close to the optimum. M2 converges fastest but the search stagnates when the fitness still is 0.02 from optimal.

Figure 10.9 shows the results for problem  $G_4$ . On this problem all GA's converge fast. Four out of five GA's get trapped at a large distance from the global optimum. Only GA M3 is able to come close to the optimal solution.

Figure 10.10 shows the results for problem  $G_6$ . The GA M3 performs best. The GA's M2 and M4 get trapped at a large distance of the global optimum. The GA's M1 and M5 do not get trapped, however these GA's converge quite slowly.

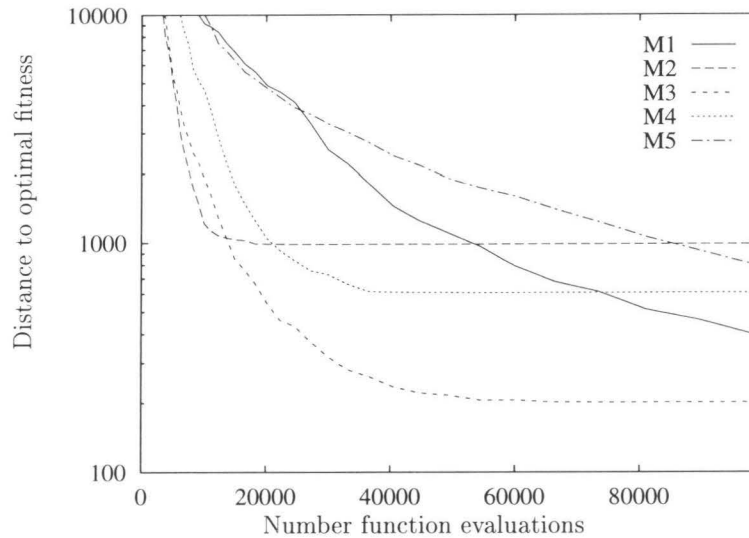
Figure 10.14: Convergence curves for problem  $G_{10}$ 

Figure 10.11 shows the results for problem  $G_7$ . The curve of GA M3 comes closest to the optimal fitness. GA M5 (the population of local optimizers) gets trapped at a large distance from the global optimum.

Figure 10.12 shows the results for problem  $G_8$ . On this problem the GA M2 converges fastest, followed in sequence by M3, M4, and M1. M5 converges much slower.

Figure 10.13 shows the results for problem  $G_9$ . The GA's M2, M4, and M5 get trapped. M5 already gets trapped at a large distance of the optimal fitness. M3 performs best of the two GA's that do not get trapped.

Figure 10.14 shows the results for problem  $G_{10}$ . The GA's M2, M3, and M4 get trapped. M3 comes closest to the optimal fitness. M1 performs best of the two GA's that did not get trapped at function evaluation number 100,000.

The next set of plots shows the same set of convergence curves, however this time the curves within a plot belong to different problems and were produced by the a single GA.

Figure 10.15 shows the results for GA M1 (CLES with  $\tau = 1.5$ ). This GA convergences relatively slowly and does not get trapped easily.

Figure 10.16 shows the results for GA M2 (CLES with  $\tau = 1.5$  and no local optimizer). The GA always converges fast, and it usually does not make much progress anymore after 20,000 function evaluations. This indicates that this GA might benefit from the usage of a larger population than the other GA's on this test-suite.

Figure 10.17 shows the results for GA M3 (CLES with  $\tau = 10$ ). This GA usually finds near optimal solutions but does not converge very fast.



| Problem | number<br>dimen. | Type of<br>$f(\vec{x})$ | optimal<br>objective | linear<br>constraints | nonlinear<br>constraints |
|---------|------------------|-------------------------|----------------------|-----------------------|--------------------------|
| G1      | 13               | minimize quadratic      | -15                  | 9                     | 0                        |
| G2      | 20               | maximize nonlinear      | <b>0.803603</b>      | 1                     | 1                        |
| G4      | 5                | minimize quadratic      | <b>-31,025.6</b>     | 0                     | 6                        |
| G6      | 2                | minimize cubic          | -6961.81381          | 0                     | 2                        |
| G7      | 10               | minimize quadratic      | 24.3062032           | 0                     | 5                        |
| G8      | 2                | maximize nonlinear      | 0.0958250414         | 0                     | 2                        |
| G9      | 7                | minimize polynomial     | 680.6300573          | 0                     | 4                        |
| G10     | 8                | minimize linear         | 7049.3307            | 3                     | 3                        |

Table 10.1: Characteristics of the test-problems (the optima shown in boldface are better than the best solutions found in literature).

Figure 10.15 shows the results for GA M4 ( $(\mu, \lambda)$  selection). This GA relatively often gets trapped at a large distance from the optimal solution. It converges relatively fast, but not as fast as the GA M2.

Figure 10.19 shows the results for GA M5 (the set of independent  $(1 + 1)$ -ES local optimizers). This GA often gets trapped at a large distance of the global optimal solution. In all other cases it converges relatively slow.

Table 10.2 shows performance measures that are based on the best solution found after the termination of the different GA's. Each column of this table corresponds to a single GA and each row corresponds to one of the test-problems. For each test-problems five rows are given. The first row, which is labelled *best*, shows the best individual obtained over all sixty independent runs. The rows labelled *low*, *median*, and *high* show the statistics for the difference between the best observed fitness and the fitness of the global optimum. The row labelled *median* shows the median best solution at termination over all sixty runs. The lines labelled *low* and *high* give the bounds on the 90% confidence interval. This interval is obtained taking the best solution of each of the sixty independent runs, sorting these solutions on fitness and removing the 5% best solutions and the 5% worst solutions. The two end points of the remaining sequence of fitness values are reported. The last line, labelled *feasible*, shows the percentage of runs for which the best solution is a feasible solution. For each of the rows labelled *best*, *median*, and *feasible* the best result is shown in bold-face.

## 10.6 Comparison to other methods

Test-problems from the test-suite presented in the previous section have been used to asses the performance of a large number of evolutionary approaches to constrained optimization [MS96]. Furthermore a comparison is made between seven different evolutionary algorithms on the test-problems G1, G7, G9, and G10.

| problem | M1              | M2              | M3             | M4                | M5                |
|---------|-----------------|-----------------|----------------|-------------------|-------------------|
| G1      | <i>best</i>     | -14.9997        | -14.9997       | -14.9999          | -15               |
|         | <i>low</i>      | 0.000278185     | 0.00045778     | 0.000291878       | $3 \cdot 10^{-8}$ |
|         | <i>median</i>   | 0.000397222     | 0.000626888    | 0.000447622       | $4 \cdot 10^{-8}$ |
|         | <i>high</i>     | 0.00061712      | 0.000868548    | 0.000631218       | $7 \cdot 10^{-8}$ |
|         | <i>feasible</i> | <b>100%</b>     | <b>100%</b>    | 98.3%             | <b>100%</b>       |
| G2      | <i>best</i>     | 0.790802        | 0.80358        | 0.80343           | <b>0.803603</b>   |
|         | <i>low</i>      | 0.016           | 0.003          | 0.0001            | -0.00004          |
|         | <i>median</i>   | 0.041           | 0.003          | <b>0.0056</b>     | 0.01              |
|         | <i>high</i>     | 0.10            | 0.09           | 0.03              | 0.03              |
|         | <i>feasible</i> | 100%            | 100%           | 100%              | 100%              |
| G4      | <i>best</i>     | -31023.9        | -31018.3       | <b>-31025.6</b>   | -31025.5          |
|         | <i>low</i>      | 4.7             | 21.6           | 0.04              | 7.0               |
|         | <i>median</i>   | 38.9            | 79.4           | <b>0.042</b>      | 38.8              |
|         | <i>high</i>     | 112.4           | 211.2          | 0.44              | 104.378           |
|         | <i>feasible</i> | 100%            | 100%           | 100%              | 100%              |
| G6      | <i>best</i>     | <b>-6961.81</b> | -6909.36       | <b>-6961.81</b>   | -6954.52          |
|         | <i>low</i>      | 0.00193931      | 152.467        | $2.5 \cdot 10^5$  | 123.531           |
|         | <i>median</i>   | 11.7058         | 643.803        | <b>0.00076073</b> | 365.341           |
|         | <i>high</i>     | 203.15          | 1572.97        | 0.0152746         | 1272.81           |
|         | <i>feasible</i> | <b>100%</b>     | <b>100%</b>    | <b>100%</b>       | <b>100%</b>       |
| G7      | <i>best</i>     | 24.5112         | 24.324         | <b>24.3195</b>    | 24.3634           |
|         | <i>low</i>      | 0.249132        | 0.0531871      | 0.0210181         | 0.0743209         |
|         | <i>median</i>   | 0.436249        | 0.250809       | <b>0.125341</b>   | 0.288405          |
|         | <i>high</i>     | 0.703945        | 0.854894       | 0.377214          | 1.31661           |
|         | <i>feasible</i> | <b>100%</b>     | <b>100%</b>    | <b>100%</b>       | <b>100%</b>       |
| G8      | <i>best</i>     | 0.09582504      | 0.09582504     | 0.09582504        | 0.09582504        |
|         | <i>low</i>      | 0.0             | 0.0014169      | 0.0               | 0.0               |
|         | <i>median</i>   | <b>0.0</b>      | 0.029          | <b>0.0</b>        | <b>0.0</b>        |
|         | <i>high</i>     | 0.0             | 0.228          | 0.0               | 0.0               |
|         | <i>feasible</i> | 100%            | 100%           | 100%              | 100%              |
| G9      | <i>best</i>     | 680.635         | <b>680.631</b> | 680.632           | 680.634           |
|         | <i>low</i>      | 0.0148805       | 0.0054169      | 0.0024507         | 0.018264          |
|         | <i>median</i>   | 0.0415661       | 0.0337994      | <b>0.0108197</b>  | 0.0535273         |
|         | <i>high</i>     | 0.0890875       | 0.232804       | 0.0349449         | 0.186989          |
|         | <i>feasible</i> | 100%            | 100%           | 100%              | 100%              |
| G10     | <i>best</i>     | 7139.87         | 7066.67        | <b>7056.52</b>    | 7094.35           |
|         | <i>low</i>      | 162.886         | 82.7099        | 32.1858           | 137.109           |
|         | <i>median</i>   | 395.008         | 986.571        | <b>201.291</b>    | 605.38            |
|         | <i>high</i>     | 765.559         | 4973.58        | 438.629           | 1924.59           |
|         | <i>feasible</i> | <b>100%</b>     | 98.3%          | <b>100%</b>       | <b>100%</b>       |

Table 10.2: Results obtained on the test-suite of constrained optimization problems (*best* reports the objective value, *low*, *median* and *high* report the distance to the optimum).

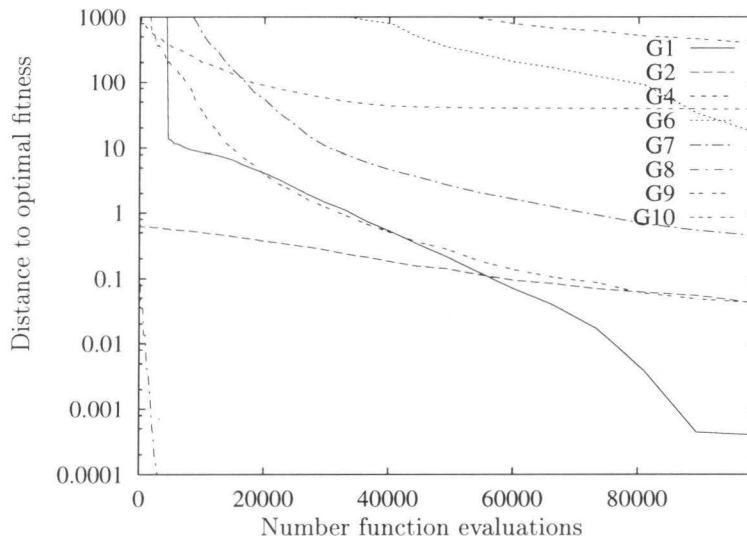


Figure 10.15: Convergence curves for method 1

On problem G2 the results are presented for a tailor-made genetic algorithm. The nonlinear constraint is taken, and the initial population is generated such that all points are located on the surface defined by this constraint, the mutation operator is defined such that if the parents are located on the surface defined by this constrained, then the offspring are located on this surface too. So, this is a problem-specific GA that only searches the surface defined by the nonlinear constraint. Both GA M2 and GA M4 located solutions that are superior to the best result reported in literature [MS96]. The best solution we obtained is located in the interior region, and therefore could not possibly be found by the problem specific GA that only search the boundary of the region. The problem specific GA found solutions with a fitness  $G_2(\vec{x}) > 0.8$  within 120,000 function evaluations (4000 generations and a population with 30 individuals). During our experiments a single run used at most 100,000 function evaluations, and the median result was above 0.8 for the GA's M2 and M3, showing that on average at least one out of two runs results in solutions with fitness  $G_2(\vec{x}) > 0.8$  (using 16% less function evaluations than the problem specific GA).

On the minimization problem G4 the best results we found are better than the results reported in literature. Homaifar, Qi, and Lai [HQL94] report results using a complex penalty method involving a set of penalty levels for each constraint. Their best result has a fitness  $G_4(\vec{x}) = -30005.7$ . The optimal solution from literature is  $G_4(\vec{x}) = -30665.5$  [MS96], and we obtained at result having fitness  $G_4(\vec{x}) = -31025.6$ .

Problem G6 has been handled by the Genocop II system (using an annealing of penal-

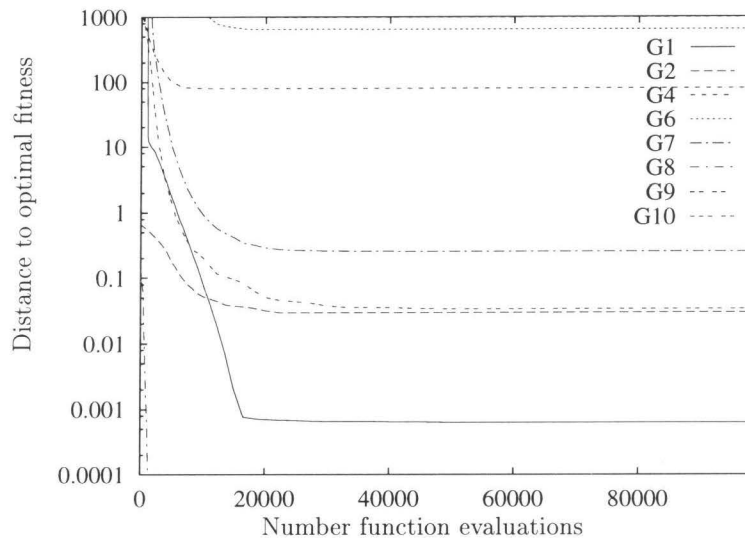


Figure 10.16: Convergence curves for method 2

ties). This system approached the optimum closely at iteration 12, finding the point (14.098, 0.849) that has objective value -6955.02 [MS96]. The GA's that use a local optimizer M1, M3, and M5 find the real optimum having fitness -6961.81.

The minimization problem G7 was handled with a GA using a death penalty method (infeasible solutions are removed from the population). The best solution found had a fitness  $G_7(\vec{x}) = 25.653$ , which still is 1.35 higher than the optimal solution [MS96]. The Genocop III system found a solution with fitness 25.883, which is even slightly worse [MS96]. This GA is outperformed by GA's M1, M2, M3, and M4, which all find better solutions. For these GA's the lower bound of the confidence interval is less than 1.35 above the optimum, meaning that 95% of the runs produced a better solution than the best run of the GA described by Michalewicz [MS96].

Schoenauer and Xanthakis developed the so-called behavioral memory method and tested it on problem G8 [SX93]. This method adds the constraints in sequence. First one searches for solutions that only satisfy the first constraint, next one searches for solutions that satisfy the first and the second constraint, etc. They use a penalty approach where a penalty term is added to the objective function in order to obtain the fitness. Small penalty terms result in convergence to locations in the infeasible region, large penalties make it difficult to find the feasible region [SX93]. Our GA's have no problems with problem G8, and all five GA's find the optimum rapidly.

The problem G9 was handled with the Genocop III system. The best result reported had a fitness of 680.640 [MS96]. The GA's M1-4 presented in this chapter found better

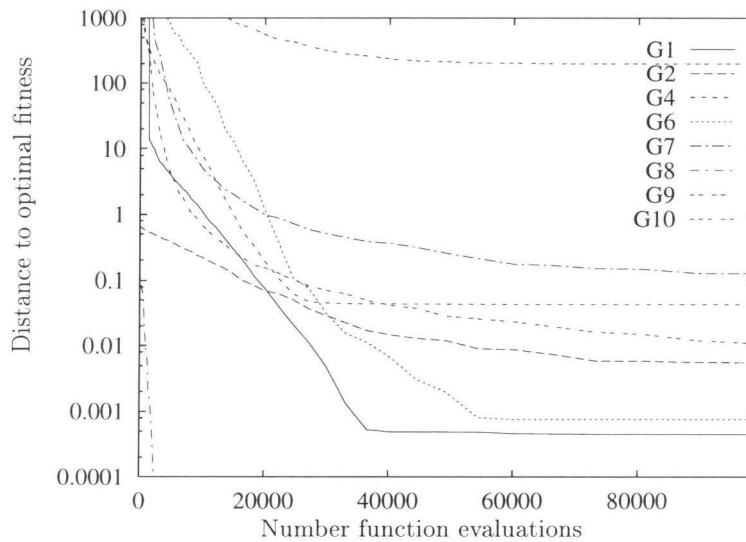


Figure 10.17: Convergence curves for method 3

solutions.

On the problem G10 the Genocop III system found 7286.650. The GA's M1-M5 gave better results.

Michalewicz and Schoenauer [MS96] give results for seven different evolutionary approaches on problems G1, G7, G9, and G10. The approaches are:

- MS1: GA with static penalties using a set of different penalty values for each constraint.
- MS2: GA with dynamical penalties, where the multiplier for the penalties is an increasing function of the number of constraints violated by the best solution in the previous generation.
- MS3: behavioral memory (adds constraints in sequence).
- MS4: annealing of constraints where the strength of constraints increases each generation.
- MS5: GA with superiority of feasible points. Feasible points are guaranteed to have a higher fitness than infeasible points.
- MS6: GA with death penalty (infeasible individuals are discarded).
- MS7: GA with death penalty and a initialization procedure that produces an initial population containing feasible solutions only.

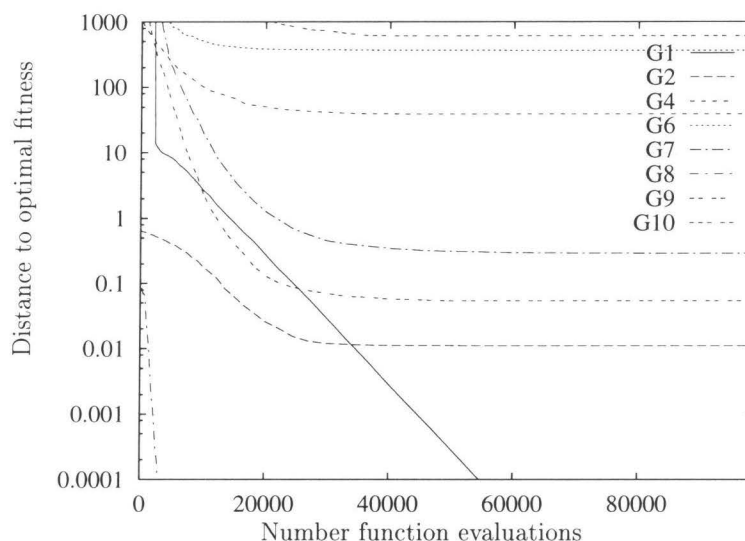


Figure 10.18: Convergence curves for method 4

Each method was applied ten times to each test-problem.

On problem G1 all methods find the optimum, although in case of method MS1 the median out of these ten runs still violates four constraints.

On problem G7 the GA MS4 and MS5 find a solution with a fitness smaller than the optimum fitness. The median solution still violates some of the constraints, so we have to assume that the best solutions violate some of the constraints, otherwise the authors probably would have reported this new superior solution. GA's MS1, MS2, and MS7 find solutions with fitnesses 24.690, 25.486, and 25.653 respectively. We assume that these solutions are feasible, although this is not stated explicitly by the authors. GA's M1, M2, M3, and M4 outperform the GA's MS1-5, and in case of M2, M3, and M4 the median solution has a fitness smaller than 24.69, which is the best solution generated by GA's MS1-5.

On problem G9 the method MS1 outperforms MS2-5, finding a best solution having fitness 680.771. This solution is 0.141 above the optimum. The best solutions obtained with GA's M1, M2, M3, and M4 are significantly better, and even the median solution obtained by these GA's are closer to the optimum (for example the median solution of M3 has a fitness of 680.641, which is 0.01 higher than the optimal objective value).

On problem G10 the GA MS4 performs best. It finds a solution having fitness 7377.976. GA's M1, M2, M3, and M4 perform better as at least 5% of the runs of these GA's locate a solution that is closer to the optimum. In case of M3 the median solution has a fitness 7250.622, which is even better than the best solution found by MS1-5.

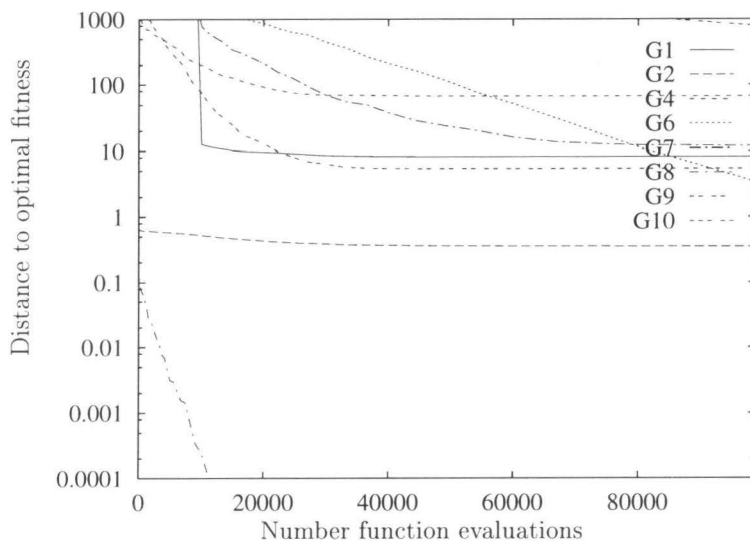


Figure 10.19: Convergence curves for method 5

## 10.7 Discussion of results on the large test-suite

The results produced by our GA's are promising. On two out of eight problems we have found better solutions than the best results reported in literature citeMS96. On problem G1 our GA's find the optimum easily, just like some other GA's. On the five remaining problems our GA's outperform the other evolutionary approaches.

Nonlinear inequality constraints are handled well: good results are produced and our approach is not tailored towards specific inequality constraints. Furthermore our method handles the constrained optimization problem as a kind of a black box problem, and the GA uses fitness values as the only information to guide the search. Such an approach is more general than the different problem specific GA's that require information about the different constraints. Our parameters were set by means of an educated guess, no further tuning of these parameters was performed, and all problems were handled with the same set of GA parameters. This suggests that our approach is quite robust, and not too sensitive to the actual settings of the different parameters.

The CLES was developed for unconstrained numerical optimizations. The algorithm was not changed for handling the constrained problems. Instead, the constraints were handled by a constrained-handling method that wraps all information about feasibility and objective value in a single fitness value, thereby casting the constrained problem to an unconstrained numerical optimization problem.

Our penalty approach is quite powerful, using a strong penalty for violating constraints,

a gradient term showing how strong these constraints are violated, and a scaled fitness-contribution for violated constraints, such that cross-competition between constraints is limited. The objective function is only computed for feasible solutions.

Local search helps a lot in finding good solutions for constraint optimization problems, although one has to take care that the local search does not consume too much time. Local search by itself does not seem powerful enough to handle complex constrained problems, because it easily gets trapped at a large distance of the global optimum (see results of method M5).

## 10.8 Summary

We introduced the stepping-stones problems SSP-a and SSP-b. These are nonlinear constraint optimization problems (with a scalable dimension  $d$ ) that have a feasible region consisting of  $2^d$  peaks (stepping-stones). The global optimum is located in the smallest peak.

Current results suggest that for this kind of problem it is advantageous for selection schemes to induce selective pressure by accepting only a limited number of offspring as parents of the next generation. Such a selection scheme outperforms selection schemes that accept all offspring and adjust the fertility based on (relative) fitness of the individuals. We expect this to be due to the fact that fertility based selection schemes use the individuals that represent infeasible solutions as parents too.

Within the current experiments the cluster-selection performs best. It can handle more difficult problems than the other selection schedules, and it does so using less function evaluations. This scheme has a high selective pressure and the application of the clustering step and the selection of representatives prevents the over-sampling of certain parts of the search-space and hence the probability of premature convergence is reduced.

The test-results on the larger test-suite are promising. On all eight problems our set of GA's M1-4 perform at least as good as the best other evolutionary approaches reported in literature. On seven of these problems all other evolutionary approaches are outperformed, and on two problems we even find optima that are better than the best results reported in literature. Given that different GA's perform with our penalty-function approach, it seems that our penalty-function approach is quite powerful. Furthermore, when comparing the different GA's we observe that the Cluster evolution strategies outperform other GA's, and that local optimization really is beneficial on the test-suite. However, the method based on local optimization only is inferior to the GA's tested in this chapter.



# Chapter 11

## Diagonal Crossover in Genetic Algorithms for Numerical Optimization

In this chapter the results of a detailed investigation on a multi-parent recombination operator, diagonal crossover, are reported. Although earlier publications have indicated the high performance of diagonal crossover on a number of problems, so far it has not been investigated whether high performance is indeed a result of using a high number of parents. Here we formulate three hypotheses to explain why GA performance increases when more parents are used. Based on an extensive study on a test-suite containing eight numerical optimization problems we are able to establish that the higher number of parents is indeed one of the sources of higher performance, if and when this occurs. By the diversity of the test functions (unimodal, multimodal, quasi-random landscapes) we can also make observations on the relationship between fitness landscapes and operator performance.

### 11.1 Introduction

Multi-parent recombination is a new research area within evolutionary computation. Although some researchers have incidentally proposed and applied recombination mechanisms using more parents, [BS92, BRS66, Müh89], the phenomenon of multi-parent recombination has not been given much attention in the past. In this chapter we study this phenomenon through investigating the behaviour of diagonal crossover (see the definition in section 11.2). Our research goals are two-fold.

1. We try to find connections between the structure of the fitness landscape and the performance of diagonal crossover. In particular we want to establish on what kind of landscapes it is advantageous to increase the number of parents. (Diagonal crossover for two parents is identical to the traditional one-point crossover operator.)
2. We try to disclose the source of increased performance of the diagonal crossover with

more parents if and when it is superior to two-parent recombination.

Further elaboration of the second research objective requires technical details, therefore, we return to this issue after the exact definition of the operator. The rest of the chapter is organized as follows. In the following section we briefly review multi-parent recombination operators in Evolutionary Algorithms. After the exact definition of diagonal crossover we formulate three hypotheses that can clarify why the performance of the GA increases when the number of parents in diagonal crossover is increased. To this end we design experiments that allow rejection or confirmation of these hypotheses. In section 11.3 we present the GA used in the experiments and discuss the performance measures to be used to monitor GA performance. The test-suite and the results of the experiments on each test function are presented in section 11.4. Finally, in section 11.5 we evaluate the results and draw conclusions.

## 11.2 Multi-parent recombination

In evolution strategies (ES) global recombination is a multi-parent operator, [Bäc96, Sch95]. This operator creates a new value in the child chromosome based on two parents, but randomly chooses two parents for each variable anew. By this particular mechanism the number of parents is undefined, thus investigations on the effects of different number of recombinants on algorithm performance could not be performed in the traditional ES framework. (Let us note that in [SR95] an extension of ES is proposed that allows tuning of the number of recombinants.) So far there are almost no experimental results available on the (dis)advantages of global recombination with respect to usual, two-parent recombination. Schwefel briefly touches on this issue in [Sch95] stating that “appreciable acceleration” is obtained by changing to bisexual from asexual scheme (i.e. adding recombination using two parents to the mutation-only algorithm), but only “slight further increase” is obtained when changing from “bisexual to multisexual recombination” (i.e. using global recombination instead of the two-parent variant).

Related work of Beyer, [Bey95], generalizes the traditional ES recombination operators by introducing the number of parents as an independent parameter  $\rho$ . The resulting  $(\mu/\rho, \lambda)$  evolution strategy is studied for the special case of  $\rho = \mu$  and theoretical analysis on the spherical function shows an advantage of using more than two parents.

Global recombination in ES also fertilized Genetic Algorithms. The gene-pool recombination of Mühlenbein and Voigt mixes information of possibly more parents by a similar mechanism as global recombination in ES, [MV95, VM95]. Hence, the number of parents is not defined here either. Mühlenbein and Voigt report an increase of performance when using gene-pool recombination (GPR) instead of two-parent recombination (TPR). GPR is showed to be approximately 25% faster than TPR on the ONEMAX problem, and the fuzzyfied GPR outperforms TPR on the spherical function in speed and in realized heritability.

The  $N$ -parent generalizations of the traditional one-point crossover and uniform crossover in GAs were introduced in [ERR94]. The resulting diagonal, respectively, scan-

ning crossover have the number of parents as parameter. This tunability is new feature compared to global recombination and gene pool recombination, where the multi-parent option can only be switched on or off, but it is not scalable. Several studies, cf. [EvKK95, ERR94, ES96, vKKE95], have shown that using more than two parents in either crossover mechanism can increase GA performance, although this does not hold for every problem and the two operators can respond differently to increasing the number of parents.

The main subject of the present investigation, diagonal crossover, generalizes one-point crossover for  $N$  parents by selecting  $(N - 1)$  crossover points and composing  $N$ -children by taking the resulting in  $N$  chromosome segments from the parents “along the diagonals”. The idea is illustrated for  $N = 3$  in Figure 11.1, left.

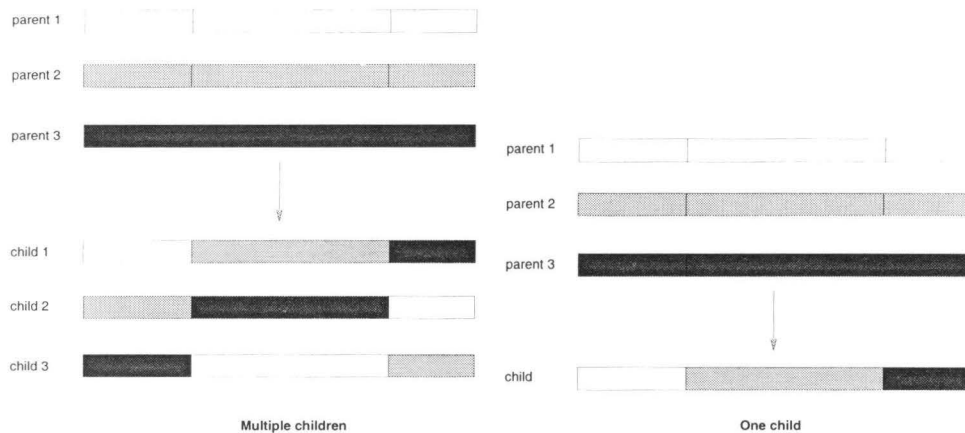


Figure 11.1: Diagonal crossover with three parents and three children (left) and with three parents and one child (right).

To investigate our second research objective let us make the following observations. First, the increase in the number of parents automatically leads to an increased number of crossover points. It can be the case that higher performance for higher  $N$ 's is not the result of using more parents, but simply comes from being more disruptive by using more crossover points. This forms our first working hypothesis.

**H<sub>1</sub>** Using more crossover points leads to better performance.

Second, notice that applying diagonal crossover,  $N$  parents create  $N$  children in one go. Since we use a steady state GA and update the population, i.e. insert offspring, after each application of crossover (followed by mutation), this means that a steady state GA using 10-parents diagonal crossover has processed more offspring before performing the selection step than a GA using the two-parents version. In other words, GAs with higher operator arity have a bigger generational gap which might cause a bias in their favour. Our second working hypothesis is accordingly the following.

|                         |                                     |
|-------------------------|-------------------------------------|
| representation          | fixed point binary with Gray coding |
| GA type                 | steady state                        |
| parent selection        | uniform random                      |
| deletion mechanism      | worst fitness deletion              |
| number of parents       | 2-30                                |
| crossover rate $p_c$    | 1.0                                 |
| mutation rate $p_m$     | 1/chromosome length                 |
| population size         | 500                                 |
| termination criterion   | population converged OR optimum hit |
| max. nr. of evaluations | 100000                              |
| results averaged over   | 50 independent runs                 |

Table 11.1: GA setup used in the experiments

**H<sub>2</sub>** Bigger generational gap leads to better performance.

Finally, we maintain our original conjecture that the advantages of using diagonal crossover with higher arities are not the results of an unintended artifact.

**H<sub>3</sub>** Using more parents leads to better performance.

Note that the hypotheses  $H_1$ ,  $H_2$  and  $H_3$  are not mutually exclusive as there might be more sources of increased GA performance when increasing  $N$  in diagonal crossover. The main contribution of the present chapter is that these sources are investigated in isolation hence providing a solid ground to check whether higher performance for higher  $N$ 's is an artifact ( $H_1$ ,  $H_2$ ), or the higher number of parents is indeed advantageous.

## 11.3 Experiment setup

All experiments are executed using a GA setup as described in Table 11.1. A non-standard option is the uniform random parent selection mechanism, whereby no selective pressure is applied when choosing recombinants. The motivation comes from [vKKE95], where we observed that this mechanism is preferable. Note, that a steady-state GA with uniform random parent selection mechanism is close to an evolution strategy, although using bit-string representation and no self-adaptation.

In order to test the working hypotheses presented in section 11.2 we run experiments a set of different crossover operators. For investigating  $H_1$  we apply the traditional two-parent two-children  $N$ -point crossover [DS92]. This operator is well known from the literature, therefore we omit a definition. If  $N$ -point crossover exhibits increasing performance when increasing  $N$ , (the experimental results reported in [EvKK95] make us expect this) then we accept  $H_1$ . To test the second hypothesis  $H_2$ , we will apply a slightly modified version of diagonal crossover that creates only one child. The right hand side of Figure 11.1 illustrates

this operator. Using the one-child version of diagonal crossover the generational gap does not increase when increasing the number of parents. If the original variant outperforms the one-child version of diagonal crossover, then we accept the hypothesis  $H_2$ . Concerning Hypothesis  $H_3$ , note that the number of chromosome segments using  $N$ -point crossover is  $N + 1$ , which equals the number of chromosome segments obtained by diagonal crossover for  $N + 1$  parents. This means that the disruptiveness of these operators grows parallelly as  $N$  increases. If higher disruptiveness increases GA performance on our test suite, then the performance of both the  $N$ -point crossover and the diagonal crossover will increase with increasing  $N$ . This, however, does not imply that more parents have no additional advantage as the performance of diagonal crossover might grow faster with increasing  $N$  than that of the two-parent  $N$ -point crossover. We accept the hypothesis  $H_3$  if diagonal crossover for  $N + 1$  parents is better than  $N$ -point crossover.

To evaluate different GA setups, that is the effect of different number of parents, respectively crossover-points, several performance measures of a run are monitored. The two main performance measures are accuracy and speed. Accuracy is measured by the error at termination. Since all functions have a minimum of zero, we use the best objective function value at termination as the accuracy measure of one run, and the median of the best objective function values, calculated over the 50 independent runs, as the accuracy belonging to a specific setting. For practical purposes we consider  $10^{-10}$  as zero and terminate the run if this value is achieved. Let us remark that using medians instead of average values has an advantage, namely medians are less sensitive for outliers in the data. On the other hand, if the optimum is found in the majority of the runs, then the median will equal the optimum. Additionally to the medians of the outcomes we also present the 90% confidence interval bars to the performance curves. The second main performance measure is the speed of the algorithm, measured by the median number of fitness evaluations before termination. If the GA with a certain setup never finds the optimum, this value equals the maximum number of fitness evaluations. A third performance measure is the success rate, i.e. the percentage of runs where the optimal objective function value has been found. We will present figures on success rates and 90% confidence intervals, whenever the accuracy or the speed curves are (nearly) constant, thus providing (almost) no basis to compare different settings. Finally, for a detailed insight in the behaviour of the GA sometimes we also depict the progress curves of the evolution for 18 parents (diagonal crossover), respectively 17 crossover points ( $N$ -point crossover). These curves (with a logarithmic  $y$ -axis) show the populations best objective function value as a function of the number of executed fitness evaluations, averaged over 50 independent runs.

## 11.4 Experiments

Experiments have been conducted on eight numerical function optimization problems. Each function is to be minimized and is scaled to have an optimal function value of zero. The fitness landscapes defined by these functions have various characteristics, unimodal, multimodal and quasi-random, i.e. very rugged with randomly distributed local optima.

Additionally, some of the functions are separable, while others are not. The exact definitions will be given in the corresponding subsections, here we only give a summary on their separability, the dimensions and the representation used in the experiments. As default, we use 20 bits for representing a single variable, but deviate from this value for F1, F2 and F8. For F1 and F2 we use the values originating from DeJong, for F8 30 bits are used, following Bäck, [Bäc96]. A concise treatment on numerical optimization problems as test functions can be found in [BM97].

|               | F1 | F2 | F3  | F4  | F5  | F6  | F7  | F8  |
|---------------|----|----|-----|-----|-----|-----|-----|-----|
| separable     | y  | n  | n   | n   | y   | y   | y   | n   |
| dimension     | 3  | 2  | 30  | 10  | 10  | 10  | 10  | 30  |
| chrom. length | 30 | 22 | 600 | 200 | 200 | 200 | 200 | 900 |

Table 11.2: Properties of the test functions

### 11.4.1 Spherical function

The first test function is the spherical function:

$$F1(\vec{x}) = \sum_{i=1}^n x_i^2,$$

where  $-5.12 \leq x_i \leq 5.12$ . This function is one of the most widely used objective functions in Evolutionary Computation, especially for convergence velocity evaluation. It has a unimodal, smooth fitness surface and is separable, making optimization rather easy. We tested the classical version of DeJong with  $n = 3$ . The GA found the optimum with every setting (every operator, for every value of  $N$ ). Therefore we omit accuracy and success rate data, only presenting the speed curves in Figure 11.2.

From the speed curves it turns out that the two variants of diagonal crossover show almost identical behaviour and both are faster than  $N$ -point crossover. Furthermore, it seems that there is a limit to increasing  $N$ : approximately up to 6 it leads to performance increase, thereafter the performance begins to deteriorate.

### 11.4.2 Rosenbrock's saddle function

F2 is the saddle function after Rosenbrock:

$$F2(\vec{x}) = 100 \cdot (x_1^2 - x_2)^2 + (1 - x_1)^2,$$

where  $-2.048 \leq x_i \leq 2.048$ . The global minimum is zero at the point  $(1, 1)$ . The Rosenbrock function is not separable and the unimodal fitness landscape is characterized by an extremely deep valley along the parabola  $x_1^2 = x_2$ .

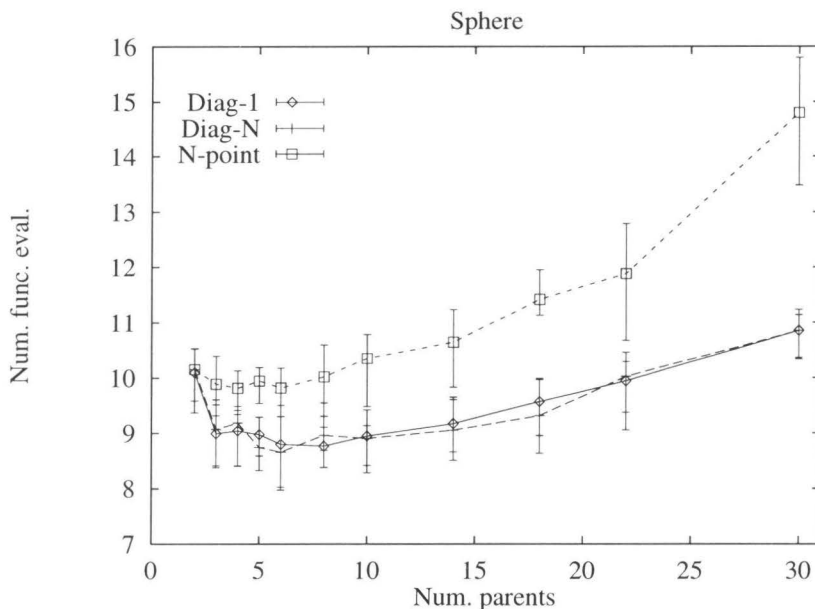


Figure 11.2: Speed curves on the spherical function

Recall from Table 11.2 that we use the classical DeJong setting with chromosome length 22 for F2. Therefore, the maximum number of parents is lowered accordingly in these experiments.

The accuracy and speed curves suggest that increasing  $N$  decreases the performance. The success rate curves in Figure 11.4 disclose that this is only partly true. The optimization performance grows with  $N$  for  $N$ -point crossover (up to  $N = 8$ ), but deteriorates for the diagonal crossovers.  $N$ -point crossover outperforms both diagonal crossovers with respect to each performance measure. The two variants of diagonal crossover are practically identical on F2.

### 11.4.3 Ackley function

Our third test function F3 is the Ackley function:

$$F3(\vec{x}) = 20 + e - 20 \exp \left( -0.2 \sqrt{\frac{1}{n} \sum_{i=1}^n x_i^2} \right) - \exp \left( \frac{1}{n} \sum_{i=1}^n \cos(2\pi x_i) \right),$$

where  $-30 \leq x_i \leq 30$ . The global minimum of zero is at the point  $\vec{x} = (0, 0, 0, \dots)$ . This function is not separable and at a low resolution the fitness landscape is unimodal,

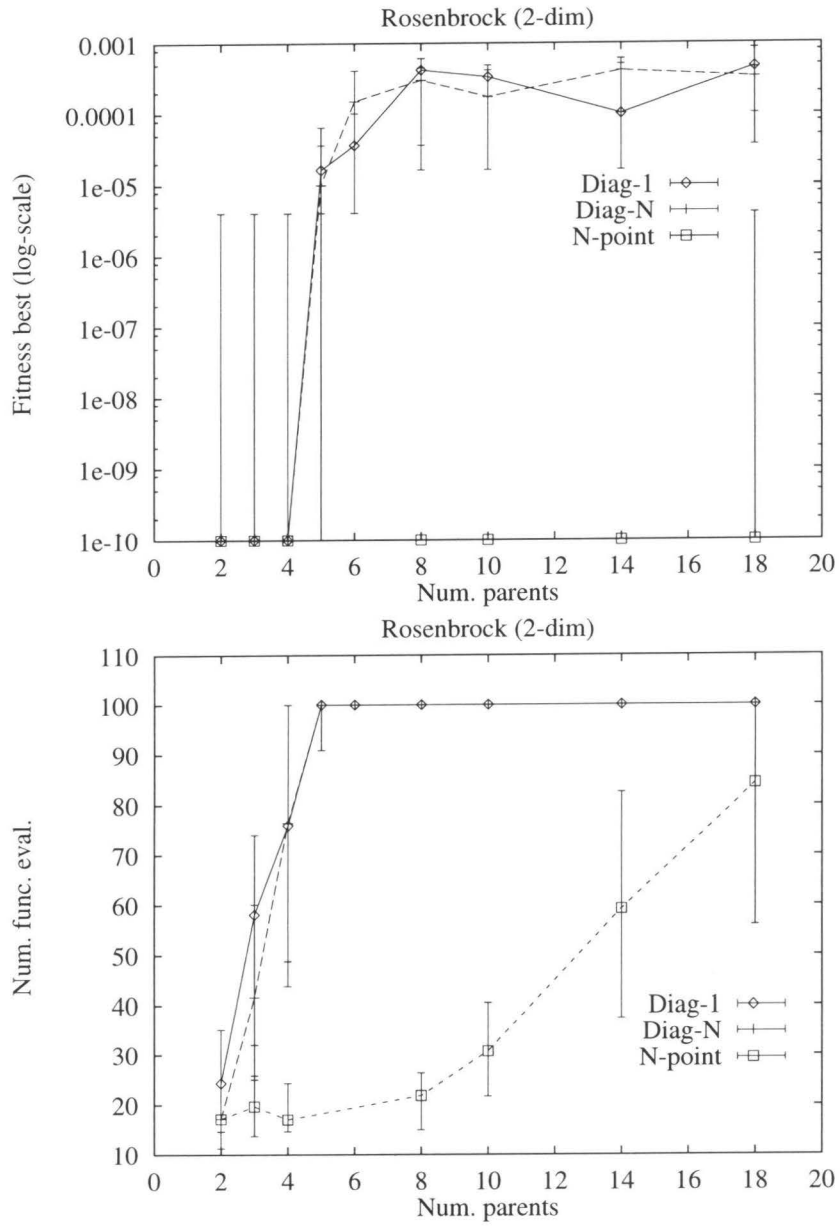


Figure 11.3: Accuracy (left) and speed (right) on the Rosenbrock function



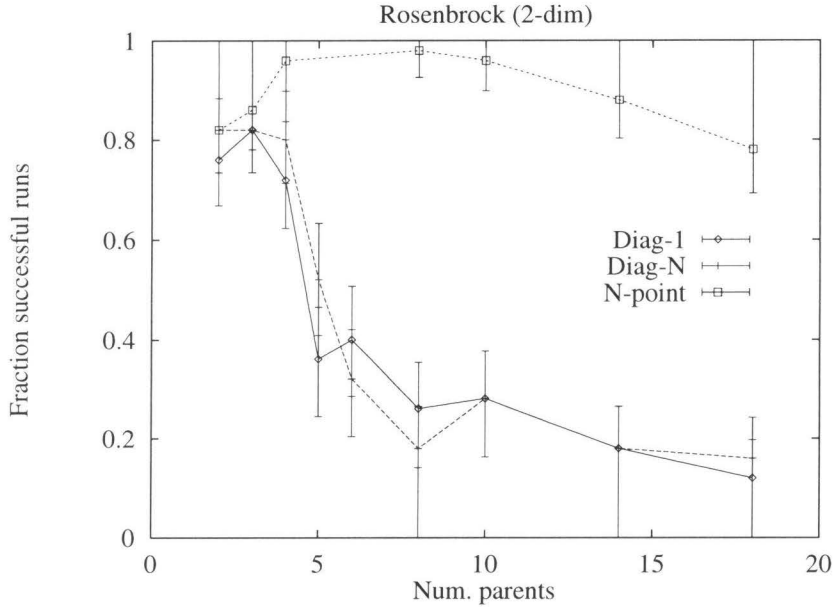


Figure 11.4: Success rates on the Rosenbrock function

but the second exponential term covers the landscape with many small peaks and valleys, i.e. many local optima. We tested F3 for  $n = 30$  and observed that the GA never found the optimum. Accordingly, the speed and the success rate curves are constant, therefore omitted here. We present the accuracy curves in Figure 11.5.

The effect of higher  $N$ 's is clear from the accuracy curves. Increasing  $N$  is advantageous for each operator up to the upper limit we tested. The one-child and the  $N$ -children versions of diagonal crossover perform identically also on this function, and both diagonal crossovers are consistently better than  $N$ -point crossover.

#### 11.4.4 Griewangk function

F4, the Griewangk function is defined as follows.

$$F4(\vec{x}) = 1 + \sum_{i=1}^n \frac{x_i^2}{400n} - \prod_{i=1}^n \cos\left(\frac{x_i}{\sqrt{i}}\right),$$

where  $-600 \leq x_i \leq 600$ . The global minimum of zero is at the point  $\vec{x} = (0, 0, 0, \dots)$ . This function has a product term introducing an interdependency between the variables, thus it is not separable. F4 was tested for 10 dimensions, the results are exhibited in Figure 11.6.

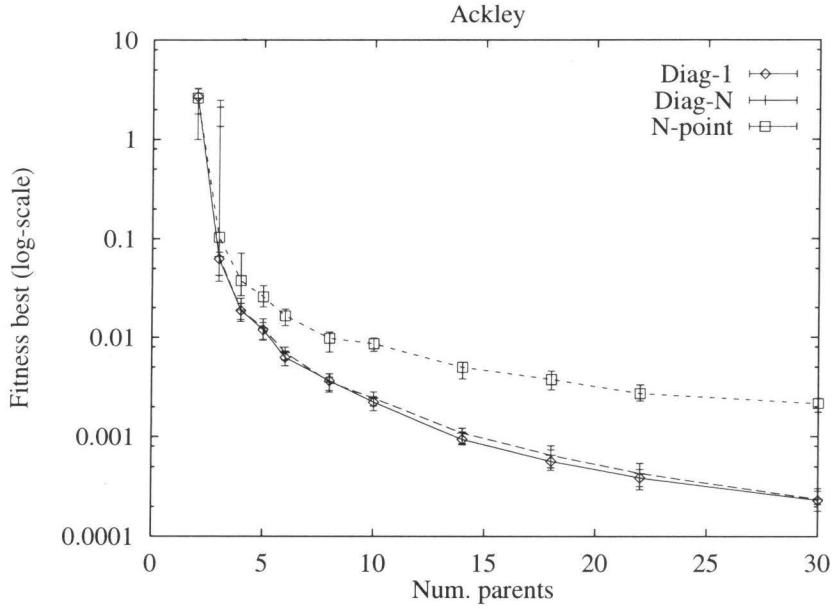


Figure 11.5: Accuracy curves on the Ackley function

On this function the advantages of higher  $N$ 's are clear, but the performance increase of accuracy stops at about  $N = 15$ . While the accuracy curves show only modest differences between the operators, the results on the speed of the algorithm disclose that the diagonal crossovers are significantly faster after  $N = 5$ . The two variants of diagonal crossover do not differ significantly.

#### 11.4.5 Michalewicz function

The fifth test function is taken from Michalewicz, [BDL<sup>+</sup>96].

$$F5(\vec{x}) = - \sum_{i=1}^n \sin(x_i) \cdot \sin^{2n} \left( \frac{ix_i^2}{\pi} \right),$$

where  $0 \leq x_i \leq \pi$ . We tested F5 for  $n = 10$  and observed that the GA found the optimum in the majority of the runs. Hence the medians of the accuracy results are equal to the optimal value. Therefore we rather present the success rates instead of accuracy data.

Increasing  $N$  above 2 on the Michalewicz function results in the highest gains so far. The success rates show a spectacular grow from approximately 30% for two parents to 80-90% for  $N$  between 5-20 and the GA becomes approximately four times faster for  $N = 5 - 10$ ,

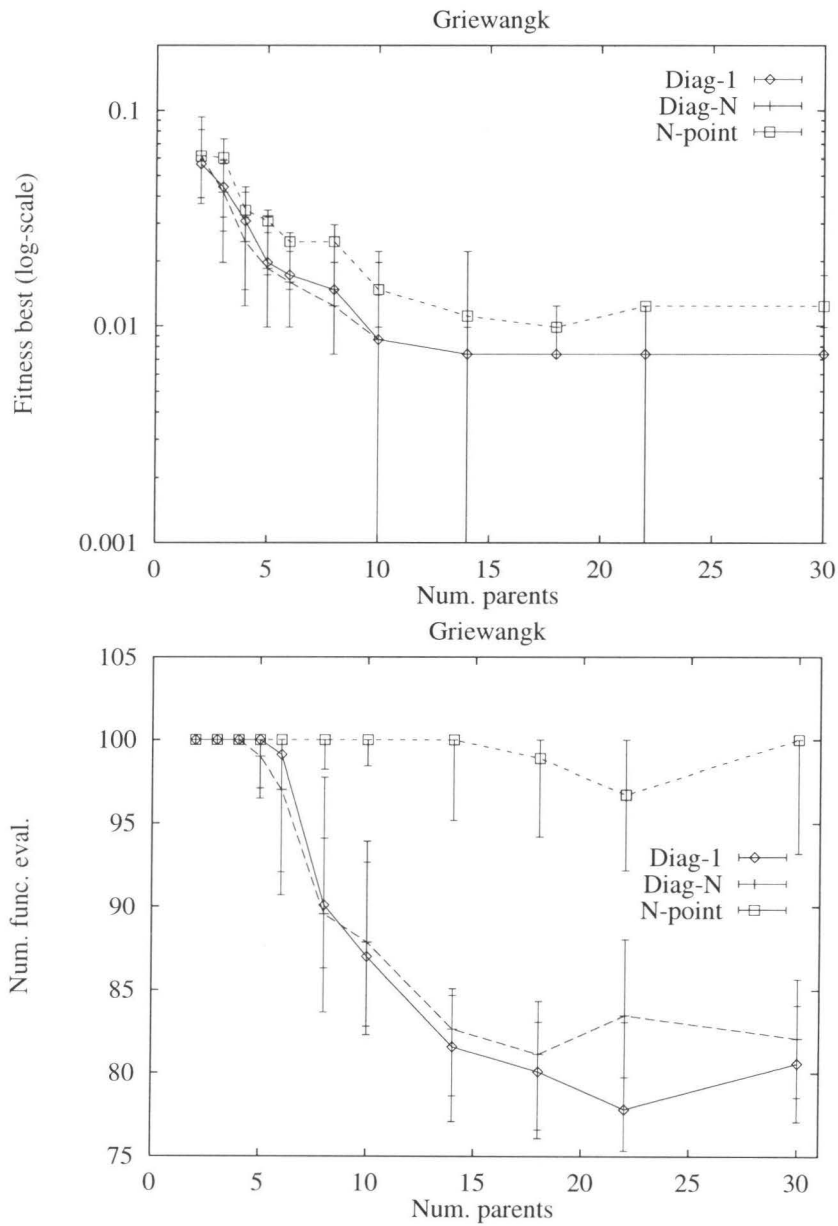


Figure 11.6: Accuracy (left) and speed (right) on the Griewangk function

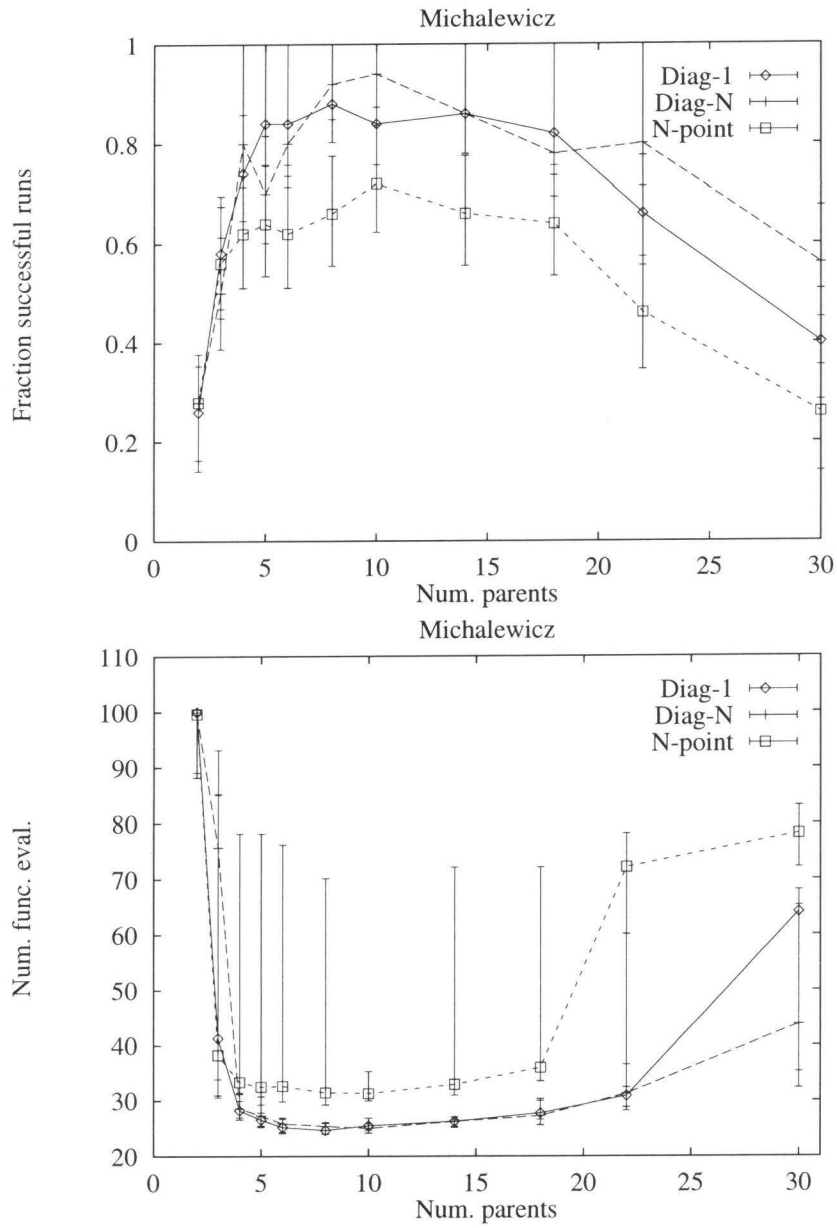


Figure 11.7: Success rates (left) and speed (right) on the Michalewicz function

than for  $N = 2$ . Comparing the operators we see again the superiority of diagonal crossover and no significant difference between the one-child and the  $N$ -child version.

### 11.4.6 Rastrigin function

F6 is the Rastrigin function:

$$F6(\vec{x}) = \alpha n + \sum_{i=1}^n x_i^2 - \alpha \cos(2\pi x_i),$$

where  $-5.12 \leq x_i \leq 5.12$ . The global minimum of zero is at the point  $\vec{x} = (0, 0, \dots)$ . This function is separable and its primary characteristic is the existence of many suboptimal peaks whose values increase as the distance from the global optimum point increases. In our tests we used  $\alpha = 10.0$  and  $n = 10$ .

Since many runs found the optimum, accuracy figures are replaced by success rates curves (notice the 0.65 - 1.0 scaling in Figure 11.8, left). These show that increasing  $N$  is advantageous, but the differences between various operators are small. Looking at the speed curves the effect of higher  $N$ 's and the differences between operators become clear. We can observe that each operator becomes better for higher  $N$ 's and that the two diagonal crossovers (identical again) outperform  $N$ -point crossover.

### 11.4.7 Schwefel function

F7 is obtained by generalizing Schwefel's 2.26 function (cf. [Sch95], pg. 344):

$$F7(\vec{x}) = 418.9829n - \sum_{i=1}^n x_i \sin\left(\sqrt{|x_i|}\right),$$

where  $-512.03 \leq x_i \leq 511.97$ . The global minimum of zero is at the point  $\vec{x} = (420.9687, 420.9687, \dots)$ . Although this function is separable, it is interesting because of the presence of a second-best minimum far away from (in the "opposite corner" to) the global minimum. This feature, just like two-peaks landscapes, makes the GA sensitive for early commitment with respect to the search direction. F7 was tested for  $n = 10$  and turned out to be easy. Nearly all runs ended with the global optimum, implying that accuracy and success rates would give no information for comparing the operators. The results on speed, however, show that the GA performance quickly and consequently improves when increasing  $N$  from 2 to approximately 10-15, and stagnates thereafter. The algorithm becomes approximately twice as fast for high  $N$ 's as for  $N = 2$ . Once again, there is no significant difference between the two diagonal crossovers that both outperform  $N$ -point crossover.

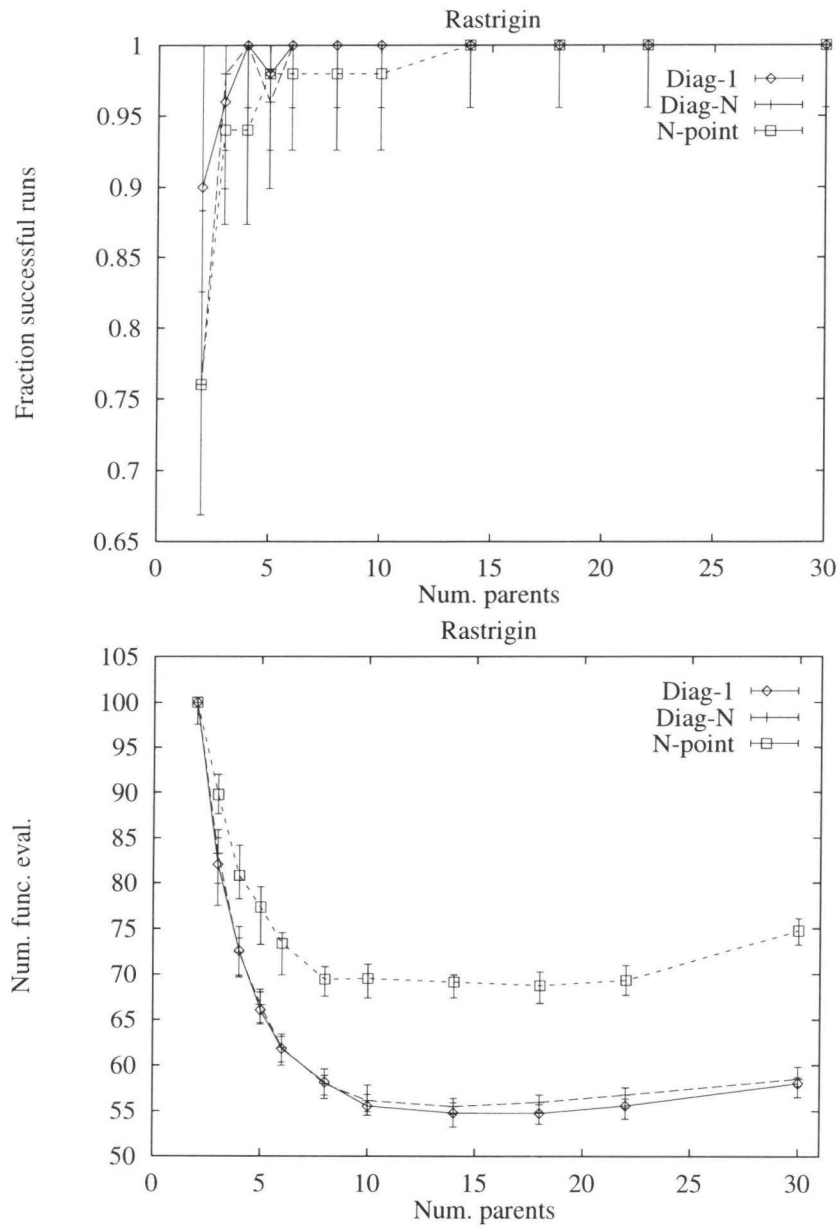


Figure 11.8: Success rates (left) and speed (right) on the Rastrigin function

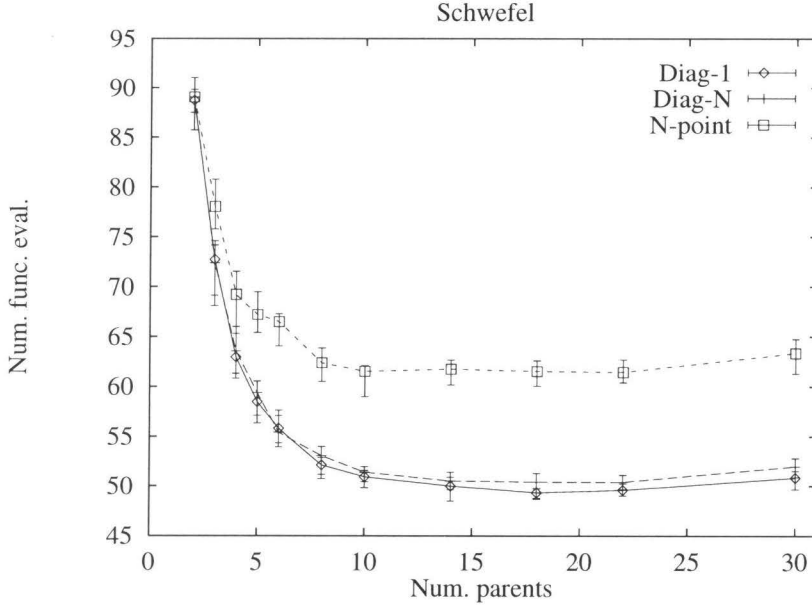


Figure 11.9: Speed curves on the Schwefel function

#### 11.4.8 Fletcher-Powell function

F8, the Fletcher-Powell function is retrieved from [Bäc96]:

$$\begin{aligned}
 F8(\vec{x}) &= \sum_{i=1}^n (A_i - B_i)^2 \\
 A_i &= \sum_{j=1}^n (a_{ij} \sin \alpha_j + b_{ij} \cos \alpha_j) \\
 B_i &= \sum_{j=1}^n (a_{ij} \sin x_j + b_{ij} \cos x_j),
 \end{aligned}$$

where  $n = 30$ ,  $n_\sigma = 30$ , and  $-\pi \leq x_i^0 \leq \pi$ . The  $a_{ij}, b_{ij} \in \{-100, \dots, 100\}$  are random integers, and  $\alpha_j \in [-\pi, \pi]$  is the randomly chosen global optimum position. For the matrices  $\mathbf{A}, \mathbf{B}$  and the vector  $\vec{\alpha}$  we used the values given in [Bäc96] (pp. 265–267).

No runs found the optimum on this function, resulting in constant speed and success rate curves. Accuracy curves reveal differences between performance for different  $N$ 's, showing advantageous effects of higher  $N$ 's, up to approximately 10. The three operators, however, hardly differ in performance and, as the progress curves for  $N = 18$  in Figure 11.10 indicate, their search behaviour is very similar too.

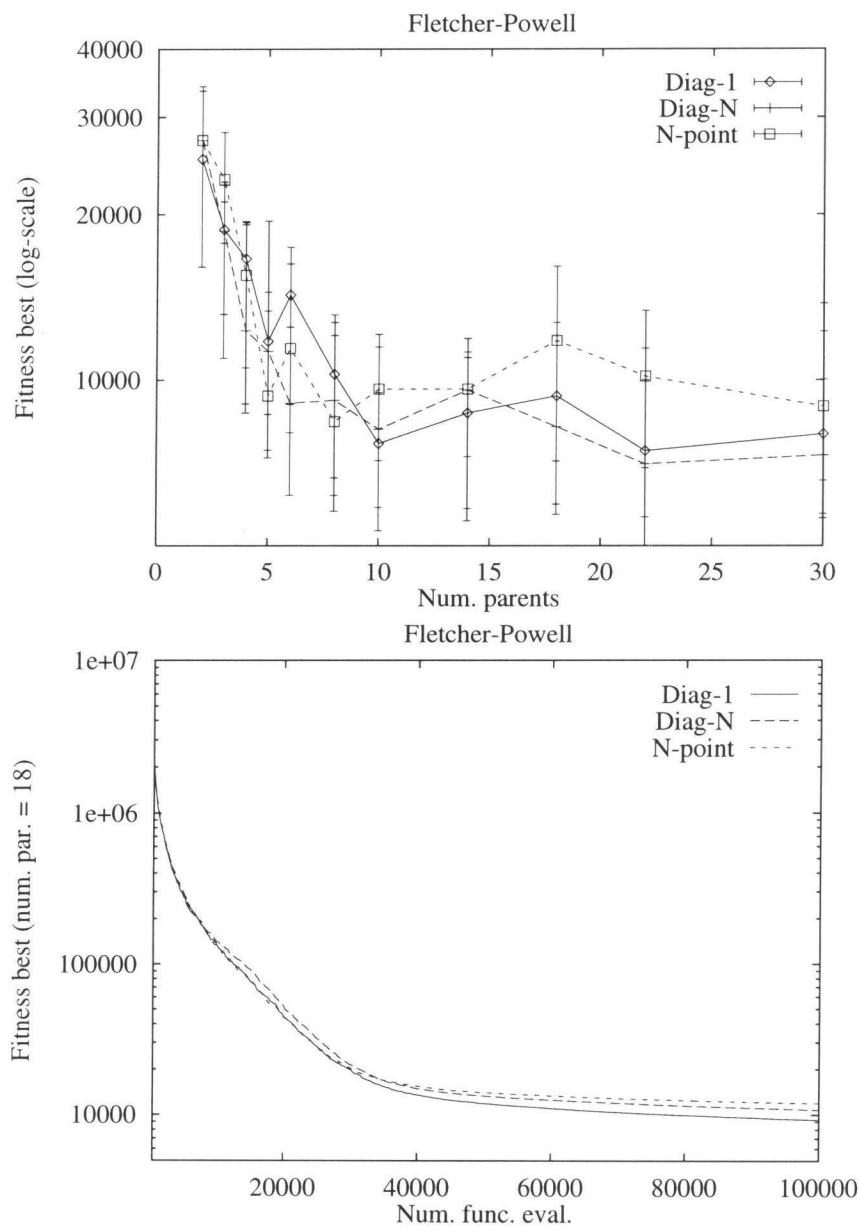


Figure 11.10: Accuracy (left) and progress curves (right) on the Fletcher-Powell function



## 11.5 Conclusions

Concerning our first research objective, i.e. finding connections between the characteristics of the objective functions and the usefulness of applying more parents, we can observe the following. With the single exception of Rosenbrock's saddle (F2) it is useful to apply diagonal crossover with an arity  $N > 2$ . Looking for particular features of F2 that may cause this deviance let us note that it has the lowest dimensionality ( $n = 2$ ) and the shortest chromosomes (of length 22) as opposed to the other functions (200-900 for F3-F8). This makes the disruptiveness of the crossover operators relatively high even for low  $N$ 's. The other unimodal function in the test set, F1, is apparently so easy to optimize that the GA does not suffer from this effect (all first order schemata point towards the global optimum), but on F2 where the optimum is "hidden at the bottom of a long bent valley", cf. [BM97], this seems to be disastrous.

Our second research objective concerned the identification of the source(s) of increased performance of diagonal crossover when used with more parents. As for the hypothesis  $H_1$ , i.e. that more crossover points increase performance, observe that  $N$ -point crossover did become better for higher  $N$ 's on all functions of our test-suite. Therefore, we accept  $H_1$  and conclude that higher performance partially comes from a higher number of crossover points. Explanations for this fact are the better mixing of information, cf. [DS92], and the increased macro-mutation-like effects of crossover in case of using higher  $N$ 's. The diagonal crossover strongly decorrelates loci that are far apart because the values of these loci are likely to be taken from different parents, this contrary to the  $N$ -point crossover that takes values of loci from the same parent when an even number of crossover-points falls between these loci.

Hypothesis  $H_2$  about the advantages of a bigger generational gap is clearly rejected, since the one-child and the  $N$ -children variants of diagonal crossover exhibited the same behaviour on each function. Hence, we can conclude that the advantage of applying diagonal crossover with higher  $N$ 's is not the result of a bigger generational gap in the Steady State GA we use (see section 11.2 for discussion).

Recall, that our working hypotheses are not mutually exclusive. Accepting  $H_1$  does not imply rejection of  $H_3$ , i.e. that better performance for higher  $N$ 's would only come from having more crossover points. In fact diagonal crossover was better than  $N$ -point crossover on all but two functions: on Rosenbrock's saddle (F2) and on the Fletcher-Powell function (F8). On the Fletcher-Powell function diagonal crossover was not significantly better than  $N$ -point crossover. Such little differences in performance do not clearly justify the acceptance of the hypothesis  $H_3$  on the function F8. Here, there is no clear advantage of using more parents for recombination, increased performance for higher  $N$ 's seems to be the result of crossovers effect as macro mutation, which effect is intensified by more crossover points. Recall, that F8 spans a very rugged landscape with randomly distributed local optima, which makes it similar to NK-landscapes with relatively high  $K$  values. These observations are thus in agreement with earlier conclusions for NK-landscapes, [ES96, HM95, Kau93], stating that on such surfaces crossover is not useful at all. On Rosenbrock's saddle (F2) diagonal crossover was clearly worse than  $N$ -point

crossover, besides, the performance of diagonal crossover decreased for increasing  $N$ .

According to the above considerations, hypothesis  $H_3$  has to be refined. On quasi-random landscapes, such as F8, increased performance of diagonal crossover for higher  $N$ 's may occur, but it seems not to be the result of using more parents, i.e.  $H_3$  does not hold. On other types of landscapes (the unimodal F1 and the multimodal, but somewhat regularly shaped F3-F7), diagonal crossover exhibits increased performance when increasing  $N$ , and it does outperform  $N$ -point crossover, thus confirming  $H_3$ . F2 remains an exception, showing that even for unimodal landscapes it is not guaranteed that diagonal crossover will become better when increasing  $N$ . Yet, with this exception in mind, we can draw the conclusion that if diagonal crossover becomes better for higher  $N$ 's then this improvement is not only the consequence of using more crossover points, but also that of using more parents.

Let us close our conclusions with noting that the most gain occurred to be in the speed of the GA, diagonal crossover is primarily faster than two-parent  $N$ -point crossover (if and when). Clearly, if we had set the maximum number of evaluations lower than this difference in speed would also have resulted in differences in success rate and accuracy. Thus, although we definitely do not claim that diagonal crossover is a universally superior operator. However, especially when one expects that the linkage within a problem is relatively tight, it is a sound design heuristics to implement diagonal crossover in a GA and set the number of parents above two.

## Chapter 12

# Evolutionary 3D-Air Traffic Flow Management

The increasing amount of air traffic requires more efficient use of airspace. One possible way to do this is to use a new planning model which is called the free-route model. New optimization tools are required to create a planning according to this type of model. We present an evolutionary tool to solve free-route Air Traffic Flow Management problems within a three-dimensional airspace. To the best of our knowledge, this is the first (evolutionary) tool that can solve free-route planning problems involving a few hundred aircraft. The performance of the tool has been tested on a set of randomly generated problem instances, where we were especially interested in the robustness of the tool and the scaling of the amount of computation with respect to the size of the problem instances.

### 12.1 Project overview

Air Traffic Flow Management is the general name for the routing and the scheduling of aircraft from half a year before departure until shortly after departure. An important task is the creation of a planning for trajectories of aircraft in a certain area for a certain time horizon (for example for the next twelve hours). Other tasks include for example dynamic replanning. Though some automated tools are used to make a global planning long before departure, especially the detailed planning just before or after departure is still mainly done manually. Due to the increasing volume of air traffic, improved planning methods and models become necessary, and automated tools to create plans become very useful. One of the new planning models is the free-route model. Under this model the trajectories of aircraft are less restricted than under the more traditional corridor-model. Currently, there are no tools that are able to create good solutions for the free-route model for a large number of flights. In a cooperation between the Centre for Mathematics and Computer Science CWI, Leiden University, and the National Aerospace Laboratory NLR the free-route model was studied with the objective to develop a planning tool for this model. Figure 12.1 shows the environment of such a planning tool. On the left we see the

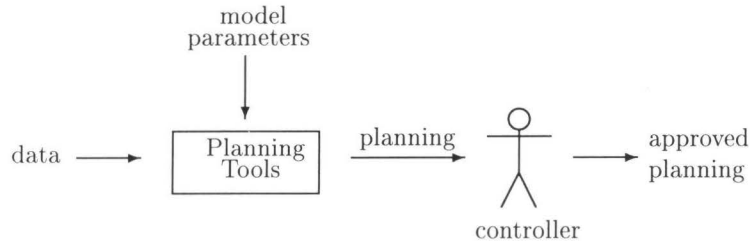


Figure 12.1: Environment of the ATFM planning tool

data to be input to the tool. The data consists of a list of flights that have to be scheduled. For each flight the coordinates of departure, the time of departure and the coordinates of the destination are given. We also have to give the parameters of the free route model. The box represents the planning tool that creates a plan given the current data. The plan describes a schedule for a period of several hours and will be passed to a human controller for approval. Only after this approval will the plan be used.

## 12.2 Air Traffic Flow Management Problem

A good reference to Air Traffic Flow Management (ATFM) is the book [Obe85]. A plan describes the trajectories of all involved aircraft. A trajectory defines the exact position of an aircraft as a function of time, so it corresponds to a path with additional temporal information. Two trajectories are conflicting when the separation standards, as stated by the ICAO (International Civil Aviation Organization), are violated. Depending on the area and weather circumstances different separation standards apply. In this chapter we use the following standards. The minimal required separation between different routes in the horizontal plane is about 16 nautical miles (1 nautical mile = 1,852 metres), the vertical separation for the higher flight levels is 2000 ft (1 foot = 0.3048 metres). The airspace is partitioned into a set of sectors. A sector can contain a number of layers, called flight-levels. The controller assigned to a sector is responsible for the planning of the trajectories of all the aircraft that fly in this sector. When an aircraft wants to pass a boundary between two sectors the corresponding controllers first have to agree on the time and location of the crossing of the boundary.

An ATFM plan assigns a single trajectory to each aircraft. The primary goal is to choose in such a way that there are no conflicts between aircraft. Secondary targets are:

- to keep all trajectories as short as possible,
- to minimize the number of manoeuvres (especially those manoeuvres that are uncomfortable to the passengers), and

- to have a fair plan with respect to all involved aircraft (i.e. the additional flying distance should be divided over all involved aircraft in a reasonable manner).

Currently ATFM is based on a network model. This model assumes a fixed network of corridors within the airspace, each containing a number of flight-levels. An aircraft is assumed to fly through a corridor from beacon to beacon. Intersections of corridors are always marked by beacons. Only near those beacons, an aircraft is allowed to switch to a different corridor. So this model can be seen as a kind of three-dimensional highway network.

The network model restricts the number of possible trajectories in order to make the planning manageable. In this sense, the network model does not use the available airspace in an efficient manner. Due to the increasing amount of air traffic, the airspace above Europe is almost saturated. Increased accuracy of navigation equipment and the availability of better computers allows for other airspace models. One such model is the free-route model that allows arbitrary shaped trajectories, has a larger degree of flexibility. A different model is the free-flight model, in which the aircraft are autonomous instead of being guided by air traffic control. It is clear that new methods have to be developed for planning problems in these models. This was the starting point of the research described in this chapter.

## 12.3 Design process

Currently no practical applications of the free-route model for ATFM exist, hence we have to use artificially generated problem instances. Moreover, if the free-route planning model is going to be used, it might be used in a different form than the form that is studied in this chapter. Therefore one of the requirements is that the tool should be robust in the sense that it is able to handle new constraints. Moreover, a certain degree of flexibility is desired. For example, a tool should offer a way to handle soft constraints. A characteristic of the ATFM problems is their dynamic nature: it is possible that additional aircraft have to be added to the plan and planned aircraft can deviate from their planned trajectory. If the current plan violates constraints due to these changes a modified plan has to be created on the fly. The number of affected trajectories should not be too large. It is also desirable that a tool can create alternative plans, among which a human controller can choose. It should also be adaptive in the sense that it can cope with additional constraints imposed by a human controller. Hence, the construction of a tool for the free-route model is a challenge. An evolutionary algorithm is used to have the flexibility described above and to handle the many additional constraints that can arise during the development of a plan.

### 12.3.1 ATFM and Evolutionary Computation

The first paper about the application of evolutionary techniques to ATFM problems was [AGJS93]. In this paper a binary genetic algorithm (GA) was used to do (short-term)

conflict resolution. The problems handled involved two or three aircraft and the GA was shown to outperform  $A^*$ -search. In France, research on conflict resolution continued, and currently they can handle problems involving up to 20 aircraft all having a conflict at approximately the same location [DAC95]. This genetic conflict solver is used as a part of their so-called ATC test bench. The ATC test bench has been applied to a problem involving 4835 aircraft. It detects all conflicts within a time window of five minutes, partitions the conflicts over independent subsets, applies the genetic solver to each of these subsets independently, and combines the results to obtain a new planning. This procedure is repeated until no further progress is possible. Evolutionary algorithms have also been applied to ATFM problems using the network model [DO97]. For each flight a small set of possible routes and departure times is created. An evolutionary algorithm is used to assign the departure time and the route for each flight in order to minimize the workload over all involved sectors. Problems involving up to 3000 aircraft have been handled. Also research on the combination of neural networks and GA's for (short-term) conflict resolution involving two aircraft [DAN96] was investigated. In Germany at the DLR evolutionary algorithms were used to solve ATFM problems involving the free-flight model and the free-route model [Ger94].

Our first system was a mutation based hybrid EA that could handle two-dimensional problem instances, so all trajectories were restricted to a horizontal plane. The system could solve problem instances involving up to 20 aircraft within a square 2D sector of size  $200 \times 200 \text{ km}$  [vKHHK95]. Recombination operators were used, but these operators did not increase the performance of the system. This is expected to be a consequence of the strong dependence on the context of the trajectories that are close to other trajectories. A new system was developed that could handle three-dimensional problem instances where the altitude of the aircraft is the additional degree of freedom [vKvdAK96]. This system could easily find conflict-free solutions to problems involving up to 800 aircraft in a 3D-sector of size  $2000 \times 2000 \text{ km}$  that were spread over an interval of four hours. These problems can be handled on a standard workstation within roughly 15 minutes of computation. For these larger instances recombination is useful because on these larger problem instances many trajectories are either temporal or spatial independent of each other. We expect that some conflicts can be solved in parallel and the resolutions to these conflicts can then be merged by means of recombination. A more direct way to exploit this kind of parallelism would be to make a decomposition of the problem in a set of smaller independent problems. Unfortunately finding an optimal decomposition is far from trivial and the choice of the decomposition will restrict the set of possible solutions. In order to exploit the potential parallel conflict resolution we apply an evolutionary algorithm with recombination. This system is described in the rest of the chapter.

### 12.3.2 Evolutionary Algorithm

The elitist recombination algorithm [TG94] is a simple evolutionary algorithm, that involves a direct competition between the offspring and both of their own parents. Figure 12.2 shows a single generation of this algorithm. It shows how the next population

$P_{t+1}$  is produced from the current population  $P_t$ . To the left we see the current population

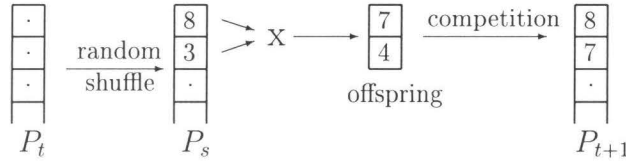


Figure 12.2: Elitist recombination

$P_t$ , and each box represents a single individual. The values in the boxes denote the fitness of the corresponding individuals. An intermediate population  $P_s$  is generated by doing a random shuffle of the individuals in population  $P_t$ . Partition population  $P_s$  in a set of adjacent pairs, and for each pair apply the recombination operator in order to obtain two offspring. Next, hold a competition among the two offspring and their two parents, and transfer the best two out of these four to the next population  $P_{t+1}$ . This competition between parents and offspring prevents rapid duplication of relatively fit individuals, and as a result decreases the probability of premature convergence. Elitist recombination resembles the deterministic crowding scheme [Mah92], but deterministic crowding lets each offspring compete with only one of its parents, i.e. the most similar parent.

We are using the elitist recombination algorithm as basis for our evolutionary planner. The (population-wide) elitism prevents a decay of the average fitness. Inferior offspring will not enter the population and due to the direct competition between offspring and its parents the duplication of the best few individuals is slowed down. Furthermore elitist recombination tends to be less sensitive to undersized populations than most other evolutionary algorithms [TG94].

### 12.3.3 Requirements

The ATFM planning tool should be able to create a plan for an airspace under the free-route model. Such a planning should be free of conflicts, the amount of additional distance travelled by all aircraft should be minimized and the number of manoeuvres should be kept low. The planning tool should have appropriate scaling properties with respect to the number of flights to be scheduled. In order to be applicable such a tool should be able to create a planning for problem instances involving a few thousand flights in less than an hour.

### 12.3.4 Representation

A single individual represents a complete plan, describing the trajectories of all the flights that have to be scheduled. We represent a single trajectory by means of an ordered list of four-dimensional coordinates. Each coordinate is a tuple  $(x, y, l, t)$  where  $x$  and  $y$  represent

a location in a two-dimensional plane,  $l$  represents the flight-level of the aircraft, and  $t$  represents the time of passing the location. An aircraft is assumed to fly in a straight line and at a constant velocity from one coordinate to the next coordinate. A complete plan is represented as:

|            |                      |                      |                      |                      |                      |
|------------|----------------------|----------------------|----------------------|----------------------|----------------------|
| flight 1   | $(x, y, z, t)_{1,1}$ | $(x, y, z, t)_{1,2}$ | $(x, y, z, t)_{1,3}$ | $(x, y, z, t)_{1,4}$ | $(x, y, z, t)_{1,5}$ |
| flight 2   | $(x, y, z, t)_{2,1}$ | $(x, y, z, t)_{2,2}$ |                      |                      |                      |
| $\vdots$   |                      | $\vdots$             |                      |                      |                      |
| flight $n$ | $(x, y, z, t)_{n,1}$ | $(x, y, z, t)_{n,2}$ | $(x, y, z, t)_{n,3}$ | $(x, y, z, t)_{n,4}$ |                      |

Note that the number of coordinates can differ for each trajectory (flight).

The fitness  $f(I)$  of the individual  $I$  is defined by

$$f(I) = -f_c(I) - \frac{1}{d} \left( \sum_{x \in I} f_m(x) + f_d(x) \right),$$

where  $f_c(I)$  is the number of conflicts between trajectories in individual  $I$ ,  $f_m(x)$  is the number of manoeuvres in trajectory  $x$ , and  $f_d(x)$  is the relative length of trajectory  $x$  with respect to the shortest possible trajectory for the corresponding flight. The constant  $d$  is a large constant chosen such that the value of the second term is in the range  $[0, 1]$ . Hence minimization of the number of conflicts is the primary goal, while minimization of the distances and the number of manoeuvres is of secondary importance.

### 12.3.5 Operators

For each conflict we can identify a set of involved flights. Such a set consists of the two aircraft that are actually in conflict augmented with aircraft that are in their vicinity. If two conflicts involve disjoint sets of aircraft then often it is possible to resolve these conflicts independently. The possibility to resolve conflicts independently and to merge the results, increases the efficiency of the planner significantly. This is the reason to emphasize on the recombination operator in the evolutionary algorithm. Recombination considers the trajectories as atomic entities and therefore recombination will not generate new trajectories, but will only generate new combinations of (existing) trajectories.

We first experimented with a uniform recombination operator which took approximately half of the trajectories from the first parent while the other trajectories were taken from the second parent. This operator worked fine for small problems, but did not scale well when increasing the size of the problem. The main reason for this bad scaling behaviour is that a uniform recombination operator can not keep large sets of conflict-free trajectories together as the probability that  $m$  trajectories are taken from the same parents is  $(1/2)^m$ .

In order to handle large problems we need a better scaling behaviour. Therefore we developed a heuristic recombination operator. This operator starts with an empty plan for the offspring. Then it iteratively selects a nonscheduled flight and a primary parent; both are selected at random. The trajectory corresponding to this flight is taken from



the selected parent and it is checked whether this trajectory will introduce a conflict in the current plan of the offspring. If no conflict is introduced then the trajectory is added to the offspring, otherwise the trajectory from the other parent is added to the offspring. This process continues until all flights of the offspring are scheduled. This recombination operator is unbiased in the sense that on average half of the flights will be taken from one of the parents.

The mutation operator is the only operator that introduces new trajectories. A new trajectory is created by making a copy of one of the existing trajectories and add one additional manoeuvre. The mutation operator introduces a detour in a trajectory by changing its heading of the aircraft by  $\pm 45^\circ$ , let it fly in this new direction for a random duration, and then change the heading of the aircraft such that it flies to its destination along a straight line. Note that the mutation operator can cancel previously inserted manoeuvres. Figure 12.3 shows a trajectory obtained after applying the mutation operator one time. It might seem a bit restricted to consider only changes of  $\pm 45^\circ$ , but in practice this works well. A much smaller change of heading results in a new trajectory that is quite similar to the non-mutated trajectory, and a much larger change of heading will move the aircraft too far off course. Although this mutation operator is relatively easy to

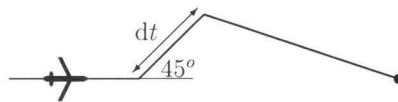


Figure 12.3: Trajectory obtained after applying the mutation operator once

implement one can not guarantee that a single application of the operator will produce a good trajectory. New trajectories that do not resolve any conflicts will rapidly be thrown out of the population by the EA as such trajectories only introduce a penalty and therefore a decrease of fitness.

### 12.3.6 Evolutionary planner

In order to get an efficient evolutionary planner we have to obtain a good balance between the creation of new trajectories and the assessment of the quality of each of the trajectories. If too few new trajectories are generated then the search will lack diversity and we are likely to get premature convergence. On the other hand a too rapid production of new trajectories will result in a waste of computational effort and will likely result in the introduction of unnecessary complex trajectories.

In order to balance the two processes we have split the inner loop of the evolutionary algorithms in two parts that are alternated. The first part is an elitist recombination algorithm involving recombination only. Here the quality of the available trajectories is assessed by mixing trajectories in order to get a complete plan. The second part contains the mutation operator. Here we take an arbitrary plan and select an arbitrary conflicting

trajectory within this plan. Mutation is applied to this trajectory until a better trajectory is obtained within the context of the selected plan or the maximal number of trials for a single mutation is exceeded. So the amount of computational effort used to generate new trajectories will increase when the generation of new trajectories becomes more difficult, thereby resulting in a more constant rate of introducing new trajectories. The following pseudo-code gives a more detailed description of the evolutionary planner.

```

ATFM( $N_{gen}$ ,  $F_{mut}$ , and  $N_{try}$ )
   $t = 0$ ;
   $P_t = \text{initialPopulation}(\mu)$ ;
  while (not ready) do
    ## Part 1: apply Elitist recombination for  $N_{gen}$  generations
    for  $i = 1$  to  $N_{gen}$  do
       $P_{t+1} = \text{ElitistRecombinationStep}(P_t)$ ;
       $t = t + 1$ ;
    od;
    ## Part 2: introduce new trajectories by means of mutation
     $best = \text{SelectBest}(P_t)$ ;
    for  $i = 1$  to  $F_{mut}f_c(best)$  do
       $plan = \text{RandomPlan}(P_t)$ ;
       $flight = \text{randomConflictFlight}(plan)$ ;
       $k = 0$ ;
      repeat
         $k = k + 1$ ;
        if  $k < N_{try}$  then
           $newPlan = plan - flight + \text{mutate}(flight)$ ;
        else
           $newPlan = plan - flight + \text{randomStraight}(flight)$ ;
        fi;
      until  $f_c(newPlan) < f_c(plan)$  or  $k \geq N_{try}$ ;
      if  $f_c(newPlan) < f_c(plan)$  then
         $P_t = P_t - plan + newPlan$ ;
      fi;
    od;
     $od$ ;
     $ATFM = \text{SelectBest}(P_t)$ ;
  end

```

The function  $f_c(plan)$  counts the number of conflicts in a plan,  $\text{randomStraight}(flight)$  creates a straight-line trajectory for the flight at a random flight level, and  $\text{mutate}(flight)$  creates a new flight by mutation. The number of mutations is related to the number of conflicts in the current best solution by means of the parameter  $F_{mut}$ .

### 12.3.7 Domain knowledge

Domain knowledge is used in several places in our algorithm. During initialization we have to choose an initial set of trajectories for all flights. Because we know that a good plan should involve as few manoeuvres as possible we initialize all plans with straight line trajectories or trajectories involving only a single manoeuvre. More complex trajectories will be introduced by mutation and recombination only if necessary. Domain knowledge is also incorporated in the mutation operator that generates relatively smooth trajectories, that do not contain too large deviations from the straight line trajectories.

## 12.4 Development and implementation

The evolutionary planner has been implemented in *C++*. All experiments were conducted on a Silicon Graphics Indy 120 MHz workstation. A single run is terminated as soon as the fitness of the best individual is higher than -1, which indicates that a conflict-free solution is found.

### 12.4.1 Generation test problems

The free-route model is currently not in use in practice. There is no practical situation nor a golden standard to compare results to. Usually models are compared on the basis of the number of aircraft they can handle on random problem instances. Hence we generated a reasonable set of randomly generated test problems for this model.

We decided to model a square sector of  $2000 \times 2000$  km, containing 12 independent flight-levels. This sector is assumed to be located at an altitude of roughly 10 km above sea-level. Independent flight levels imply that aircraft located in different flight levels will never be in conflict. The source and the destination of the aircraft are 2D-locations, chosen at random within the sector, using a uniform distribution. The flight-level of the aircraft at the given entry and exit locations can be chosen freely.

The entry and exit locations of flights in a plan do not need to correspond to actual locations of airports. The reason for this is as follows. The sector we are modelling is located at an altitude of approximately 10 km above sea level. Our tool plans flights only for higher regions of the airspace. A local Air Traffic Control center at an airport usually manages lower regions of the airspace. When aircraft depart or approach an airport their trajectory is managed by controllers at the airport. When aircraft reach the sector we are interested in, then their locations are already spread over a rather large area. Moreover, in Western Europe airports are relatively close to each other. So the entry and exit locations can be considered as uniformly distributed over the 2D-space.

The evolutionary algorithm can handle a problem where all aircraft have a different, but known, velocity. For convenience we assume that all the aircraft have a velocity of 900 km/hr. The flights are to be planned within an interval of 4 hours. The time of entry is selected at random within this interval. This time of entry is accepted if the aircraft can

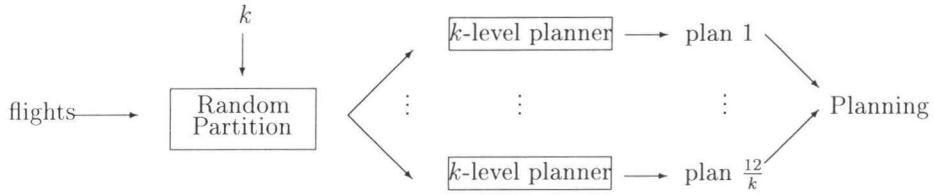


Figure 12.4: Approach to solve large scale problems

reach its exit location within the interval of 4 hours, when flying in a straight line from the entry to the exit location.

In order to predict the number of conflicts we can use physical models describing the number of collisions between a set of gas molecules in a box per unit of time [EO83]. Application of such a model to our case results in

$$E[\#Conf] = \alpha \frac{t_{hor} d_{sep} v}{s^2 l} n^2$$

where  $t_{hor}$  is the interval of the planning,  $d_{sep}$  is the minimal horizontal separation between any two aircraft,  $v$  is the average velocity of the aircraft,  $s$  is the length of the side of the sector,  $l$  is the number of independent flight levels, and  $n$  is the number of aircraft. The constant  $\alpha$  is introduced to account for the non-uniform distribution of aircraft over the sector. The density of aircraft will be highest near the center of the sector, and lowest near its borders. Within the gas model this constant is one as the gas molecules are distributed uniformly over the complete volume. The validity of the formula has been tested experimentally [vKvdAK96].

## 12.5 Handling large-scale problems

The expected number of conflicts scales quadratically with respect to the number of aircraft. Therefore it can be beneficial to split a large-scale problem in a set of smaller independent subproblems and merge the solutions of these subproblem to a solution of the large-scale problem. Whether we can find a set of independent subproblems will depend on the set of trajectories that we allow. The class of possible trajectories is determined by the mutation operator. Because the initial population contains trajectories that are only within a single level and the mutation operator does not introduce level changes, the search is restricted in this case to solutions that only involve trajectories located in a single flight level. A population consists of flight plans and within the different flight plans aircraft can have different flight levels. The recombination operator combines trajectories from different plans.

However, quite some efficiency can be gained by splitting the flight levels in disjoint subsets. We can first do an assignment of flights over these subsets, and then use inde-

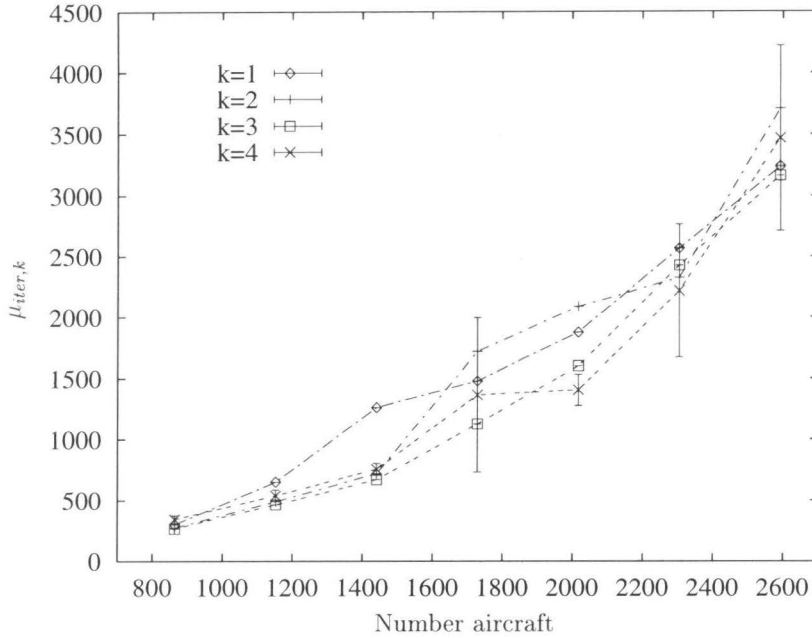


Figure 12.5: Number of iterations required when extrapolated to the 12-level planning problem

pendent evolutionary algorithms for each of the subsets. In this way, each flight is first assigned a restricted number of flight levels, among which a good one is chosen by the evolutionary algorithm. This approach reduces the search space significantly. However simultaneously the set of solutions is reduced.

There are two extremes: take one subset, or take as many subsets as there are flight levels. The first one would take a lot of time and memory, and the second one is too restrictive and does not give very good results. In our experiments we tried different values for the number of subsets. We considered 12, 6, 4, and 3 subsets. In fact, in our formulae and figures we use a variable  $k$  that ranges from 1 to 4, where  $k$  stands for the number of flight levels within the subproblem. The actual approach can be represented schematically as shown in figure 12.4. On the left we see all flights that have to be scheduled. The number of subsets  $k$  is given as an input and the flights are randomly partitioned over  $12/k$  sets. For each set a  $k$ -level planner is applied that schedules the flights within a range of  $k$  consecutive flight levels. The resulting  $12/k$  plans are combined to a plan for the 12-level problem.

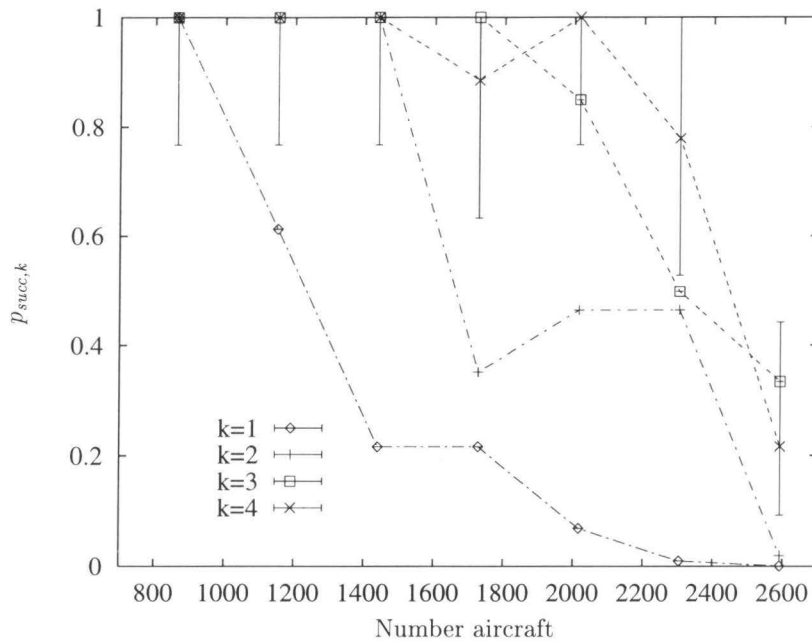


Figure 12.6: Fraction of successful runs when extrapolated to the 12-level planning problem

### 12.5.1 Performance measures

We were interested in feasibility of tackling free-route problems using evolutionary computation. By feasibility we mean that we were interested to obtain evidence that this type of problem can be solved in a reasonable amount of time and a conflict-free solution will be found with a reasonable probability of success.

Moreover, we are interested in the quality of the solutions. The quality of a conflict-free solution is determined by the expected total number of manoeuvres and by a distance penalty with respect to a situation that all flights go along a path of minimal length.

## 12.6 Results

During the experiments we varied the number of aircraft to be planned thereby varying the complexity of the problem instances. We have applied the procedure described in section 12.5 to partition large-scale problems into a set of subproblems.

In order to measure the expected performance of the system we did run the  $k$ -level for different values of  $k$ . All results have been extrapolated to a 12-level problem instance.

During a single iteration each of the  $k$ -level planners is allowed to perform a single fitness evaluation. If  $\sigma'_{iter,k}$  is the standard deviation in the number of iterations used to

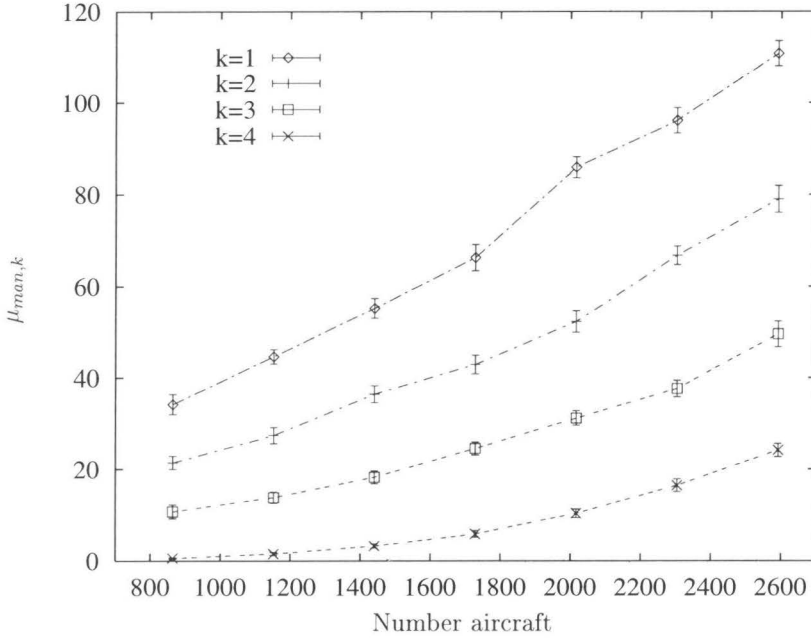


Figure 12.7: Average number of manoeuvres when extrapolated to 12-level planning problem

find a conflict-free solution for a problem with  $k$  levels then we can estimate the standard deviation for the 12-level problem by

$$\sigma_{iter,k} = \frac{\sigma'_{iter,k}}{(12/k)^{1/2}}.$$

The success-rates are determined experimentally. The 90%-confidence intervals for the success-rate have been computed. Given the confidence interval of the success rate for the  $k$ -level planner we can estimate the confidence interval for the 12-level problem when using a set of  $k$ -level planners by

$$p_{succ,k} = p'_{succ,k}^{(12/k)}.$$

Given the expected value  $\mu'_{man,k}$  of the total number of manoeuvres for the  $k$ -level sub-problems, we have for the whole problem

$$\mu_{man,k} = (12/k)\mu'_{man,k},$$

$$\sigma_{man,k} = (12/k)^{1/2}\sigma'_{man,k}.$$

Similar estimates can be used for values  $\mu_{dev,12}$  and  $\sigma_{dev,12}$  for the cumulated deviations from the shortest paths.

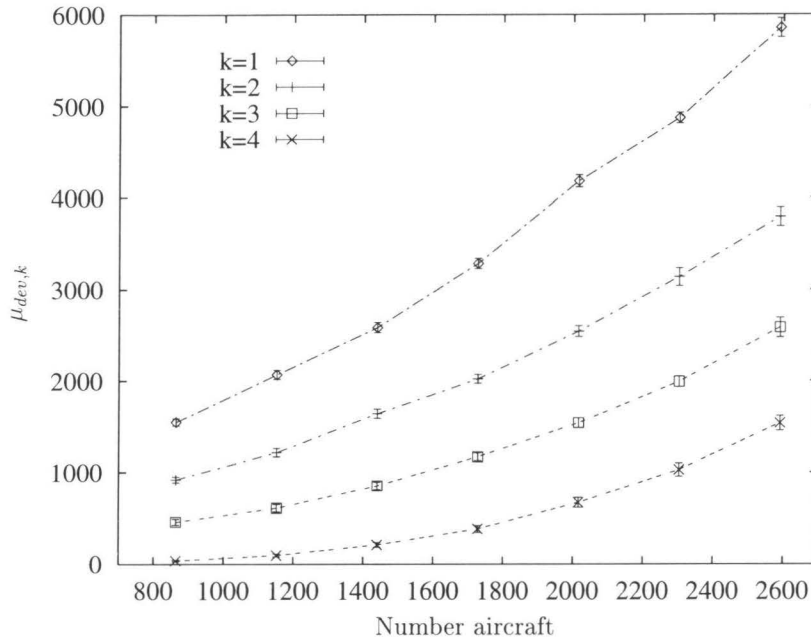


Figure 12.8: Average total deviation when extrapolated to 12-level planning problem

All experiments were performed with an upper bound of 5000 fitness evaluations, a maximal number of allowed avoidance manoeuvres of 12 per aircraft,  $N_{gen} = 2$ ,  $F_{mut} = 0.5$ ,  $N_{try} = 10$ ,  $\#I = 16$ . All results are averaged over 20 independent runs.

Figure 12.5 shows the number of iterations. The horizontal axis shows the number of aircraft routed in 12 levels. The error bars correspond to the standard deviation. These error bars are only given for the case  $k = 4$ . Note the large standard deviation for the number of iterations. It is remarkable that the actual number of iterations does not change much as the number of levels handled simultaneously increases. Figure 12.6 shows the fraction of successful runs. The error bars correspond to the 90%-confidence intervals. Again the error bars are only given for the case  $k = 4$ . Optimizing multiple levels simultaneously results in a significantly higher success rate. For 2592 aircraft  $k = 3$  performs best. We expect this to be a result of the rather arbitrary limit of 5000 fitness evaluations. The experiments for  $k = 4$  terminate 10 times by reaching this upper limit while the experiments for  $k = 3$  terminate only 6 times by reaching this limit. Figures 12.7 and 12.8 show the extrapolated number of manoeuvres and the amount of deviation. As expected, using larger values of  $k$  results in a lower value of the number of manoeuvres needed and a smaller total distance being travelled.



## 12.7 Summary

The free-route planning problem has a search space that grows exponentially in the number of aircraft. To be able to handle large scale ATFM problems it is important to incorporate knowledge regarding the problem domain. We have done so by means of a non-uniform seeding of the initial population and by designing problem-specific evolutionary operators. Introduction of such operators has to be done carefully in order to prevent certain good solutions from being ignored and to prevent premature convergence.

Large problem instances can be handled relatively easily by splitting them in a number of smaller problem instances each involving a limited range of flight levels. We have investigated this approach and observed that optimizing multiple levels simultaneously results in a significant improvement of the probability of finding a solution and also in the quality of the obtained solutions.

Using the current tool we are able to generate a planning involving up to 2592 flights within a 4-hour interval. Straightforward extrapolation suggests that the tool can create a plan for 7776 flights in a 12-hour interval.



# Bibliography

- [Ack87] D.H. Ackley. *A connectionist machine for genetic hillclimbing*. Kluwer Academic Publishers, 1987.
- [AGJS93] J. Alliot, H. Gruber, G. Joly, and M. Schoenauer. Genetic algorithms for solving air traffic control conflicts. In *Ninth Conference on Artificial Intelligence for Applications*, pages 338–344. IEEE Computer Society press, 1993.
- [AK89] E. Aarts and J. Korst. *Simulated Annealing and Boltzmann machines*. Wiley, 1989.
- [Alt94] L. Altenberg. The evolution of evolvability in genetic programming. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic programming*, chapter 3, pages 47–74. M.I.T. Press, 1994.
- [AM94] H. Asoh and H. Mühlenbein. On the mean convergence time of evolutionary algorithms without selection and mutation. In Y. Davidor and Männer [YDM94], pages 88–97.
- [Bäc92] T. Bäck. The interaction of mutation rate, selection, and self-adaptation within a genetic algorithm. In Männer and Manderick [MM92], pages 85–94.
- [Bäc93] T. Bäck. Optimal mutation rates in genetic search. In Forrest [For93], pages 2–8.
- [Bäc96] Th. Bäck. *Evolutionary Algorithms in Theory and Practice*. Oxford University Press, 1996.
- [Bäc97] Th. Bäck, editor. *Proceedings of the 7th International Conference on Genetic Algorithms*. Morgan Kaufmann, 1997.
- [Bak87] J.E. Baker. Reducing bias and inefficiency in the selection algorithm. In Grefenstette [Gre87], pages 14–21.
- [Bar62] N.A. Barricelli. Numerical testing of evolution theories. part I: Theoretical introduction and basic tests. *Acta Biotheoretica*, Vol. 16(1–2):69–98, 1962.

- [Bar63] N.A. Barricelli. Numerical testing of evolution theories. part II: Preliminary tests of performance, symbiogenesis and terrestrial life. *Acta Biotheoretica*, Vol. 16(3-4):99-126, 1963.
- [BB91] R.K. Belew and L.B. Booker, editors. *Proceedings of the 4th International Conference on Genetic Algorithms*. Morgan Kaufmann, 1991.
- [BDL<sup>+</sup>96] H. Bersini, M. Dorigo, S. Langerman, G. Seront, and L. Gambardella. Results of the first international contest on evolutionary optimisation. In IEEE [IEE96], pages 611-615.
- [Bey95] H.-G. Beyer. Toward a theory of evolution strategies: On the benefits of sex- the  $(\mu/\mu, \lambda)$  theory. *Journal of Evolutionary Computation*, 3(1):81-111, 1995.
- [BFM97] T. Bäck, D. Fogel, and Z. Michalewicz, editors. *Handbook of Evolutionary Computation*. Oxford University Press and Institute of Physics Publishing, 1997.
- [BHmS91] T. Bäck, F. Hoffmeister, and H.-P. Schwefel. A survey of evolution strategies. In Belew and Booker [BB91], pages 2-9.
- [BKdGK97] Th. Bäck, J.N. Kok, J.M. de Graaf, and W.A. Kusters. Theory of genetic algorithms. *Bulletin of the EATCS*, 63:161-192, 1997.
- [BKW98] S. Bandyopadhyay, H. Kargupta, and G. Wang. Revisiting the GEMGA: Scalable evolutionary optimization through linkage learning. In IEEE [IEE98], pages 603-608.
- [BM97] T. Bäck and Z. Michalewicz. Test landscapes. In Bäck et al. [BFM97], pages B2.7:14-B2.7:20.
- [Box57] G.E.P. Box. Evolutionary operation: A method for increasing industrial productivity. *Applied statistics*, Vol. 6(2):81-101, 1957.
- [BP96] G. Bilchev and I. Parmee. Inductive search. In IEEE [IEE96], pages 832-836.
- [Bra90] R.N. Brandon. *Adaption and Environment*. Princeton University Press, 1990.
- [Bri81] A. Brindle. *Genetic algorithms for function optimization*. Doctoral dissertation, University of Alberta, Department of computer science, 1981. Technical Report TR81-2.
- [BRS66] H.J. Bremermann, M. Rogson, and S. Salaff. Global properties of evolution processes. In H.H. Pattee, E.A. Edlsack, L. Fein, and A.B. Callahan, editors, *Natural Automata and Useful Simulations*, pages 3-41. Spartan Books, Washington DC, 1966.

- [BS92] H. Bersini and G. Seront. In search of a good evolution-optimization crossover. In Männer and Manderick [MM92], pages 479–488.
- [BS93] Th. Bäck and H.-P. Schwefel. An overview of evolutionary algorithms for parameter optimization. *Journal of Evolutionary Computation*, 1(1):1–23, 1993.
- [CJ91] C. Caldwell and V.S. Johnston. Tracking a criminal suspect though face-space with a genetic algorithm. In Belew and Booker [BB91], pages 416–421.
- [Cul93] J. Culberson. Crossover versus mutation: fueling the debate: TGA versus GIGA. In Forrest [For93], page 632.
- [DAC95] N. Durand, J.-M. Alliot, and O. Chansou. Optimal resolution of en route conflicts. *Air Traffic Control Quarterly*, 3(3):139–161, 1995.
- [DAN96] N. Durand, J.-M. Alliot, and J. Noailles. Collision avoidance using neural networks learned by genetic algorithms. In *Proceedings of the Ninth International Conference on Industrial & Engineering*, 1996.
- [Dar59] C. Darwin. *On the origin of species by means of natural selection*. London: Everyman's Library, J.M. Dent & Sons, 1951, 1859.
- [DeJ75] K.A. DeJong. *An analysis of the behavior of a class of genetic adaptive systems*. Doctoral dissertation, University of Michigan, 1975.
- [DeJ93] K.A. DeJong. Genetic algorithms are NOT function optimizers. In Rawlins [Raw93], pages 5–18.
- [DG93] K. Deb and D.E. Goldberg. Analyzing deception in trap functions. In Rawlins [Raw93], pages 93–108.
- [DG94] K. Deb and D.E. Goldberg. Sufficient conditions for deceptive and easy binary functions. *Annals of Mathematics and Artificial Intelligence*, 10:385–408, 1994.
- [DG97] K. Deb and M. Goyal. Optimizing engineering designs using a combined genetic search. In Bäck [Bäc97], pages 521–528.
- [DO97] D. Delahaye and A.R. Odoni. Airspace congestion smoothing by stochastic optimization. In P.J. Angeline R.G. Reynolds and J.R. McDonnell, editors, *Proceedings of proceedings of Evolutionary Programming VI*, pages 163–176, 1997.
- [DS91] K.A. DeJong and W.M. Spears. An analysis of multi-point crossover. In Rawlins [Raw91], pages 301–315.

- [DS92] K.A. DeJong and W.M. Spears. A formal analysis of the role of multi-point crossover in genetic algorithms. *Annals of Mathematics and Artificial Intelligence*, 5:1–26, 1992.
- [DS93] K.A. DeJong and J. Sarma. Generation gaps revisited. In Rawlins [Raw93], pages 19–28.
- [DW91] R. Das and D. Whitley. The only challenging problems are deceptive: Global search by solving order-1 hyperplanes. In Belew and Booker [BB91], pages 166–173.
- [ECS89] L.J. Eshelman, R.A. Caruana, and J.D. Schaffer. Biases in the crossover landscape. In Schaffer [Sch89], pages 10–19.
- [Eib98] A.E. Eiben. Multi-parent recombination. In Bäck et al. [BFM97], pages C3.3.7:1–C3.3.7:8.
- [EO83] S. Endoh and A.R. Odoni. A generalized model for predicting the frequency of air conflicts. In *Proceedings of the Conference on Safety Issues in Air Traffic Systems Planning and Design*, pages 226–251. Department of Civil Engineering, Princeton University, 1983.
- [ERR94] A.E. Eiben, P-E. Raué, and Zs. Ruttkay. Genetic algorithms with multi-parent recombination. In Y. Davidor and Männer [YDM94], pages 78–87.
- [ES93] L.J. Eshelman and J.D. Schaffer. Crossover's niche. In Forrest [For93], pages 9–14.
- [ES95] L.J. Eshelman and J.D. Schaffer. Productive recombination and propagating and preserving schemata. In Whitley and Vose [WV95], pages 299–313.
- [ES96] A.E. Eiben and C.A. Schippers. Multi-parent's niche: n-ary crossovers on NK-landscapes. In Voigt et al. [VERS96], pages 319–328.
- [Esh91] L.J. Eshelman. The CHC adaptive search algorithm: How to have safe search when engaging in nontraditional genetic recombination. In Rawlins [Raw91], pages 265–283.
- [EvK97] A.E. Eiben and C.H.M. van Kemenade. Diagonal crossover in genetic algorithms for numerical optimization. *Journal of Control & Cybernetics*, 26(3):447–466, 1997.
- [EvKK95] A.E. Eiben, C.H.M. van Kemenade, and J.N. Kok. Orgy in the computer: Multi-parent reproduction in genetic algorithms. In Morán et al. [MMMC95], pages 934–945.

- [FA90] D.B. Fogel and J.W. Atmar. Comparing genetic operators with gaussian mutations in simulated evolutionary processes using linear systems. *Biological Cybernetics*, 63:111–114, 1990.
- [FDN59] R.M. Friedberg, B. Dunham, and J.H. North. A learning machine: Part II. *IBM J. Research and Development*, 3:282–287, 1959.
- [FF95] C.M. Fonseca and P.J. Fleming. An overview of evolutionary algorithms in multiobjective optimization. *Journal of Evolutionary Computation*, 3(1):1–16, 1995.
- [FGMV96] C. Fleurent, F. Glover, P. Michelon, and Z. Valli. A scatter search for unconstrained continuous optimization. In IEEE [IEE96], pages 643–648.
- [FM93] S. Forrest and M. Mitchell. Relative building block fitness and the building block hypothesis. In Rawlins [Raw93], pages 109–126.
- [FM96] D.S. Falconer and T.F.C. MacKay, editors. *Introduction to quantitative genetics*. Longman, fourth ed. edition, 1996.
- [Fog89] T.C. Fogarty. Varying the probability of mutation in the genetic algorithm. In Schaffer [Sch89], pages 104–109.
- [Fog94] D.B. Fogel. *Evolving artificial intelligence*. Doctoral dissertation, University of California, 1994.
- [Fog95] D.B. Fogel, editor. *Evolutionary Computation: towards a new philosophy of machine intelligence*. IEEE Press, New York, 1995.
- [Fog98] D.B. Fogel, editor. *Evolutionary Computation, the fossil Record*. IEEE Press, New York, 1998.
- [For93] S. Forrest, editor. *Proceedings of the 5th International Conference on Genetic Algorithms*. Morgan Kaufmann, 1993.
- [For95] S. Forrest, editor. *Proceedings of the 6th International Conference on Genetic Algorithms*. Morgan Kaufmann, 1995.
- [FOW65] L.J. Fogel, A.J. Owens, and M.J. Walsh. Artificial intelligence through a simulation of evolution. In *Biophysics and Cybernetics: Proc. of the 2nd cybernetic sciences symp.*, pages 131–155. Spartan Books, 1965.
- [Fra57] A.S. Fraser. Simulation of genetic systems by automatic digital computers. I. introduction. *Australian J. Biological Sciences*, Vol. 10:484–491, 1957.
- [Fri56] G.J. Friedman. Selective feedback computers for engineering synthesis and nervous system analogy. Master's thesis, UCLA, 1956.

- [Fri58] R.M. Friedberg. A learning machine: Part I. *IBM J. Research and Development*, 2(1):2–13, 1958.
- [FS94] D.B. Fogel and L.C. Stayton. On the effectiveness of crossover in simulated evolutionary optimization. *Biosystems*, 32:171–182, 1994.
- [GDC93] D.E. Goldberg, K. Deb, and J.H. Clark. Accounting for noise in the sizing of populations. In Rawlins [Raw93], pages 127–151.
- [GDK90] D.E. Goldberg, K. Deb, and B. Korb. Messy genetic algorithms revisited: Studies in mixed size and scale. *Complex Systems*, 4:415–444, 1990.
- [GDKH93] D.E. Goldberg, K. Deb, H. Kargupta, and G. Harik. Rapid, accurate optimization of difficult problems using the fast messy genetic algorithms. In Forrest [For93], pages 56–64.
- [GDT93] D.E. Goldberg, K. Deb, and D. Thierens. Towards a better understanding of mixing in genetic algorithms. *Journal of the Society for Instrumentation and Control Engineers, SICE*, 32(1):10–16, 1993.
- [Ger94] I.S. Gerdes. Application of genetic algorithms to the problem of free-routing for aircraft. In IEEE [IEE94], pages 536–541.
- [GKD89] D.E. Goldberg, B. Korb, and K. Deb. Messy genetic algorithms: Motivation, analysis, and first results. *Complex Systems*, 3(5):493–530, 1989.
- [Gol89a] D.E. Goldberg. Genetic algorithms and walsh functions: Part II, deception and its analysis. *Complex Systems*, 3:153–171, 1989.
- [Gol89b] D.E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.
- [Gol93] D.E. Goldberg. The wright brothers, genetic algorithms, and the design of complex systems. In *proceedings of SYNAPSE '93 (Japan)*, 1993.
- [Gol94] D.E. Goldberg. Genetic and evolutionary algorithms come of age. *Communications of the ACM*, 37(3):113–119, 1994.
- [Gre85] J.J. Grefenstette, editor. *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*. Lawrence Erlbaum Associates, 1985.
- [Gre87] J.J. Grefenstette, editor. *Proceedings of the 2nd International Conference on Genetic Algorithms and Their Applications*. Lawrence Erlbaum Associates, 1987.
- [Gre93] J.J. Grefenstette. Deception considered harmful. In Rawlins [Raw93], pages 75–91.



- [GS87] D.E. Goldberg and P. Segrest. Finite markov chain analysis of genetic algorithms. In Grefenstette [Gre87], pages 1–8.
- [Har95] G.R. Harik. Finding multimodal solutions using restricted tournament selection. In Forrest [For95], pages 24–31.
- [Hil82] W.G. Hill. Predictions of response to artificial selection from new mutations. *Genet. Res.*, 40:255–278, 1982.
- [Him72] D.M. Himmelblau. *Applied Nonlinear Programming*. McGraw-Hill Book Company, 1972.
- [HM95] W. Hordijk and B. Manderick. The usefulness of recombination. In Moran et al. [MMMC95], pages 908–919.
- [Hol75] J.H. Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. The University of Michigan Press/Longman Canada, 1975.
- [Hol92] J.H. Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT Press, 1992.
- [Hol95] J.H. Holland. *Hidden order: how adaptation builds complexity*. MIT Press/Addison-Wesley, 1995.
- [HQL94] A. Homaifar, C.X. Qi, and S.H. Lai. Constrained optimization via genetic algorithms. *Simulation*, 62(3):242–253, 1994.
- [HS81] W. Hock and K. Schittkowski. *Test Examples for Nonlinear Programming Codes*, volume 187 of *Lectures Notes in Economics and Mathematical Systems*. Springer-Verlag, 1981.
- [IEE94] *Proceedings of the First IEEE Conference on Evolutionary Computation*. IEEE Press, 1994.
- [IEE95] *Proceedings of the 2nd IEEE Conference on Evolutionary Computation*. IEEE Press, 1995.
- [IEE96] *Proceedings of the 3rd IEEE Conference on Evolutionary Computation*. IEEE Press, 1996.
- [IEE98] *Proceedings of the IEEE World Congress on Computational Intelligence/fifth IEEE Conference on Evolutionary Computation (Vol. 1)*. IEEE Press, 1998.
- [KA96] K.E. Kinnear, Jr. and P.J. Angeline, editors. *Advances in Genetic programming*, volume 2 of *Complex adaptive systems*. MIT Press, 1996.

- [Kar95] H. Kargupta. SEARCH, polynomial complexity, and the fast messy genetic algorithm. Technical Report IlliGAL-95008, University of Illinois, October 1995.
- [Kar96a] H. Kargupta. Gene expression messy genetic algorithm. In IEEE [IEE96], pages 814–819.
- [Kar96b] H. Kargupta. The performance of the gene expression messy genetic algorithm on real test functions. In IEEE [IEE96], pages 631–637.
- [Kar96c] H. Kargupta. Search, evolution, and the gene expression messy genetic algorithm. Technical Report 96-60, Los Alamos National Laboratory, February 1996.
- [Kau93] S.A. Kauffman. *The origins of order*. Oxford University press, New York/Oxford, 1993.
- [KD96] J. Kingdon and L. Dekker. Morphic search. In IEEE [IEE96], pages 837–841.
- [KF95] J.N. Kok and P. Floréen. Tracing the behavior of genetic algorithms using expected values of bit and walsh products. In Forrest [For95], pages 201–208.
- [KGV83] S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.
- [Kin94] K.E. Kinnear, Jr., editor. *Advances in Genetic programming*. Complex adaptive systems. MIT Press, 1994.
- [Koz94] J.R. Koza. *Genetic Programming*. MIT Press, 1992–1994.
- [KR90] L. Kaufman and P.J. Rousseeuw. *Finding Groups in data, an introduction to cluster analysis*. John Wiley & Sons, Inc., 1990.
- [KR92] J.R. Koza and J.P. Rice. *Genetic Programming*. MIT Press, 1992.
- [LS96] D-G. Li and C. Smith. A new global optimization algorithm based on latin square theory. In IEEE [IEE96], pages 628–630.
- [Mah92] S.W. Mahfoud. Crowding and preselection revisited. In Männer and Manderick [MM92], pages 27–36.
- [MG92] S.W. Mahfoud and D.E. Goldberg. A genetic algorithm for parallel simulated annealing. In Männer and Manderick [MM92], pages 301–310.
- [MG95] S.W. Mahfoud and D.E. Goldberg. Parallel recombinative simulated annealing: a genetic algorithm. *Parallel computing*, 21:1–28, 1995.

- [Mic92] Z. Michalewicz. *Genetic Algorithms + Data structures = Evolution programs*. Springer-Verlag, Berlin Heidelberg, 1992.
- [Mic94] Z. Michalewicz. *Genetic Algorithms + Data structures = Evolution programs*. Springer-Verlag, 2nd edition, 1994.
- [Mic96] Z. Michalewicz. *Genetic Algorithms + Data structures = Evolution programs*. Springer, Berlin, 3rd edition, 1996.
- [Mit96] M. Mitchell. *An introduction to genetic algorithms*. MIT Press, 1996.
- [MJ91] Z. Michalewicz and C.Z. Janikow. Handling constraints in genetic algorithms. In Belew and Booker [BB91], pages 151–157.
- [MM92] R. Männer and B. Manderick, editors. *Proceedings of the 2nd Conference on Parallel Problem Solving from Nature, 2*. North-Holland, 1992.
- [MMMC95] F. Morán, A. Moreno, J.J. Merelo, and P. Chacón, editors. *Advances in Artificial Life. Third International Conference on Artificial Life*, volume 929 of *Lecture Notes in Artificial Intelligence*. Springer, Berlin, 1995.
- [MR98] H. Mühlenbein and A.O. Rodriguez. Schemata, distributions and graphical models in evolutionary optimization. Submitted for publication. Available via [ftp://borneo.gmd.de/pub/as/ga/gmd.as.ga-98\\_02.ps](ftp://borneo.gmd.de/pub/as/ga/gmd.as.ga-98_02.ps), May 1998.
- [MS96] Z. Michalewicz and M. Schoenauer. Evolutionary algorithms for constrained parameter optimization problems. *Journal of Evolutionary Computation*, 4(1):1–32, 1996.
- [MSV94] H. Mühlenbein and D. Schlierkamp-Voosen. Predictive models for the breeder genetic algorithm. *Journal of Evolutionary Computation*, 1(1):25–49, 1994.
- [Müh89] H. Mühlenbein. Parallel genetic algorithms, population genetics and combinatorial optimization. In Schaffer [Sch89], pages 416–421.
- [Müh92] H. Mühlenbein. How genetic algorithms really work I. mutation and hill-climbing. In Männer and Manderick [MM92], pages 15–25.
- [MV95] H. Mühlenbein and H.M. Voigt. Gene pool recombination in genetic algorithms. In I.H. Osman and J.P. Kelly, editors, *Metaheuristics international Conference, Norwell*. Kluwer Academic Publishers, 1995.
- [Obe85] Arnold Field Obe. *International Air Traffic Control; Management of the World's Airspace*. Pergamon Press, Oxford, 1985.

- [OO94] U.-M. O'Reilly and F. Oppacher. Program search with a hierarchical variable length representation: Genetic programming, simulated annealing and hill climbing. In Y. Davidor and Männer [YDM94], pages 397–406.
- [Rad91a] N.J. Radcliffe. Equivalence class analysis of genetic algorithms. *Complex Systems*, 5:183–205, 1991.
- [Rad91b] N.J. Radcliffe. Forma analysis and random respectful recombination. In Belew and Booker [BB91], pages 222–229.
- [Rad93] N.J. Radcliffe. Genetic set recombination. In Rawlins [Raw93], pages 203–219.
- [Raw91] G. Rawlins, editor. *Foundations of Genetic Algorithms*. Morgan Kaufmann, 1991.
- [Raw93] G. Rawlins, editor. *Foundations of Genetic Algorithms-2*. Morgan Kaufmann, 1993.
- [Rec73] I. Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Frommann-Holzboog, 1973.
- [Rec94] I. Rechenberg. *Evolutionsstrategie '94*. Frommann-Holzboog, 1994.
- [RS95] N.J. Radcliffe and P.D. Surry. Fitness variance of formae and performance prediction. In Whitley and Vose [WV95], pages 51–72.
- [RS97] N.J. Radcliffe and P.D. Surry. Real representations. In R.K. Belew and M.D. Vose, editors, *Foundations of Genetic Algorithms-4*, pages 343–363. Morgan Kaufmann, 1997.
- [Rud94] G. Rudolph. Convergence analysis of canonical genetic algorithms. *IEEE Transactions on Neural Networks*, 5(1):96–101, 1994.
- [Rud96] G. Rudolph. Convergence of evolutionary algorithms in general search spaces. In IEEE [IEE96], pages 50–54.
- [Sal71] W.C. Salmon. *Statistical Explanation and Statistical Relevance*. University of Pittsburgh Press, 1971.
- [SB96] G. Seront and H. Bersini. Simplex GA and hybrid methods. In IEEE [IEE96], pages 845–848.
- [Sch81] H.-P. Schwefel. *Numerical Optimization of Computer Models*. Wiley, Chichester, 1981.
- [Sch89] J.D. Schaffer, editor. *Proceedings of the 3rd International Conference on Genetic Algorithms*. Morgan Kaufmann, 1989.

- [Sch95] H.-P. Schwefel. *Evolution and Optimum Seeking*. Sixth-Generation Computer Technology Series. Wiley, New York, 1995.
- [Sco93] D.W. Scott. *Multivariate Density Estimation*. Wiley series in probability and Mathematical Statistics. John Wiley & Sons, INC., New York, 1993.
- [SEO91] J.D. Schaffer, L.J. Eshelman, and D. Offutt. Spurious correlations and premature convergence in genetic algorithms. In Rawlins [Raw91], pages 102–112.
- [Sla70] M. Slatkin. Selection and polygenic characters. In *Proceedings of the National Academy of Science U.S.A.* 66, pages 87–93, 1970.
- [SM91] H.-P. Schwefel and R. Männer, editors. *Proceedings of the 1st Conference on Parallel Problem Solving from Nature*, volume 496 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.
- [SP93] M. Srinivas and L.M. Patnaik. Binomially distributed populations for modelling gas. In Forrest [For93], pages 138–145.
- [SP96] R. Storn and K. Price. Minimizing the real functions of the icec'96 contest by differential evolution. In IEEE [IEE96], pages 842–844.
- [SR95] H.-P. Schwefel and G. Rudolph. Contemporary evolution strategies. In Morán et al. [MMMC95], pages 893–907.
- [SX93] M. Schoenauer and S. Xanthakis. Constrained GA optimization. In Forrest [For93], pages 573–580.
- [Sys89] G. Syswerda. Uniform crossover in genetic algorithms. In Schaffer [Sch89], pages 2–9.
- [TG93] D. Thierens and D.E. Goldberg. Mixing in genetic algorithms. In Forrest [For93], pages 38–45.
- [TG94] D. Thierens and D.E. Goldberg. Elitist recombination: an integrated selection recombination GA. In IEEE [IEE94], pages 508–512.
- [TGP98] D. Thierens, D.E. Goldberg, and A.G. Pereira. Domino convergence, drift, and the temporal-salience structure of problems. In IEEE [IEE98], pages 535–540.
- [Thi95] D. Thierens. *Analysis and Design of Genetic Algorithms*. Doctoral dissertation, University of Leuven, Belgium, 1995.
- [Thi96] D. Thierens. Dimensional analysis of allele-wise mixing revisited. In Voigt et al. [VERS96], pages 255–265.

- [TŽ89] A. Törn and A. Žilinskas. *Global optimization*, volume 350 of *Lecture Notes in Computer Science*. Springer, 1989.
- [VERS96] H.-M. Voigt, W. Ebeling, I. Rechenberg, and H.-P. Schwefel, editors. *Proceedings of the 4th Conference on Parallel Problem Solving from Nature – PPSN IV*, volume 1141 of *Lecture Notes in Computer Science*. Springer, Berlin, 1996.
- [vK95] C.H.M. van Kemenade. A two-level evolution strategy: balancing global and local search. In C. Decaestecker and T. van de Merckt, editors, *Proceedings of the Fifth Belgian-Dutch Conference on Machine Learning (TR/IRIDIA/95-16)*, pages 49–60. Université Libre de Bruxelles, 1995.
- [vK96a] C.H.M. van Kemenade. Cluster evolution strategies, enhancing the sampling density function using representatives. In IEEE [IEE96], pages 637–642.
- [vK96b] C.H.M. van Kemenade. Comparison of selection schemes for evolutionary constrained optimization. In *Proceedings of the Eighth Dutch Conference on Artificial Intelligence*, pages 245–254, 1996.
- [vK96c] C.H.M. van Kemenade. Explicit filtering of building blocks for genetic algorithms. In Voigt et al. [VERS96], pages 494–503.
- [vK97a] C.H.M. van Kemenade. Cross-competition between building blocks, propagating information to subsequent generations. In Bäck [Bäc97], pages 1–8.
- [vK97b] C.H.M. van Kemenade. The mixing evolutionary algorithm, independent selection and allocation of trials. In *Proceedings of the 4th IEEE Conference on Evolutionary Computation*, pages 13–18. IEEE Press, 1997.
- [vK97c] C.H.M. van Kemenade. Modeling elitist genetic algorithms with a finite population. In *Proceedings of the Third Nordic Workshop on Genetic Algorithms*, pages 1–10, 1997.
- [vK97d] C.H.M. van Kemenade. Modeling of genetic algorithms with a finite population. Technical Report SEN-R9726, CWI, Amsterdam, 1997.
- [vK98a] C.H.M. van Kemenade. Analysis of NK-xor landscapes. In IEEE [IEE98], pages 735–740.
- [vK98b] C.H.M. van Kemenade. Building block filtering and mixing. In IEEE [IEE98], pages 505–510.
- [vKE95] C.H.M. van Kemenade and A.E. Eiben. Multi-parent recombination to overcome premature convergence in genetic algorithms. In J.C. Bioch and Y.-H Tan, editors, *Proceedings of the Seventh Dutch Conference on Artificial Intelligence*, pages 137–146. Erasmus University, 1995.

- [vKHHK95] C.H.M. van Kemenade, C.F.W. Hendriks, H.H. Hesselink, and J.N. Kok. Evolutionary computation in air traffic control planning. In Forrest [For95], pages 611–616.
- [vKK95] C.H.M. van Kemenade and J.N. Kok. Time constrained routing. In J.T. Alander, editor, *Proceedings of the First Nordic Workshop on Genetic Algorithms*, pages 107–121. University of Vaasa, 1995.
- [vKKE95] C.H.M. van Kemenade, J.N. Kok, and A.E. Eiben. Controlling the convergence of genetic algorithms by tuning the disruptiveness of recombination operators. In IEEE [IEE95], pages 345–351.
- [vKKar] C.H.M. van Kemenade and J.N. Kok. Cluster evolution strategies for constrained numerical optimization. In *proceedings of the Eleventh International Symposium on Methodologies for Intelligent Systems*, LNCS/LNAI. Springer-Verlag, 1999 (to appear).
- [vKKvdA97] C.H.M. van Kemenade, J.N. Kok, and J.M. van den Akker. Evolutionary 3D-air traffic flow management. In Bäck et al. [BFM97], chapter G3.9.
- [vKPM98] C.H.M. van Kemenade, J.A. La Poutré, and R.J. Mokken. Density-based unsupervised classification for remote sensing. Technical Report SEN-R9810, CWI, Amsterdam, 1998. Also available via <http://www.cwi.nl/~hlp/PAPERS/RS/dense98.ps>.
- [vKPM99a] C.H.M. van Kemenade, J.A. La Poutré, and R.J. Mokken. Density-based unsupervised classification for remote sensing (extended abstract). In Kanellopoulos, Wilkinson, and Moons, editors, *Proceedings of Machine Vision In Remotely sensed Image Comprehension (MAVIRIC)*. Springer, 1999. Extended abstract of CWI report SEN-R9810.
- [vKPM99b] C.H.M. van Kemenade, J.A. La Poutré, and R.J. Mokken. Unsupervised class detection by adaptive sampling and density estimation. In A. Stein and F. van der Meer, editors, *Spatial Statistics and Remote Sensing*. Kluwer, 1999.
- [vKvdAK96] C.H.M. van Kemenade, J.M. van den Akker, and J.N. Kok. Evolutionary air traffic flow management for large 3D-problems. In Voigt et al. [VERS96], pages 910–919.
- [VM95] H.-M. Voigt and H. Mühlenbein. Gene pool recombination and utilization of covariances for the Breeder Genetic Algorithm. In IEEE [IEE95], pages 172–177.
- [VMC95] H.-M. Voigt, H. Mühlenbein, and D. Cvetković. Fuzzy recombination for the Breeder Genetic Algorithm. In Forrest [For95], pages 104–111.

- [vWEvK<sup>+</sup>97] M.C. van Wezel, A.E. Eiben, C.M.H. van Kemenade, J.N. Kok, W.A. Kusters, and I.G. Sprinkhuizen-Kuyper. Natural solutions to practical problems: An overview of marketing, scheduling and information filtering problems solved by neural and evolutionary techniques. In *Neural Networks: Best Practice in Europe, Proceedings Conference SNN'97, Amsterdam*, pages 202–205. World Scientific, Singapore, 1997.
- [Whi87] D. Whitley. Using reproductive evaluation to improve genetic search and heuristic discovery. In Grefenstette [Gre87], pages 108–115.
- [Whi89] D. Whitley. The GENITOR algorithm and selective pressure. In Schaffer [Sch89], pages 116–121.
- [Whi91] D. Whitley. Fundamental principles of deception in genetic search. In Rawlins [Raw91], pages 221–241.
- [Whi93] D. Whitley. An executable model of a simple genetic algorithm. In Rawlins [Raw93], pages 45–62.
- [WS90] D. Whitley and T. Starkweather. GENITOR II: A distributed genetic algorithm. *Journal of Experimental and Theoretical Artificial Intelligence*, 2:189–214, 1990.
- [WV95] L.D. Whitley and M.D. Vose, editors. *Foundations of Genetic Algorithms-3*. Morgan Kaufmann, 1995.
- [YDM94] H.-P. Schwefel Y. Davidor and R. Männer, editors. *Proceedings of the 3rd Conference on Parallel Problem Solving from Nature - PPSN III*, volume 866 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.



# Samenvatting

Evolutionair rekenen behelst het gebruik van een gesimuleerd evolutieproces voor het oplossen van optimalisatieproblemen, voor het verkrijgen van (zelf-)adaptieve systemen en voor het modelleren van het evolutieproces. Evolutionaire rekenmethoden gebruiken een populatie van individuen. Deze individuen zijn in competitie met elkaar voor spaarzame hulpbronnen, waarbij de verdeling van deze bronnen over de individuen gebaseerd wordt op de verschillen in prestatie tussen de verschillende individuen. Naarmate een individu beter presteert, maakt het een grotere kans op het verkrijgen van hulpbronnen.

Het hoofdthema van dit proefschrift is recombinationeel evolutionair zoeken. Een van de redenen waarom evolutionair zoeken zo succesvol is in de natuurlijke evolutie is het parallelle karakter van dit zoekmechanisme. Dit parallellisme vloeit voort uit het feit dat evolutionair zoeken een populatie-gebaseerde zoekmethode is. Dit parallellisme wordt nog eens versterkt door het gebruik van recombinitie, want dan is het mogelijk verschillende sterke eigenschappen van goede oplossingen onafhankelijk van elkaar te ontdekken en daarna pas te recombineren. Recombinatieel evolutionair zoeken tracht dezelfde eigenschap uit te buiten voor gesimuleerde evolutie. De primaire componenten van recombinationeel evolutionaire zoekmethoden zijn een fitness-gebaseerde selectie en een recombinitie operatie. Beide componenten zijn van belang voor het verkrijgen van een betrouwbare evolutionaire rekenmethode. In dit proefschrift bestuderen we de condities voor het verkrijgen van betrouwbare zoekmethoden, zoeken we naar de redenen dat traditionele recombinationeoperators soms falen op bepaalde binair gecodeerde optimalisatieproblemen, tonen we aan hoe de toepasbaarheid van recombinitie vergroot kan worden voor sommige van deze problemen, en geven we enkele succesvolle toepassingen van recombinitie-gebaseerde evolutionaire rekenmethoden. In het bijzonder kijken we naar het ontdekken en mengen van bouwblokken, ontwikkelen we meer effectieve recombinitieoperators die additionele informatie uitbuiten, en ontwikkelen we selectiemechanismen en recombinitieoperators voor een aantal verschillende toepassingsgebieden.

Evolutie kan beschouwd worden als een robuust controlemechanisme voor het bijsturen van een complex adaptief systeem, maar het is ook mogelijk om evolutie te zien als een optimalisatieproces. Deze twee perspectieven op evolutie zijn beide zinvol. In mijn onderzoek heb ik de nadruk gelegd op evolutie voor optimalisatie.

Dit proefschrift bevat twee delen. Het eerste deel (hoofdstuk 1–5) geeft een korte introductie en heeft betrekking op theorie voor genetische algoritmen. Het tweede deel (hoofdstuk 6–12) gaat over empirisch en toegepast onderzoek.

Hoofdstuk 1 geeft een overzicht over het gebied van de evolutionaire rekenmethoden. Een meer gedetailleerde introductie volgt in hoofdstuk 2. In hoofdstuk 3 wordt een eenvoudig optimalisatieprobleem geïntroduceerd, en het gedrag van evolutionaire algoritmen toegepast op dit probleem wordt onderzocht door middel van enkele modellen. Hoofdstuk 4 beschrijft het zogenaamde transmissiefunctieraamwerk, en bevat implementaties van transmissiefunctiemodellen voor een aantal verschillende evolutionaire algoritmen. Deze algoritmen beschrijven het gedrag van een genetisch algoritme (GA) met een oneindig grote populatie. GA's gebruiken eindige populaties, en daarom zijn extensies van de transmissiefunctie modellen voor het modelleren van GA's met eindige populaties ontwikkeld in hoofdstuk 5. Hoofdstuk 6 introduceert het "mixing evolutionary algorithm (MixEA)". Dit algoritme legt een grotere nadruk op het mengen van bouwblokken dan de meer traditionele GA's. In hoofdstuk 7 wordt een drie-fase methode, het zogenaamde bbf-GA, geïntroduceerd. Dit algoritme separeert het exploratie- en het exploitatieproces dat simultaan plaatsvindt in de meer standaard GA's. Hoofdstuk 8 geeft een serie testproblemen, en gebruikt deze om een prestatievergelijking te maken tussen de verschillende GA's die beschreven zijn in dit proefschrift. Een van de interessante eigenschappen van GA's is dat deze algoritmen relatief eenvoudig verrijkt kunnen worden door middel van het invoegen van heuristieken en lokale optimalisatiemethoden. In hoofdstuk 9 introduceren we de "Cluster Evolution Strategies (CLES)" voor numerieke optimalisatie. CLES maakt een expliciet onderscheid tussen globaal zoeken (exploratie) en lokaal zoeken (exploitatie). Hoofdstuk 10 toont de toepassing van een aantal GA's, waaronder CLES voor numerieke optimalisatieproblemen met restricties. In hoofdstuk 11 wordt de toepassing van GA's met zogenaamde diagonale crossover-operatoren besproken. Een toepassing van GA's voor het zogenaamde "Air Traffic Flow Management" is beschreven in hoofdstuk 12.

# Curriculum Vitae

Cornelis (Cees) Hendricus Maria van Kemenade werd op 24 maart 1969 geboren te Uden. Hij behaalde op 13 juni 1988 zijn VWO diploma aan het Kruisheren Kollege te Uden. Tevens was hij deelnemer van het Nederlandse team bij de Internationale Natuurkunde Olympiade 1988. Hij ging Natuurkunde studeren aan de Universiteit Utrecht en behaalde een propaedeuse in de Natuurkunde op 26 februari 1990, een propaedeuse in de Wiskunde op 26 februari 1990 en een propaedeuse in de Informatica op 26 februari 1990. Op 29 november 1993 behaalde hij zijn doctoraal in de Natuurkunde en op 31 augustus 1994 behaalde hij zijn doctoraal in de Informatica aan de universiteit van Utrecht. In dezelfde periode was hij actief als penningmeester van een studievereniging, secretaris van een stichting en mede-organisator van een studiereis naar Zwitserland en Spanje. Op 1 september 1994 trad hij in dienst van het Centrum voor Wiskunde en Informatica (CWI) te Amsterdam. In dezelfde periode was hij werkzaam als wetenschappelijk onderzoeker binnen een internationaal samenwerkingsverband met het "Cold Regions Research and Engineering Laboratory (RSGIS/CRREL)" en het "Center for Computer Science in Organization and Management (ALL/CCSOM)" van de Universiteit van Amsterdam. Tevens was hij voorzitter van het organiserend comité van de Nederlandse/Belgische conferentie voor kunstmatige intelligentie (NAIC'98) die plaatsvond op 17 en 18 november 1998 in Amsterdam.



# Index

- air traffic flow management, 240–241
- allele, 3
- ATFM, *see* air traffic flow management
- BBF, *see* building block filtering
- BGA, *see* breeder genetic algorithm
- bias, 8
- BinInt, 32
- breeder genetic algorithm, 7, 56
- building block, 28–31
  - compatible, 29
  - non-overlapping, 29
- building block filtering, 122–126
- building block filtering GA, 121–128
  - filtering method, 122–126
  - masked crossover, 126–127
- chromosome, 3
- CLES, *see* cluster evolution strategy
- cluster evolution strategy, 180–188
  - clustering method, 182–184
  - evolutionary operators, 184–187
  - local search strategy, 187–188
- compatible, 138
- complex adaptive system, 5–6
- conferences, 3
- cross-competition, 34, 42–47, 64–65
- crossover, *see* recombination, 4
  - diagonal, 21–22
  - n*-point, 21
  - one-point, 20–21
  - uniform, 21, 37–42
- Darwin, 2
- deception, 29–31
- deceptive problem, 29–31, 116, 129, 139–140
- deterministic crowding, *see* selection
- dimensional analysis, 34–36
- DNA, 3
- ecology, 5
- elitist recombination, *see* selection
- epistasis, 4, 29
- evolution, 3
- evolution strategies, 12
- evolution strategy, 2
- evolutionary computation, 1
- evolutionary operators, 20–22
- evolutionary programming, 12
- fitness, 3
  - scaling, 20
- forma analysis, 28
- frustration, 138
- gene, 3
- genetic
  - drift, 32, 36
  - hitch-hiking, 31
- genetic algorithms, 11–12
- genetic programming, 12–13
- genome, 3
- genotype, 3
- I*-box, 80
- implicit parallelism, 27
- intrinsic parallelism, 27
- linkage, 4, 29, 137–138
- locus, 3
- masked crossover, 135
- masked uniform crossover, 126–127

- massively multimodal problem, 139
- Mendel, 2
- mixEA, *see* mixing evolutionary algorithm
- mixing, 34, 40–42, 47–51
- mixing evolutionary algorithm, 109–116
- multimodality, 138
- mutation, 4, 20, 22
  
- niche, 5
- NK-bb problem, 129, 144–145
- NK-landscape, 36, 144
- numerical constrained optimization, 194–195
  
- oneMax problem, 20, 36
- optimization
  - black-box, 8
  - direct, 8
  - function, 19
  - indirect, 8
  
- P*-box, 80
- penalty functions, 194–195
- phenotype, 3
- population flow diagram, 65–73
  
- random sampling method, 175–177
- random-mutation hill-climber, 146
- recombination, *see* crossover, 4, 20
  - assortment, 28
  - efficiency, 37–40
  - gene-pool, 21–22
  - multi-parent, 21–22, 222–224
  - respectfulness, 28
- reproduction
  - asexual, 4
  - sexual, 4
- RMHC, *see* random-mutation hill-climber
- royal road problem, 138–139
  
- sampling error, 32–33
- schema, 27
  - defining length, 27
  - fitness, 27, 29–30
  - order, 27
- schema theorem, 27
- search
  - hill-climbing, 9
  - path-oriented, 9
  - recombination-based, 10–11
  - volume oriented, 9
- selection
  - CHC, 57
  - deterministic crowding, 26–27
  - elitist recombination, 26–27, 58–61
  - fitness proportional, 20, 23–24, 54–55
  - local competition, 26–27
  - $(\mu + \lambda)$ , 26, 56–57
  - $(\mu, \lambda)$ , 24–25, 55–56
  - roulette-wheel, 23–24
  - soft-brood, 59
  - steady-state, 25–26
  - stochastic universal sampling, 23–24, 33
  - tournament, 24, 55
  - triple-competition, 25–26, 57–58
  - truncation, 24–25
- selection scheme, 23–27
  - generational, 23–25
- selective pressure, 4
  - fertility, 4
  - response, 5
  - viability, 4
- SP*-box, 80
- species, 5
- stepping-stones problem, 195–197
  - SSP-a, 195–197
  - SSP-b, 197
- stochastic universal sampling, *see* selection
  
- transmission function, 54
- trap function, 29–31
  
- unitation, 20
  - equivalence classes, 63–64