GENE REORDERING AND CONCURRENCY IN GENETIC ALGORITHMS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
THE MIDDLE EAST TECHNICAL UNIVERSITY

BY

ONUR TOLGA ŞEHİTOĞLU

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN

THE DEPARTMENT OF COMPUTER ENGINEERING

AUGUST 2002

Approval of the Graduate School of Natural and Applied Sciences.

Prof. Dr. Tayfur Öztürk
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Doctor of Philosophy.

Prof. Dr. Ayşe Kiper
Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Doctor of Philosophy.

Assoc. Prof. Dr. Göktürk Üçoluk
Supervisor

Examining Committee Members

Prof. Dr. Faruk Polat

Assoc. Prof. Dr. İlyas Çiçekli

Assoc. Prof. Dr. Hakkı Toroslu

Assoc. Prof. Dr. Göktürk Üçoluk

Inst. Dr. Cevat Şener

ABSTRACT

GENE REORDERING AND CONCURRENCY IN GENETIC ALGORITHMS

Şehitoğlu, Onur Tolga

Ph.D., Department of Computer Engineering

Supervisor: Assoc. Prof. Dr. Göktürk Üçoluk

August 2002, 90 pages

This study first introduces an order-free chromosome encoding to enhance the performance of genetic algorithms by learning the linkage of building blocks in non-binary encodings. The method introduces a measure called *affinity* which is based on the statistical properties of gene valuations in the population. It uses the affinity values of the local and global gene pairs to construct a global permutation with tight building block positioning. Method is tested and experimental results are shown for a group of deceptive and real life test problems.

Then, study proposes a gene level concurrency model where each gene position is implemented on a different process. This combines the advantages of implicit parallelism and a chromosome structure free approach. It also helps implementation of gene reordering method introduced and probably other non-linear chromosome encodings.

Keywords: genetic algorithms, concurrency, reordering, linkage learning, deceptive problem

ÖZ

GENETİK ALGORİTMALARDA GEN YENİDEN SIRALAMA VE EŞZAMANLILIK

Şehitoğlu, Onur Tolga
Doktora, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi: Doç. Dr. Göktürk Üçoluk

Ağustos 2002, 90 sayfa

Bu çalışma öncelikle genetik algoritmaların başarımlarını ikili olmayan alfabelerde yapıtaşı bağlılıklarını öğrenerek arttırmayı amaçlayan sıralamadan bağımsız bir kodlama önermektedir. Sunulan yöntem geçinebilirlik olarak adlandırılan ve nüfustaki gen değerlerinin istatistiksel özelliklerine dayanan bir ölçüt tanımlar. Yöntem yerel ve genel gen çiftleri arasındaki *geçinebilirlik* değerlerini kullanarak yakın yapıtaşı yerleşimleri oluşturan genel bir permütasyon oluşturmaya çalışır. Bu yöntem bir grup yanıltıcı ve gerçek yaşam probleminde denenmiş ve deney sonuçları gösterilmiştir.

Çalışma daha sonra her gen konumunun ayrı bir işlem olarak gerçekleştirildiği gen düzeyinde bir eşzamanlılık modeli önermektedir. Model, bu tanımın doğasından gelen paralelikle kromozom yapısından bağımsız bir yaklaşımın avantajlarını birleştirmektedir. Ayrıca ilk kısımda tanımlanan gen sıralama yöntemi ve olası diğer doğrusal olmayan koromozom kodlamalarını olanaklı kılmaktadır.

Anahtar Kelimeler: genetik algoritmalar, eşzamanlılık, yeniden sıralama, bağlılık öğrenme, yanıltıcı problem

ACKNOWLEDGMENTS

I would like to thank Dr. Göktürk Üçoluk for his warm friendship and support. It would be impossible to complete this study without his motivation, guidance and encouragement.

I want to thank all my friends, especially Dr. Meltem Turhan Yöndem and Dr. Cevat Şener for their continuous help and motivation. I also want to thank my family for their patience, and their limitless and precious love.

To my mother and my father

TABLE OF CONTENTS

ABSTRACT iii

ÖZ iv

ACKNOWLEDGMENTS v

DEDICATON vi

TABLE OF CONTENTS vii

LIST OF FIGURES x

LIST OF TABLES xii

CHAPTER

1 INTRODUCTION 1

1.1 Genetic Algorithms: Introduction and Basic Definitions 3

1.2 Scope of the Thesis 5

2 BUILDING BLOCKS AND LINKAGE LEARNING 7

2.1 Schema Theorem and Building Block Hypothesis 7

2.2 Mixing of Building Blocks and Linkage 10

2.3 Linkage Learning Problem 13

2.3.1 Early Efforts 13

2.3.2 Symbiotic Evolution 14

2.3.3 Messy GA 15

2.3.4 Linkage Learning GA 16

2.3.5 Distribution Estimation and Bayesian Optimization Algorithm 17

3 PARALLEL GENETIC ALGORITHMS 18

3.1 A Classification of PGAs 19

3.1.1 Global Parallelization 20

3.1.2 Coarse Grained PGAs 22

3.1.2.1 Traditional GAs 22
3.1.2.2 Non-traditional GAs 25
3.1.3 Fine Grained PGAs 26
3.1.4 Hybrid Algorithms 28
3.2 Underlying Problems 29
4 GENE REORDERING GENETIC ALGORITHM 31
4.1 Global Reordering and Affinity 32
4.1.1 Genetic Algorithm Engine 35
4.1.2 Implementation 36
4.1.3 Test results 38
4.1.4 Affinity Functions 40
4.2 Greedy Maximum Affinity Sequence 43
4.2.1 Generalized Assignment Problem 45
4.2.2 Problem Encoding and Test Parameters 46
4.3 Test Problems 49
4.3.1 Experiment Setting 50
4.3.2 Experiment Results 51
4.3.3 Observing the Linkage 54
4.4 Non-binary Deceptive Functions and the Linkage 55
4.4.1 Experiment Setup 58
4.4.2 Experimental Results 59
5 GENE LEVEL CONCURRENCY 64
5.1 Gene Process Model 64
5.2 Simple GA Implementation 67
5.3 Parallel Implementation Considerations 70
5.4 Gene Reordering Genetic Algorithms and Gene Level Concurrency 72
5.5 Implementation 75
5.6 Other Possible Architectures 75
6 SUMMARY AND CONCLUSIONS 78
6.1 Summary 78
6.2 Conclusions 80
6.3 Future Work 83
REFERENCES 84

VITA 89

LIST OF FIGURES

FIGURES

1.1 Single point crossover 4
1.2 Mutation of the fifth gene 5

2.1 Mixing cases of single point crossover for different configuration of building blocks: $tight - random$, and $random - random$ 11
2.2 Expected value plots for mixing cases $tight - tight$, $tight - random$, and $random - random$ configurations of building blocks. 13
2.3 Cut and splice operators of the messy GA 15

4.1 Modification of order according to affinity values 35
4.2 Evolution of the best individual for a typical run 39
4.3 Distribution of best fitness values for $Keep$ value 10 and 5 for 20 executions 40
4.4 Best individuals fitness value after 1000 generation. Negative difference value means gene moved version has a better solution 41
4.5 General assignment problem results: (a) 60 jobs 10 agents (b) 100 jobs 10 agents. Average of 100 runs 48
4.6 Plots of the two functions that are used in the tests.(a) is the $f\_6(x, y)$ plot and (b) is the $f\_10(x, y)$ plot. 49
4.7 Results of the f_6 function execution for four test cases, average of the 100 repetition (b) is the zoomed version of the same plot 52
4.8 Results of the F_10 function execution for four test cases, average of the 100 repetition (b) is the zoomed version of the same plot 53
4.9 Average linkage of the F_10 function execution for four test cases, average of the 100 repetition 55
4.10 A trap function of order 3 and the corresponding graph. 56
4.11 Integer trap function used in the experiments The $f\_intdec(d)$ function where d is the Euclidean distance of the point from the predefined global optimum. 56
4.12 3D plot of the order 2 fitness function derived from $f\_intdec(d)$ function for global optimum is at (0,0) 57
4.13 (a):Best fitness evolution of the deceptive function with $k = 2, N = 16$, (b): Average linkage 61
4.14 (a):Best fitness evolution of the deceptive function with $k = 3, N = 15$, (b): Average linkage 62

4.15 Distribution of linkage values during the whole evolution states for 100 repetitions (a) order 2 and (b) order 3 63

5.1 Architecture of the gene process model 65
5.2 Speed-up plots for different parameters 71
5.3 A possible two-dimensional chromosome representation. 76
5.4 A possible hierarchical chromosome representation. 77

LIST OF TABLES

TABLE

4.1 Expected linkage values calculated from the linkage frequencies 60

5.1 Computation costs of the serial and parallel cases 70

CHAPTER 1

INTRODUCTION

Genetic algorithms are search algorithms based on the mimic of the evolution in nature. Many mechanisms in the natural evolution like mating, mutation, survival of the fittest are used to evolve a group of candidate solutions to find an optimal solution. Especially for the last 20 years, genetic algorithms have been used in many areas of the optimization and machine learning problems. They have been applied to many real life problems to find acceptable solutions in domains in which even finding an acceptable solution is not a trivial task.

The Bagley's work (1967) [7] introduced the words *Genetic Algorithms* in the scientific literature and Holland's work *Adaptation in Natural and Artificial Systems* (1975) [40] put the first theoretical foundations. Since then there has been many attempts to understand their nature and characteristics to design more competent genetic algorithms and to use their power in its maximum extend in the applications.

Currently there are two theoretical bases of how and why genetic algorithms work: schema theorem [40] and building block hypothesis [27]. Both theories suggest that the performance of the genetic algorithms depends on the combination and growth of short, low order, and high fit *building blocks* of the problem representation. This implies that the encoding and representation of the problem is one of the most critical points in the success of the genetic algorithms. The encodings and representations, providing the building block supply, growth and combination, will affect the genetic algorithm performance significantly.

Encoding a problem in genetic algorithms is to introduce a sample space that contains all possible solutions with an acceptable accuracy and represent each point in the sample space by a vector of values of a –usually binary– alphabet. This vector is called the *chromosome*. Practically, a chromosome is a vector of the encoding of the parameters of subject problem. The encoding of each parameter is called a *gene*. The placement of each gene on the chromosome is a degree of freedom, the GA implementer has. This thesis is partly concerned with this issue so that the wise use of this freedom leads to a faster and better solution.

Genetic algorithm's ability to solve a problem is provided by its ability to group this values into higher order building blocks. This construction of better solution vectors based on former solution vectors is done by means of a split-and-recombine process, which is called *crossover*. Most of the crossover operations split several vectors into segments, and then recombine the generated segments. Obviously, the order of the encoding, namely the gene order bears an importance in this splitting process.

The problem of having genes ordered in the chromosome in various permutations to allow building block growth is called the *ordering problem*, and to invent algorithms and methods for adaptively learning good orderings in the genetic algorithms is called *linkage learning*[28, 38]. Ordering problem and linkage learning have been one of the popular areas in the theory of genetic algorithms for the last 15 years.

Another popular area in the genetic algorithms is the parallelization of the genetic algorithms. In order to decrease the processing time for a solution, parallel architectures have been introduced and implemented by many researchers. While some of the works employ explicitly parallelization of the genetic algorithms (much like the parallelization of any algorithm), some introduce changes to the architectural composition the way genetic algorithms function. While parallelizing the genetic operations (e.g. crossover, mutation, fitness calculation) over the population is an example to the efforts of the first kind, population migration is a good example to the second.

This study contributes to Genetic Algorithms in two aspects:

- Introduction of a gene reordering method to improve the performance of the genetic algorithms by linkage learning, and

- the definition of a concurrent model dividing the population into a group of concurrently executing and interacting gene entities.

These two aspects are combined into a concurrent model of the gene reordering method introduced.

For the first aspect, a new measure called *affinity* among the gene positions in the population is introduced. Then, this measure is used to adaptively change the ordering of the population to favor closely placed building blocks. Two versions of the affinity-based methods are introduced. The first of these uses a local approach while the second considers all gene pairs globally. Methods are implemented on two real life problems, genetic algorithm test problems, and deceptive functions of different orders. and their performance is analyzed empirically.

For the second aspect, a concurrent definition, where gene process are modeled as communicating objects, is made. The model proposes an approach to genes which is not implementation-wise handicapped by any orientation restriction, so that variable order and/or orientation realization of genetic algorithms will be possible with the advantages of the concurrent execution and implicit parallelism. Working mechanism and implementation details of the model is given with the details of how gene reordering method is implemented.

1.1 Genetic Algorithms: Introduction and Basic Definitions

GAs involve usually a fixed-length string representation of individuals (solution candidates) in the population. Evolution is achieved by genetic operators like mutation and crossover. Each individual is assigned a *fitness score* depending on how good the encoded solution is. Individuals with high fitness values are given opportunities to live and reproduce. By genetic operators new individuals are created as *offsprings*. Individuals with low fitness values have less chance to reproduce in the next population so their code cannot live for long generations. By giving high fit members more chance to reproduce, a new *generation* with higher proportion of the characteristics possessed by the good members of the previous generation will possibly emerge. Evaluation process is iterated until a population with a good solution is found.

As it was mentioned in the previous section, one of the most critical issue in a success of an evolutionary algorithm is the *encoding* (representation) of the problem. *Encoding* is the set of parameters of a potential solution of the problem. Each parameter is called a *gene*, and genes are joined together to form a vector of parameters

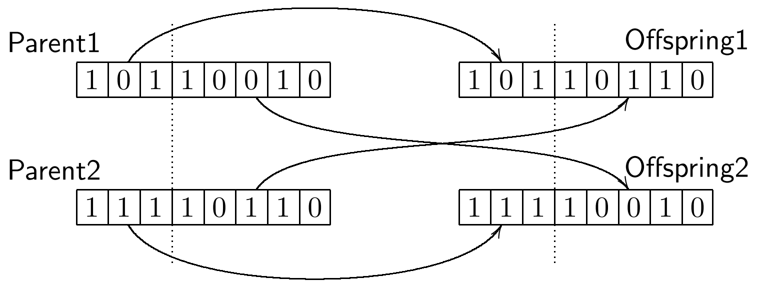


Figure 1.1: Single point crossover

which is called a *chromosome*.

The set of terms represented by a particular chromosome is referred to as a *genotype* whereas the solution is called a *phenotype*. For example, if a problem seeks for the solution of a vector of 10 integer values, this vector is the phenotype. Usually binary alphabet is used for genes in the genetic algorithms so each integer is converted into a sequence of bits let's say 8 bits each. The resulting bit string of length 80 will be the genotype. The set of feature values that a gene can take is called an *allele*. For a representation, a function returning a single comparable value evaluating the success of the represented chromosome should be defined. This function is called as the *fitness function*. The selection among individuals is made according to this function by emulating the *survival of the fittest* rule of the nature.

Reproduction of the population is possible by using *crossover* and *mutation* operations. The randomly selected individuals produce new *offsprings* by the application of these operators.

There are various crossover techniques. *One point crossover* takes two individuals, and cuts their chromosome strings at some randomly chosen position to produce two head and two tail segments. Two offsprings are produced by swapping the tail segments. Crossover is applied with a probability typically greater than 0.6. So that individuals are given a chance of passing its features to new generations without disruption. Instead of *single-point crossover*, crossover can be applied at two points (*2-point crossover*) or at more points (*n-point crossover*) which might improve the performance. In *uniform crossover*, among two or more parents, each gene is transferred to an offspring by a preassigned global probability.

Mutation randomly alters or modifies any gene with a small probability. It is usually applied after crossover. It is mimicking the anomalies that rise up due to external disruptive effects or any faulty genetic process, in nature. Computationally speaking, it is a stochastic mechanism which enables escapes from local minima.

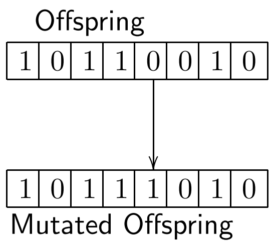


Figure 1.2: Mutation of the fifth gene

These operators provide two different search characteristics. Crossover does a rapid exploration of a region of a search space whereas mutation directs search to random points in the search space to assure no unsearched region is left.

A typical generation loop for a genetic algorithm is:

- Select chromosomes to be reproduced
- Crossover chromosome pairs
- Mutate resulting offsprings
- Evaluate fitness of the new generation
- Replace the current generation
- Go to the beginning

The term *epistatis* has been defined by geneticists to name the influence of the dependency among two genes on the global fitness of the chromosome. A particular change in a gene may produce a major change in the fitness value of the chromosome depending on the value of the other genes. This small change may cause a chain reaction where dependent genes depend on others. Depending on the problem domain and coding this may lead to a divergence instead of a convergence which is called as *deception*.

1.2 Scope of the Thesis

Chapter 2 starts with the explanation of the Schema Theorem and Building Block Hypothesis. Then, mixing of building blocks and the relation of linkage with mixing is analyzed. At the end of this chapter a summary of related work on linkage learning is given.

Chapter 3 provides a brief survey of the studies on parallelization of genetic algorithms. Three class of approaches, (i) global parallelization, (ii) coarse grain parallelization and (iii) fine grain parallelization (iv) hybrid algorithms are described with references to the existing studies.

Chapter 4 introduces a new concept: "affinity" and based on this structures a method for gene reordering. The proposed method is explained in detail. Two basic gene reordering models, (i) with local neighborhood, and (ii) by looking at all pairs are described. The experimental results on the minimum bounding area rectangle placement problem and several test problems are given. To analyze the relation between linkage and affinity, non-binary deceptive functions are introduced and the evolution and linkage learning performance is tested with order 2 and order 3 building blocks.

Chapter 5 gives the details of the gene level concurrency model. The model is described formally for a simple genetic algorithm. Then, by extending the model gene reordering method is implemented. Also other possible architecture are proposed.

Chapter 6 than summarizes the dissertation and discusses the results of the experiments and proposed methods.

CHAPTER 2

BUILDING BLOCKS AND LINKAGE LEARNING

There have been many researches to understand the theoretical foundations of genetic algorithms, but many of the questions do not have a complete answer yet. Those questions are [45]:

- Which laws describe the general behavior of GA?
- How genetic operators effect GA behavior?
- Which type of problems are suitable and which are not for GA?
- What is the definition of good performance for GA?
- Under which conditions GA outperforms other search methods?

Answers of these questions play a crucial role in the application and development of genetic algorithms.

2.1 Schema Theorem and Building Block Hypothesis

Holland's book "Adaptation in Natural and Artificial Systems" [40] gives the earliest explanation of why genetic algorithms work. A *schema* is a template to analyze the similarities among the chromosomes. In addition to the binary alphabet $\{0, 1\}$, a *don't care* symbol $*$ is introduced in the schema vectors. This symbol matches both 0 and 1. While a binary chromosome vector represents a point in the sampled hyperspace, a schema represents an hyperplane, collection of all points matching the schema. For example, schema (0**) matches all binary strings of length 3 starting with 0. So (000), (001), (010), (011) are matched by this schema. Similarly schema (01*01**0) matches 8 different strings, (11001100) matches only itself, and (********) matches all binary strings of length 8 (2^8 different strings).

A schema with r * symbols will match 2^r schema. Similarly a binary string of m is matched by 2^m different schemata. In GA interpretation, the *fixed* (determined) symbols in the schema represent the bits matching the solution, * symbol represents the bits that the correctness is not known. For example, in a case where the global optimum is obtained by setting all bits to one (1111111111), the (1*1**11***) schema represents the hyperplane where 4 bits are fixed. So this schema refers to a reduced form of the search space.

The *order* of a schema S, $o(S)$, is defined as the number of fixed points in the schema. The *defining length* of a schema S, $\delta(S)$ is defined as the distance between the leftmost and the rightmost fixed points of the schema. For example schema (*100**1**1**) has an order 5 and defining length 8. The order of a schema determines how much of the solution space is partitioned by the schema, and defining length determines the compactness of the schema which will be used in the following analyses of the genetic operators.

Initial GA population is assumed to contain schemata that is sufficient to construct the solution by smart recombinations of the individuals. Given a schema S, the number of representatives of this schema in the population at time t is represented by $\xi(S,t)$. Another property of the schema is the *fitness* of a schema, $f(S)$, which is the average fitness of the individuals matching the schema S. When the fitness proportionate selection is used the selection ratio of the individuals of the schema is $f(S)/f_a$ where f_a is the average fitness of the whole population. Therefore, only considering the selection the expected number of the schema representatives in the next generation can be written as:

$$\xi(S,t+1) = \xi(S,t)f(S)/f_a$$

To include *crossover* and *mutation* in the analysis, assumptions about the effect of these operators on the number of the schema should be done. Assuming the worst case, these operators are always assumed to be destructive on the schemata. When a crossover point is enclosed by the fixed points of the schema, it will destruct the schema. The probability of this to happen is directly related with the defining length of schema so with probability of $p_c\delta(S)/(l-1)$ the schema will be destroyed where p_c is the crossover probability, l is the length of the problem.

Similarly when *mutation* matches one of the fixed points of the schema, it destroys the schema. The probability of schema to be destroyed by a mutation is $p_m o(S)$ where p_m is the mutation probability. Adding these two into the $\xi(S, t+1)$ estimate the following inequality is found:

$$\xi(S, t+1) \geq \frac{\xi(S,t)f(S)}{f_a}\left(1 - \frac{p_c\delta(S)}{(l-1)} - p_m o(S)\right)$$

Looking at this formula, the schema growth is related to three parameters: $f(S)$, $\delta(S)$ and $o(S)$. Final conclusion of the growth equation is:

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

This theorem is followed by the building block hypothesis by Goldberg [28, 45]:

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

Relaying on this hypothesis, having encodings with short and low-order building blocks is critical for the performance of the GA. However there is a class of problems in which short, low-order schemata lead to suboptimal solutions instead of the global one. This class of problems are called *deceptive* problems. Considering only low order building blocks genetic algorithm converges to a local optimum. This is related to the term *epistatis* which is the strong relation among the genes in the chromosome. In other words the contribution of the gene to the overall fitness is dependent on the values of the other genes. It is just the opposite of what the "applicability of divide-and-conquer" implies for a problem.

In [27] three ways of dealing with deception is suggested. In the first one it is assumed that the characteristics of the fitness function and deception is already known and related genes are placed closely to have tight building blocks, but this assumption usually does not hold in real life problems. The other two suggestions are about changing the encoding of the problem adaptively to have building blocks shorter. These two suggestions, the *inversion operator* and the *messy GA*, will be mentioned later in this chapter.

2.2 Mixing of Building Blocks and Linkage

In [29] GA is defined as a 6 piece puzzle with pieces:

1. knowing what GAs process: building blocks,
2. ensuring an adequate initial building block supply,
3. guaranteeing that building blocks grow,
4. making building block decisions well,
5. solving problems that are not building block difficult, and
6. ensuring that building blocks exchange to form better solutions.

These items emphasize the importance of GAs design choices, including initialization of the population, parameters like population size, crossover and mutation probabilities, selection strategy etc. In [32], it is expressed that although there have been a substantial progress in the first 5 items almost no work has considered the building block exchange.

While it is critical to supply, grow and select the building blocks it is also very crucial to combine a building block in one structure with the building block in some other structure in order to form a structure with a larger number of building blocks. So –especially nondestructive– effects of crossover need to be analyzed further.

Authors combine and verify empirical and theoretical results to define a region between forces of selection and crossover where proper mixing can occur. This region is defined by the inequality:

$$p_c < \frac{e^{l/n} \ln s}{n \ln n}$$

Where p_c is the crossover probability, l is the problem length, s is the tournament size of the selection. Outside of the region, with small sizes of selection genetic drift, the domination of a few schemata in the entire population occurs. With large sizes of selection, cross-competition of building blocks, where no mixing could happen, occurs.

Another important parameter in the successful mixing of the building blocks is the linkage, namely *how the building blocks are grouped together to facilitate the growth and mixing of them.* As the schema theorem concludes the defining length of the schemata should be small for the schema growth. *When the crossover strategy*

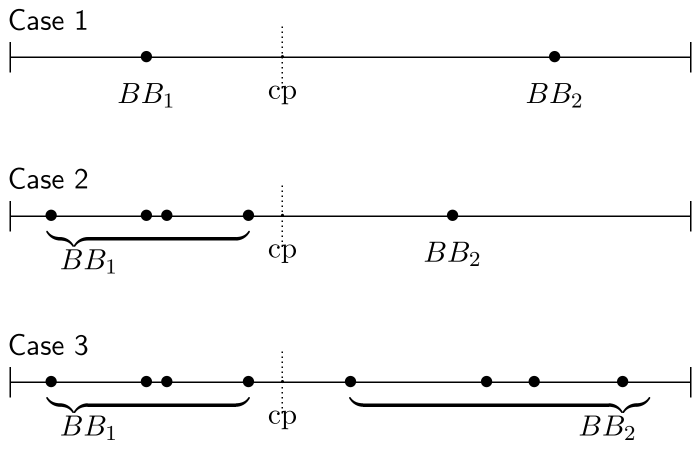


Figure 2.1: Mixing cases of single point crossover for different configuration of building blocks: $tight - random$, and $random - random$

is one-point crossover it is sufficient to define the linkage of a building block as the defining length of the schema.

In order to observe the effect of linkage on the mixing of building blocks it is assumed that a chromosome consists of infinite number of genes, the length of a chromosome is 1 and the genes have real valued positions in the interval $[0, 1]$ on a chromosome. Considering the 3 cases (Figure 2.1) for a crossover of two chromosomes, containing different building blocks of order k:

1. Both building blocks have a *tight* linkage (the gene points are close to each other so that they are considered to be a single point),

2. one of them has a tight linkage but the other has a random linkage,

3. both have random linkages.

In the first case, it is assumed that both building blocks are represented as points on the chromosome since they have a defining length 0. In this case, crossover of these two chromosomes will produce an offspring containing both building blocks if the crossover point is lying somewhere in between them. If crossover point divides the chromosome into two parts α and $(1 - \alpha)$, then the probability of these two building blocks to be mixed is $P_{tight-tight}(k) = 2\alpha(1 - \alpha)$. In order to find the expected value of the mixing, the integral of this probability in $[0,1]$ interval is taken:

$$\begin{aligned} EV_{tight-tight}(k) &= \int_0^1 2\alpha(1-\alpha) \\ &= 2 \Big|_0^1 (\frac{\alpha^2}{2} - \frac{\alpha^3}{3}) \\ &= \frac{1}{3} \end{aligned}$$

For the second case, it is assumed that the first building block is a point and the other consists of randomly distributed k points. If crossover point similarly divides the chromosome into α and $(1-\alpha)$ sized regions the probability of the mixing is the probability of the first building block falling in one region while k points of the second falling into the other region. This probability can be expressed as $P_{tight-random}(k) = \alpha(1-\alpha)^k + (1-\alpha)\alpha^k$. Taking the integral and using the symmetry in the interval $[0,1]$, we calculate the expected value as:

$$\begin{aligned} EV_{tight-random}(k) &= 2\int_0^1 \alpha^k(1-\alpha) \\ &= 2 \Big|_0^1 (\frac{\alpha^{k+1}}{k+1} - \frac{\alpha^{k+2}}{k+2}) \\ &= \frac{2}{(k+1)(k+2)} \end{aligned}$$

Third case is similarly calculated as the probability of having k points of the first building block at one side of the crossover point and k points of the second is in the other side. This probability can be written as $P_{random,random}(k) = 2\alpha^k(1-\alpha)^k$. The integration of this function results in an hypergeometric function factor. Evaluating it for the definite integral points 1 and 0 results in:

$$\begin{aligned} EV_{random-random}(k) &= 2\int_0^1 \alpha^k(1-\alpha)^k \\ &= 2 \Big|_0^1 \frac{\alpha^{k+1} Hypergeometric2F1[1+k, -k, 2+k, \alpha]}{k+1} \\ &= \frac{2\Gamma(k+1)^2}{\Gamma(2k+2)} \end{aligned}$$

Knowing that the k values are positive integers, it can be written as:

$$EV_{random-random}(k) = \frac{2(k!)^2}{(2k+1)!}$$

These expected value functions are also verified with a computer simulation. Plots of the functions are shown in Figure 2.2. As the order of building blocks increases

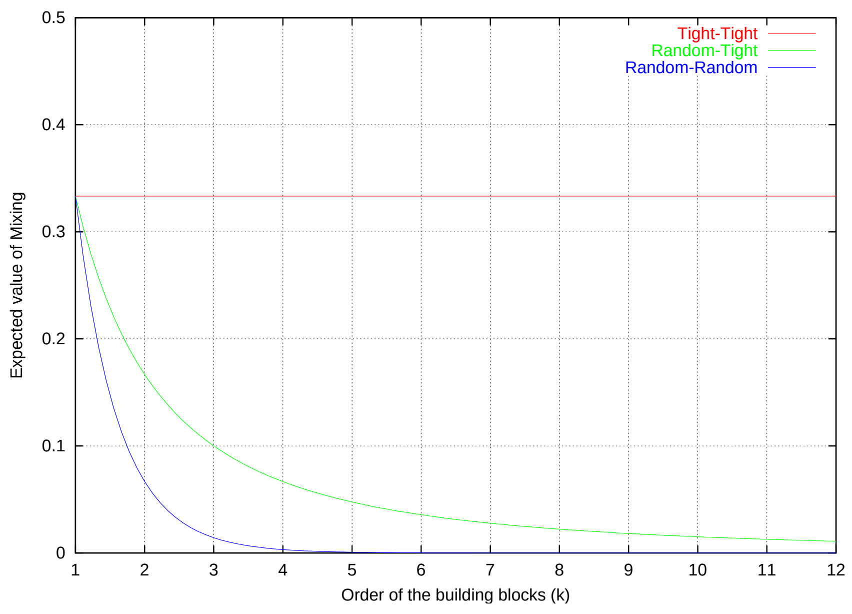


Figure 2.2: Expected value plots for mixing cases $tight - tight$, $tight - random$, and $random - random$ configurations of building blocks.

the linkage gets more critical in mixing. When both building blocks are randomly distributed along the chromosome, full proper mixing gets very rare for orders greater than 3. Even keeping one of the building blocks tight improves the expected value of proper mixing significantly.

2.3 Linkage Learning Problem

2.3.1 Early Efforts

Although the problem of changing linkage adaptively to favor better mixing of building blocks has been studied in a better detail for last 15 years, the first clue of having an adaptive ordering of genes was introduced long time ago by Bagley in his dissertation [7]. Bagley introduced the *inversion operator* on a special encoding. In this encoding each gene consists of a tuple of gene numbers and values. Instead of a fixed ordering, an ordering in which a gene can be in any position of the chromosome is employed. For example, the chromosome `(00101)` can be represented by `((3,1), (5,1), (2,0), (1,0), (4,0))` or `((4,0), (2,0), (5,1), (1,0), (3,1))` in this encoding. Inversion operator works like a positional mutation and randomly reverses a portion of the chromosome. For example when the individual `((3,1),⟨(5,1), (2,0), (1,0)⟩, (4,0))` is inverted on the ⟨...⟩ enclosed partition resulting string will be `((3,1), (1,0), (2,0), (5,1), (4,0))`.

By this operator it is possible to produce any orderings in some number of steps. However when two such chromosomes are crossovered in a standard way, resulting offsprings may contain ambiguous and missing gene values. Bagley suggested to crossover only the chromosomes with the same order, but this restriction decreased the rate of crossover to very low values. As a result, Bagley concluded that inversion in this case was not useful at all.

Goldberg and Lingle [33] later analyzed inversion operator in detail but concluded that it had not been useful in the solution of the problem. They introduced the *Partial Match Crossover* (PMX) operator to exchange allele and ordering information together among the chromosomes.

2.3.2 Symbiotic Evolution

Paredis [48] proposed a method called *symbiotic evolution* where two populations, the gene values, and the possible orderings, were evolved simultaneously. He used a type of genetic algorithm called GENITOR [65]. When crossover is to be made, one chromosome is selected randomly from the permutation pool and the selected chromosomes from the first pool are first mapped into their new positions according to this permutation. Than, they are crossovered and the offsprings generated are inverse mapped into the original representation. While the first pool evolves according to the fitness function to be optimized, fitnesses of the second pool depend on the performance of the crossovers that the permutations involve.

The fitnesses of the offsprings are compared with the fitnesses of the parents and the fitness of the permutation chromosome is updated accordingly. Fitness of the permutations are calculated according to the last 20 crossover they are involved in. According to these fitness values this second pool evolved together with the solution.

Paredis showed the empirical results in an artificial problem by comparing tight and loose fixed linkages, one-point and two-point symbiotic evolution and random linkage. He reported that symbiotic evolution was successful in both performance and grouping of building blocks together compared to loose and random linkages.

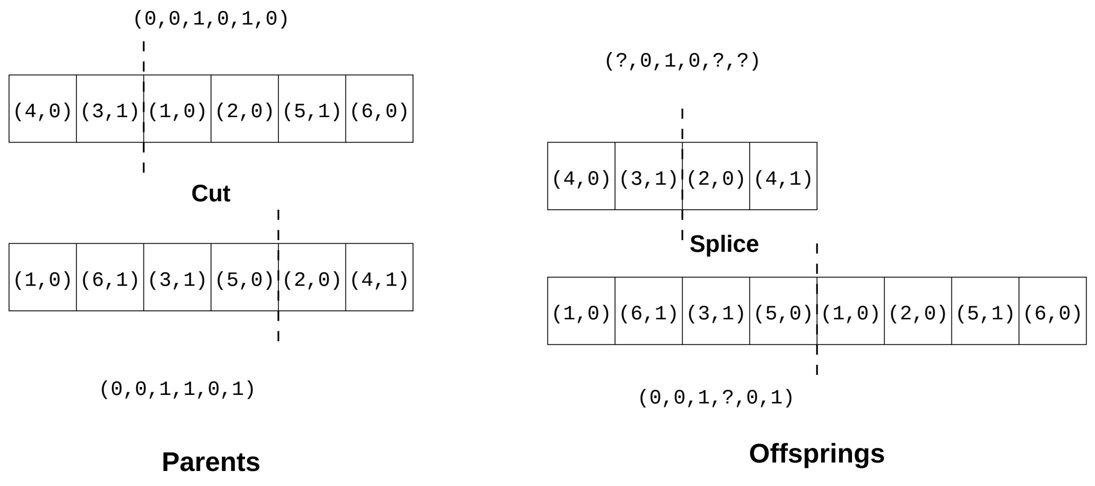


Figure 2.3: Cut and splice operators of the messy GA

2.3.3 Messy GA

Goldberg, Korb and Deb [28] introduced Messy GA where like the inversion operator, a tuple representation having both number and value of the gene is used in the chromosome encoding. However in messy GA, evolution is divided into two phases: in the *primordial phase* building blocks are constructed, and in the *juxtapositional phase* these building blocks are recombined.

Messy GA aims to solve problems up to a predefined order of deception: let's say k, Therefore all possible order k building blocks are constructed initially. In the primordial phase these building blocks are combined together into chromosomes of length l (problem size), randomly. These randomly created combinations of building blocks are selectively updated to construct an initial population containing high fit building blocks.

In the juxtapositional phase, chromosomes are combined to mix the building blocks. Special operators *cut* and *splice* are used in place of crossover to mix the building blocks. In Figure 2.3 these operators are illustrated. Two chromosomes are cut at arbitrary positions. Then, the second partition of the first is appended to the first partition of the second to form one of the offsprings and similarly, the second offspring is formed by appending the second partition of the second to the first partition of the first.

However, these operators are messy in the way that they produce *overspecified* and *underspecified* chromosomes. In the overspecification, value of one gene variable is presented by 2 or more genes in the chromosome ambiguously. The proposed solution by the authors is simply to take the first gene supplying the value. In the underspecification, one of the gene variables is not specified at all, in the chromosome. Solution of this problem is not simple. Messy GA fills missing gene values from a template.

Determination of this template is not a trivial task. This is called the *bootstrapping dilemma*: to solve a problem that was assumed to be deceptive of order k, a template which is a solution of the same problem with assumption of $k-1$ deception is required. Thus, messy GA starts from $k = 1$ and iterates up to the desired order of deception.

Goldberg, Deb and Korb empirically showed that he messy GA is capable of solving a variety of boundedly-deceptive problems of mixed size and scale.

2.3.4 Linkage Learning GA

Harik [38] introduced another method called Linkage learning GA in his dissertation. In LLGA a circular representation of chromosome structure where the chromosome end is connected to its beginning and 2-point crossover is used. Similar to messy GA tuples of values are used in the encodings of the gene. However interpretation of a gene is defined by a clock-wise scan of the chromosome. Also he defined an *exchange* operator where a partition of one of the chromosome (donor) is inserted into the other one (recipient) to an arbitrary point. Then, the overfull second chromosome is scanned clockwise starting from a random location and repaired by deleting the overspecification.

For example, having a donor chromosome `((4,1), (2,1), (3,0), (5,1))` onto the recipient `(((2,1), (4,1), (1,0), (5,0), (3,1))` by donating the `(2,1), (3,0)` portion after `(4,1)` in the recipient, an overfull chromosome `((2,1), (4,1), (2,1), (3,0), (1,0), (5,0), (3,1))` is produced. Then, deleting the excess genes, the resulting chromosome will be `((2,1), (4,1), (3,0), (1,0), (5,0))`.

Harik analyses the linkage change in the population by exchange and 2-point crossover operators and affects of the selection on the linkage. Another extension he introduced is the probabilistic expression of the circular chromosomes. Instead of a fixed single mapping from a chromosome to its meaning, a chromosome represents a probability distribution of individuals. In order to achieve this, the circular chromosome contains overspecifications ($2l$ values instead of l values) and is scanned starting from a random position. According to this starting point interpretation changes stochastically.

2.3.5 Distribution Estimation and Bayesian Optimization Algorithm

A general class of genetic algorithms are based on the estimation of distributions that are called *estimation of distribution algorithm* (EDA) [46]. In EDAs a distribution estimate is done from the current population and this estimate is used to construct a new population with this distribution. Then genetic operators are applied on this population and a new distribution is estimated. The varieties of this class of algorithms depend on how the estimation is done and how this estimate is used to produce a population.

In [50] an overview of EDAs are given and *Bayesian Optimization Algorithm* is introduced by Pelikan, Goldberg and Cantú-Paz. In this algorithm the joint distribution of the population is estimated by learning Bayesian networks. Bayesian networks [49] encode the relationships between the variables of the modeled data and the structure of a problem. They can be used both to describe a model from the data and to generate new instances from a model.

Each node in the network represents a variable and edges between the nodes represent the relation among the variables. Mathematically, a directed and acyclic Bayesian network with directed edges encode a joint probability distribution. In the proposed method a Bayesian network is learnt from a population of individuals and then this distribution is used to generate a new population. Genetic operators are applied on this population and a new network is learnt and so on.

The relation of this method with linkage learning is that the indegree of the Bayesian network can represent the gene relations of order k. Hence the population distribution is estimated with the information of building block groupings. The proposed algorithm is experimented on additively and hierarchically composed k deceptive problems [51, 52].

CHAPTER 3

PARALLEL GENETIC ALGORITHMS

The main goal of this chapter is to summarize the recent research on parallel genetic algorithms (PGAs). In the past few years, PGAs have been used to solve difficult problems. Hard problems need a bigger population and this translates directly into higher computational costs. The basic motivation behind many early studies of PGAs was to reduce the processing time needed to reach an acceptable solution. This was accomplished by implementing GAs on different parallel architectures. In addition, it was noted that in some cases the PGAs found better solutions than comparably sized serial GAs. However, most of the research has been empirical and, until recently, we lacked a theory that helped to resolve the fundamental questions that arise in the design of these algorithms.

Genetic algorithms are not guaranteed to find an optimal solution and their effectiveness is determined largely by the population size. As the population size increases, the GA has a better chance of finding the global solution, but the computation cost also increases as a function of the population size. With serial GAs, one has to choose between getting a good result with a high confidence and pay a high computational cost, or loosen the confidence requirement and get results fast but possibly poor. In contrast, PGAs can keep the quality of the results high and find them fast because, using parallel machines, larger populations can be processed in less time. This keeps the confidence factor high and the response time low, opening opportunities to apply genetic algorithms in time-constrained applications. Additionally, PGAs may evolve several different independent solutions that may be recombined at later stages to form better solutions.

3.1 A Classification of PGAs

As a first step in this investigation of PGAs, one must recognize that there are different ways to parallelize GAs [15]. This section presents a classification of the efforts made up to date. Evolution is a highly parallel process. Each individual is selected according to its fitness to survive and reproduce. GAs are an abstraction of the evolutionary process and are indeed very easy to parallelize. The first approach in parallelizing GAs is to do a global parallelization. In this class of PGAs, the evaluation of individuals and the application of genetic operators are explicitly parallelized. Every individual has a chance to mate with all the rest (i.e., there is random mating). Therefore, the semantics of the operators remain unchanged. This method is relatively easy to implement and a speedup proportional to the number of processors can be expected.

A more sophisticated idea is used in coarse grained PGAs (i.e. where the ratio of the time spent in computation and the time spent in communication is high). The population is divided into a few subpopulations keeping them relatively isolated from each other. This model of parallelization introduces a migration operator that is used to send some individuals from one subpopulation to another.

Two population genetic models for population structures are used in different implementations of coarse grained GAs: the island model and the stepping stone model. The population in the island model is partitioned into small subpopulations by geographic isolation and individuals can migrate to any other subpopulation. In the stepping stone model, the population is partitioned in the same way, but migration is restricted to neighboring subpopulations. Both models have been used in PGAs. Sometimes coarse grained PGAs are known as "distributed" GAs since they are usually implemented in distributed memory MIMD computers.

The third approach in parallelizing GAs uses fine grained parallelism. Fine grained PGAs partition the population into a large number of very small subpopulations. The ideal case is to have just one individual for every processing element available. This model calls for massively parallel computers.

In the last two approaches, selection and mating occur only within each subpopulation. Based on this the term deme is borrowed from biology to refer to a subpopulation. Since the size of the demes is usually smaller than the population used by a serial GA, one can expect that the PGA will converge faster. However, when it is considered that there is communication between the demes it is no longer clear whether the PGA will converge faster or not. It is important to notice that while the global model did not affect the behavior of the algorithm, the last two approaches are introducing fundamental changes in the way the GA works. In subsequent sections these changes are identified.

The final method to parallelize GAs uses some combination of the first three methods, called hybrid PGAs. There are not many papers published describing implementations with hybrid techniques, but some of them have achieved very interesting results.

3.1.1 Global Parallelization

In this model there is a single population and the evaluation of the individuals and sometimes the application of the genetic operators is done in parallel. The evaluation can be parallelized assigning a subset of individuals to each of the processors available. There is no communication between the processors during the evaluation because the fitness of each individual is independent from all the others. Communication only occurs at the start and at the end of the evaluation phase.

On a shared memory multiprocessor, the individuals could be stored in shared memory. Each processor can read the individuals assigned to it and write the evaluation results back without any conflicts. One should note that there is some synchronization needed between generations. In a multiuser environment, it may be necessary to balance the computational load among the processors using a dynamic scheduling algorithm. In any case, a speedup proportional, but probably sublinear, to the number of processors can be expected.

On a distributed memory computer, the population can be stored in one processor to simplify the application of the genetic operators. This "master" processor would be responsible for sending the individuals to the other processors (i.e. the "slaves") for evaluation, collecting the results, and applying the genetic operators to produce the next generation. There would be a bottleneck in the algorithm since the slave processors sit idle while the master is doing its work.

An example of global parallelization is the work of Fogarty and Huang [24] . They used a transputer network to evolve a set of rules for a pole balancing application. The simulation of the cart and the pole takes a considerable time. They connected the transputers in different topologies, but they conclude that in cases where the computation time dominates over the communication time, the configuration of the network is not important. They noted that the speedups were sublinear because the communication overhead increased very fast as they added more transputers to the network.

Another more recent implementation of a Global GA is the work by Hauser and Männer [39]. They used three different parallel computer architectures, but only got reasonable speedups on a NERV multiprocessor (speedup of 5 using 6 processors), that has a very low overhead. On the other systems they used (a SparcServer and a KSR1), they did not get good speedups because the thread libraries they used did not gave them complete control over the scheduling of threads to processors.

A further step in global parallelization is to apply the genetic operators in parallel. Each of the operators of the simple GA is examined. First, consider the selection operator. There are several variants of selection and some require a global statistic (like the population average) which could introduce a serial bottleneck in the algorithm. Fortunately, if some form of tournament selection is used, only the fitness of a subset of the individuals (usually only two) is needed. Now consider crossover, it can be applied simultaneously over $n = 2$ pairs of individuals. Mutation can be applied to each individual (and even to each bit) independently of the others.

On the other hand, it is not clear whether parallelizing the application of the operators would result on a performance improvement or not. The genetic operators are very simple and the time spent in communication could easily be longer than the time doing any computation. This is specially true in distributed memory machines, where the overhead for each message might be considerable.

Abramson implemented a GA on a shared memory computer (an Encore Multimax with 16 processors) to search for efficient timetables for schools [1] . He parallelized the creation and evaluation of individuals. Abramson reported sublinear speedups, and blamed a few sections of serial code on the critical path of the program for the results. He later used a distributed memory machine (a Fujitsu AP1000 with 128 processors) and changed his application to train timetables. In both systems, he reports almost linear speedups up to 16 processors. In the distributed memory machine the speedup degrades fast with more processors probably for the reasons outlined above.

3.1.2 Coarse Grained PGAs

The important characteristics of this class of algorithms are the use of few relatively large demes and the introduction of a migration operator. Coarse grained PGAs are the most popular model and many papers have been written describing many aspects and details of their implementation. A thorough review of this area is given below.

3.1.2.1 Traditional GAs

One of the pioneering efforts is Grosso's dissertation [35]. He simulated diploid individuals and the population was divided in five demes. The demes exchanged individuals using a fixed migration rate and several rates were tried in the experiments. Grosso found that the rate of improvement was faster in the smaller demes than in a single large panmictic population. However, when the demes were isolated, this rapid rise in fitness stopped at a lower level than with the larger population. At intermediate migration rates, the divided population displayed a behavior similar to the panmictic population. With a lower migration rate, the demes had the opportunity to behave independently and explore different regions of the search space. But, if the migration rate is to low, the migrants might not have a significant effect on the receiving deme. These results point out that there is a critical migration rate below which the performance of the algorithm is obstructed by the isolation of the demes and above which the partitioned population behaves as a panmictic one.

In the first International Conference of Genetic Algorithms (ICGA), there were no papers about PGAs at all. This situation changed in the second ICGA in 1987, where six papers were published. From then on there has been a steady flow of papers published in conferences and journals of GAs and parallel computation.

Tanese, for example, proposed a PGA that used a 4-D hypercube topology to communicate individuals from one deme to another [62]. In Tanese's algorithm migrations occurred at uniform periods of time between neighbor processors along one dimension of the hypercube. The migrants were chosen probabilistically from the best individuals in the subpopulation and they replaced the worst individuals in the receiving deme. It is reported that the PGA found results as good as a serial GA, with the advantage of near-linear speedups. Tanese continued her work making a very exhaustive experimental study on the frequency of migrations and the number of individuals exchanged [63]. She found that migrating too many individuals too frequently or too few individuals very infrequently degraded the performance of the algorithm. However, good quality results were found faster than with a serial GA even in cases with no migration at all. More details about the experiments and the conclusions she derived can be found in her dissertation [64]. A recent paper by Belding attempts to extend Tanese's work using the Royal Road functions [11] . Migrants were sent to a random destination, rather than using a hypercube topology, as in Tanese's original study, but the author claims that experiments with a hypercube yielded similar results. In most cases, the global optimum was found more often when migration was used than in the completely isolated cases. Surprisingly, convergence was faster when a high migration rate was used (r = 0:5) than with a low rate.

Also in 1987, Cohoon, Hedge, Martin, and Richards proposed an implementation of a PGA based on the theory of "punctuated equilibria" [18]. One aspect of the punctuated equilibria theory is that new species are likely to form quickly in relatively small isolated populations after some change in the environment occurs. The genetic composition of a population is an important part of an organism environment and thus, immigration can be a trigger for evolutionary changes. Cohoon et al. noticed that the number of migrants affected the level of disruption in the demes and that new solutions were found shortly after migration occurred. Keeping the total cost constant, they found that the PGA with migration outperformed both a PGA without migration and a serial GA. This work was later extended by the same group using a VLSI application (a graph partitioning problem) on a 4-D hypercube topology [20, 19].

The last paper from Pettey, Leuze and Grefenstette in 1987 uses parallelism to deal with the problem of getting fast results while using increasingly larger population sizes that are needed in some applications [54]. This is a direct implementation of Grefenstette's fourth prototype outlined in the previous section. In this algorithm, a copy of the best individual found in each deme is sent to all its neighbors after every generation to ensure good mixing. In this paper, Pettey, Leuze, and Grefenstette recognize that this approach can be regarded as equivalent to a sequential GA with a single large panmictic population, with the only difference being that selection occurs at the deme level.

From the papers examined above, one can recognize some very important issues emerging. First, migration is controlled by several parameters; the topology that defines the connections between the subpopulations, a migration rate that controls how many individuals migrate, and a migration interval that affects how often migrations occur. Second, the values for these parameters are chosen using intuition rather than analysis. Although some tried to make some analysis on PGAs, intuition continued to be the norm in the following years.

Probably the first attempt to provide some theoretical foundations to the performance of a PGA is a 1989 paper by Pettey and Leuze [53]. In this article they describe a derivation of the schema theorem for PGAs. They show that the expected number of trials allocated to schemata can be bounded by an exponential function when randomly selected individuals are broadcast every generation.

It can be expected that relatively isolated demes will find different partial solutions to a given problem. Starkweather, Whitley, and Mathias made an important observation regarding this: if partial solutions can be combined to form a better solution, then a PGA would probably outperform a serial GA [61]. On the other hand, if the recombination of partial solutions results in a less fit individual, then the serial GA might have an advantage.

Note that in all the papers cited so far, migration occurs at predetermined constant intervals. A different approach was introduced in a 1990 paper presented at the first Parallel Problem Solving from Nature workshop (PPSN) where migration occurs after the subpopulations converge (the author uses the term "degenerate") [17]. The same concept was later used (with some alterations) by Munetomo, Takai, and Sato [18]. These researchers raised an important (and yet unanswered) question: when is the right time to migrate? If migration occurs too early the number of correct building blocks in the migrants may be too low to influence the search on the right direction and expensive communication resources would be wasted.

As seen above, the bulk of the work on PGAs has been empirical, but few researchers have identified and studied a particular issue. A traditionally neglected aspect of PGAs has been the topology of the interconnection between demes. The topology is an important factor in the performance of the PGA, because it determines how fast (or how slow) a good solution disseminates to other demes. If the topology has a dense connectivity (or a short diameter, or both), good solutions will spread fast to all the demes and may quickly take over the population. On the other hand, if the topology is sparsely connected (or has a long diameter), solutions will spread slower and the demes will be more isolated from each other, permitting the appearance of different solutions. These solutions may come together at a later time and recombine to form potentially better individuals.

Bianchini and Brown attempted to tackle the topology problem considering the problems and restrictions that arise in a transputer implementation [12] . These two researchers tried different topologies and presented experimental results that support the idea of using isolated and semi-clustered demes. These topologies have low connectivity (making them realizable with transputers which can connect to only four other transputers) and good results were found using them. A more recent empirical study of different topologies showed that topologies with higher arity find the global solution faster than the more sparsely connected ones [14]. This study considered completely connected topologies, 4-D hypercubes, a toroidal mesh and unidirectional and bidirectional rings.

An unconventional approach to migration was recently developed by Marin, Trelles-Salazar, and Sandoval [42]. They propose a centralized model where the processes that are executing the GAs periodically send their best partial results to a master process. Then, the master process chooses the fittest individuals among those that it receives and broadcast them to all the nodes. Experimental results show that near linear speedups can be obtained with a small number of nodes (around six), and the authors claim that this approach can scale up to larger workstation networks.

3.1.2.2 Non-traditional GAs

Some non-traditional GAs have also been parallelized and some improvement has been reported over their serial versions. Maresky used Davidor's ECO model as a basis of his distributed algorithm [41]. Each node in this algorithm is a ECO GA. Maresky explored different migration schemes with varying complexity. The simplest scheme is to do no migration at all and just collect the results from each node at the end of a run. The more complex algorithms involve the migration of complete ECO demes. Several replacement policies were explored with only marginal benefits.

GENITOR is a “steady-state” GA where only a few individuals change in every generation. In the distributed version of GENITOR , individuals migrate at fixed intervals to neighboring nodes [61]. Immigrants replace the worst individuals in the target deme.

There have also been efforts to parallelize messy GAs. Messy GAs have two phases. In the primordial phase the initial population is created using a partial enumeration and it is reduced using tournament selection. Next, in the juxtapositional phase partial solutions found in the primordial phase are mixed together. The primordial phase dominates execution time and several data distribution strategies were tried by Merkle and Lamont [43]. The results indicate that the quality of the solutions was not statistically different in most of the cases varying the distribution strategy, but that gains in execution time could be possible.

More recently, Koza and Andre used a network of transputers to parallelize a Genetic Programming application [4]. In their report they make an analysis of the requirements of their algorithm and choose transputers as the most cost-effective solution to implement it. The report also includes a very detailed description of the implementation of their algorithm. They used an island model and experimented with different migration rates using 64 demes. They reported super linear speedups over the panmictic version using moderate migration rates (around 8 percent).

3.1.3 Fine Grained PGAs

In this section, some publications on the fine grained approach to parallelize GAs are examined. In this model the population is divided into many small demes. The demes overlap providing a way to disemminate good solutions across the entire population. Again, selection and mating occur only within a deme.

In the second ICGA, Robertson published a paper describing the parallelization of a GA in a classifier system using a Connection Machine [55]. He parallelized the selection of parents and of classifiers to replace and also the mating and crossover operations. The execution time of this algorithm is independent of the number of classifiers (up to 16K).

In 1989 the ASPARAGOS system was introduced in the paper by Gorges-Schleuter [34]. It uses a population structure that looks like a ladder with the upper and lower ends tied together (forming a ring). ASPARAGOS was used to solve some difficult combinatorial optimization problems with great success. Later, a linear variation on the population structure was tried. There is no question on the effectiveness of this system, but because it uses non- conventional genetic operators (like the PMX crossover) and a hill-climbing algorithm to further optimize the individuals in the population, it is difficult to use the results from this system to help us understand the dynamics of a fine grained PGA.

Another massively PGA with a more traditional population structure was studied [60]. In this algorithm the population is distributed on a 2-D mesh topology. Selection and mating are only possible with neighboring individuals, therefore the demes are defined by the spatial distribution of the individuals. The authors noted that the performance of the algorithm degraded as the size of the deme increased, On the extreme, if the size of the neighborhood was big enough, this PGA was reduced to a conventional panmictic population. Their analysis showed that, for a deme size s and a string length l, the time complexity of the algorithm was $\mathcal{O}(s + l)$ or $\mathcal{O}(s(\log s) + l)$ time steps depending on the selection scheme used.

Davidor also used 2-D grid to place the individuals, but on his algorithm offspring are placed in their parents' deme [21]. A conflict occurred if the grid element is occupied by another individual and it is solved probabilistically using the fitness of both individuals as bias. Davidor named his algorithm the "ECO model".

It is common to place the individuals of a fine grained PGA in a 2-D grid because in many massively parallel computers the processing elements are connected using this topology. However, most of these computers have a global router that can send messages to any processor in the network (at a higher cost) and other topologies can be simulated on top of the grid. There is a study that compares the effect of using different interconnection networks on fine grained GAs [56]. In this paper a graph partitioning problem is used as a benchmark to test different topologies simulated with a MasPar MP-1 computer. The topologies tested were a ring, a torus, a cube, and a 10-D binary hypercube. The results for this problem showed that a torus had a better performance than the other higher or lower dimensionality topologies.

Anderson and Ferris tried different topologies and different replacement algorithms [3]. They experimented with two rings, a hypercube, two meshes and an "island" topology where only one individual of each deme overlapped with other demes. They concluded that for the particular problem they were trying to solve (assembly line balancing) the ring and the "island" topologies were the best. From the results of these papers one can conclude that the topology affects the performance of the algorithm. It seems that a topology with a medium diameter gives good results.

Shapiro and Navetta successfully applied a fine grained algorithm to predict the secondary structure of RNA [59]. They used the native X-Net topology of the MasPar-2 computer to define the neighborhood of each individual. The processing elements on the MasPar-2 are placed on a 2-D grid and are directly connected to their eight closest neighbors. Different logical topologies were tested with the GA: all eight nearest neighbors, four nearest neighbors, and all neighbors within a specified distance r.

3.1.4 Hybrid Algorithms

A few researchers have tried to combine two of the methods to parallelize GAs. Some of these new hybrid algorithms add a new degree of complexity to the already complicated scene of PGAs, but other algorithms manage to keep the same complexity as one of their components.

Gruau invented a "mixed" PGA. In his algorithm the population of each deme is distributed in a 2-D grid [36]. The demes themselves are connected as a 2-D torus and migration between neighboring demes occurs at regular intervals. Good results were reported for a novel neural network design and training application.

Another way to hybridize a PGA is to use global parallelization on each of the demes of a coarse grained GA. Migration occurs between demes as in the coarse grained algorithm, but the evaluation of the individuals is handled in parallel. This approach does not introduce new analytic problems and can be useful when working with complex applications with objective functions that need a considerable amount of computation time.

This approach has not been used yet in any application, but it does not introduce more complexity into its analysis. Recall that the global parallelization model has the same properties as a serial GA, thus the analysis for this hybrid model will follow the exact same path as the analysis for coarse grained PGAs. Good speedups can be expected from this method when applied to problems that need a long time to be evaluated and extremely large populations.

3.2 Underlying Problems

It has been observed in previous sections that PGAs are effective in solving a number of difficult problems and that they can be mapped to different kinds of parallel machines. PGAs promise a lot both on terms of speedups and in improving the search quality of the serial GAs. With all the good results published it is easy to miss that there are several issues that are not understood at all. This section will go over the most important problems and the advances in their understanding. Coarse grained GAs are focused on, since they are the more realistic simulation of natural processes, they are better understood than fine grained GAs, and they are the most popular approach.

A first step in a formal investigation of PGAs is to decompose the problem into manageable parts that can be studied independently and that one can reunite later. A simple first level decomposition divides the study of PGAs in two parts: population sizing and migration.

The size of the population is probably the parameter that affects most the performance of the GA. There is a population sizing theory for serial GAs that is based on statistical decision making theory and gives conservative lower bounds on the population sizes needed to solve problems with a given confidence factor [26]. In a PGA, the population sizing question is very closely related to the deming issue. That is, how many demes and of what size does the PGA need to solve a problem with a given confidence? This questions have been addressed recently and solved for the extreme cases where perfect mixing between the demes is assumed and for isolated demes [25].

After the number and the size of demes to be used are determined, it has to be established how they are going to communicate. Migration should be the next step in the investigation of PGAs. It is a very complex operator and the migration of individuals across demes is affected by several parameters. To be able to study migration, one have to study each of its facets independently.

A possible decomposition of the migration parameter could be:

A migration timescale

Intuitively, migration should occur after the expected number of building blocks in each individual is relatively high. Migrating before that is probably a waste of communication resources.

Migration rate

The number of individuals that should be migrated is very closely related with the migration timescale. If individuals are 'rich' in building blocks, maybe one can get good results migrating just a few.

Topology

What is the best way to connect the demes? If a topology with a long diameter is used, good solutions will take a longer time to propagate to all the demes. On the other hand, if a topology that regards the mixing of partial solutions is chosen, different 'species' might appear at they may not mix to produce a better solution.

CHAPTER 4

GENE REORDERING GENETIC ALGORITHM

Although there are many linkage learning approaches reported to be successful for artificial and real world problems, most of them have been defined and examined on binary alphabets. There are only a few approaches that are suitable on linkage learning in non-binary genotype representations.

The majority of the real world problems try to find solutions of real or integer vector encodings. When using binary alphabets each integer or real phenotype value is converted into a sequence of bits in the genotype encoding. For example, a fixed or floating point real value, encoded in binary, an integer value encoded in binary or grey coding. However, when a linkage learning strategy is tried to be applied in these problems without special treatment bits are free to float in the chromosome regardless of the integer or real value that they belong to. As a result, the linkage information known a priory, namely the information that a certain group of bits contribute to the same number, will possibly be lost.

Therefore a linkage learning strategy should try to preserve these gene boundaries while discovering the inter-gene relations and keeping the related genes together. Thus the primary focus in problems with non-binary phenotype values should be to learn the linkage among these allele groups instead of the linkage of the bits.

The approach introduced in this thesis is to use non-binary alphabets equivalent to the phenotype representations in the genotypes while trying to learn the linkage. Although lower cardinality alphabets are suggested by the schema theorem and analysis, there are some claims that non-binary alphabets have the same power of searching the hyperspace with the binary alphabets. In [44] the advantages of floating point representations are described and it is shown that floating point representations work faster in many problems especially when special genetic operators are used. This claim is more convincing when the underlying architecture is considered. All contemporary computers provide their most primitive operations on integer and floating point values.

Some of the the existing linkage learning systems are suitable and/or adapted for non-binary representations (like symbiotic evolution [48] and messy GA [23], see Chapter 2), but most of the others are designed to work only with binary alphabets.

4.1 Global Reordering and Affinity

The concept of affinity and gene reordering genetic algorithm have been introduced in our earlier work [58] and results are shown in a non-intersecting rectangle placement problem in a minimized bounding box. This subsection explains this proposed method.

In gene reordering genetic algorithm, a single global permutation, which applies to all members of the population, is kept. This permutation is a mapping between the actual order of the gene in the phenotype and the effective position of it which is subject to genetic operators. This mapping is especially significant for the crossover operator. Crossover operation is based on this effective position instead of the actual order in the representation. During the evolution of the genetic algorithm, this global permutation is readjusted according to a statistical analysis on neighboring genes, calculated for the bests of the pool.

The proposed method is defined as follows:

- Consider that the solution to a subject problem requires the determination of a set of parameters $\mathcal{X}$. The first step of GA requires the determination of a one-to-one mapping $\mathcal{E}$ from the set of parameters to the set of strings $\mathbf{\Gamma}$, this is called the *encoding* of the parameters.

  $$\mathcal{E} : \mathcal{X} \mapsto \mathbf{\Gamma}$$

  We name the members Γ_i of the set $\mathbf{\Gamma}$ as *genes*. The Γ_i strings are concatenated into a one dimensional array $\mathbf{\Gamma}$ so that Γ_i is followed by Γ_{i+1}. An instance of $\mathbf{\Gamma}$ is called a *chromosome*.

- We define a *permutation of genes* as an ordering relation of the genes in a chromosome. It maps the i^{th} gene in the problem into it is position interpreted by the genetic operators. If $\mathcal{P}_p$ represents such a permutation operator on a chromosome, it is defined by means of its gene elements as:

$$[\mathcal{P}_p \Gamma]_i \overset{\triangle}{=} \Gamma_{p(i)}$$

  Here p is a permutation function:

$$p : \{1, 2, \ldots, n\} \mapsto \{1, 2, \ldots, n\} \quad \text{where}$$

$$n = |\mathbf{\Gamma}| \quad \text{and} \quad p^{-1} \quad \text{exists.}$$

- As a part of the proposed method we define for each gene position i of the chromosome a function f_i that admits a gene value of that position as argument and is defined as:

$$F_i : Allele(\Gamma_i) \mapsto \mathcal{R} \quad \text{where} \quad \Gamma_i \in \mathbf{\Gamma}$$

  Also note that, in the general case, Γ_i can be a tuple representation of various features. Therefore F_i may be defined over domain that has tuples as elements.

  When the building blocks of a GA is reverse mapped to the actual problem domain, it is usually observed that the formation of blocks correspond to some paternal, structural, mathematical invariance or covariances.

  The functions F_i will serve to extract the features which may lead to some invariance or covariances and express them as real numbers. Thus, we are defining a mechanism where the GA user has a chance to hook-in his hint for defining the features which may lead to building blocks.

- For denotational simplicity, we will define

$$f_i|_\xi \overset{\triangle}{=} F_i(Value(\Gamma_i|_\xi))$$

  with the semantics: The f_i value of the chromosome ξ is defined to be the result of the F_i function applied to the value of the gene Γ_i in the chromosome ξ of the pool.

- During the evolution we calculate a *neighbor-affinity-function* $\mathcal{A}_i$ that is defined for each neighbor gene pair position in the current permutation over the fitness-wise top M elements of the pool. We propose it to be most generally of the form:

  $$\mathcal{A}_i^p : \{\langle f_{p^{-1}(i)}\big|_\xi , f_{p^{-1}(i+1)}\big|_\xi\rangle\} \mapsto [0,1] \quad \text{where}$$

  $$i = 1, 2, \ldots, n-1 \quad \text{and} \quad \xi \in [\text{Fitnesswise top } M \text{ elements of the pool}]$$

  Of course if the chromosome composition is a homogeneous vector of genes (due to the nature of the problem) then all f_i's will reduce to a single f and hence $\mathcal{A}_i$ will reduce to a single function $\mathcal{A}$.

  $\mathcal{A}_i$ is a problem specific function and should be coded according to the characteristics of the problem encoding. Since the aim is to construct good building blocks, the result of $\mathcal{A}_i^p$ should be closer to 1 for gene values acting coherently and contributing for better solutions. Correlation and standard deviation analysis can be used as alternatives for $\mathcal{A}_i$.(See 4.1.4)

- The next step is to modify the permutation mapping by looking at the results of $\mathcal{A}_i^p$ evaluated using the instances of the current generation of the pool, namely the *neighbor-affinity values*. We will be calling these values A_i^p.

  In doing so every gene position in the permutation will be considered and based on the right and left neighbor-affinity values of each gene, a decision will be made. If this value is found to be less then a threshold value τ, these two genes with low affinity will be considered as unrelated to each other and the permutation will be altered to to separate them. Actually there exists 4 possible affinity cases for a gene:

  1. *Affinity value is greater then τ for both neighbors* so there is no problem with the current ordering. Gene position is kept in the permutation.
  2. *Affinity value with left gene is greater than τ but it is less than τ for the right gene.* This means left neighbor is okay but it should be separated from the right. Thus gene exchanges position with the left neighbor. In this way good affinity with the left (new right) neighbor is maintained.
  3. *Affinity value with left gene is less than τ but it is greater than τ for the right gene.* Similarly gene exchanges its position with the right neighbor.

4. *Both affinity values are less than* τ. Gene is moved to a random position in the permutation.

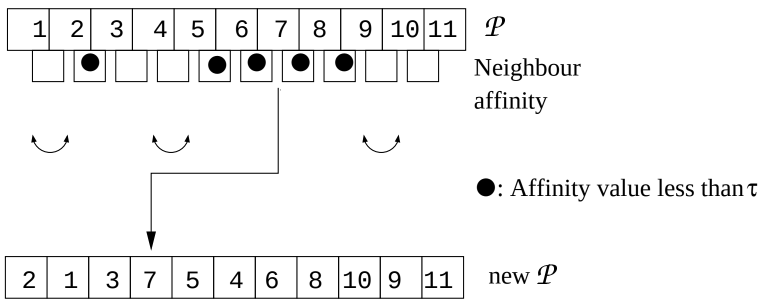


Figure 4.1: Modification of order according to affinity values

Modifications in the permutation mappings are done locally when it is possible. The algorithm to do this is given as follows:

```
for $i \leftarrow 2 \ldots n$ do
  {
    if $p^{-1}[i-1]$ is not modified in previous iteration
      if $A_{i-1} > \tau \wedge A_i < \tau$
        $p^{-1}[i-1] \leftrightarrow p^{-1}[i]$
      else if $A_{i-1} < \tau \wedge A_i > \tau$
            $p^{-1}[i] \leftrightarrow p^{-1}[i+1]$
      else if $A_{i-1} < \tau \wedge A_i < \tau$
            Move $p^{-1}[i]$ to a random location
  }
```

$p^{-1}[i]$ represents the inverse of the permutation mapping. Hence it should be interpreted as the actual gene number of the gene in the i^{th} position in the ordering. Thus neighbor relation holds between the genes $p^{-1}[i]$ and $p^{-1}[i+1]$.

4.1.1 Genetic Algorithm Engine

The genetic engine is similar to a simple genetic algorithm implementation. Only two modifications are made:

1. Crossover is based on the permutation p

2. This permutation p is modified according to the affinity values.

Genetic algorithm engine works as follows:

```
• Initialize the global permutation $p$ to $[1, 2, \ldots, n]$
• Generate a random population
Repeat:
• Select individuals to crossover
• Perform crossover based on $p$
• Mutate some of the genes
• Evaluate the new generation
• If reorder period is reached
    • Calculate $A_i$ values according to $M$ and the current $p$
    • Modify $p$ according to $A_i$ values
• Applying elitism with a keep ratio of $Keep$ replace old generation by
  new generation.
• If maximum number of generations is reached or the solution is sat-
  isfactory terminate. Otherwise goto repeat.
```

4.1.2 Implementation

To implement and test the the proposed system, the *rectangle placement problem with minimum bounding box* is chosen. In this problem, n rectangle's width and height information is given as input. The aim is to find a placement of all rectangles such that all rectangles are in a bounding box which is minimized in the area and no two rectangles intersect.

Problem encoding

- $I_i = \langle w_i, h_i \rangle$ where $i = 1 \ldots n$ are the dimension data of the n rectangles to be placed. This is input.

- $\Gamma_i = \langle x_i, y_i, o_i \rangle$ where x_i and y_i are the offsets of (of the lower-left corner) from the origin and $o_i \in \{0, 1\}$ is the orientation (not rotated, 90° rotated) at the placement instance. Each gene represented by Γ_i defines a placement for I_i. Combining I_i and Γ_i gives a rectangle area with absolute addresses.

- The fitness function for a chromosome is defined as:
  $V = cZ + dO, +eB \quad j = 1, \ldots, poolsize$
  where c, d, e are constant weights and Z is the total area of the rectangles placed out of the placement area, O is the total overlapping area among all rectangle pairs, B is the area of the minimum bounding box covering all rectangles in the placement. The aim of the genetic algorithm is to find a chromosome that *minimizes* V.

- One point crossover is defined according to a permutation mapping $\mathcal{P}_p$ for crossover point is k: [1]

$$\Gamma_i^{AB} = \begin{cases} \Gamma_i^A & \text{if } p(i) < k \\ \Gamma_i^B & \text{if } p(i) \geq k \end{cases}$$

$$\Gamma_i^{BA} = \begin{cases} \Gamma_i^B & \text{if } p(i) < k \\ \Gamma_i^A & \text{if } p(i) \geq k \end{cases}$$

  where Γ_i^A and Γ_i^B are crossovered two produce two offsprings Γ_i^{AB} and Γ_i^{BA}

- The neighbor affinity function is defined as the total standard deviation of offset values describing the placement:

$$A_i^p = 1 - \frac{t\sigma_x(i) + u\sigma_y(i) + v\sigma_o(i)}{c}$$

  i = 1...n-1

  where $\sigma_\alpha(i)$ is defined as the standard deviation of the difference of the α feature

[1] Actually, the correct notation would be to use $Value(\Gamma_i^{\cdot\cdot})$. Because Γ_i refers to the data structure at the gene position i, and not the data content. However, from now on, to keep the denotation simple and understandable, we will drop the $Value()$ operation whenever this degenerecy does not lead to a semantic ambiguity.

(one of x, y, o) of the $\Gamma_{p(i)}$ and $\Gamma_{p(i+1)}$ values in the population. t, u, v are constant weights:

$$\sigma_\alpha(i) = \sqrt{\frac{\sum_{j=1}^{M} \delta_{i,j,\alpha}^2 - (\sum_{j=1}^{M} \delta_{i,j,\alpha})^2/M}{M}}$$

$$\delta_{i,j,\alpha} = \Gamma_{p^{-1}(i),j,\alpha} - \Gamma_{p^{-1}(i+1),j,\alpha}$$

Due to the proposed algorithm, when the standard deviation is high, A_i becomes less (closer to zero). Which is interpreted that they do not exhibit a cooperation towards a good solution. They contribute to the success mutually independently. Thus, they are unlikely components of the same building block. The algorithm, in this case, will separate these two neighbor genes and place them in relatively arbitrary positions in the chromosome. On the contrary, when the standard deviation is small, then A_i is close to 1, which, again due to the algorithm, keeps the relative distance of the placements of neighbor genes fixed. Thus, a building block is established by these neighbors.

- Since the gene values are triples of numerical values, there is no need to define an auxiliary function f_i which will map them to $[0, 1]$. Their numerical values are directly used in the formula.

- The following GA settings are used:

Poolsize	: 100
Keep ratio	: 5%, 10%, 20%
Mutation	: 5%
M	: 20
Reorder period	: 5 generations

4.1.3 Test results

The implementation of the proposed algorithm is compared to a normal GA implementation with the same parameters, random seed and initial population. Tests are performed on different *Keep* values indicating the number of best members of current generation to be kept in the next generation. Having a large *Keep* ratio, GA tries to conserve its existing solution candidates, and thus converges faster to a local minima. Smaller *Keep* ratios results in a more chaotic system with slow convergence which perhaps yields a better solution.

In the experiments, both normal and gene reordering versions are tested for three *Keep* ratios 5%, 10% and 20% and for each ratio 20 executions with different random seeds have been carried out. In the test results the gene reordering algorithm GA version will be abbreviated *gene moved* version.

In almost all cases gene moved version established a faster convergence. A typical evolution of the best fitness is given in (Figure 4.2). It is also observed that as the *Keep* parameters decreases, normal version gets more chaotic and convergence significantly gets slower. However, gene moved version keeps its convergence speed (Figure 4.3).

When the fitness values of the best individuals after 1000 iterations are compared it is observed that, though proposed algorithm converges significantly faster, it does not fall into a local minima worse than the original algorithm (Figure 4.4). Furthermore, observations show that the quality of the resulting solutions is usually better.

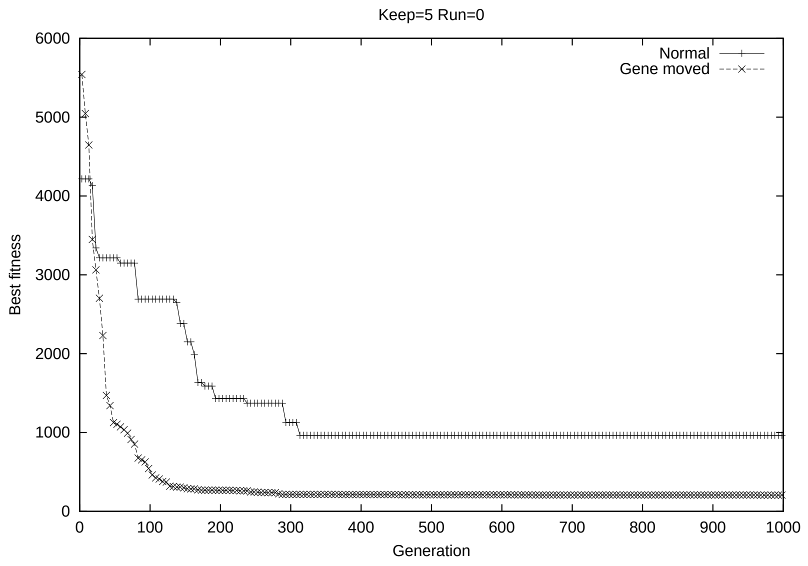


Figure 4.2: Evolution of the best individual for a typical run

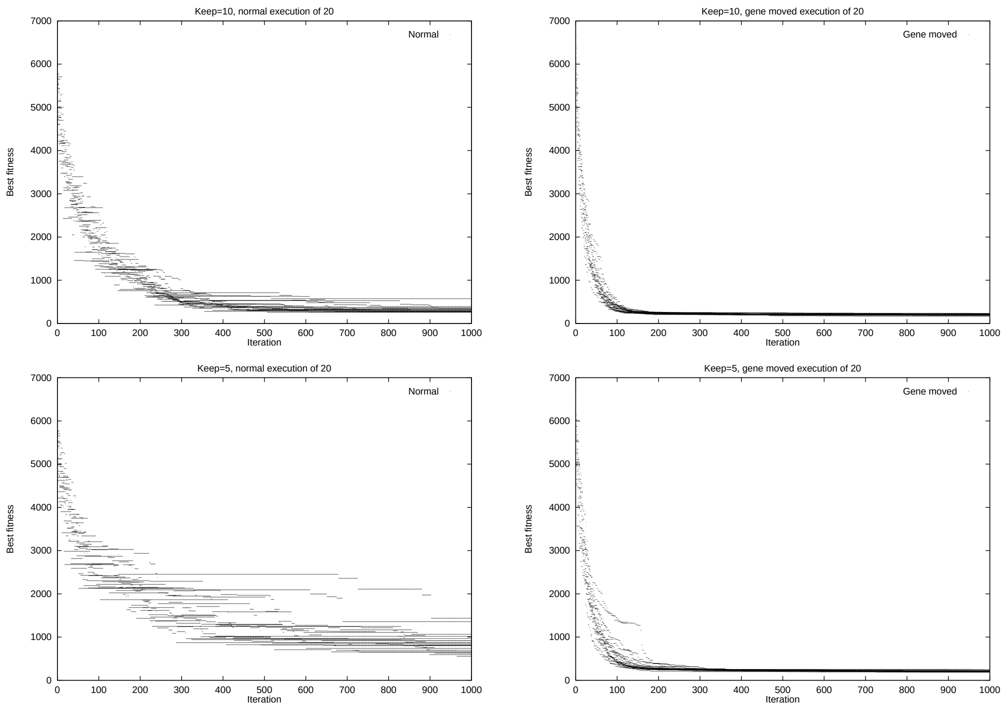


Figure 4.3: Distribution of best fitness values for $Keep$ value 10 and 5 for 20 executions

4.1.4 Affinity Functions

One of the most crucial points of the proposed method is the design of the affinity function $\mathcal{A}$. In addition to the fitness function and other parameters of the genetic algorithm (like crossover, mutation rates and selection mechanism) the GA implementer has an additional choice to be made. Instead of fixing affinity functions to a limited set of possibilities, it is better to leave the definition to the genetic algorithm designer.

The underlying idea of affinity functions is to capture a relation or dependency among two vectors of possibly converged gene values. When a topmost portion of the population is considered, the individuals possessing this relation have the higher fitness values and the genetic algorithm has favored the schemata that these two gene positions has defined.

In other words, $\mathcal{A}$ is a proposition, the GA implementer makes, claiming that s/he has a measure $\mathcal{A}$ for two genes whether they formed a building block or not. Thus, using $\mathcal{A}$ to discover building block forming genes, we are able to move them closer to each other and hence reduce the defining schema length (of the building block).

Therefore, the first consideration in the design of the affinity functions is to capture a statistical dependency among the best individuals. This can be based on a relative

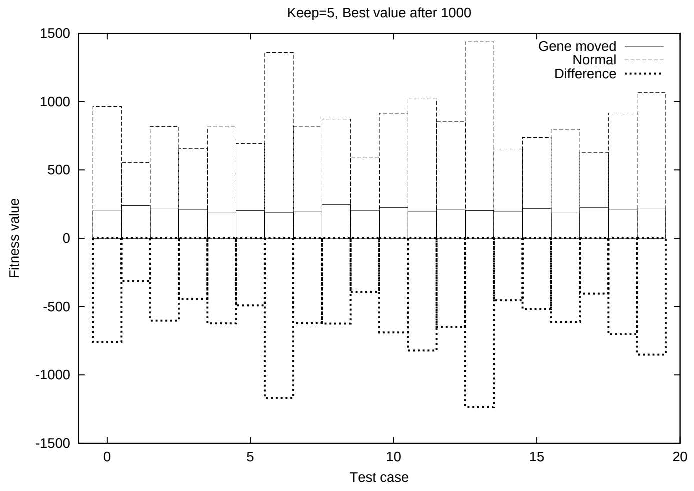


Figure 4.4: Best individuals fitness value after 1000 generation. Negative difference value means gene moved version has a better solution

difference distribution, a linear or higher order correlation. According to the problem one may customize these first candidates introduced in the following paragraphs or come up with a totally new function.

In the following part a list of possible affinity functions and the semantics they imply, are given.

Standard deviation of the differences

When this value is small that means the genes with relatively the same values have higher fitnesses in the population. Thus, for example, when the gene representation defines some spatial relation, having this value close to 0 implies that these gene values correspond to an almost fixed mutual positioning of spatial entities.

To normalize this value and make it conform with the affinity definition, the values should be in $[0, 1]$ interval and small deviation values should end up with better affinities. The deviation is divided by the average of the absolute differences and it is subtracted from 1.

$$A(\overline{x}, \overline{y}) = 1 - \frac{\sigma(x - y)}{\mu|x - y|}$$

In the rectangle placement problem. A similar affinity function is used where a fixed relative placement of rectangle positions in 2 dimensional space implies high affinity value.

Average of the absolute differences

When this value is small, gene values have the same values. In other words, they converge to the same value. When the equality relation is tried to be favored in the schema this function can be used. Similarly, normalization is applied to conform the affinity semantics.

$$A(\overline{x},\overline{y}) = 1 - \frac{\mu(\mid x - y \mid)}{max(\mid x - y \mid)}$$

As an example, this function can be used when the gene value represents an integer identifier, like city number, knapsack number etc. When the average difference is close to 0 that means gene values refer to the same entity and a possible building block.

Absolute value of the correlation

Correlation coefficient of two vectors is -1 or 1 if there is a full linear dependency among the sample data. In other words one of the values can be represented as a linear function of the other. This affinity function can be used to favor the linear dependency of the gene values, e.g. if one gene value is always the multiple of the other. The only normalization required is to take the absolute value. Thus negative slopes may also end up with the same affinities as of the positive ones.

$$A(\overline{x},\overline{y}) = \left| \frac{\sum xy - \sum x \sum y}{\sqrt{[n\sum x^2 - (\sum x)^2]\,[n\sum y^2 - (\sum y)^2]}} \right|$$

Such an affinity relation can be expected in problems that has an inherent linear relation among the variables.

Average Hamming distance

Although the proposed methods is motivated by the linkage learning in non-binary alphabets, binary alphabets are not ruled out completely. Hamming distance is the simplest relation among two binary strings. If it is close to 0 two vectors are almost equal and both genes converged to the same exact bit values. Thus an order 2 schema is found. $\oplus$ is used to represent bitwise exclusive or:

$$A(\overline{x},\overline{y}) = 1 - \mu(x \oplus y)$$

Bipolar average Hamming distance

It is not always the equality that is desired to be favored in the binary schema. One may decide to favor also if a vector is the complement of the other or not. To achieve this, the affinity should be large for both 0 and 1 Hamming distance averages. This can be interpreted as one gene value is either equal to the other or the complement of the other.

$$A(\overline{x}, \overline{y}) = 2|\mu(x \oplus y) - 0.5|$$

Independent standard deviation

In real life problems the input data and encoding of the problem does not always exhibit a dependency clue to be used in the design of the affinity function. However, the affinity can be based on the convergence state of the values and one may want to favor already converged gene values to the diverse gene values and try to get the converged ones closer in the representation during the algorithms execution. In this case the standard deviation of the gene values are calculated independently and the affinity will based on the normalized sum of the two deviation values.

$$A(\overline{x}, \overline{y}) = 1 - \frac{1}{2}\left(\frac{\sigma(x)}{max(x)} + \frac{\sigma(y)}{max(y)}\right)$$

4.2 Greedy Maximum Affinity Sequence

Although the gene reordering genetic algorithm has worked for the rectangle placement problem, there are two main questions to be asked: Is it a generalizable method to any genetic algorithm problem and is there really an interaction between the linkage and the affinity?

To clarify these questions, instead of considering the local affinity, a better idea would be to consider all affinity interactions and then to form a sequence of maximum affinities from which a permutation is derived. In this way the interaction of the linkage with the affinity can be observed more clearly. Thus, the next method that is proposed, regarding the affinity, is to find a permutation that maximizes the total affinity among all the neighbors. Namely, to find a p that maximizes the $\mathcal{S}$ in the following definition:

$$\mathcal{S} = \sum_{i=1}^{n-1} A_p(i, i+1)) \ , \quad \text{where} \ \ A_p(i,j) = A(p^{-1}(i), p^{-1}(j))$$

This problem however turns out to be equivalent to *Traveling Salesman Problem* where gene positions in the permutation correspond to nodes of the graph and the $1 - affinity\ values$ correspond to the cost of the edges and the aim is to minimize the Hamiltonian cycle length. Since this problem is possibly as complex as the original problem that the genetic algorithm tries to solve, it is not feasible to find the optimal affinity sequence.

However, a greedy approximation to this problem can be implemented and is sufficient to observe the linkage relation. Before starting the greedy algorithm, affinity values for all pairs of gene positions should be calculated. Then the algorithm starts with the largest affinity value of all pairs and the first two members of the permutation are set to be the elements of this pair. Then all affinities of the last element in the pair is searched and the unvisited one with the maximum affinity is taken as the third element. The same process is repeated from the third, forth and so on until all genes are included in the permutation. This approximation is identical to the traveling salesman solution where the closest two cities are started with and then the next travel is always done to the closest city.

The algorithm can be stated as the following:

```
calculate $A_p(i,j)$ for all $i = 1,\ldots,n$ and $j = 1,..,n$
find $k$ and $l$ such that $A_p(k,l)$ is the maximum affinity value
$p(1) \leftarrow k$
$p(2) \leftarrow l$
$s \leftarrow l$
visited[$k$] $\leftarrow$ $true$
visited[$l$] $\leftarrow$ $true$
for $c \leftarrow 3,\ldots,n$
        find $t$ such that $t$ is not visited and $A_p(s,t)$ is the maximum
        $p(c) \leftarrow t$
        visited[$t$] $\leftarrow$ $true$
        $s \leftarrow t$
```

This algorithm approximates the optimum permutation. The neighborhood affinity technique is a further approximation to this one. In terms of time complexity, compared to the neighborhood algorithm, the greedy maximum affinity algorithm is more expensive. Affinity values of all pairs has to be calculated, costing $\mathcal{O}(Mn^2)$.

Here M is the count of the fitnesswise best chromosomes that will be considered for affinity and n is the gene count in a chromosome. Then the maximum affinity in the array is found in $\mathcal{O}(n^2)$. For each node the next minimum node is calculated in $\mathcal{O}(n^2)$ so overall complexity is $\mathcal{O}(Mn^2)$.

In gene reordering, only neighbor affinities are calculated which has a cost of $\mathcal{O}(Mn)$. Then the permutation is changed in a single pass, costing $\mathcal{O}(n)$. Thus the overall complexity is $\mathcal{O}(Mn)$.

However, depending on the complexity of the fitness calculations, these time costs can be less significant. Their constant factors are quite low because they are only executed once in t generations. For example, the fitness function of the rectangle placement problem, described in the previous section, requires $\mathcal{O}(n^2)$ steps to determine if any pair of rectangles intersect. Thus the cost of the fitness evaluations is already $\mathcal{O}(Mn^2)$ which makes affinity calculations and permutation changes insignificant.

4.2.1 Generalized Assignment Problem

In order to test the proposed methods, another real-life problem, *generalized assignment problem* has been chosen [16, 17]. Generalized assignment problem can be used to formulate a group of real-life problems.

One possible scenario is to assign multiple jobs to multiple agents. Each jobs should be assigned to exactly one agent. There is a cost of assigning a job i to agent j is given as $cost_{ij}$. Each agent has a capacity of resources cap_j, and assigning a job i to agent j consumes res_{ij} from the capacity of the agent j. Aim is to minimize the cost. Formally:

$$J = \{1, 2, ..., m\} \quad \text{is the set of jobs} \tag{1}$$

$$A = \{1, 2, ..., n\} \quad \text{is the set of agents} \tag{2}$$

$$x_{ij} = 1 \;\; \text{if job } i \text{ is assigned to the agent } j\text{, otherwise it is } 0 \tag{3}$$

$$\sum_{j \in A} x_{ij} = 1 \;, \;\; \forall i \in J \;\; \text{each job should be assigned to exactly 1 agent} \tag{4}$$

$$\sum_{i \in J} res_{ij} x_{ij} \leq cap_j \;, \;\; \forall j \in A \;\; \text{resources limited by the capacity of the agent} \tag{5}$$

$$minimize \sum_{i \in A} \sum_{j \in J} cost_{ij} x_{ij}$$

Considering cost values as profits, problem can be converted into a maximization problem.

4.2.2 Problem Encoding and Test Parameters

In the implementation of the generalized assignment problem a chromosome of length m, number of jobs is used. Each gene value is an integer representing which agent that the job is assigned. So each gene value takes a value in the set A.

$$\Gamma_i = k \ , k \in \{1, 2, ..., n\} \ \ , \ \text{if } x_{ik} = 1$$

This encoding guarantees that no job is assigned to more than one agent, so constraint (4) is satisfied in the problem description. However the capacity constraint, constraint (5), should be checked to avoid assignments with excess capacity to agents. This is achieved by introducing a penalty in the fitness function for the excess resource use. The resulting fitness function for a maximization problem is:

$$f(\Gamma) = \sum_{i=1}^{i \leq m} cost_{i\,\Gamma_i} - c \sum_{j=1}^{j \leq n} penalty(j) \ \ , \quad \text{where}$$

$$penalty(j) = \begin{cases} 0 & \text{if } \sum\limits_{i|\Gamma_i = j} res_{ij} \leq cap_j \\ (\sum\limits_{i|\Gamma_i=j} res_{ij}) - cap_j & \text{otherwise} \end{cases}$$

In order to solve the minimization problem sign of the *cost* sum can be change to negative.

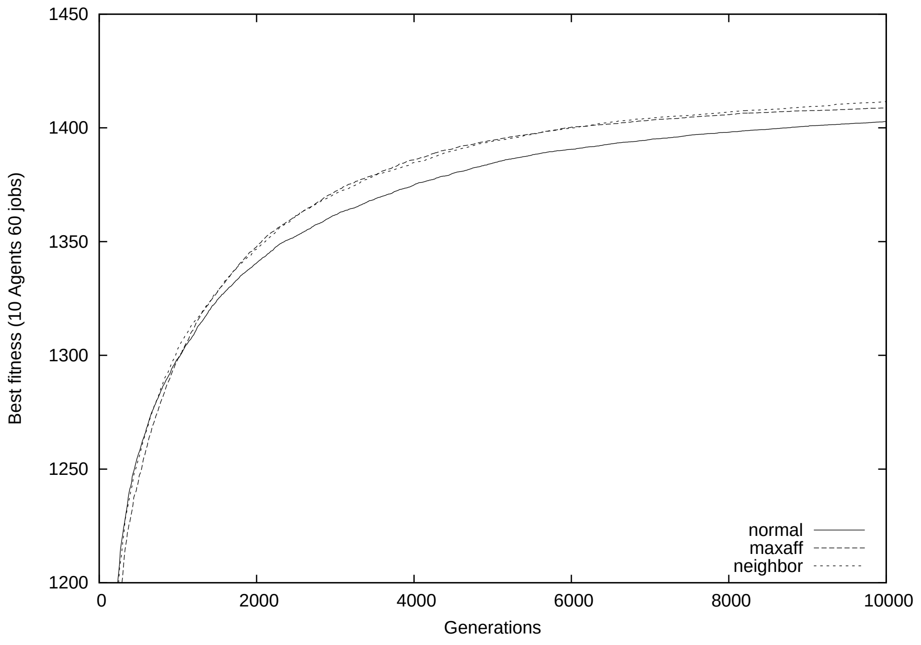
Introduced methods are tested for two test data from [10] for 10 agent 60 jobs and 10 agent 100 jobs cases solved as maximization problems. Experiments are repeated 100 times for each of the neighbor affinity, *neighbor*, greedy maximum affinity, *maxaff* and fixed permutation, *normal* cases. All cases start with an initial permutation $p(i) = i$. The following genetic algorithm parameters are used in the experiment:

Population size (Q)	200
Chromosome length (m)	60 and 100
Gene values	Integers in $1, \ldots, n$
Crossover ratio	$Q/2$ crossovers are made
Mutation ratio	2.5% of the genes are mutated randomly
Selection method	Elitism (Rank based)

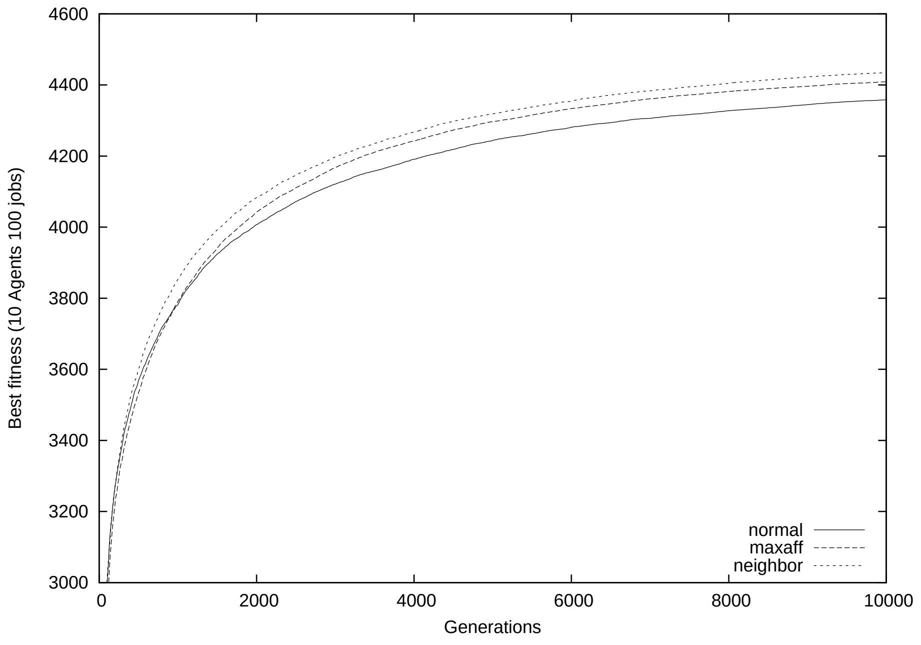
Number of generations	10000
Keep ratio	top 1% of the old generation is processed in the selection
Count of best (M)	20
Affinity function	average of absolute differences (see 4.1.4)
Reorder frequency	Once in 10 generations permutation is changed in *maxaff* and *neighbor* cases.

Experiment results are show in Figure 4.5. In this problem neighbor affinity and greedy maximum affinity cases performed better than the normal fixed order case as expected.

In this problem however maximum affinity case did not outperform the neighbor affinity version. This show the more chaotic nature of neighbor affinity method helped construction of better building blocks for this problem.

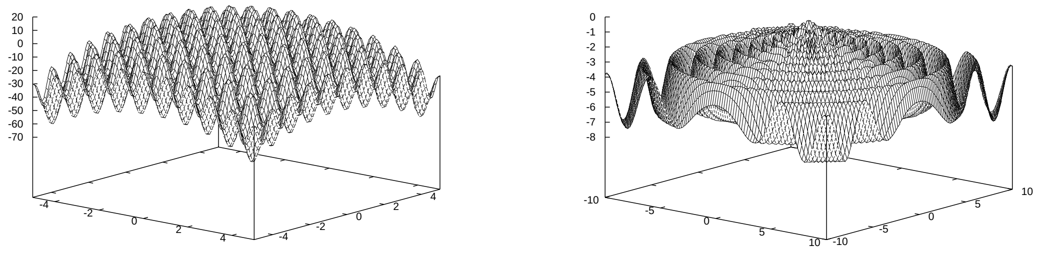


(a)



(b)

Figure 4.5: General assignment problem results: (a) 60 jobs 10 agents (b) 100 jobs 10 agents. Average of 100 runs



(a) (b)

Figure 4.6: Plots of the two functions that are used in the tests.(a) is the $f_6(x, y)$ plot and (b) is the $f_{10}(x, y)$ plot.

4.3 Test Problems

Introduced neighbor affinity and greedy maximum affinity based methods are also tested on the artificially generated floating point problems to observe their generality. These two functions are defined in [66] and contain many local optima whereas the global optimum for both of them is at the point where all arguments are 0.0. They can be generalized in N dimensions where two dimensional versions are plotted in Figure 4.6.

$$f_6(x_1, \ldots, x_N) = -\sum_{i=1}^{N} \left(x_i^2 - 10cos(2\pi x_i)\right)$$
$$f_{10}(x, y) = -\sqrt[4]{x^2 + y^2}\left(sin^2(50\sqrt[10]{x^2 + y^2}) + 1\right)$$

The first function f_6 is a function with many local optima. It is an example of linearly separable functions, since it can be decomposed into a linear combination of other functions (of single variables). For instance, it is possible to write f_6 as $f_6(x, y, z) = g(x) + g(y) + g(z)$. In the optimization perspective all gene values can be optimized independently and results can be combined. The representation order is not important because of these independency.

The second function takes two arguments. To generalize it to N dimensions, a composition should be made. But to make the problem more interesting, instead of as linearly separable composition we choose to link consequent variables in tuples and join them as: $f(x, y, z, w) = f_{10}(x, y) + f_{10}(y, z) + f_{10}(z, w)$. As a result, the neighbor gene values will be related in the fitness function. In general terms the following N

parameter function is used:

$$F_{10}(x_1, x_2, \ldots, x_N) = \sum_{i=1}^{N-1} f_{10}(x_i, x_{i+1})$$

4.3.1 Experiment Setting

In the experiments four different cases are tested. All four cases start with the same random initial population. In the *normal* case the original representation with perfect ordering is assumed to be known and this fixed permutation is used during the evolution.

In other three cases, it is assumed that the perfect permutation is not known and all of them start with the same random initial permutation. In the *permuted* case this random permutation is fixed and not changed. In the *neighbor* case gene reordering based on the neighbor affinities is applied. In the *maxaff* case the greedy maximum affinity method is applied.

In the gene representation, floating point numbers are used. The Allele for all of them is the interval $[-10.0, 10.0]$. Initial population and mutations are based on random floating point values in this interval. Crossover is applied differently at the exact crossover point. Instead of copying the values of the parents directly, a linear combination of both of the parent values are transferred to the offsprings, depending on a random weight value w:

$$\Gamma_i^{AB} = \begin{cases} \Gamma_i^A & \text{if } p(i) < k \\ \Gamma_i^B & \text{if } p(i) > k \\ \Gamma_i^A \, w + \Gamma_i^B \, (1-w) & \text{if } p(i) = k \end{cases}$$

$$\Gamma_i^{BA} = \begin{cases} \Gamma_i^B & \text{if } p(i) < k \\ \Gamma_i^A & \text{if } p(i) > k \\ \Gamma_i^B \, w + \Gamma_i^A \, (1-w) & \text{if } p(i) = k \end{cases}$$

This enriches the variety of values in the population while preserving the parents contribution at the crossover point.

Following parameters are used in the algorithm:

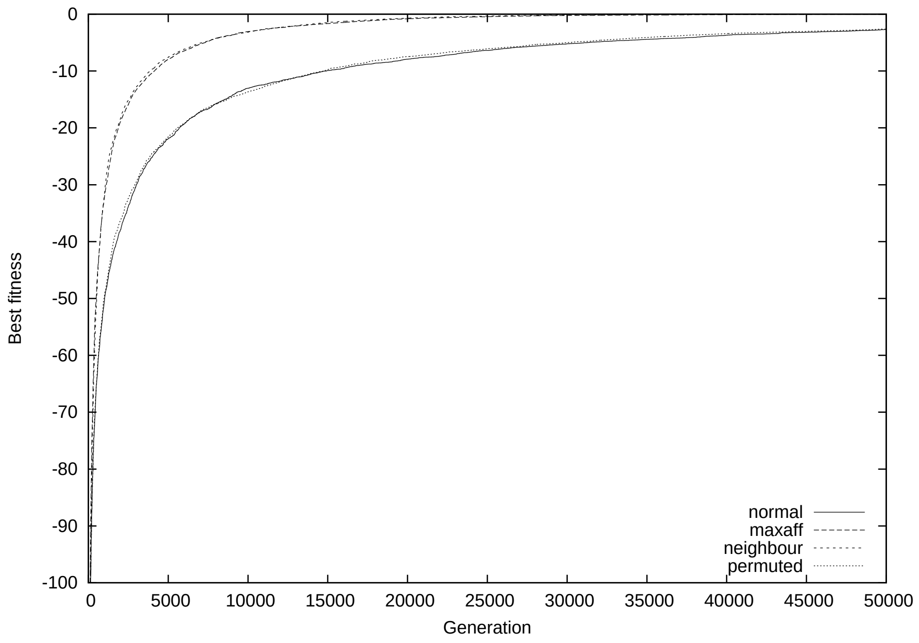
Population size (Q)	100
Chromosome length (n)	16

Gene values	Floating point values in [-10,10]
Crossover ratio	$Q/2$ crossovers are made
Mutation ratio	2.5% of the genes are mutated randomly
Selection method	Elitism (Rank based)
Number of generations	50000
Keep ratio	top 1% of the old generation is processed in the selection
Count of best (M)	20
Affinity function	Standard deviation of absolute differences (see 4.1.4)
Reorder frequency	Once in 10 generations permutation is changed in *maxaff* and *neighbor* cases.

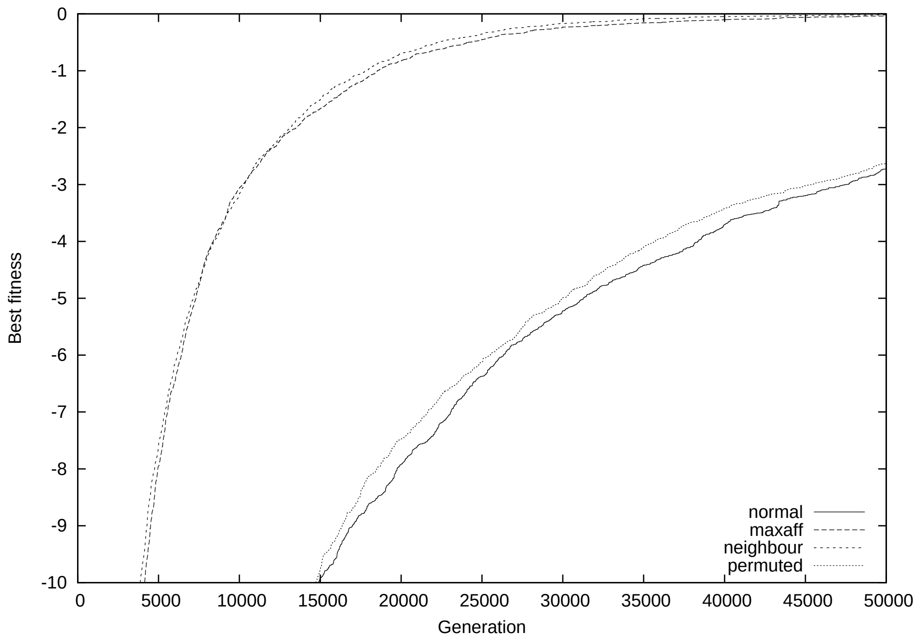
All four cases are repeated with 100 different initial random seeds and results are collected for two functions f_6 and F_{10}. Since f_6 is a linearly separable problem no difference between the fixed order permutation cases *normal* and *permuted* is expected. Because there is no predefined linkage in the problem any crossover strategy will work the same.

However, in the F_{10} case there is a tight linkage among the consequent couples since they contribute to the fitness together. In this function the cases respecting the linkage are expected to perform better.

4.3.2 Experiment Results

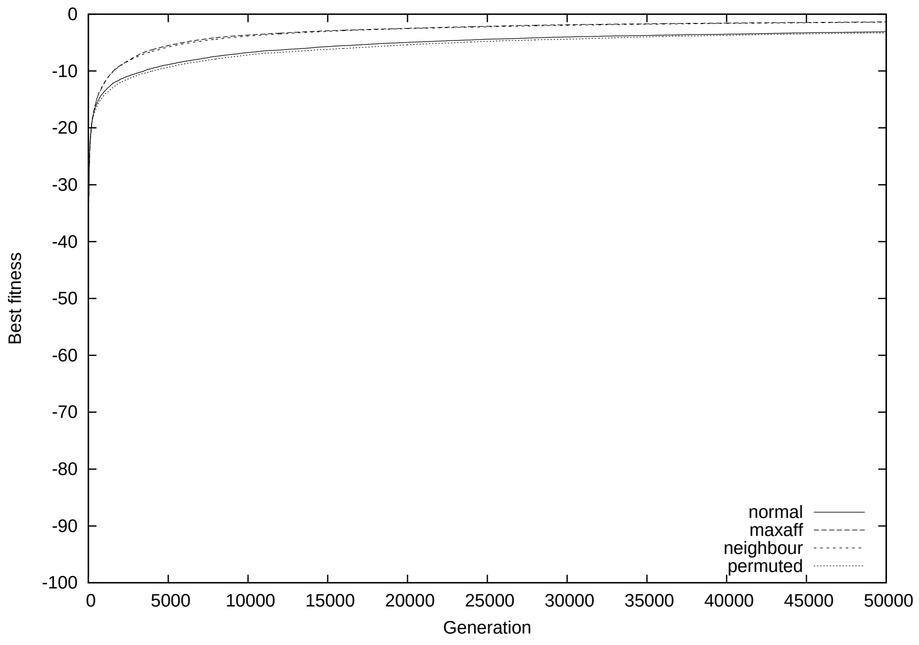


(a)

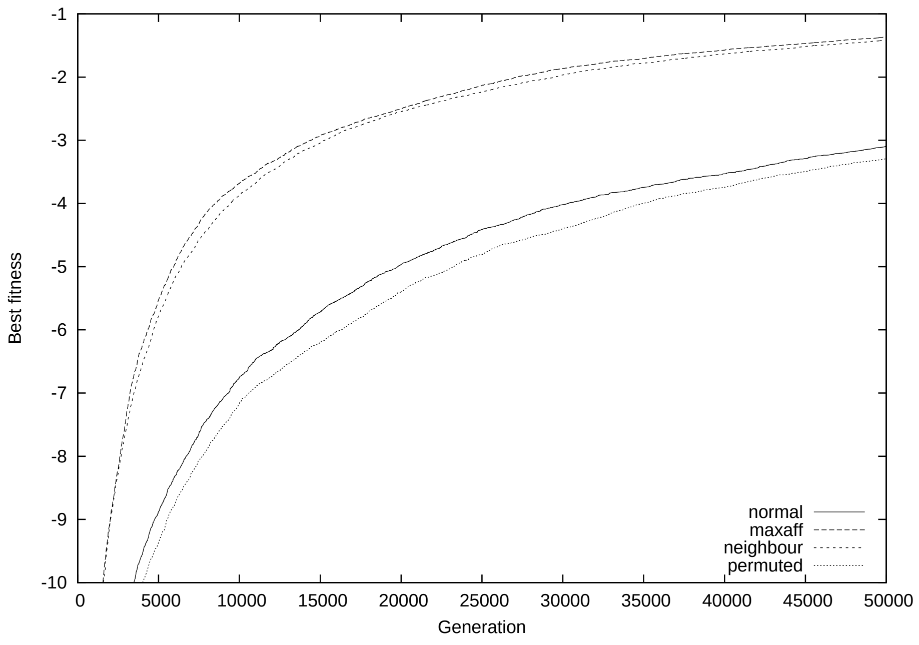


(b)

Figure 4.7: Results of the f_6 function execution for four test cases, average of the 100 repetition (b) is the zoomed version of the same plot



(a)



(b)

Figure 4.8: Results of the F_{10} function execution for four test cases, average of the 100 repetition (b) is the zoomed version of the same plot

Results of these experiments are shown in the Figure 4.7 and Figure 4.8. Evolution of the best individuals' fitness is shown as the average of 100 executions for 4 cases.

Results of f_6 shows an interesting property of the proposed methods. Although there is no predefined linkage in the problem they performed significantly better than the fixed order versions. Although 50000 generations is not sufficient to find the global optimum with the fixed orders, they could find it in almost all cases. This implies that they provide good means of construction and mixing of the building blocks. Thus possibly they could construct, capture and preserve the inherent building blocks of the population based on the already converged values of the current state. This implies that these proposed methods can perform well even if there is no predefined linkage.

Results of F_{10} function similarly shows that the reordering methods are better than the fixed order ones. However, in this problem, since the perfect linkage is known, the *normal* case performs slightly better than the other fixed order case which is the *permuted* one. The proposed methods are still better than the *normal* case. This can again be explained by the building block construction abilities of the methods. Since it is not always possible to have all schemata in the initial population present in a close neighborhood, in the initial stages of the algorithm it might be better to split up linked positions so that good schema have a chance to emerge from this separation. After the construction of the schema, it is important to preserve the constructed building blocks and it would be better to have a tight linkage. Thus the algorithm has a stirring and shuffling effect in the schema space. This might be the explanation of why the proposed methods have performed better than even the *normal* case.

4.3.3 Observing the Linkage

The F_{10} function of the previous subsection gives us the opportunity to observe the linkage change during the evolution. Since the positions of the genes contributing to the fitness function together is known, the average defining length of the order two schemata in a given permutation can be determined. It is simply the average distance of the permutation placement of the consequent gene positions. For example, if x_1 is placed in the 3^{rd} position in the permutation mapping and x_2 is placed in the 10^{th},i.e. $p(1) = 3, p(2) = 10$ the linkage of these two genes is 7. Taking the average of such calculated values for the position pairs $1-2, 2-3, \ldots, 14-15, 15-16$, the average

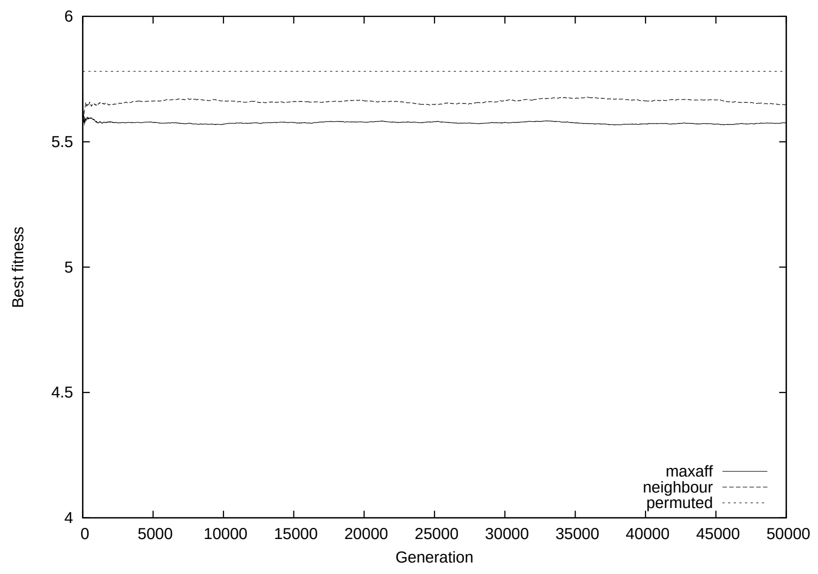


Figure 4.9: Average linkage of the F_{10} function execution for four test cases, average of the 100 repetition

linkage of the current permutation is found. Average linkage $L(p)$ of permutation p is defined as:

$$L(p) \triangleq \frac{\sum_{i=1}^{N-1} |p(i) - p(i+1)|}{N-1}$$

In the *normal* case the algorithm always works with the perfect linkage, $L(p) = 1$. In the *permuted* case, the average of the $L(p)$ values is equal to the expected value of the random permutation for N for all of the generations since it is fixed at the beginning and that permutation is kept until the termination. However, in the proposed methods *permuted* and *maxaff* we can expect to observe a decrease from this expected linkage. This is observable in the results displayed in Figure 4.9.

4.4 Non-binary Deceptive Functions and the Linkage

In order to make controlled experiments about the relation between the linkage and the affinity, it is possible to define some trap functions [22] where finding the linkage is important in the performance of the genetic algorithms. Also the linkage of a permutation could be observed in this way.

The deceptive functions that are used in other linkage learning approaches like [30, 48, 38, 50, 51, 52] are not suitable for our purposes, since we aim to find the

$$f_{3deceptive}(u) = \begin{cases} 0.9 & \text{if } u = 3 \\ 0.8 & \text{if } u = 2 \\ 0 & \text{if } u = 1 \\ 1 & \text{if } u = 0 \end{cases}$$

$u$is the number of ones in the 3-bit string

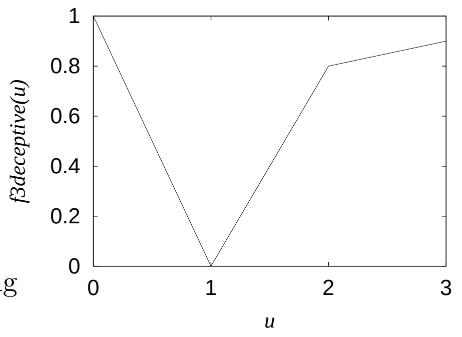


Figure 4.10: A trap function of order 3 and the corresponding graph.

d		$f_{intdec}(d)$
$[0, \alpha_1]$	:	$1 - \frac{x}{\alpha_1}$
$[\alpha_1, \alpha_2]$	:	0
$[\alpha_2, \frac{\alpha_2+\alpha_3}{2}]$	:	$\frac{2\gamma}{\alpha_3-\alpha_2}(x - \alpha_3)$
$[\frac{\alpha_2+\alpha_3}{2}, \alpha_3]$	:	$\frac{2\gamma}{\alpha_2-\alpha_3}(x - \alpha_2)$
$[elsewhere]$	:	0

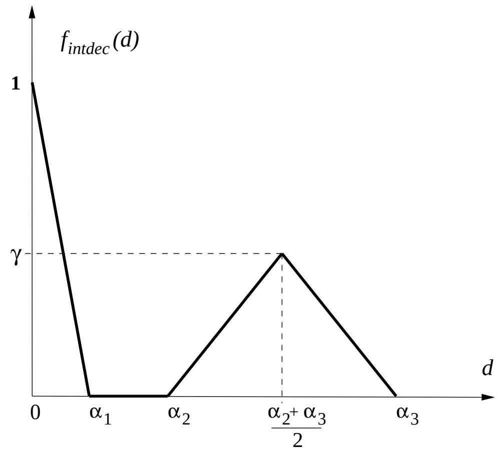


Figure 4.11: Integer trap function used in the experiments The $f_{intdec}(d)$ function where d is the Euclidean distance of the point from the predefined global optimum.

linkage in non-binary alphabets and they were defined on binary gene values. Though it is meaningful to investigate the nature of bit deceptive problems. The the fitness function displayed in Figure 4.10 is defining such a problem. When one or two digits of the 3 digit string are considered, one would tend to set the digits to 1 which is unfortunately the local minima. Similarly, considering only a subset of the dimensions should lead to a local minima in the design of the non-binary deceptive functions. We use this idea and propose a deceptive function working on integer or floating point values.

The proposed deceptive function is shown in Figure 4.11. The parameters $\alpha_1, \alpha_2, \alpha_3$ and γ can be adjusted according to the desired characteristics. α_1 defines the slope of the global optimum hill. α_2, α_3 define the width and distance of the local optima range. γ defines the local optima weight.

f_{intdec} function can be used to form order k deceptive functions by simply calculat-

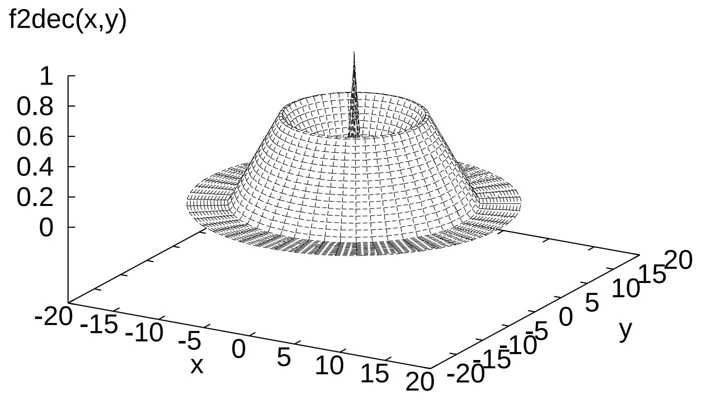


Figure 4.12: 3D plot of the order 2 fitness function derived from $f_{intdec}(d)$ function for global optimum is at (0,0)

ing the Euclidean distance d of a k dimensional point to the global optimum point and getting the value of $f_{intdec}(d)$. The resulting function will have a hyphersphere where global optimum is in the center and local optima points will have the same distance from the global optimum coordinate $\overline{o}$. In Figure 4.12 a 3D plot of the function is given for $k = 2$.

$$f_{kdec}(x_1, .., x_k)|_{\overline{o}} = f_{intdec}(\sqrt{\sum_{i=1}^{k}(x_i - o_i)^2})$$

After defining the deceptive functions the test problems are constructed by linear compositions of these order k functions. Thus an order 2 problem of length N is defined as:

$$f_{2,N}(x_1, x_2, \ldots, x_N) = \sum_{i=1}^{N/2} f_{2dec}(x_{2i-1}, x_{2i})|_{\overline{o}_i}$$

Resulting function is a combination of $N/2$ independent problems of order 2. $\overline{o}_i$ are vectors describing the desired coordinates of the optimum points of the order 2 problems.

The average linkage $L(p)$ in such a problem is similarly defined as:

$$L(p) = \frac{\sum_{i=1}^{N/2} |p(2i-1) - p(2i)|}{N/2}$$

Generalizing deceptive functions to an order k problem of size N

$$f_{k,N}(x_1, x_2, \ldots, x_N) = \sum_{i=1}^{N/k} f_{kdec}(x_{(ki-k+1)}, x_{(ki-k+2)}, \ldots x_{(ki)})\big|_{\overline{o}_i}$$

In the linkage of the general case the defining length of the schema containing all order k subproblem elements should be considered:

$$L(p) = \frac{\sum_{i=1}^{N/2} L_i(p)}{\frac{N}{k}}$$

where

$$L_i(p) = \max(p(m)) - \min(p(m)) \text{ for all } m \text{ in } \{(ki-k+1), (ki-k+2), \ldots, (ki)\}$$

4.4.1 Experiment Setup

In the experiments two problems, first with order 2 and size 16 ($k = 2, N = 16$) and the second with order 3 and size 15 ($k = 3, N = 15$) are constructed as described above. However, instead of using the same global optimum for all of the subproblems, a different optimum point is chosen for each. This choice is made to rule out the possibly of false linkages to be discovered among the genes of two different subproblems. This is very probable when two subproblems search for the same global optimum.

We have used the freedom of setting the coordinates of the optimum for the sub problems

$$[\overline{o}_i]_j \overset{\triangle}{=} k(i-1)$$

where k is the order, j refers to any component of the coordinate and $i = 1, \ldots, N/k$. This will obviously minimize any undesired entanglement among building blocks of (unrelated) sub problems.

Thus resulting fitness functions are:

$$\begin{aligned} f_{2,16}(x_1, \ldots, x_{16}) &= f_{2dec}(x_1, x_2)|_{(0,0)} + f_{2dec}(x_3, x_4)|_{(2,2)} + \ldots \\ &\quad + f_{2dec}(x_{15}, x_{16})|_{(14,14)} \\ f_{3,15}(x_1, \ldots, x_{15}) &= f_{3dec}(x_1, x_2, x_3)|_{(0,0,0)} + f_{3dec}(x_4, x_5, x_6)|_{(3,3,3)} + \ldots \\ &\quad + f_{3dec}(x_{13}, x_{14}, x_{15})|_{(12,12,12)} \end{aligned}$$

Resulting global optimum will be `[0,0,2,2,4,4,6,6,8,8,10,10,12,12,14,14]` for $f_{2,16}$ and `[0,0,0,3,3,3,6,6,6,9,9,9,12,12,12]` for $f_{3,15}$. Local optima are all the points having an Euclidean distance of 7 to the subproblem optimum points. The local optima receive a fitness value that is 0.6 of optimum. The first problem has a maximum fitness value of 8 and the second has one of 5.

Experiments similarly repeated for four cases *normal*, *permuted*, *maxaff* and *neighbor* and results are collected for 100 executions with different random seeds.

Parameters of the experiments are similar to the Section 4.3.1 and same mutation and crossover strategies are used:

Population size (Q)	100
Chromosome length (N)	16 in $f_{2,16}$, 15 in $f_{3,15}$
Gene values	Integers in [0,15]
Crossover ratio	$M/2$ crossovers are made
Mutation ratio	2.5% of the genes are mutated randomly
Selection method	Elitism (Rank based)
Number of generations	30000 and 40000
Keep ratio	top 1% of the old generation is processed in the selection
Count of the best (M)	20
Affinity function	Standard deviation of absolute differences and average absolute differences (see 4.1.4)
Reorder frequency	Once in 10 generations permutation is changed in the *maxaff* and *neighbor* cases.

4.4.2 Experimental Results

Similar to the other test cases, the proposed methods increased the online performance of the algorithm significantly as shown in Figure 4.13(a) for order 2 and Figure 4.14(a) for order 3. However, a negative change in performance is observed as the order of the problems increase.

In order 2 both *neighbor* and *maxaff* performed almost equivalent to *normal* case with perfect linkage and could find the global optimum in almost all executions. On the other hand evolution of the *permuted* case is significantly slow and it could not find the global optimum in the given number of generations. This shows the importance of linkage in deceptive problems.

In order 3, however *neighbor* case had a lower performance but still better than the *permuted*. Considering all methods, none of them could find the global optima

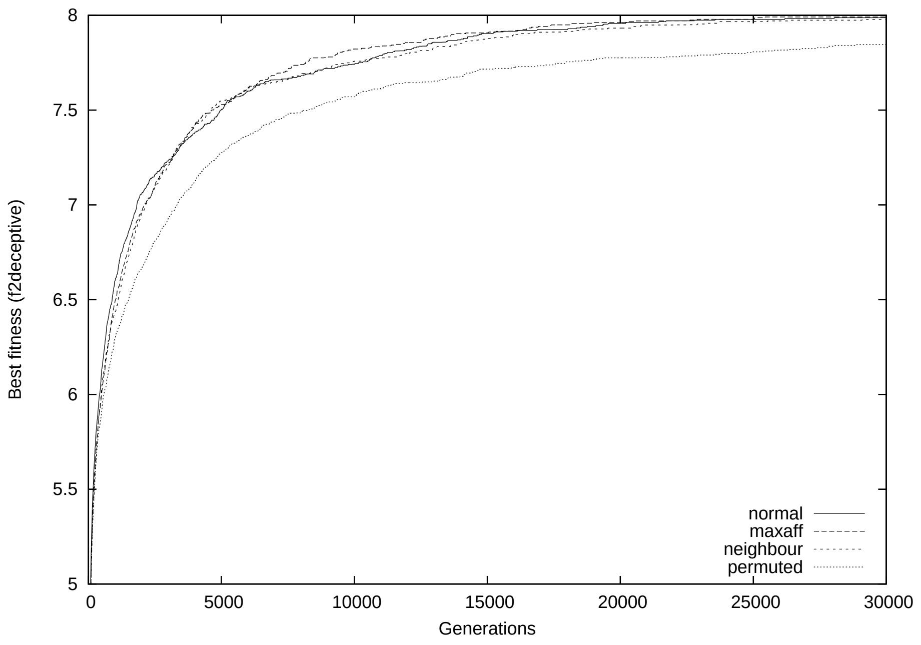
Table 4.1: Expected linkage values calculated from the linkage frequencies

	$f_{2,16}$	$f_{3,15}$
normal	1	2
permuted	5.66232	7.99949
neighbor	5.49344	7.70071
maxaff	3.66196	6.162223

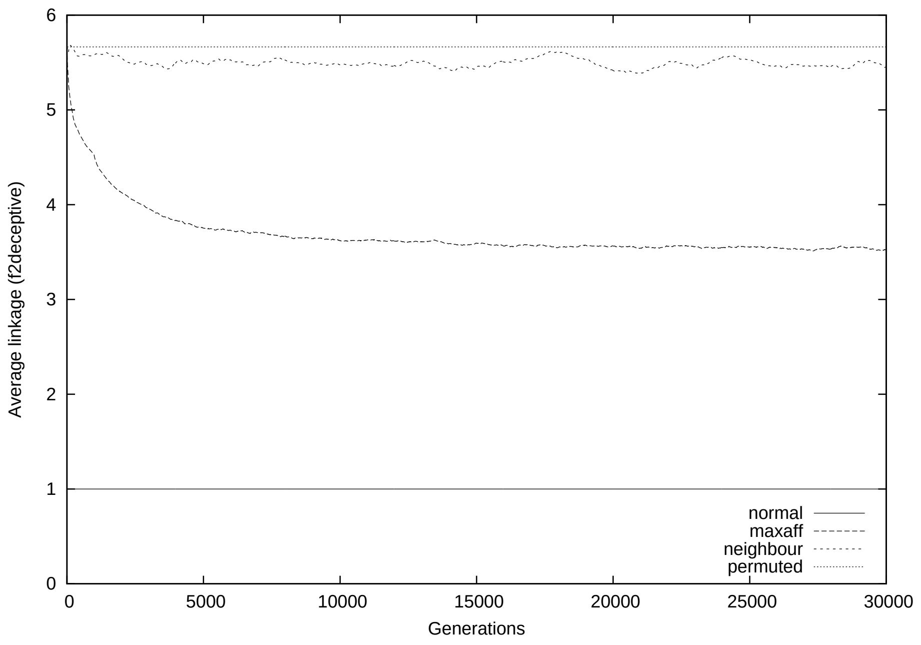
5 in the given number of generations. This is due to the exponential increase in the complexity of the problem as the order of subproblems increases.

In figures Figure 4.13(b), 4.14(b) the average linkage values $L(p)$ during the evolution are shown. Linkage of the *normal* case is always 1 in $f_{2,16}$ and 2 in the $f_{3,16}$. In the *permuted* case the average linkage values in all of the generation steps are the same, 5.665 and 7.936 respectively. These values are sample averages of the random permutation distribution. Figure shows that the *maxaff* method could decrease the linkage by approximately 2. The *neighbor* method could decrease the linkage a little and more slowly compared to *maxaff*.

In order to observe the linkage change better, it is also required to look at the frequency distribution of the linkage values. This shows which linkage values are used in the crossover throughout the evolution and will give an idea of the expected linkage value for all generations. This distributions are given in Figure 4.15. The linkage frequency of the *permuted* case is generated in another environment with the same number of instances to have a better sample space (instead of 100). Results show that the linkage is distributed almost evenly by the methods and *maxaff* shifted the distribution better than the *neighbor* did. In Table 4.1 expected linkages calculated from the frequency distributions are given.

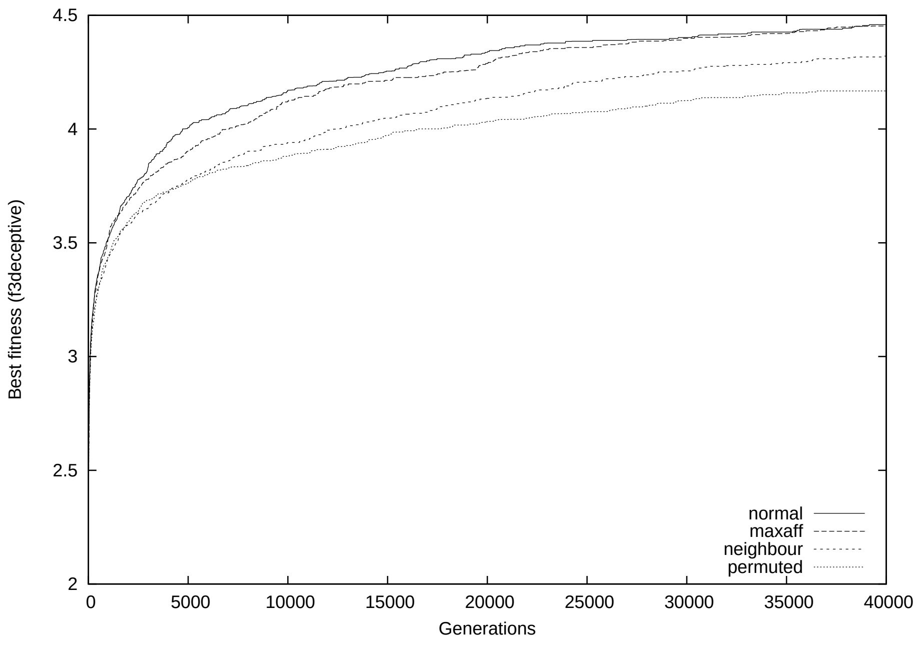


(a)

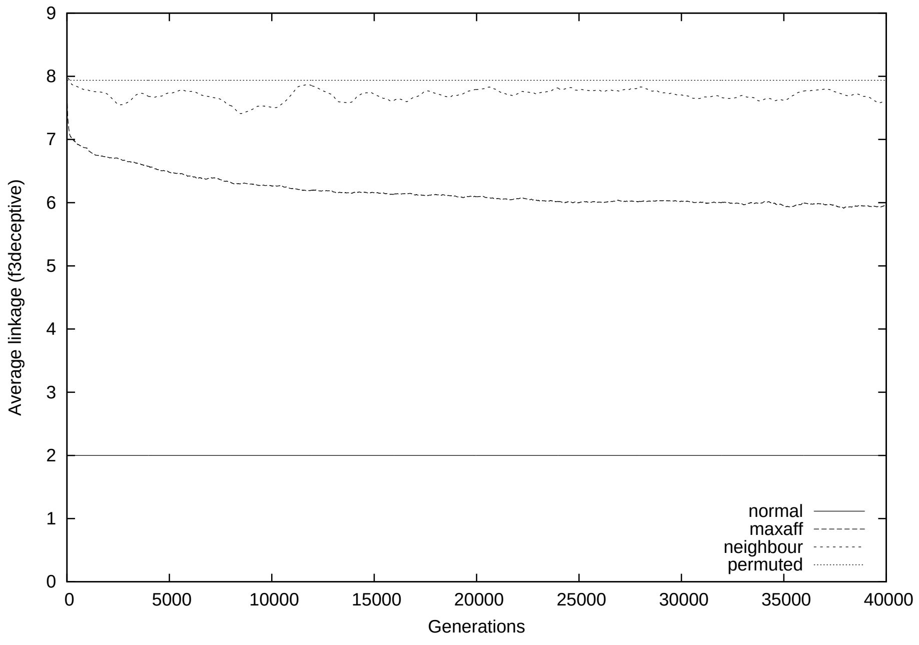


(b)

Figure 4.13: (a):Best fitness evolution of the deceptive function with $k = 2, N = 16$, (b): Average linkage

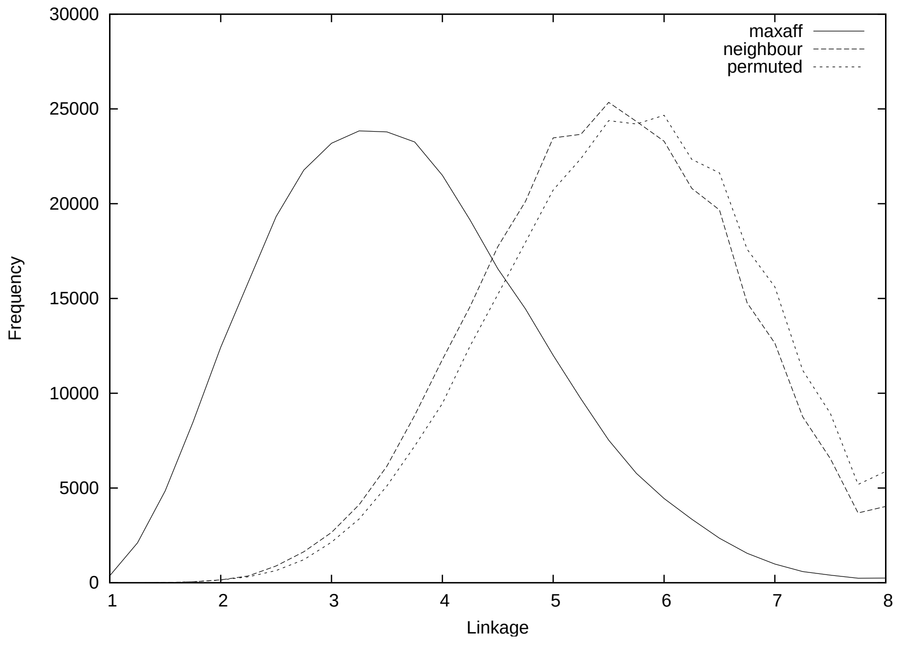


(a)

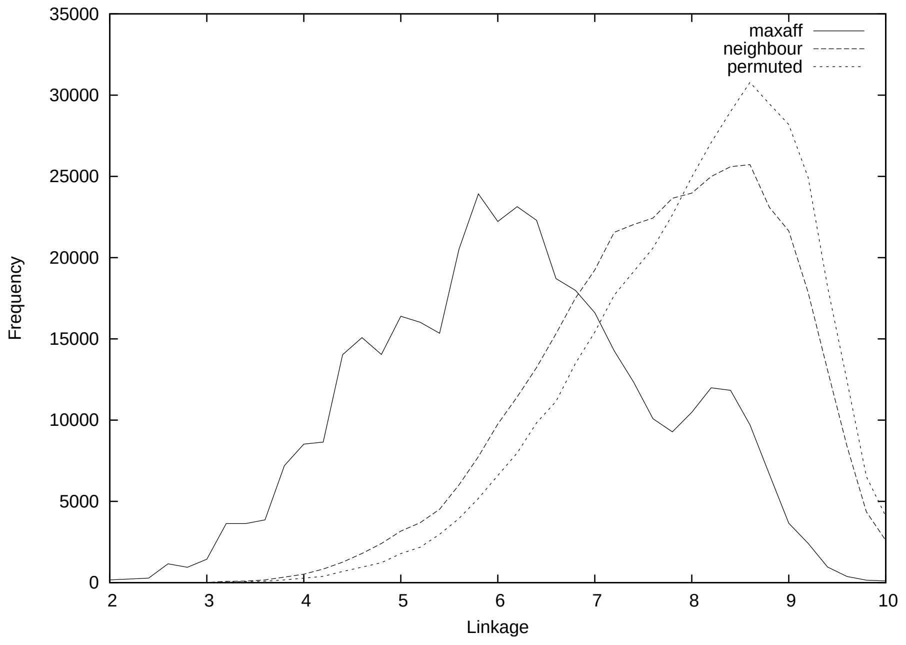


(b)

Figure 4.14: (a):Best fitness evolution of the deceptive function with $k = 3, N = 15$, (b): Average linkage



(a)



(b)

Figure 4.15: Distribution of linkage values during the whole evolution states for 100 repetitions (a) order 2 and (b) order 3

CHAPTER 5

GENE LEVEL CONCURRENCY

In Chapter 3 a survey of parallelization attempts of genetic algorithms are given. In global parallelization genetic operators are parallelized while in coarse grain and fine grain parallelism genetic algorithm is modeled as concurrently evolving and interacting subpopulations, namely demes.

All these approaches apply chromosome level or population level parallelism. In this chapter an alternative way of concurrent modeling of genetic algorithms is introduced. This model, named gene level concurrency, considers genes as individual entities evolving concurrently.

In the following sections this model is explained and implementation details are given for the realization of the model in a concurrent environment. This environment can be any multiprocess system with interprocess communication. In the model, following assumptions are made for the underlying architecture:

- Processes or threads can be created and executed asynchronously
- Order preserving and lossless communication channels can be established
- Implementation of broadcast or multicast message passing is possible
- Primitives for synchronization among processes exist

5.1 Gene Process Model

In the proposed model each gene position in the phenotype is executed by a separate process. In each process an object, keeping the specific genom data of the whole

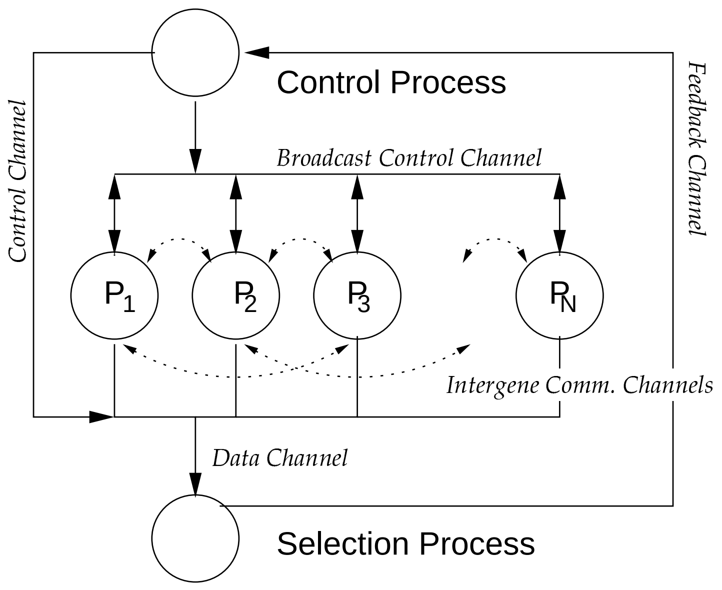


Figure 5.1: Architecture of the gene process model

population, is alive. These objects are connected to each other and to two other special processes via communication channels. These two special processes are the *Control Process* and the *Selection Process*. Architecture of the model is given in Figure 5.1

Execution of the entire model is based on message passing. All gene processes and the selection process execute a *receive message-do corresponding action* loop. The control process initiates and supervises the genetic algorithm.

If i^{th} chromosome in the population is represented as a vector of genes, $\Gamma_i = \langle g_1, g_2, \ldots, g_n \rangle$ where n is the phenotype length, and the population size is m, the whole population can be denoted as $\mathbf{\Gamma}$, a vector of chromosomes. A particular gene value in the population is then represented as $\Gamma_{i,j}$ denoting the j^{th} gene of the i^{th} chromosome.

In the proposed architecture, this representation is transposed into a collection of vertical gene vectors instead of a collection of chromosomes. A gene vector Λ_i is the vector containing all gene values in the i^{th} position of the chromosomes of the pool $\Lambda_i = \langle \Gamma_{1,i}, \Gamma_{2,i}, \ldots, \Gamma_{m,i} \rangle$.

All gene processes are identical objects defined by a set of attributes and actions (methods). Thus, a gene process P_i is a tuple:

$P_i = \langle i, \mathsf{pos}, \Lambda, new\Lambda, \mathsf{Broad}, \mathsf{Inport}, \overline{\mathsf{Outports}}, \mathsf{Actions}\rangle$

Where:

i: Position of the gene implemented by this process in the phenotype.

pos: Phenotype independent position information. It is used by variable permutation representations and may contain any spatial or hierarchical information. In simple GA implementation it is set to i.

Λ: Gene vector Λ_i of the current population.

$new\Lambda$: Gene vector Λ_i of the new generation.

Broad: Broadcast command channel to be listened

Inport: Private input channel of the process

$\overline{\mathsf{Outports}}$: A vector of output ports that this process uses to send data. It includes the data channel flowing through the *selection process*, the inter-gene channels and the broadcast channel. In simple GA implementations only the data channel is sufficient.

Actions: A set of actions. Each action is identified by a named tuple of any arity. When a message of the same pattern is received, the corresponding action is executed by the gene process.

A gene process executes a very simple command sequence:

1. Wait for a message in Inport or Broad
2. Execute the matching action
3. Go to step 1

Selection process S is similarly a tuple:

$S = \langle \mathbf{\Gamma}, new\mathbf{\Gamma}, \overline{\mathsf{fitnesses}}, \mathsf{Inport}, \mathsf{Outport}, f, \mathsf{Actions}\rangle$

Where:

$\mathbf{\Gamma}$: Data of the entire pool of the current generation.

$new\mathbf{\Gamma}$: Data of the entire pool of the new generation.

$\overline{\mathsf{fitnesses}}$: Fitnesses calculated for the individuals in the pool

Inport: Input channel to be listened.

Outport: Feedback channel to send results of the selection.

f: Fitness function.

Actions: A set of actions. Similar to gene processes.

Selection process executes the same loop with the gene processes.

1. the pool data is received from the gene processes,
2. the fitness calculation
3. the selection action is initiated by a message coming from the control process
4. the selection information is sent back to the control process via the feedback channel.

The control process executes the genetic algorithm loop by sending

1. genetic operator messages to all gene processes
2. the fitness calculation message to selection process
3. the select message to the selection process

in each iteration. Repeating this iteration for many generations will make genetic algorithm work.

5.2 Simple GA Implementation

In a simple GA implementation a gene process implements the following messages:

`init(`M`,[`min`, `max`])`

Initializes the gene vector Λ with random values in $[min, max]$ interval. If min, max values are skipped, binary encoding is assumed (0 and 1 respectively).

$$\Lambda_i \leftarrow random([min, max]) \text{ for all } i = 1, .., M$$

`mutate(`$prob$`,[`min`, `max`])`

All genes in the new population $new\Lambda$ is mutated with a probability of $prob$. If min and max is skipped binary encoding is assumed and bit is complemented instead of a random value.

$$new\Lambda_i(t+1) \leftarrow \begin{cases} random(min, max) & \text{if } random([0,1]) \leq prob \\ new\Lambda_i(t) & \text{otherwise} \end{cases} \quad \text{for all } i = 1, .., M$$

`crossover(`cp`, `$p1$`, `$p2$`)`

Parents $p1$ and $p2$ in the current population are crossovered to produce two new offsprings in the new population. cp is the crossover point where offsprings switch parents. ($\bullet$ is used to denote the append operation)

$$new\Lambda \leftarrow new\Lambda \bullet \langle g_A, g_B \rangle \text{ where } \langle g_A, g_B \rangle = \begin{cases} \langle \Lambda_{p1}, \Lambda_{p2} \rangle & \text{if } \mathsf{pos} \leq cp \\ \langle \Lambda_{p2}, \Lambda_{p1} \rangle & \text{otherwise} \end{cases}$$

`uniformcrossover(`$p1$`, `$p2$`,[`$prob$`])`

In uniform crossover, offspring value is taken from one of the parents, probabilistically. Thus, the position of a gene is not significant. $prob$ is the probability of taking the value from the first parent. If it is skipped this value is assumed as 0.5.

$$new\Lambda \leftarrow new\Lambda \bullet \langle g_A, g_B \rangle \text{ where } \langle g_A, g_B \rangle = \begin{cases} \langle \Lambda_{p1}, \Lambda_{p2} \rangle & \text{if } random([0,1]) \leq prob \\ \langle \Lambda_{p2}, \Lambda_{p1} \rangle & \text{otherwise} \end{cases}$$

`senddata()`

This message initiates the sending of the gene information to the selection process so that it could make calculations on this global data.

$\mathsf{send}_{datachannel}(\mathtt{data}(i,\ new\Lambda))$

`select(`$List$`)`

The control process sends a list of chromosome identifiers to be selected via this message. The $List$ stems from the selection process and contains a sequence of integers representing the position of the chromosomes to be selected from the new population into the current population.

$$\Lambda \leftarrow \langle new\Lambda_k \mid k \in List \rangle$$
$$new\Lambda \leftarrow \langle\ \rangle$$

Similar to the gene processes, selection process implements a set of messages as actions. Those messages are received over the same channel from gene processes or control process.

`data(`i`,`L`)`

Gene vector $new\Lambda$ values are received from P_i in vector L and $new\Gamma$ is updated with the gene values

$$\mathbf{\Gamma}_{k,i} \leftarrow L_k \text{ for all } k = 1, \ldots, \mid L \mid$$

`setfitnesses()`

Fitness values of the chromosomes are calculated by calling fitness function f.

$$\overline{\mathsf{fitness}}_i \leftarrow f(\Gamma_i) \text{ for all } i = 1,..,\mid \Gamma \mid$$

`select(`K`)`

K element is selected from the pool for the next generation.

$$\Gamma \leftarrow \langle Chrom \mid Chrom \in \mathsf{selected}(K, new\Gamma)$$
$$new\Gamma \leftarrow \langle\ \rangle$$

selected is the result of the implemented selection mechanism. Any of *tournament selection, roulette wheel selection, rank selection* or any other operator can be implemented.

`sendselected()`

Sends the result of the selection as a list of chromosome identifiers (order of the chromosome in the pool vector).

$$\mathsf{send}_{feedbackchannel}(\mathtt{select}(List)) \text{ where } List = \langle i \mid \Gamma_i \text{ is selected from the } \rangle$$

After this definitions of messages (or actions) control process executes the following loop:

```
send_{broadcast}(init($M$))
DO {    REPEAT ($CrossoverRatio \times M$) times {
                 $A \leftarrow random(1,M)$
                 $B \leftarrow random(1,M)$
                 $CP \leftarrow random(1,N)$
                 send_{broadcast}(crossover($CP, A, B$))
        }
        send_{broadcast}(mutate($MutateProb$))
        send_{broadcast}(senddata())
        send_{selection}(setfitnesses())
        send_{selection}(select($M$))
        send_{selection}(sendselected($M$))
        receive_{feedback}(selected($List$))
        send_{broadcast}(select($List$))
} WHILE Maximum number of generations is not reached
```

Most of the communication can be done asynchronously, however there are two places where synchronization is required. First one is the fitness calculation on the selection process. The system should wait for all data of $new\mathbf{\Gamma}$ to become available

Table 5.1: Computation costs of the serial and parallel cases

Operation	Serial	Parallel
Crossover	mn	$max(\frac{mn}{p}, m)$
Mutation	mn	$max(\frac{mn}{p}, m)$
Fitness calc.	mnT_{fit}	$max(\frac{mnT_{fit}}{p}, nT_{fit})$
Selection	m	m
Communication	0	$T_{com}m(p-1)$

to the selection process before calculating the fitnesses. Thus, control process should wait until all gene processes have completed sending their data, and then continue with `setfitnesses(...)` message.

The other synchronization is required when the selection process sends the selected individual identifiers over the feedback channel. Control process should wait until this message is received before starting a new generation.

5.3 Parallel Implementation Considerations

This model best fits under global parallelization methods where same population and generation loop is parallelized with a sequential implementation. However in contrast to other —per chromosome, horizontal— global parallelization approaches, in this model parallelization is achieved in a per gene basis (vertical).

Since the model uses message passing and interprocess communication heavily, it is best suited for shared memory architectures where communication cost is relatively small. Furthermore, since most of the communication takes place in a broadcast channel, architectures with broadcast message passing is provided will have an advantage. For example, in a distributed system, if multicast or broadcast messaging is possible it will work much better than a unicast only messaging system.

In systems with insignificant communication cost, the model can achieve better speed-up values. However this ideal case is not achieved easily since communication is usually quite slow compared to processing. Table 5.1 describes the computational costs of operations. m is the poolsize, n problem length, p number of processors, T_{fit}, fitness calculation cost for a chromosome,T_{com} communication cost constant. In the communication cost, the cost of all gene processes sending their gene vectors to the selection process requires the communication of mn elements. This cost depends on the amount of data that can be transsmitted on a single transmission cycle and the cost of this cycle. The other communication costs are in order of m so they can be neglected.

Substituting these values, speed-up can be written as:

$$S(p) = \frac{2mn + T_{fit}mn + m}{2max(\frac{mn}{p}, m) + max(\frac{mnT_{fit}}{p}, nT_{fit}) + m + T_{com}m(p-1)}$$

Plot of the speed-up for functions are given in Figure 5.2. As the communication cost increases the speed-up decreases. On the other hand as the problem length and the population size increase the speedup curve is better.

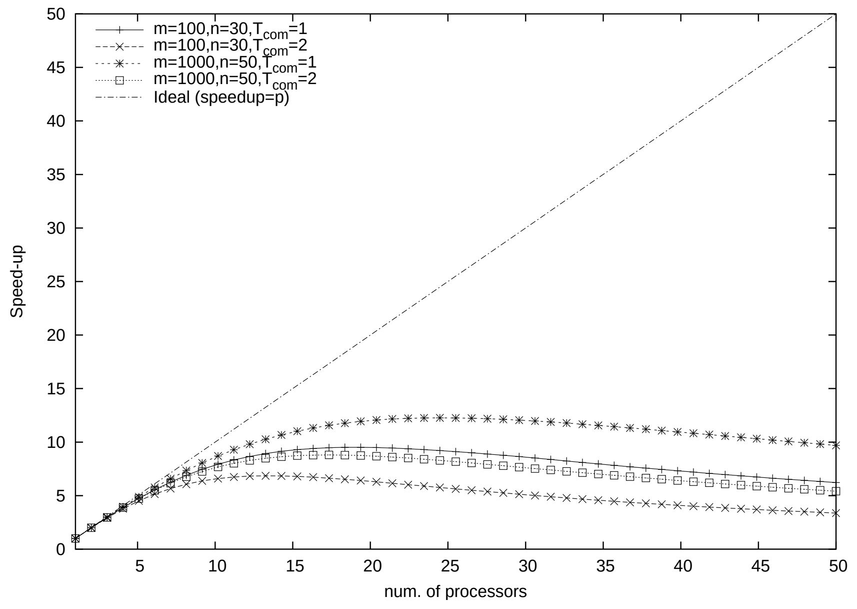


Figure 5.2: Speed-up plots for different parameters

However, the purpose of the model is not to have a more efficient parallelism but to establish a natural and effective way of representing order independency in genetic algorithms and to model a gene as an independent, self functioning entity. Efficiency considerations are left out of the scope of this thesis and expected to be solved by the developments in the parallel architectures.

On the other hand, having representation independent genes provides a natural implementation of non-linear or varying order genetic algorithms with a small effort.

The next section explains how *gene reordering genetic algorithm* is implemented in gene level concurrency.

5.4 Gene Reordering Genetic Algorithms and Gene Level Concurrency

In *gene reordering genetic algorithms* global ordering of genes with respect to crossover operator is changed by the local decisions to maximize affinity values among the gene positions.

In gene level concurrency, crossover operation is based on the pos attribute of each gene process. Thus, moving a gene in the global permutation is just simply updating the pos values of the gene processes. Since the affinity calculations are done among the neighboring genes, they can be calculated by means of local communications. In this way simple genetic algorithm implementation can be extended into gene reordering genetic algorithm with a few additions.

In the gene reordering genetic algorithm, in addition to simple GA, a gene process includes LeftAffinity and RigthAffinity attributes. Furthermore, $\overline{\mathsf{Outports}}$ includes private ports to the other gene processes.

The new gene process is the following tuple:
$P_i = \langle i, \mathsf{pos}, \Lambda, new\Lambda, \mathsf{Broad}, \mathsf{Inport}, \overline{\mathsf{Outports}}, f, \mathit{aff}, \mathsf{Actions}, \mathsf{LeftAff}, \mathsf{RightAff}\rangle$
Only affinity values to the left and right neighbor processes and the user defined affinity function *aff* are added. This *neighborhood* is defined as to possess consecutive pos attribute values.

New message and actions are required to calculate and process affinities. Affinity calculation requires the data of the two processes. Thus, each process sends its data to the left neighbor. Left neighbor calculate and set the right affinity and sends it to the right so that it can update the left affinity. When this cycle is completed all affinity values would be known for all gene processes.

The next step is to update the positions based on the affinities obtained. Since the position updates are local in the implementation of the gene reordering genetic algorithm, this decision is left to the gene process. In the sequential implementation this is done deterministically in a linear scan of the permutation. In the concurrent version this is done probabilistically. Thus, there is no need to check whether the left or right neighbor has already changed position, or not. Only a few of the genes will change position, so the process is local and non-deterministic.

Following messages are introduced in order to implement gene reordering GA:

`sendtoleft()`

This initiates the gene vector exchange among the neighbors. Each gene process sends its Λ to the left neighbor (having pos-1 position) in a `data(...)` message.

$$\mathsf{send}_{gene(\mathsf{pos}-1)}(\mathtt{data}(i, \Lambda))$$

`data(`k`,`$Neigh\Lambda$`)`

Data is available from the right neighbor (having pos+1 position). The gene calculates and updates the right affinity by calling a user defined aff function and sends back the calculated value to its neighbor.

$$\mathsf{RightAff} \leftarrow aff(\Lambda, Neigh\Lambda)$$
$$\mathsf{send}_{gene(\mathsf{pos}+1)}(\mathtt{setaff}(i, \mathsf{RightAff}))$$

`setaff(`k`,`v`)`

Left neighbor is sending the calculated affinity. Just set the LeftAff to the value sent.

$$\mathsf{LeftAff} \leftarrow v$$

`moveme(`k`,currpos,topos)`

This messages is received by all processes via the broadcast channel since order change of a process may affect positions of all genes. If a move is requested from a left position to another left position or a right position to another position, message is just ignored. If a left to right move is requested, pos attribute is decremented and if a right to left move is requested, pos attribute is incremented. After this change the gene process listens to the channel of its new position. The gene requesting the move, simply updates its position.

$$\mathsf{pos} \leftarrow \begin{cases} \mathsf{topos} & \text{if } \mathsf{currpos} = \mathsf{pos} \\ \mathsf{pos} - 1 & \text{if } \mathsf{currpos} < \mathsf{pos} \text{ and } \mathsf{topos} > \mathsf{pos} \\ \mathsf{pos} + 1 & \text{if } \mathsf{currpos} > \mathsf{pos} \text{ and } \mathsf{topos} < \mathsf{pos} \\ \mathsf{pos} & \text{otherwise} \end{cases}$$

As a result, in a right to left jump all genes having smaller pos from topost and all genes having greater pos from currpos will retain their positions but the other genes in the middle will be shifted right. Similarly, in a left to right jump genes in the middle will be shifted left.

Only sendtoleft(...) is initiated by the control process. data(...) and setaff(...) is generated among the gene processes as a consequence of this message. moveme(...) on the other hand is a probabilistically generated message. After a gene process completes a select(...) request, it probabilistically (with a probability of $\frac{1}{t}$ for an expected period of t (generations) makes a movement decision and sends a moveme(...) message to the broadcast channel.

This decision is based on its left and right neighbor affinities and a threshold value τ, of them. If gene process have good affinities with both neighbors it will not send any message and retain its existing position. If left neighbor is good in affinity but right is below the threshold, it sends a message to move itself to the left across the left neighbor. Similarly if right is good but left is below the threshold, it sends a message to move itself to the right across the right neighbor. Otherwise, when both affinities are below the threshold, it sends a message to move itself to a random location. This can be formalized as:

```
if  LeftAff > $\tau$ and  RightAff > $\tau$
            Do nothing
else if  LeftAff > $\tau$ and  RightAff < $\tau$
            send$_{broadcast}$(moveme($i$, pos, pos-1)
else if  LeftAff < $\tau$ and  RightAff > $\tau$
            send$_{broadcast}$(moveme($i$, pos, pos+1)
else
            send$_{broadcast}$(moveme($i$, pos,$random(1,N)$))
end
```

After introducing these messages into the gene processes, the only additional update required for the control process is to have a sendtoleft(...) message to initiate the affinity calculation in the main loop.

Although moves to left and right of the neighbors can be done in local channels, there is a possibility of having two move requests to interfere with each other. For example, a gene requesting a move to its left, possibly before its own request is completed, will receive a move request from its right neighbor. To resolve this, moveme(...) requests are sent over the broadcast channel so there will be no such inconsistencies. Because this channel is common to all genes and is inherently order preserving.

5.5 Implementation

Simple GA and gene reordering GA are implemented in a concurrent programming environment *Mozart* [37]. Mozart is an environment based on the concurrent constraint programming language *Oz*. *Oz* provides logic variables, functional, concurrent and some object oriented features.

Interprocess communication is provided by means of Port typed object. On the receiving end, a stream is instantiated as the data is received.

Each process is implemented as an instance of a class. Attributes are mutable cells that can be inspected and updated. Actions are implemented as methods. Since method names are first order values, method calls can be sent along the channels as named tuples.

Concurrency in the *Mozart* can work both in a single machine multi threaded environment and can be distributed on different hosts or processes connected on a TCP/IP network. It implements shared logical variable semantic also in a distributed environment.

The choice of language *Oz* is due to ease of development. However, any MIMD parallel environment that provides message passing can be used to implement the model as long as it fulfills the assumptions that are defined in the beginning of the chapter: asynchronous creation and execution of processes (or threads), order preserving and lossless channels, native or indirect implementation of broadcast or multicast messages is possible, and the primitives for synchronization. Basic concern here is the availability of native broadcast or multicast message passing to accelerate the transmission of the control messages. Otherwise communication cost of genetic operators will be much significant in the total cost.

5.6 Other Possible Architectures

Gene reordering GA is not the only possible architecture that can be implemented in the gene level concurrency model. Since the model eliminates the requirement for

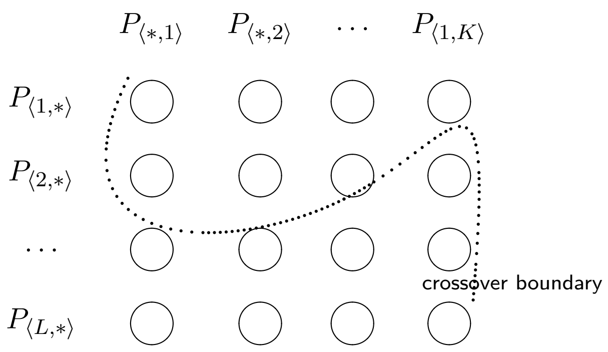


Figure 5.3: A possible two-dimensional chromosome representation.

a linear fixed order, any non-linear and/or variable topology can be implemented by using it.

For example, a matrix notation can be suitable to represent 2-D spatial chromosome encodings (Figure 5.3). Image processing is a good example for such an application area. The only change required is to have a tuple for the pos attribute. The crossover point is going to be any shape in the 2-D representation. This shape will separate the chromosome into two partitions. Thus, the crossover will decide about a particular gene of a chromosome, considering which partition it belongs to.

So, resulting gene process tuple will be:

$$P_i = \langle \langle i_x, i_y \rangle, \langle \mathsf{pos}_x, \mathsf{pos}_y \rangle, \Lambda, new\Lambda, \mathsf{Broad}, \mathsf{Inport}, \overline{\mathsf{Outports}}, \mathsf{Actions} \rangle$$

Any shape description which divides the matrix into two partitions is suitable. It can be a line, a polyline, a bezier curve or a polygon, as long as given the shape and the $\langle \mathsf{pos}_x, \mathsf{pos}_y \rangle$ attribute the partition of the gene can be decided. Thus, a if a function returning a binary choice $partition : Shape \times Position \rightarrow \{0, 1\}$ can be defined, than the crossover based on this function can be made.

For example, if the shape is a line passing from two points $\langle a_x, a_y \rangle$ and $\langle b_x, b_y \rangle$ the *partition* function can be defined as:

$$partition(\langle a_x, a_y \rangle, \langle \mathsf{pos}_x, \mathsf{pos}_y \rangle) = \begin{cases} \text{if } \mathsf{pos}_y < \dfrac{b_y - a_y}{b_x - a_x}(\mathsf{pos}_x - a_x) + a_y \ , & 0 \\ \text{otherwise} \ , & 1 \end{cases}$$

Another possible extension of the method is the hierarchical chromosome representation shown in (Figure 5.4). Instead of a single level orientation of gene processes an intermediate level of processes are inserted between the control process and the gene processes. These processes capture the incoming control messages and generate

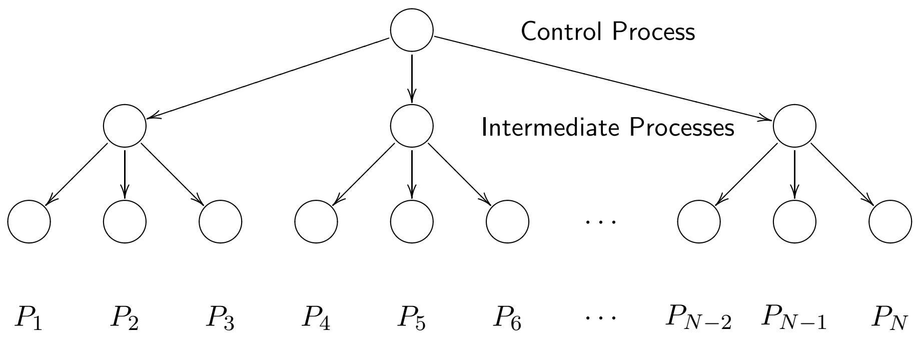


Figure 5.4: A possible hierarchical chromosome representation.

modified messages through their child processes. This allows different level of genetic operator interactions possible.

One possible scenario of this implementation is the hierarchical representation of building blocks. Assume sibling processes in the leaf level are building blocks and intermediate level processes filter incoming crossover messages so that building blocks are preserved and mixed properly. In such a configuration linkage learning problem reduces to the learning of this tree structure.

CHAPTER 6

SUMMARY AND CONCLUSIONS

6.1 Summary

In the first part of this study, a method for improving the performance of genetic algorithm is introduced. The key point in the performance improvement is to favor building block growth and mixing by learning a suitable ordering adaptively during the evolution of the genetic algorithm. Since

- there is no general way to priorly judge about the quality of an ordering, and
- the performance of the encoding totally depends on the subject problem,

the only way of adaptation is to collect information from the state of the pool, during the run, while using this information to introduce better orderings on the fly. This process, which is called linkage learning, has the primary goal to shorten the defining lengths of the building blocks.

In order to achieve this, relations among pairs of gene positions, based on the valuations of them, are tried to be captured. The introduced term *affinity* describes a measure of this relation according to some collected statistics of gene value pairs of the best fit individuals of the population. By using the affinity measure, a global permutation, affecting especially the crossover operations, is updated to maximize the overall affinity among the neighboring pairs.

Two different approaches are proposed to achieve this:

1. Gene reordering is based on *neighbor* pair affinities by performing local updates

on the permutations.

2. Gene reordering is based on a greedy algorithm which seeks a maximized affinity sequence considering *all* pairs. This approximates to maximization of the overall affinity.

Proposed methods are tested first in a minimum area rectangle placement problem where the input rectangles are placed on a two dimensional grid and the minimum area bounding box containing all non-intersecting rectangles are sought. A significant decrease in the number of generations required for an acceptable local optimum is observed.

Secondly, methods are tested on the generalized assignment problem where a group of jobs are assign to a group of agents with each jobs is assigned to exactly one agent end there are costs and constraints of assigning a job. Methods converged faster in the proposed method.

In the third test, methods are experimented on two test problems. One of the problems is linearly decomposable and the other is not. Similarly proposed methods converged faster.

In the forth experiment, the relation of affinity and the linkage is tested on an artificially created non-binary deceptive problem of order 2 and 3. While performance is similarly better a positive linkage change is observed. The greedy maximum sequence method is more successful than the neighborhood affinity version as the order of the problem increases.

In the second part of the thesis, a concurrent model of genetic algorithms is proposed. In contrast to the existing parallelization studies, our model divides the population into gene processes and each gene evolves its own part of the population receiving control commands from a control process. After the genetic operators are applied, a selection process takes the data, and selects the individuals which will reproduce in the next generation.

The most important advantage of this model is that it is independent of the ordering and orientation. Each gene works in its own space without caring in which part of the chromosome it is. This enables the implementation of varying orders and orientations without changing much of the gene code.

By using this property, the gene reordering method is implemented on this architecture with a small number of additions. Similarly, two other architectures, mesh and tree topologies, are proposed.

6.2 Conclusions

In the experiments the cases for the

- ideal permutation,
- a random fixed permutation and
- the proposed methods

are executed many times and results are collected. Experimental results show that the proposed methods, gene reordering by using neighbor affinity and greedy maximum affinity improves the online performance of the genetic algorithm, namely how fast the genetic algorithm explores the search space and finds a solution, significantly. While doing this there is no loss of offline performance, i.e.. the quality of the final solution. The results indicate that *the methods accelerate the search procedure for different problem cases.*

Experimented problems had different characteristics. The rectangle placement problem is a real optimization problem with high epistasis since the intersection of the rectangles decreases the fitness drastically, so even if a good placement is found for one rectangle, a single bad placement cause a very low fitness. *In this problem, though only a local optimum could be found, the neighbor affinity gene reordering outperformed the simple genetic algorithm in both speed and the quality.*

In one of the test problems, the problem was linearly decomposable. That means the problem could be divided into order 1 subproblems and could be solved independently. As expected normal and permuted cases had the same performance since they had both fixed orderings and in order 1 blocks the ordering was not important. However, the proposed methods were still better than the fixed order methods. This shows that *the methods work also for order 1 building building blocks*. That could mean that the exploration abilities of the methods does not depend on the explicit existence of building blocks which is enforced by the fitness function itself. The method is still powerful when the building blocks are hidden in divide-and-conquer type problems.

In the second test problem, an order 2 chain of functions were used so the problem was not linearly decomposable. Similar results were collected. However, in this problem it was possible to observe the linkage. Linkage in this case is defined by the order 2 average proximity of the gene positions. Proposed method lowered the linkage some, relative to the average, but not a drastic amount.

In order to observe linkage changes better, another artificial problem set is constructed. However this time a deceptive problem in which smaller order building blocks were leading to local minima is used. With the order 2 deceptive function both proposed methods successfully found the global optimum. Greedy maximum affinity sequence version exhibit a significant linkage drop in a short time, and then the linkage remained constant. On the other hand, the neighbor version slowly took the linkage down and was not as successful as the greedy one.

When the order increases the difference between the performance of two proposed methods became more clear. Greedy maximum affinity method was still competitive, although the complexity of the problem had increased exponentially and even the ideal case (permutation) was far from the catching the global optimum. Also a positive linkage shift has been observed. On the other hand the neighbor affinity method lost some of its performance and only was better than the fixed permutation case.

As a result, we can conclude that the *greedy maximum affinity method continues its performance for higher orders, where the neighbor affinity method cannot handle higher orders well.*

Comparing the method introduced with other linkage learning methods:

- Proposed methods are quite easy to implement. With a small amount of changes any simple genetic algorithm implementation can be converted into a gene reordering genetic algorithm.
- They are general in the sense that in any domain where a plausible affinity function definition can be made, they can be useful. There is no arbitrary restrictions on the alphabet and domain.
- Their complexity is not so significant in the genetic algorithm. Neighbor affinity method has the same complexity with the fitness evaluation. Greedy maximum affinity has an $\mathcal{O}(Mn^2)$ instead of the $\mathcal{O}(Mn)$ where M is proportional to the population size Q. However since the constant factors are very small (i.e. in

orders of magnitude 1/100), the overall complexity difference may not be crucial in some of the problems. Especially fitness calculations with many floating point arithmetic or $\mathcal{O}(n^2)$ or higher complexity, reordering cost will be relatively small.

In the second part of the thesis a new concurrent model, *gene level concurrency model* have been introduced. The model proposes a completely different approach than the current parallel genetic algorithm practices. The purpose of the model is not to achieve a high speed-up and efficiency but to study gene interaction in an new way while keeping advantages of the parallelism as it is available also in nature.

Model decomposes the population into vertical entities of gene processes where all pool data of a gene position is stored and processed on this process. This helped decomposing the problem into independently and concurrency working genes. Concurrency of genes is also the way nature functions.

The model based on a simple genetic algorithm is explained and is implementation on a concurrent programming language *Oz* has been done.

The model is implementable in any concurrent environment providing asynchronous processes and/or threads as long as the interprocess communication primitives and order preserving and lossless channels are available. Also primitives for the synchronization is required in several places. Although it is possible to implement the model in any concurrent architecture with properties mentioned above, the model uses broadcast message passing and extensive communications. *In order to take the advantage of parallelism, architectures natively supporting the broadcast communication and fast communication are required.* Therefore, currently the most suitable architectures for the model are the shared memory systems.

The flexible ordering and orientation provided by the model is than used to implement neighbor gene reordering method described in Chapter 4. Since the method is local, learning of the linkage is implemented via local decisions and interactions of the gene processes. *The extension of the method requires the introduction of only a few messages and actions.* This shows the flexibility of the model.

This flexibility is used to propose two new possible architectures, matrix and hierarchical structure of the genes. With a small number of extensions, this structures can also be implemented by the model.

6.3 Future Work

Although two basic parts in the thesis are connected in an implementation, they may lead to distinct paths for a future research.

- Affinity based reordering has been proven to work in a group of problems. However its performance in different genetic algorithm approaches and parameters could be a promising direction for the future studies. The interaction of different affinity functions with a group of problems can also be investigated in the future.
- Although affinities of gene pairs are handled in two different ways in this study, these are not the only ways possible. Affinity, the statistical analysis of gene vector relations, can be adapted in the other linkage learning algorithms or new affinity based methods can be introduced. Furthermore, the affinity concept can be generalized to n-groupings of genes.
- The gene level concurrency model, on the other hand, promises a flexible architecture instead of a performance seeking parallelism. However the amount of speed-up and parallelism achievable on an ideal parallel architecture can be studied.
- The flexibility of the model can be used to exploit different orientations and adaptive structures of linkage learning. Different encodings turn out to be different topologies in the gene process model. Similarly the linkage learning problem turns out to be the learning of this topology.
- It may also be desirable to observe further finer grains of parallelism by hybrid models like islands of gene per processes or gene per process.

REFERENCES

[1] D. Abramson. Constructing school timetables using simulated annealing: Sequential and parallel algorithms. *Management Science*, 37, 1:98–113, 1991.

[2] D. Abramson and J. Abela. A Parallel Genetic Algorithm for Solving the School Timetabling Problem. In Hobart, editor, *Proceedings of the 15. Australian Computer Science Conference*, February 1992.

[3] Edward J. Anderson and Michael C. Ferris. A genetic algorithm for the assembly line balancing problem. Technical Report CS-TR-1990-926, University of Wisconsin, Madison, March 1990.

[4] David Andre and John R. Koza. Parallel genetic programming on a network of transputers. In Justinian P. Rosca, editor, *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pages 111–120, Tahoe City, California, USA, 9 July 1995.

[5] T. Bäck, F. Hoffmeister, and H.-P. Scwefel. A survey on evolution strategies. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 2–9. Morgan Kaufman, 1991.

[6] Thomas Bäck. *Evolutionary Algorithms in Theory and Practice.* Oxford University Press, 1996.

[7] J.D. Bagley. *The Behavior of Adaptive Systems Which Emply Genetic and Correlation Algorithms.* PhD thesis, University of Michigan, 1967.

[8] David Beasley, David R. Bull, and Ralph R. Martin. An overview of genetic algorithms: Part 1, fundamentals. *University Computing*, 15(2):58–69, 1993.

[9] David Beasley, David R. Bull, and Ralph R. Martin. An overview of genetic algorithms: Part 2, research topics. *University Computing*, 15(4):170–181, 1993.

[10] J.E. Beasley. OR-Library: distribution test problems by electronic mail. *Journal of the Operational Research Society*, 41:1069–1072, 1990.

[11] T. Belding. The distributed genetic algorithm revisited. In D. Eshelmann, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*, San Moteo, CA, 1995. Morhan Kaufmann.

[12] R. Bianchini and C.M. Brown. Parallel genetic algorithms on distributed-memory architectures. In *Transputer Research and Applications 6*, pages 67–82. IOS Press, Amsterdam, 1994.

[13] H. Braun. On solving travelling salesman problems by genetic algorithms. In H.-P. Schwefel and R. Männer, editors, *Parallel Problem Solving from Nature - Proceedings of 1st Workshop, PPSN 1*, volume 496 of *Lecture Notes in Computer Science*, pages 129–133, Dortmund, Germany, 1-3 October 1991. Springer-Verlag, Berlin, Germany.

[14] E. Cantú-Paz and M. Mej'ia-Olvera. Experimental results in distributed genetic algorithms. In *International Symposium on Applied Corporate Computing*, pages 99–108, Monterrey, Mexico, 1994.

[15] Eric Cantú-Paz. A summary of research on parallel genetic algorithms. Technical Report 95007, IlliGAL, July 1995.

[16] D. Catrysse and L.N Van Wassenhove. A survey of algorithms for the generalized assignment problem. *European Journal of Operational Research*, 60:260–272, 1992.

[17] P.C. Chu and J.E. Beasley. A genetic algorithm for the generalized assignment problem. *Computers and Operations Research*, 24:17–23, 1997.

[18] J. Cohoon, S. Hegde, W. Martin, and D. Richards. Punctuated equilibria: a parallel genetic algorithm. In J.J. Grefenstette, editor, *Proceedings of the Second International Conference on Genetic Algorithms.* Lawrence Erlbaum Associates, 1987.

[19] J. Cohoon, W. Martin, and D. Richard. A multi-population genetic algorithm for solving the k-partition problem on hyper-cubes. In R.K. Belew and L.B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms,*, San Moteo, CA, 1991. Morgan Kaufmann.

[20] J. Cohoon, W. Martin, and D. Richards. Genetic algorithms and punctuated equilibria in vlsi. In H.P. Schwefel and R. Männer, editors, *Parallel Problem Solving from Nature - PPSN I*, volume 496 of Lecture Notes in Computer Science, pages 133–144. Springer-Verlag, Berlin, Germany, 1991.

[21] Yuval Davidor. A naturally occurring niche and species phenomenon: The model and first results. In Lashon B. Belew, Richard K.; Booker, editor, *Proceedings of the 4th International Conference on Genetic Algorithms*, pages 257–263, San Diego, CA, July 1991. Morgan Kaufmann.

[22] K. Deb and D.E. Goldberg. Analyzing deception in trap functions. In K.D. Whitley, editor, *Foundations of Genetic Algorithms 2*, pages 93–108, San Mateo, CA, 1993. Morgan Kaufmann.

[23] Kalyanmoy Deb. *Binary and Floating-point Function Optimization using Messy Genetic Algorithms.* PhD thesis, University of Alabama, 1991.

[24] T. C. Fogarty and R. Huang. Implementing the genetic algorithm on Transputer based parallel processing systems. *Lecture Notes in Computer Science*, 496:145, 1991.

[25] D. E. Goldberg, H. Kargupta, J. Horn, and E. Cantu-Paz. Critical deme size for serial and parallel genetic algorithms. IlliGAL Report 95002, University of Illinois, Urbana-Champaign, 1995.

[26] David E. Goldberg, Kalyanmoy Deb, and James H. Clark. Genetic algorithms, noise, and the sizing of populations. *Complex Systems*, 6(4):333–362, August 1992.

[27] D.E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley, 1989.

[28] D.E. Goldberg. Messy genetic algorithms: Motivation, analysis, and first results. *Complex Systems*, 3:493–530, 1989.

[29] D.E. Goldberg, K. Deb, and J.H. Clark. Genetic algorithms, noise, and the sizing of populations. Technical Report IlliGAL Report No. 91010, University of Illinois at Urbana-Champaign, Illinois Genetic Algorithms Laboratory, 1991.

[30] D.E. Goldberg, K. Deb, and J Horn. Massive multimodality, deception, and genetic algorithms. Technical Report IlliGAL Report No. 91010, University of Illinois at Urbana-Champaign, Illinois Genetic Algorithms Laboratory, April 1992.

[31] D.E. Goldberg, K. Deb, H. Kargupta, and G. Harik. Rapid, accurate optimization of difficult problems using fast messy genetic algorithms. Technical Report IlliGAL Report No. 93004, University of Illinois at Urbana-Champaing Illinois Genetic Algorithms Laboratory, February 1993.

[32] D.E. Goldberg, K. Deb, and D. Thierens. Toward a better understanding of mixing in genetic algorithms. Technical Report IlliGAL Report No. 92009, University of Illinois at Urbana-Champaign, Illinois Genetic Algorithms Laboratory, 1992.

[33] D.E. Goldberg and R.Jr. Lingle. Alleles, loci and the traveling salesman problem. In *Proceedings of the an International conference on Genetic Algorithms and Their Applications*, pages 154–159, 1985.

[34] Martina Gorges-Schleuter. ASPARAGOS an asynchronous parallel genetic optimization strategy. In J. David Schaffer, editor, *Proceedings of the 3rd International Conference on Genetic Algorithms*, pages 422–427, George Mason University, June 1989. Morgan Kaufmann.

[35] P. Grosso. *Computer Simulations of Genetic Adaptation: Parallel Subcomponent Interaction in a Multilocus Model.* PhD thesis, University of Michigan, 1985.

[36] F. Gruau. *Neural Network Synthesis using Cellular Encoding and the Genetic Algorithm.* PhD thesis, Laboratoire de l'Informatique du Parallilisme, Ecole Normale Supirieure de Lyon, France, 1994.

[37] Seif Haridi and Nils Franzn. *Tutorial of Oz.* DFKI, draft edition, Feb 1999. `http://www.mozart-oz.org/documentation`.

[38] G.R. Harik. *Learning Gene Linkage to Efficiently Solve Problems of Bounded Difficulty Using Genetic Algorithms.* PhD thesis, University of Michigan, 1997.

[39] R. Hauser and Reinhard Männer. Implementation of standard genetic algorithm on MIMD machines. In Yuval Davidor, Hans-Paul Schwefel, and Reinhard Männer, editors, *Parallel Problem Solving from Nature – PPSN III*, pages 504–513, Berlin, 1994. Springer. Lecture Notes in Computer Science 866.

[40] J.H. Holland. *Adaptation in Natural and Artificial Systems.* MIT Press, 1975.

[41] J. Maresky. On efficient communication in distributed genetic algorithms. Master's thesis, Institute of Computer Science, Hebrew University of Jerusalem, 1994.

[42] F. Marin, O. Trelles-Salazar, and F. Sandoval. Genetic algorithms on LAN-message passing architectures using PVM: application to the routing problem. In Y. Davidor, H.P. Schwefel, and R. Männer, editors, *Parallel Problem Solving from Nature - PPSN III*, pages 534–543. Springer-Verlag, Berlin, Germany, 1994.

[43] Laurence D. Merkle and Gary B. Lamont. Comparison of parallel messy genetic algorithm data distribution strategies. In Stephanie Forrest, editor, *Proceedings of the 5th International Conference on Genetic Algorithms*, pages 191–198, San Mateo, CA, USA, July 1993. Morgan Kaufmann.

[44] Zbigniew Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs.* Springer, third edition, 1992.

[45] Melanie Mitchell. *An Introduction to Genetic Algorithms.* MIT press, 1996.

[46] H. Mühlenbein and G. Paaß. From recombination of the genes to the estimation of distributions i. binary parameters. *Parallel Problem Solving from Nature, PPSN IV*, pages 178–187, 1996.

[47] M. Munetomo, Y. Takai, and Y. Sato. An efficient migration scheme for subpopulation based asynchronously parallel genetic algoritms. In S. Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, page 649, San Mateo, CA, 1993. Morgan Kaufmann.

[48] Jan Paredis. The symbiotic evolution of solutions and their representation. In L. Eshelman, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms (ICGA 95)*, pages 359–365. Morgan Kaufmann, 1995.

[49] J. Pearl. *Probabilistic Reasoning in the Intelligent Systems.* Morgan Kaufmann, California, 1988.

[50] M. Pelikan, D.E. Goldberg, and E. Cantú-Paz. Linkage problem, distribution estimation and bayesian netowkrs. Technical Report IlliGAL Report No. 98013, University of Illinois at Urbana-Champaign, Illinois Genetic Algorithms Laboratory, November 1998.

[51] Martin Pelikan and David E. Goldberg. Escaping hierarchical traps with competent genetic algorithms. In L. Spector, E.D. Goodman, and A. Wu et.al., editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 511–518, San Francisco, California, USA, 7-11 July 2001. Morgan Kaufmann.

[52] Martin Pelikan, David E. Goldberg, and Kumara Sastry. Bayesian optimization algorithm, decision graphs, and occam's razor. In L. Spector, E.D. Goodman, and A. Wu et.al., editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 519–526, San Francisco, California, USA, 7-11 July 2001. Morgan Kaufmann.

[53] C. Pettey and M. Leuze. A theoretical investigation of a parallel genetic algorithm. In J.D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*. Morhan Kaufmann, 1989.

[54] C. Pettey, M. Leuze, and J. Grefenstette. A parallel genetic algorithm. In J. Grefenstette, editor, *Proceedings of the Second International Conference on Genetic Algorithms*. Lawrance Erlbaum Associates, 1987.

[55] George G. Robertson. Parallel Implementation of Genetic Algorithms in a Classifier System. In John J. Grefenstette, editor, *Proceedings of the 2nd International Conference on Genetic Algorithms (ICGA87)*, pages 140–147, Cambridge, MA, July 1987. Lawrence Erlbaum Associates. Also Technical Report TR-159 RL87-5 Thinking Machines Corporation.

[56] M. Schwehm. Implementation of genetic algorithms on various interconnection networks. In M. Valero, E. Onate, M. Jane, J. L. Larriba, and B. Suarez, editors, *Parallel Computing and Transputer Applications*, pages 195–203, Amsterdam, 1992. IOS Press.

[57] Onur Tolga Sehitoglu. A concurrent constraint programming approach to genetic algorithms. In Conor Ryan, editor, *Graduate Student Workshop*, pages 445–448, San Francisco, California, USA, 7 July 2001.

[58] Onur Tolga Sehitoglu and Gokturk Ucoluk. A building block favoring reordering method for gene positions in genetic algorithms. In L. Spector, E.D. Goodman, and A. Wu et.al., editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 571–575, San Francisco, California, USA, 7-11 July 2001. Morgan Kaufmann.

[59] B. A. Shapiro and J. Navetta. A massively parallel genetic algorithm for RNA secondary structure prediction. *The Journal of Supercomputing*, 8(3):195–207, November 1994.

[60] Piet Spiessens and Bernard Manderick. A massively parallel genetic algorithm: Implementation and first analysis. In Rick Belew and Lashon Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 279–286, San Mateo, CA, 1991. Morgan Kaufman.

[61] T. Starkweather, D. Whitley, and K. Mathias. Optimization using distributed genetic algorithms. In H. P. Schwefel and R. Männer, editors, *Parallel Problem Solving from Nature - Proceedings of 1st Workshop, PPSN 1*, volume 496 of *Lecture Notes in Computer Science*, pages 176–185, Dortmund, Germany, 1-3 October 1991. Springer-Verlag, Berlin, Germany.

[62] R. Tanese. Parallel genetic algorithms for a hypercube. In J. Grefenstette, editor, *Proceedings of the Second International Conference on Genetic Algorithms*. Lawrence Eribaum Associates, 1987.

[63] R. Tanese. Distributed genetic algorithms. In J.D.Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*. Morgan Kaufmann, 1989.

[64] R. Tanese. *Distributed Genetic Algorithms for Function Optimization*. PhD thesis, University of Michigan., 1989.

[65] D. Whitley. The genitor algorithm and selection pressure: Why rank-based allocation of reproductive trails is best. In *Proceedings of the Third Internetional Conference on Genetic Algorithms*. Morgan Kaufmann, 1989.

[66] Darrel Whitley. An introduction to genetic algorithms, 2001 genetic and evolutionary computation conference tutorial. San Francisco, CA, July 2001.

VITA

Onur Tolga Şehitoğlu was born in Burdur on August 3, 1971. He received his B.S. degree in Computer Engineering from the Middle East Technical University in July 1992. He received his M.S. degree from the same institute in January 1996. He worked in the Computer Engineering Department of the Middle East Technical University as a computer systems expert between 1992 and 1999. Since 1999 he has been working as a lecturer in the same department.