Development of an Experimental System for the use of Genetic Algorithms

Andreas Grimm

Dipl.-Inform., B.Sc.(Hons)

December 17, 2015

Abstract

The following report is a reviewed verison of a report which I wrote in 1992 as a Project Report for the University of Sunderland. It consists of five parts. The first part is a short introduction into the area of genetic algorithms. It shows the fundamentals of the theory of genetic algorithms and explains most of the terminology. Also a short example is given, so that the elementary work of the genetic algorithms is explained. The second chapter shows how the problem of the first chapter can be programmed. The third chapter expands the theory and try to explain certain mechanisms which are developed in the last years. An example of these advanced mechanisms is shown in the fourth chapter. The last part is an introduction into the area of two dimensional genetic algorithms and a short discussion of the possibilities for the use of these.

The appendix gives a documentation for the used packages and introduces an optional random number generator, which could be used independently from this work.

Contents

0.1 Introduction

0.1.1 Objectives

The following objectives should be solved in my final year project:

1. Constructing an experimental environment for users of Genetic Algorithms.
   Learning the use of Genetic Algorithms is very complicated for newcomers in this area. Aim in this project is the construction of a number of libraries which will help a potential user to construct genetic systems.
2. The environment should be programmed in Ada.
   Basic idea is that the programming should be in an object orientated language which could be used by a greater number of users. The computer language, which is best known by the students and also most portable between different system is Ada. An alternative would be the language C++, but there are not enough platforms availiable.
3. The packages should represent the different levels of Genetic Algorithms. A first package cover simple Genetic Algorithms as described by Goldberg[Gol89]. econd package is an extension of the first also covering the advanced mechanims. A third package is the implementation for two dimensional individuals.
4. The report should help a newcomer to learn the use of Genetic Algorithms. The report has two parts. The first part is a short introduction in the theory of Genetic Algorithms, the second part is a case study on an existing problem.

The packages will be available on three platforms :

- on the Apollo Workstations

- on MS-Dos Computers with an Ada compiler
- on Unix System V/3.2 with Meridian Ada compiler

The report is written with the TEX wordprocessor, using the LATEX macro package. æ

0.2 Acknowledgements

At this part I want to thank my superviser, Dr. Peter Dunne for his support, especially for his help by writting this report.

Chapter 1

The Basics of Genetic Algorithms

The idea of Genetic Algorithms is not a very new one, it was developed over the last twenty years. But it was kept in darkness for a long time and became popular with the upcoming of the *Neuronal Networks*. In this chapter I want to declare the main principles of the theory of Genetic Algorithms. To understand every detail it would be the best to read the book of GOLDBERG [Gol89]. Some more experiences can be found in DAVIS[Dav91] and in the report of PROF. JOCHUM at COLOGNE POLYTECHNIC[Joc91].
In this paper I use different fonts for special purposes.

- The 'small capital'- font is used for names of persons or institutes, e.g. GOLDBERG.
- The 'italic'- font is used for expression declared in a different part of the report, e.g. *reproduction*.

- The 'typewriter' - font is used for programs and computer outputs, e.g. `procedure Text is`.
- References to the bibliography are enclosed in '[' and ']', e.g. [Joc91].

1.1 What are *Genetic Algorithms* ?

Genetic Algorithms are a very unknown way in the development of programming complex processes. As can be seen by the name two very different sciences have influence in this area

- the biology with the knowledge on evolutional processes, and
- the computer science with the knowledge on algorithms and the development of algorithms on processes and natural environments. Development in this case means the formulation of the problems in procedings and data[Wir85].

The Science of the evolution and of the reproduction and selection of the best adapted lifeforms is known as genetic. So subject of the genetic algorithms are procedures with the property to change themselfs to fit into the environment. In this context we speak about changes of a specimen living in a form of environment with almost dangerous or unhealthy attributes. In real life it is impossible for a specimen to survive without this way of changing, therefor nature developed in millions of year mechanisms to optimize lifeforms, so that they fit into their environment. These mechanisms are in the detail very complicate, but it is possible to simplify them by abstraction so that they can be used in computer science. J.Holland developed this direction of computer science by trying to adapt natural systems by the help of the computer. For him it was most important to develop systems which work alike the processes of the nature[Gol89].

1.1.1 A First Example

The following very nice example is given by A.K.Dewdney:

> Imagine an abstract sea inhabited by abstract organisms called finite living blobs, or flibs. Each flib is equipped with the simplest decisionmaking apparatus possible. This is the biological equivalent of what computer scientiests call a finite automaton. Each flib also containts a single chromosom consisting of a string of symbols that encodes the automaton. The flibs inhabit a primordial, digital soup in constant flux. These changes must be predicted accurately by the flib if it is to survive.
>
> In the primordial soup I recently set simmering in my computer, flibs that predicted poorly died out. The best predicted left progeny that sometimes improved on ancestral performance. Eventually a line of perfect predictors evolved.[Dew85]

In this example are all the very important properties of the genetic algorithms:

1. The individual, called the flip, which tries to adapted into the environment over generations of time
2. The environment or the civilisation pressure, which is the messure of of the fitness and of the ability to survive. In this example the civilisation pressure is the ability to make prognoses.

It is now time to come a little closer to the subject. A very special interest of this first chapter is the summary of areas where genetic algorithms are used and to discribe the method of genetic algorithms in general.

1.1.2 Where are Genetic Algorithms useful ?

Genetic Algorithms were meanly in use in the area of optimization. Here it is not important which kind of optimization, numerical or non-numerical optimization should been solved.

Many of the problems of numerical optimization can be solved by using programs of the area of numerical mathematics. As an example there is the algorithm of regula falsi and the algorithms of Newton[Pre89].

By comparing one conventional algorithm (like given above) with a genetic algorithm it first looks like there is really no reason to use a new method like this one. Algorithms of the modern numerics are very efficent and, because they are specialized they are in many cases quicker then the genetic algorithm. The results are often better, which means that they are exacter. But they have no robustness as it is called by Goldberg[Gol89].

Robustness is the ability to produce good results even when the environment is not made for this algorithm. What I call the environment is the problem, which should be solved by the algorithm. A bad environment has areas which do not fit to the rules given by the algorithm. This could be i.e. a point of discontinuity or more then one extremals, which are common in the area of numerical mathematics.

As a example I want to use the algorithm of Newton to find the maximum of the function $f(x) = -x^2$, which is now the environment of the algorithm. This environment can be called positive, it has exactly one extremal (the point 0) and by using the Newton algorithms we find this extremal very quick.

But if this algorithms is started on a environment which can be called malicious it will

- not terminate, or

- not find a greater number of the solutions

A very good example is the sinus-function. This function has an infinity number of extremals and leads into a errorous situation for sure, because the number of unfound extremals is also infinity.

Genetic Algorithms will not find all solutions to, but the probability to find more the one solution is proportial to the number of the individuals in the search.

The following example is given by Goldberg[Gol89]. Imagine a machine with five switches, each switch gives the values 0 or 1. Depending on the positions of the machine's switches the user will win a certain reward. So we are looking for the position of the switches with the highest reward.

A normal solution of this problem would test all combination of the switches and remember all the results (or would remember the result with the highest reward). So there would be $2^5 = 32$ iterations of the program. As can be seen later a genetic algorithms normally uses much less iterations. I will keep this example in thenext chapters to descripe the fundamentals of the genetic algorithms.

To characterise genetic algorithms in general:

1. Genetic algorithms work with a code on a set of parameters, not with the parameters.

2. Genetic algorithms are searching with a set, a population and not with a single point.

3. Genetic algorithms check the results and do not use information from outside the system.

4. Genetic algorithms calculate the next points of search by the methods of probability theory, not by deterministic rules.

1.2 A Closer Look on Genetic Algorithms

For the understanding of the next parts of this report it is necessary to introduce a special terminology. This is most important even to keep the distance between the real world and the world build up in the computer.

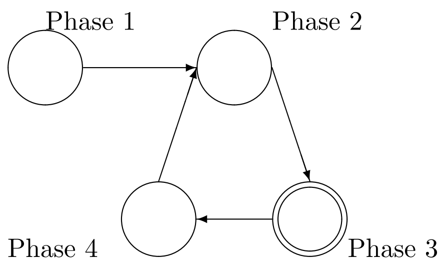
Individual An Individual $\mathcal{I}$ is an element with at least one gene. This individual is definedand gets all it special properties by the gene. As a difference the biological term in this paper is individual.

Gene is a chain of symbols, most simply consisting on the symbols 0 and 1. For the change of the chain there are three more, later defined operations: the reproduction, the crossover and the mutation. In some books the name chromosome is used, too. The number of genes in an individual is defined by the variable lchrom.

Population is the set of the existing individuals. If this number is to small the species will disappear or the algorithm will not find a good solution in a certain time. If the number is to big the time to calculate a single generation will grow and the system will not work sufficent. The size of the population will be called maxpop.

Fitness is the criterium to decide who good a single individual fits into the environment. The development of a proper fitnessfunction is necessary for the success of the genetic algorithm. Older experiment show that the development of a good fitnessfunction is the most difficult exercise in the area of genetic algorithms.

With is definitions it is possible to have a closer look on the mechanism of genetic algorithms. A genetic algorithm uses four phases, the sequence of the algorithm is shown in the following graph:



Phase 1 : The Start: The individuals are produced by a random number generator. This is called the initialisation of the system

Phase 2 : The Calculation: The individuals pass the function or the environment.

Phase 3 : The Control: Depending on the results of phase 2 the fitness of the individual is checked. This is the normal place for the algorithm to terminate.

Phase 4 : Change of Generations: In this phase new indivduums are developed.

The Start

In this very first phase n individuals I will be created. Every individual x is represented by it's genes, so that

$$x \in \{\mathcal{I} | x_k \in \{0,1\} fork = 1, 2, \ldots, lchrom\}$$

with $lchrom = \|x\|$.
So an individual $x \in \mathcal{I}$ is generated by using a random number generator on every x_k. A special random number generator is defined in the appendix [1].

[1]There is no random number generator defined in Ada !

It is important that the width[2] of this generator is big enough. If it is too small, like in some system developed generators, the system will not work properly.

Also the the sizes lchrom for the individuals and maxpop for the population must be defined. This variables must be changed, so they fit into the problem. Therefor it is necessary to get the size of the search-space, called $\mathcal{D}$. Then the number of genes is given by

$$lchrom = \log_2(\|\mathcal{D}\|)$$

The search-space must be a discret and countable. If it is not we have to find a transformation so that we can work with a discret set of numbers.

There is no direct rule for the size of the population. It depents on the behaviour of the system and should be defined by doing some tests on the computer. The literature presents some formulas to calculate a population size but my experience in early experiments[Gri91a, Gri91b] and also information of other sources [Joc91] show that these formulas should be ignored.

The Calculation

The individual passes the environment, means the system calculates a special result for the specific individual. It is important the this result is not immediatly scored, this should take place in the next phase when all results are calculated.

The environment should be seen as a mathematical function, therefor the result of the calculation is a characterisation of the individual. We have to discuss this part of the cyklus later.

The Control

This phase follows the phase of calculation. It could be connected with the change of generations but it is better to keep the phases seperated. The phase

[2]this is the number of bits used for the generation of random number

of control compares and scores the results of the individuals, so a ranking of the individuals can be made. The result is a direct scale for the fitness for each individual. Therefor the results of the population is added, the sum is defined by

$$F = \sum_{k=1}^{maxpop} f(x_k)$$

where $f(x_k)$ is the result of the individual x_k. By using this value it is possible to calculate the value of fitness for each individual, which is defined by the formula

$$f_p(x) = 100 \cdot \frac{f(x)}{F}$$

Notice that Goldberg[Gol89] uses a different formula, which gives nearly the same results.

Also an interesting value is the difference between the calculated value $f(x)$ and the expected value $\bar{f}$. The expected value is given by the average

$$\bar{f} = \frac{F}{maxpop}$$

With this formula it is possible to define the difference by

$$\sigma(x) = |f(x) - \bar{f}|$$

At this point it is necessary to think about the end of the iteration in the genetic algorithm. As could be seen above, the genetic algorithm runs in a circle which in nature will never stop[3]. This point to stop could be

- a certain value of optimization or
- a certain number of generations

One border should be defined before starting the genetic algorithm. In some cases it is possible to define a certain result as a optimum. Then the reaching

[3]I hope so

of this value should be the criteria to stop the genetic algorithm. If this is not possible a maximum number of generations should be defined. This criteria could be changed depending on the behaviour of the population. If there is no effect in increasing the fitness over a longer time the fitness function should be changed or modified. As a fact it is very difficult to find the right time to stop a genetic algorithm. I will give some hints later in the report.

Change of Generations

The next phase is the change of generations. This phase is the speciality of the genetic algorithms. There are three subphases in the evolution of the individuals:

1. the reproduction of the individual, selecting a number of parents depending on the fitness,
2. the cross-over, which means the exchange of genetic informations between to individuals, and
3. the mutation, a randomized change of genes.

These phases are used under a number of specified rules.

The first phase is the reproduction of the individuals. Imagine a roulette-wheel in the common sense. The control phase produced a fitness $f_p(x)$ value for each individual. This value descripes the fitness of each individual as a part of the population.

To take this value as direct measure for a segment on the wheel it is possible to calculate the size of the segment for each individual by the formula

$$\mathcal{P}(x) = f_p(x) \cdot 3.6$$

Because turning an ideal roulette-wheel is a random experiment, like throwing an ideal dice, the calculation of the wheel's segment shows that the probability to pass the reproduction is proportional to the fitness of the individual. It is important to notice that a good fitness is not a ticket to the next generation, but only increases the changes to get one of the places to crossover and to mutate. Also a individual with a bad fitness can reach the next level, the changes are less. This effect is important if the differences between the individuals are small. Also a bad fitness does not mean that there are no important informations in the genes, they only could be hidden.

The first step in the changing of the generations is to turn the roulette-wheel for each new individual needed. The wheel will stop on one segment which belongs to an old individual. This one will be transfered into the next generation. So in the next generation there could be more the one individual of the same type[4].

The next step is called the cross-over. Here the surving individuals are randomly paired, this process could be called the marriage of the individuals. This pairs will exchange parts of their specific genes. Therefor the system will get a place in this string of genes. The string will be cutted and the parts will be exchanged.

e.g. Given are the following strings:
$A_1 = \texttt{01101010}$
$A_2 = \texttt{11011011}$
The random number generator gives position $p = 5$ for the exchange. So this place is marked:
$A_1 = \texttt{01101}|\texttt{010}$
$A_2 = \texttt{11011}|\texttt{011}$
and after the cross-over:
$A'_1 = \texttt{01101011}$
$A'_2 = \texttt{11011010}$
On this way the information between the generations is exchanged.

[4]with the same genetic structure

The last step of this phase is the mutation. Here the idea is that some genes are changing without a normal reason from one generation to the next. Reasons in nature are i.e. radioactivity, influences of chemestry,.... Putting this into a formula:

$$\forall x \in \mathcal{I} : x_i = \begin{cases} \bar{x_i} & : \quad X = 1 \\ x_i & : \quad X = 0 \end{cases}$$

with are very small probability $P(X = 1)$. In Literature this probability is assumed $\ll 0.01$, in nature this value is much smaller[5]. As a repeation: The effect of a mutation is a effect of nature and happens always with a very small probability. The change of genes in small populations, like here it is necessary to refresh the process of evolution.

By using this rules a number of new individuals is generated and the cycle of calculating, controlling and changing the generation can start again. The special effect of the last phase is, that old information is stored, the system is learning [6] but also new information is generated, so that a probably better fitness[7] should be appear. The following bigger example should illustrate the work of a genetic algorithm.

1.3 Demonstration of a Genetic Algorithm

The following example is given in this form in the book by Goldberg[Gol89] as a very elementary demonstration of genetic algorithms. One complete generation in the lifecycle of a population will be executed and the rules given above will be used.

As an example the function $f(x) = x^2$ is used on the set $\mathcal{D} = [0, \dots, 31]$. The

[5]≈ 0.000001

[6]in biology there is the term of genetic learning

[7]for a single individual or for the whole population

algorithm should find the maximum of this function. This size of the set is 32 elements, with the formula $lchrom = \log_2(\|\mathcal{D}\|)$ we calculate that we need 5 genes in each individual, so $lchrom = 5$.

Simply every gene x_k is significant for one binary digit of the variable x, so that $x = x_k \cdot 2^k$. The fitness function is then defined by

$$f(x) = (\sum_{k=0}^{5} x_k \cdot 2^k)^2$$

Assuming the size of the population *maxpop* is 4, the next step will be the generation of the basic population with the random number generator. This gives the following individuals:

Individual	Genetic Structure
1	01101
2	11000
3	01000
4	10011

As can be seen, the individuals are all different, in the population are no twins. The algorithms will now calculate the result of the function for each of the individuals. After proceding this step there is the following structure:

Individual	Value of x	Value of x^2
1	13	169
2	24	576
3	8	64
4	19	361

The next step is the control phase. Here the system additionaly calculates the values for F, $f_p(x)$, $\bar{f}$ and to find the maximal value in this generation $f_{max}(x)$. The results are :

Parameter	Value
F	1170
$\bar{f}$	293
$f_{max}(x)$	576

The complete tabular looks as follows :

Individual	Genetic Structure	x-Value	$f(x)$	$f_p(x)$	$\sigma(x)$
1	01101	13	169	14	0.58
2	11000	24	576	49	1.97
3	01000	8	64	6	0.22
4	10011	19	361	31	1.23

This result means:
Individual 2 has the best fitness with 49% fitness. On the second and third place are individuals 4 and 1 with 31% and 14% fitness. The last place has individual 3 with 6% fitness. With this results the system enters the next phase, called changing the generations. The first step in this phase is the reproduction. In this as in every following example I use the roulette-wheel selection by Goldberg[Gol89]. The segments on the roulette-wheel are proportional to the percentage of the fitness. In this example four individuals are used, so the wheel must be turned four times. I assume that as a result I get individual 1 once, individual 2 twice and individual 4 once. Individual 3 does not enter the next generation. The new tabular looks like this :

Individual	Genetic Structure
1'	01101
2'	11000
3'	11000
4'	10011

The second step in this phase is the crossover. By using random numbers the individuals are paired and the tabular looks like this:

Individual	Genetic Structure	Partner
1'	01101	2'
2'	11000	1'
3'	11000	4'
4'	10011	3'

Now the system has to calculate the point in the string where the string will be cutted and the information will be exchanged. For the first pair this point is after the fourth gene, for the second pair it is after the second gene. The results of the crossover can be seen at the next tabular:

Individual	Genetic Structure	Partner	Point of Crossover	new Individual
1'	01101	2'	4	01100
2'	11000	1'	4	11001
3'	11000	4'	2	11011
4'	10011	3'	2	10000

The next step would be the mutation. But by using a normal probability of mutation $p_{mut} = 0.001$ the number of mutations will be $\mu = 5 \cdot 4 \cdot 0,001 = 0,02$. So in this generation there is no mutation.

With the next generation I recalculate the function :

Individual	Genetic Structure	Value x	Value x^2
1'	01100	12	144
2'	11001	25	625
3'	11011	27	729
4'	10000	16	256

Comparing the two generations will give :

	Generation 1	Generation 2	Change
Sum of Fitness F	1170	1754	+49,91 %
Average Fitness $\bar{f}$	293	439	+49,82 %
best value x	576	729	+26,56 %

Which shows a improving of fitness in all sectors after the change of generations. There is an increase in the fitness of nearly 50 %.

1.4 The Results of a Genetic Algorithm

The mechanisms introduced so far are known as simple genetic algorithms. They only consist on the three basic mechanism by producing a new generation and no extra mechanisms in checking the fitness. These mechanisms are defined in chapter 3.

There are three indicators which discripe the algorithm's behaviour. These indicators are

1. the best fitness value,
2. the average fitness value, and
3. the worst fitness value.

By ploting the values into a coordinate system generation by generation we get three curves. This curves look simular for all systems working with genetic algorithms and give a lot of information. In this section I will introduce this curves and show which informations could be taken from them.

The Curve of the Best Value

The first curve is the curve of the best values. As can be seen the curve first

Figure 1.1: The Curve of Genetic Algorithms

raises from the origin to a very good value, nearly to the optimum. After this it went back to a value which is good, but with a certain distance to the former optimum. After this it keeps swinging around a later balanced static point which is below the first reached optimum.

The curve displayed has the ideal shape of the curve and in the most experiments the curve will look different. These different shapes are a result of the circumstances the genetic algorithm is used. In the chapter on advanced genetic algorithms are mechanisms to improved the performance of a genetic algorithm. But this tools have risks, too. Doing experiments with genetic algorithms is reading the curve and changing the system parameters.

Therefor I introduce a scheme of three phases. These phases are typical and should be found in any curve produced by a genetic algorithm. A picture of the phases could be seen in figure 1.2 on page 21. The first phase starts with the first generation and stops with the reach of the first maximal. In a normal curve the size of the phase is between 10 to 200 generations. The second phase starts with the end of the first and ends with the reach of the first minimal. The last phase starts at the end of the seconds and contains the whole swinging of the curve to the end of the experiment.

The Curve of the Average Value

This curve looks familiar to the curve of the best values. The difference is that every swing of the curve has a little delay and the values reached are not so high as the values of the first curve. The really imortant fact is the first one, that the shape of the curves are simular. This means that the population follows in the behaviour the best individual. If this does not happens this could have only one reason. The programs and routines in the internal routines of the genetic algorithms are not implemented properly. The system should be checked on errors in the procedures of reproduction.

Figure 1.2: The Phases of the Curve

For the same reason this curve should be continious. Otherwise a great number of individuals will change in one generation. By using the introduced mechanism this could only happen if the mutation quote is too high. Even at points of incontinuity in the curve of the best values this curve should remain at most in its behaviour.

There is the possibility that the curve of the average value merges the curve of the best value. Than the whole population has become equal, there is no other individual then the one with the best value. In this case the genetic algorithms should terminate, there will be no more change in the behaviour and no more optimization.

The curve of the worst values

There is no special behaviour for this curve. Of course it will be under the average curve, but it could have a quite different shape. This curve is not important when simple genetic algorithms are used. It will be important later by using the advanced genetic algorithms.

Chapter 2

Programming a SGA

Even there are not many mechanisms introduced in the system so far, the concept of simple genetic algorithms is very mighty. Almost all rational functions could be solved by using a SGA. Only functions with singularities will produce some problems. Also the SGAs will work on many non-rational functions.

In this chapter I want to demonstrate how a problem could be solved by using the SGA package. This package is an improved implementation of the basic system by Goldberg[Gol89].

2.1 The Problem

To demonstrate the work with a simple genetic algorithm I use the example of the last chapter. Simply I want to find the maximal value of the function

$$y = x^2$$

Of course by using common mathematics it is very simple to find this maximum. It will not need any computer program to show the behaviour of the curve and to illustrate the complete function.

But it makes sense to use this function because it is well known and the results can be interpreted without any other help.

As known, the function $y = x^2$ has no maximum, the limit $\lim_{x\to\infty} x^2$ does not exist. It is necessary to reduce the size of x artificially. The program may only check the integers on the interval $[0, 2^{10}]$.

The program has to find the maximum at $2^{10} = 1024$, this is the maximum value for this problem.

This example is very suitable to define most of the other mathematical problems. The only changing the user has to make is in the definition of the mathematical function and depending on the mathematical function the search space and the fitness function. Also the function, which forms a number or a value out of the string must be changed, if this is needed.

If a floating point number should be calculated, the string is divided in two parts, a mantisse and a exponent. Most of the fundamental books on computer science show how a floating number is coded in a binary string.

2.2 Transforming the Problem

After the analysis of the problem there are two major problems to build the genetic algorithm. The first problem is the fitness function, i.e. the measure for the quality of the individuals. The second problem is the form and the size of the population and of the single individual. In the next subsections I try to explain which criterias are important by defining the fitness function and the population.

2.2.1 Defining the Fitness Function

Defining the fitness function is the most critical part using genetic algorithms. In the area of mathematical function this part is not so hard. But in most other cases even this is more difficult. The user of a genetic algorithm should think about

- that the fitness function should be a injective relation from string to value.
- that the fitness function should work in one direction only. Experiments in literature[Gol89] show, that an optimization in more than in one direction causes problems, even if the problems look very simular.

In some cases it is necessary to redefine the fitness function, if e.g. the experiments show that the rate of optimization is not high enough.

It is very important to say, that finding the fitness function is an empiric process. No place in the known literature gives an algorithm to form a fitness function out of a problem. Also there is not only an single fitness function for a problem, in most cases it is so that more then one fitness function fit in a problem. It is not proven, that a difference in a fitness function makes differences in the efficency of the process, but it is the most common opinion.

I cannot give a special recipe to define a fitness function, too. But in the area of mathematical functions normally the function itself is used. In the forth chapter at page 42 I use a more difficult fitness function, trying to solve the traveling salesman problem.

2.2.2 Defining the Individuals

An other important problem using genetic algorithms is the question on the size of the population and the single individuals. I start with the single individual.

Size of the Individual

The size of the indvidual is a bijective projection between a number $lchrom$ and the size of the interval $\mathcal{D}$, called the search space. On this way every element in the search space gets a representation as a string.

Simply the interval in the exercise is the integer numbers $[0, 2^{10}]$, so that the size of the interval is also 2^{10}. By using the formula of the chapter before I define the length of the string $lchrom = 10$.

The size of the individual would be different if the search space is not a set of integers but an interval of real or complex numbers. In this case another projection will be taken.

The Size of the Population

In difference to the size of the individuals there is no formula to calculate the size of the generation.

An good empiric rule is that the size to the population should be the same than the size of the inidividual.

Experiments have shown, that it is absolutly possible that this generation size is too small, so the information in the strings is not different enough. The information is too homogenous. This must no happen in the first phase. It may happen in the third phase, when the optimization is nearly finished. But if the generation is too homogenous, the number of inidividuals must be increased. The best size of the population should be found by a number of experiments. The user should think on the following : if the generation is too big the positive properties of genetic algorithm are lost. The golden rule of the generation size is

"As big as necessary and as small as possible"

2.3 Running the System

Before giving explainations on the following program I assume that the reader knows the syntax and the mechanisms of the programming language ADA. The following programs and packages are tested on at least three different platforms. These three platforms are

1. an Unix System V Release 3.2 with a Meridian Ada Compiler,
2. an Apollo Workstation, and
3. an IBM PC under MS-Dos

The following source code works in all cases without change. The source code will be commented line by line if necessary. It dependes on the theoretical view in the section above.

2.3.1 Requirements and Implementation of the SGA's

What is needed to program a genetic algorithm ? First there are satisfying data structures, i.e. data structures which represent the structures of the individuals and the population. As descripted before, the individual is nothing more but an array of the value 0 and 1, a binary string. A generation is an array of individuals. The package implementing the functions of genetic algorithms should provide these data structures.

The package should also provide the three basic genetic functions of the reproduction of the individuals. For the use of these elementary genetic algorithms no more special functions from a library are needed.

So does the package used in this example. It is a basic library with the elementary data structures and functions, as descriped in the book of Goldberg[Gol89], called `simple_genetic_algorithms`. This is a improved portation of all the function invented by Goldberg, using the advanced mechanisms of Ada. The library is a generic package, this means, that the size of the individual and the size of the generation are kept variable.

To use an Ada package it is important to know the header of this package. The header of the used package is the following :

```
with text_io; use text_io;
with random; use random;
with fio; use fio;
with iio; use iio;

generic
```

To be defined :

MaxAllele = Numbers of Alleles in an Individuum

MaxPopSize = Number of Individuums in a Population

```
   MaxAllele            : integer;
   MaxPopSize           : integer;

package Simple_Genetic_Algorithms is
```

First definition of the most important types for genetic algorithms

```
  type Individuum       is array (1..MaxAllele) of integer range 0..1;
  type Population_Array is array (1..MaxPopSize) of Individuum;
  type Fitness_Array    is array (1..MaxPopSize) of float;
```

Definition of the initialization of the population

```
  procedure Init_Population (Population : in out Population_Array);
```

Definition of the elementary functions of SGAs

```
  procedure Roulette_Wheel (Population : in out Population_Array;
                             Fitness    : in     Fitness_Array);
  procedure Crossover      (Population : in out Population_Array);
  procedure Mutation       (Population : in out Population_Array;
                             Mutationquote: in   float);
  procedure Simple_Display(Generation : integer;
                           Population : in Population_Array;
                           Fitness    : in Fitness_Array;
                           Org_Value  : in Fitness_Array;
                           Calc_Value : in Fitness_Array);

end Simple_Genetic_Algorithms;
```

2.3.2 Solving the Problem

By using the header above I generated the following program:

```
with Text_io;
with simple_genetic_algorithms;
```

This first two lines of the implementation of the program adresses the standard text input-output library and the library for the genetic algorithm.

```
procedure test_environ is
  package Genetic_Algorithms is new simple_genetic_algorithms(10,10);
  use Genetic_Algorithms;
  use Text_io;
```

Before the genetic algorithm library can be used, it must be instantiated. The size of the individual is the first, the size of the population is the second parameter. Because of the reasons above, both parameters have the size of 10. **Notice :** The size of the population must be a product of $2 \times n$.

```
Population    : Population_Array;
Fitness       : Fitness_Array;
Generation    : integer := 0;
Value_Array   : Fitness_Array;
Calc_Array    : Fitness_Array;
```

The variable `Mutation_Quote` is declared in the theoretical chapter and is need to discribe the probability to mutate.

```
Mutation_Quote : float := 0.005;
```

The next function converts a string into an integer number. This is one of the exercises a programmer has to solve if he wants to use genetic algorithms: programming a function which forms a value for the fitness function from the binary string, commonly as individual.

```
function Binary_to_Integer (Genetic_String : Individuum) return
               integer is
  Number : integer := 0;
  begin
    Number := Genetic_String(1);
    for t in 2..Genetic_String'Last loop
      Number := Number + Genetic_String(t) * (2 ** (t - 1));
    end loop;
  return Number;
  end Binary_to_Integer;
```

This is the fitness function. As can be seen, this function does nothing more but to calculate a value for each member of the whole population. It also calculates the percentage of the fitness for each individual.

```
  procedure Check_Fitness (Population : in     Population_Array;
                           Value_Array: in out Fitness_Array;
                           Calc_Array : in out Fitness_Array;
                           Fitness    : in out Fitness_Array) is
    Sum_of_Fitness : float := 0.0;
    begin
      for t in Population'Range loop
        Value_Array(t) := float(Binary_to_Integer(Population(t)));
```

The main fitness function as discribed in this chapter. By chaining this line it is possible to optimize another mathematical function or relation

```
        Fitness(t) := float(integer(Value_Array(t)) ** 2);
```

The array `Calc_Array` is used in the simple display function of the package. If the user wants to program his own output, the following line is not used.

```
        Calc_Array(t) := Fitness(t);
        Sum_of_Fitness := Sum_of_Fitness + Fitness(t);
      end loop;
      if (Sum_of_Fitness <= 0.0) then
        Sum_of_Fitness := 1.0;
      end if;
```

Also the function calculates the percentage of single fitness values.

```
      for t in Population'Range loop
```

```
          Fitness(t) := (Fitness(t) / Sum_of_Fitness) * 100.0;
        end loop;
      end Check_Fitness;
```

The following sequence is the main program. It consists of only a very few lines, initializing the population and running in the cycle of the genetic algorithm.

```
  begin
  put_line ("Test Environment for Genetics Algorithms Library started");
  Init_Population(Population);
  put_line("Start of cyklus");
  loop
    Check_Fitness(Population,Value_Array,Calc_Array,Fitness);
    Roulette_Wheel(Population,Fitness);
    Crossover(Population);
    Mutation(Population,Mutation_Quote);
    Simple_Display(Generation,Population,Fitness,Value_Array,Calc_Array);
    Generation := Generation + 1;
  end loop;
  end test_environ;
```

The program solves the problem of the exercise. This is of course a very simple one. In the next chapter I introduce some mechanisms to solve more complicated problems which leads to a more intersting problem which is solved here. This problem will be the Traveling Salesman Problem.

Chapter 3

Advanced Genetic Algorithms

In the first two chapters I had the focus on very simple problems, which do not come up with too many problems. The major problem in the area of genetic algorithms is that most of the problems are not so simple. So it is necessary to improve the mechanisms and to define some more additional functions.

In this chapter I start with the most possible problems and show, how they are handled in the common literature.

3.1 Constraints

By using genetic algorithms at more complex problems, the system cannot provide a bijective relation between the search space and the binary string anymore. In this cases an injective relation is used. This means, that there exist strings without a representation in the search space.

Of course, these strings must not enter the next generation. In other words, these strings have a fitness value of 0.

This becomes complicated, if the number of zeroed individuals is so big, that the remaining population is too small to produce an new healthy generation. In this case the whole system is not properly working and there is no simple possibility to repair it. One very small chance to save the system is provided by the concept of the handicapped individuals, but if this does not work, the fitness function must be redefined.

By defining more and more complex problems the probability of constraints destroying the system increases. So constraints should only be used with greatest caution. If possible, they should be always avoided.

3.2 The Method Of Fitness Scaling

The most familar effect in the use of genetic algorithms is that the optimization in the first phase in the run does not happens or is very slow. If this happens it could have two reasons.

1. The problem is not suitable for genetic algorithms, or
2. The results of the individuals i.e. the fitness does not show big differences.

It is very difficult to prove that a problem is not suitable, and this could not be said just after the first experiments. The second conclusion is more reasonable.

If the difference of the fitness is not big enough, the programmer should spread up this difference artificially. This means, that the difference even between two individuals with nearly the same properties makes a difference in the reproduction.

By this way it is possible to use genetic algorithms at problems were normally they would not work.

3.2.1 Spreading the Fitness

The method of fitness scaling is very simple, but also very effective. The first step is to substract the fitness of the worst individual from the fitness of the whole generation, i.e. from every single individual.

Having done this, the generation now has a fitness between 0 and a certain x. But the difference between the individuals is as big as before. The next step is simply to multiply the fitness of the individuals by a fixed number. Now the difference is wide enough to find significant differences.

By reinspecting the fitness the algorithm will find a possibility to select between the individuals, so that the optimization has a better change to work properly.

3.2.2 Using the Fitness Scaling

Because the problems with the fitness are common the fitness scaling is used very often. This means that the programmer of the genetic algorithm has to choose between two possibilities using this mechanism :

1. Constantly - the fitness of the generation is always spreaded and checked. But this also has the result, that the user cannot check, if there is really an optimization.

2. Only if needed - so that if the difference between the single individuals is big enough, then there will no fitness scaling. This gives the user the possibility to check, if the optimization over the generations is really effective.

Normally the second way of the fitness scaling is used. Only in some very extreme cases the first one will be choosen.

3.3 The Generation Gap

An unwanted effect in the use of genetic algorithms is the exist of phase 2 and phase 3 as shown in figure 1.2. A very proper behaviour of the curves would be, if they stay up and do not swing on a lower level. For the curve of the best values this is very easy to realize. But the method has no parallel in the world of nature.

The simplest method would be not to destroy the best individuum in the next cycle. Even if this will slow down the speed of optimization it also keep the best individuals alive. The way doing this is to reduce the number of the new individuals in the next generation by the size of the generation gap. The spare space in the next generation is filled by the best individuals of the present generation. The size of the gap is keeped variable, so that it can be changed in the experiments. But it should not be greater then a third of the generations size. Otherwise it is to dominant and the reproduction of generation does not have a proper effect. Best results alway occur, when the size of the gap is small, in the area of one to three individuals. By using the size like this, there is still the effect of the generation gap, but the slow down of the optimization is not to big.

The slow down of the optimization happens, because the genetic material of the best individuals does not join the generation of the best generation. This means, that the curve of the best individual does not go down, but also that the rest of the population must produce a better individual to replace the present gap. At the end of phase one this becomes more and more difficult and after the end of phase one normally only a mutation does change the gap.

A better way of the gap is also to eliminate the individual with the worst fitness and replace it with the individual with the best fitness. On this way it is possible to keep the best individual alive and also to bring the genetic material in the circle of reproduction. Then the effect of the gap is reduced on the best

Figure 3.1: The Effect of the Generation Gap

individual and the rest of the population can work on as it would do without the generation gap.

3.4 Fixed Point Crossover

Some problems works with values, which can't be simply transformed into a binary string. These problems are not suitable for a simple genetic algorithm. So a mechanism must be developed to work with these problems.

This mechanism is called fixed point crossover. This means, that the crossover will not take place at a randomly selected point, but at some predefined single points in the string. These points do not necessary be in a specific order. They only must to be defined by the exercise to be solved with the genetic algorithm.

This mechanism helps to put binary digits together to integer numbers, so that also strings of integer numbers can be used within genetic algorithms.

By defining methods for the algorithms it is important to see, if there is a equivalent in nature. This mechanism has such an equivalent. A natural genetic string consists of more than to bases, i.e. a human genetic string consists of four different bases. This means, that we do not have a binary system. The single bases do not change in the crossover. The only thing happens is, that they are recombined in a different order.

To simulate this the string in the computer must not be cut at a randomly defined point. Only every second point in the string may be used for the crossover. So the paired genes of the string form an integer value from 0 to 3, which could be an equivalent to the four bases in nature.

As far as known, this mechanism is invented by the dutch scientist Lucansius[Luc91] which made experiments to analyse to human DNA with the help of genetic algorithms. Even these kind of systems are not very common, it is good to notify that a solution to these problems is possible and approved.

3.5 The Handicapped Individual

A big problem by the use of genetic algorithms is when the defined constraints are too tough. Then the next generation does not contain enough surviving individuals to produce a healthy population. Mostly the genetic algorithms stop, because the is insufficent genetic material to generate even a new generation.

Another reason to keep these individuals is the possibility that the information provided in this individual is very important and only a little defect makes this special individual sort out off the population.

These effect occurs very often. When it is not possible to redefine the fitnessfunction the user has to find an other way to keep the system running. A method to do this is the method of the handicapped individual[Gri91b]. First experiments with a good result are made by Jochum et al[Joc91].

Individuals, which are sorted out are called **Totgeburten**. These **Totgeburten** could even provide important informations, which should not get lost. So by changing the behaviour of the constraints it is possible to keep the individual in the population. As a fact, the method does not change the constraint but the individual. This changed individual is called a handicapped individual.

Definition 3.1 (A Handicapped Individual) *is a* **Totgeburten**, *which can stay in the population if there exist a correct individual, so that the distance between these individuals is not greater then a certain number and a function, which transforms the* **Totgeburten** *to the correct individual.*

This definition can help to define a correcting function. To define this function it is necessary to define the distance between two individuals. I used this distance in the definition above.

Definition 3.2 (The Distance) *of two individuals is defined by*

$$x, y \in \mathcal{I} : x \ominus y : \sum_{i=1}^{lchrom} |x_i - y_i|$$

With this it is possible to define the correcting function:

Definition 3.3 (The Correcting Function) *is defined by*

$$k(\mathcal{I}) : \forall x : x \notin \mathcal{I} \rightarrow x' \in \mathcal{I}$$

and also the following condition must be satisfied:

$$x' \ominus x \Longrightarrow min$$

Additionaly there is a value ξ which defines a cut. A individual with more malicious informations than this number ξ must not be repaired.

3.5.1 Differences To The Normal Constraints

Normal constraints are normally destructive, they eliminate individuals if they do not fit in the environment. This does not happens in nature, in which there are handicapped individuals at all. The problem of these handicapped individuals is that they have a lot of problems to survive.

If they do it, they are very tough and have a quite good genetic material. This genetic material could be necessary to keep the whole species alive. So the concept of handicapped individuals is quite close to nature and experiments [Joc91] have shown, that genetic algorithms work more properly if the constraints are replaced by the concept of handicapped individuals.

3.6 Diploid and Haploid Crossover

In nature there are two different possibilities of exchanging information between two generations. In the chapters before I discribed a simple mechanism of cutting the string with the genetic information and exchanging the parts. This is called a halpoid crossover, according to the term in biology. But there is a different way to generate new information.

This way is called the diploid crossover. It works simular to the way a crossover with the DNA is done. The genetic string of both individuals are melting together to a new string and then this new string doubles to build the information of both new individuals.

This looks more like the way nature does the crossover, but in the area of genetic algorithms this is not very much in use. The problem of this system is, that it makes the whole genetic algorithm very complicated.

Another danger of this crossover is that the number of different individuals is artificially decreased. The programmer should be very carefully to use this mechanism.

Because of the habits of this method I like to advice the user to consult the book of Goldberg[Gol89] before installing this procedure into his program.æ

Chapter 4

Programming with AGA's

In this chapter I want to demonstrate the use of the advanced genetic algorithms. For this I use a well known problem of the copmuter science, the traveling salesman problem. This is a very good example of an $\mathcal{NP}$-complete problem[Hop88]. This kind of problems cannot be solved by computer algorithms if the size of the problem exceeds a certain number, because the number of calculations necessary to solve these problems are growing by an exponential factor. Another example of this class of functions is the Ackermann- function.

An interesting fact in the history of genetic algorithms is that in the first years the researchers believed they could solve this kind of problems. So some experiments were made, with the result that all attempts failed which means that the computer calculate a nearly infinity time to get the result or that the size of the population is so big that the result is automaticly a member of the population.

There are different solutions using other methods but none of them can ignore the fact that they will fail if the size is big enough.

The following program is not another try to solve this problem, but a demonstration how to formulate a more complex problem as a genetic algorithm. For this I first have a little revision on the problem itself.

4.1 The Traveling Salesman

A travelling salesman has to travel to a number of towns in order to sell his goods to the local shops. For this it is important for him to enter each town one time and to keep the distances between the different towns as small as possible. Also he does not want to enter the same town a second time.

There are different strategies to solve that problem.

1. Calculate all possible routes and choose the shortest. This means that the computer has to work on a huge number of possibilities. Using elementary statistics it is simple to prove that the computer has to calculate $n!$ different routes. So for 5 towns there are 120 different routes, but for 25 towns he has to calculate $1.55 \cdot 10^{25}$ different routes. I assume the user has a very quick computer which can calculate 10 routes a second this number means that the computer will have to work for approx. $5 \cdot 10^{16}$ years. If this imaginary computer would exist at the beginning of the universe it would be possible to get the results in some billion years. But raising the number of towns by ten the computer will work longer than this universe will exist.

2. Using a simple hill-climbing algorithm. This method produces a huge number of calculations, too.

3. There are also methods in the area of operation research discriped in literature. An example of this methods is the Knappsack-algorithm. But the biggest number of towns which is solved by this methods is under 50.

To simplify the problem I assumed that between each town exists a road and the distances between the towns are constant[1].

4.2 Using Genetic Algorithms

Transforming the existing problem into a form suitable for genetic algorithms needs the definition of the two major components. These components are the individual and the fitness function.

4.2.1 Defining the Individuals

The first task by using genetic algorithms to solve the traveling salesman problem is to define the individual. By examining the problem it could be seen that the main subject of this problem is the route the salesman has to take. So I am looking for a representation of the route as a binary string.

First I coded the towns as numbers, defining a function which projects a town on an integer. This is a basis to define a representation of the route. I can define the route as the order of the towns. For example the number chain '132' represents the route beginning at town no. 1 to town no. 3 and than going to town no. 2. Another route is represented by the number chain '312' or '231' and so on. As simply can be seen, each number can only be once in the string. If it would occure a second time the system would contains loops. By defining the individuals with a fixed length[2] this also excludes some towns from the route.

A special case is if the first town occurs in the string the second time. This is called a short circuit. The travel ends before all towns are visited because the salesman reaches his origin again.

[1]i.e. the distance between A and B is the same as the distance between B and A. In our normal life this is not nescessarily true.

[2]In nature the DNA has a fixed length, too.

So using this method to code the route as an individual I transfer the number chain into a binary string. For this a transform each number of the chain into its binary representation is made. It has to be made sure, that the binary numbers have the same length, i.e. that they have the same number of digits. Writing the digits in the string forms the individual I am looking for.

What makes the representation really difficult to use is the fact, that there are more individuals which represent forbidden routes than there are individuals which represent good and useful routes. As can be seen later in the experiments this problems will grow to a dimension which makes the system nearly useless.

4.2.2 Finding the Fitness-Function

The fitness function is not too complicate to find. The aim on this quest is to find the shortest route around all cities. So the first and best idea is that the driven distance is a direct measurement for the route. By following the route town by town a number of driven miles can be calculated. A route is better for the salesman if its milage is smaller than the milage of the other routes.

So the problem in this specific exercise is not the fitness function but the representation of the individual.

4.3 Using the Package

Trying to use the SGA package shows very quick, that this package is not suitable. At several points I need the mechanisms of the advanced genetic algorithms :

- The individuals must not crossover at a randomly selected point. Doing this would distroy the whole information of the string. Only at the ends of the numbers is a crossover allowed.

- The crossover also uses the haploid crossover mecahnism.
- The mutation quote should be close to zero, any change of a single binary digit will destroy the whole individual.

By using these mechanisms it should possible to solve the problem. But after a few test runs it could be seen, that the number of bad individuals is huge and that after a few generations all individuals are dead. So it is necessary to install a repair algorithm, according to the section on handicapped individuals in the last chapter.

The result of these **Gedanken** is the following program.

Initialization of the Package

A generic package in Ada must be initalized before it can be used. The first step in the program is the using of most of the standart packages.

```
with Text_io;
with iio; with fio;
with Advanced_Genetic_Algorithms;

procedure Traveling_Salesman is
-- Number of Allele in the String : 8
-- Population : 10 Strings
-- Number of Points to cut the String : 7
  package Genetic_Algorithms is new Advanced_Genetic_Algorithms(24,10,7);
```

This line initializes the package for the `Advanced_Genetic_Algorithms`. The three parameters of initialization are

1. - The size of the individual, in this case it is the number of towns times three. I use eight towns for this demonstration, these can be coded by a three digit binary number. So the size of a individual is $8 \times 3 = 24$ bits.
2. - The size of the population. This value is found by try and should be changed during the experiments.
3. - The number of crossover points. I use eight towns, so there are $8 - 1 = 7$ crossover points.

```
use Genetic_Algorithms;
use Text_io; use iio; use fio;
```

Definition of the elementary variables

In the next lines I defined the nescessary variables and arrays. This is alway a task for the programmer, because he is the only one who knows the semantic contents the genetic algorithms are used in.

```
No_of_Towns : constant integer := 8;
type Route  is array (1..No_of_Towns) of integer;
type Routes is array (1..80) of Route;
```

The following declaration is the table to change a number to a town. For example the number 1 represents London, the number 8 represents Dover. Also the distances between the towns are defined in this program segment.

```
Towns       : array (1..No_of_Towns) of string(1..10) :=
  ("London    ",
   "Edinburgh ",
   "Cardiff   ",
   "Newcastle ",
   "York      ",
   "Holyhead  ",
   "Thurso    ",
   "Dover     ");
Distance_Array : array (1..No_of_Towns,1..No_of_Towns) of integer :=
 ((  0, 378, 159, 274, 193, 259, 651,  71),
  (378,   0, 385, 110, 194, 308, 278, 449),
  (159, 385,   0, 301, 240, 201, 650, 237),
  (274, 110, 301,   0,  84, 247, 384, 345),
  (193, 194, 240,  84,   0, 188, 462, 264),
  (259, 308, 201, 247, 188,   0, 587, 339),
  (651, 278, 650, 384, 462, 587,   0, 717),
  ( 71, 449, 237, 345, 264, 339, 717,   0));
```

The table of the distances are taken from [Col91]. Are following declarations are standart and also used in the first example.

```
  Population     : Population_Array;
  Fitness        : Fitness_Array;
  Generation     : integer := 0;
  best_Generation: integer := 0;
  Value_Array    : Fitness_Array;
  Calc_Array     : Fitness_Array;
  Way            : Routes;
  Mutation_Quote : float := 0.05;
  Crossing_Dummy : Crossing_Points;
  best_Way       : Route;
  best_Value     : float := 99999.99;
  Roulette       : boolean := false;
```

The Output

This first procedure is the output of the system, it shows the found routes of the system. To keep the program small I only installed a very rough procedure, in a real system the user interface should be more satisfying.

```
procedure Show_Way (Way : in Route) is
  begin
  for t in 1..(Way'Last-1) loop
    put(Towns(Way(t))); put(" --> ");
    if ((t mod 4) = 0) then
      new_line;
      end if;
    end loop;
  put(Towns(Way(Way'Last)));
  end Show_Way;
```

The next procedure changes the binary string into the number chain which represents the route.

```
procedure Change_String_to_Route (Genetic_String  : in     Individuum;
                                  Transformed_Way : in out Route) is
  x : integer := 1;
  begin
  for t in Transformed_Way'Range loop
    Transformed_Way(t) := Genetic_String(x) * 4 + Genetic_String(x+1) * 2
                                                + Genetic_String(x+2) + 1;
    x := x + 3;
    end loop;
  end Change_String_to_Route;
```

The Fitness Function

The following lines are the implementation of the fitness function. The fitness function also contains the repair function to produce handicapped individuals. Of course this way including the repair function is not a must, it can be done differently and will work as good as this one.

```
procedure Check_Fitness(Population  : in out Population_Array;
                        Way         : in out Routes;
                        Calc_Array  : in out Fitness_Array;
                        Fitness     : in out Fitness_Array) is
  Sum_of_Fitness  : float := 0.0;
  Sum_Distance    : float := 0.0;
  Number_of_Knots : integer := 0;
  Error_Coeff     : integer := 0;
  Double          : integer := 0;
```

This function calculates the distance the salesman has to travel following the route given into the function. If the route contains a short circuit the return value is of maximum size.

```
  function Distance (Way : in Route) return float is
    Way_Gone     : float   := 0.0;
    Short_Cycle  : boolean := false;
    used_Towns   : array(1..No_of_Towns) of integer;
    Acculum_Dist : integer := 0;
    begin
    for t in used_Towns'Range loop
      used_Towns(t) := Way(t);
      end loop;
    for t in 1..(Way'Last-1) loop  -- former -1!
      for x in used_Towns'Range loop
        if ((used_Towns(x) = Way(t)) and (x /= t)) then
          Short_Cycle := true;
          Error_Coeff := Error_Coeff + 1;
          Double      := t;
          end if;
        end loop;
      used_Towns(t) := Way(t);
      Acculum_Dist := Acculum_Dist + Distance_Array(Way(t),Way(t+1));
      end loop;
    if (Short_Cycle = true) then
      return 99999.99;
    else
      return float(Acculum_Dist);
      end if;
    end Distance;
```

The following procedure is the repair function. As can be seen, this function corrects every route which has less then two errors.

```
procedure Repair(Element : in out Individuum; Way : in Route) is
  missing      : integer := 0;
  First_Allele : integer := 0;
  in_it        : boolean := false;
  begin
  for t in 1..No_of_Towns loop
    for x in 1..No_of_Towns loop
      if (Way(x) = t) then
        in_it := true;
        end if;
      end loop;
    if (in_it = false) then
      missing := t;
      in_it := false;
      end if;
    end loop;
  First_Allele := (Double * 3) - 2;
  for t in First_Allele..(First_Allele+2) loop
    Element(t) := missing / 2;
    missing := missing mod 2;
    end loop;
  end Repair;
```

The following procedure is the main part of the fitness function. It works as follows: First the binary string will be converted to a route. This route is checked and repaired if there exists a chance for this. After this a fitness is calculated.

```
begin
for t in Population'Range loop
  Change_String_to_Route (Population(t),Way(t));
  Calc_Array(t) := Distance(Way(t));
  if (Error_Coeff = 1) then
    Repair(Population(t),Way(t));
    Change_String_to_Route (Population(t),Way(t));
    Calc_Array(t) := Distance(Way(t));
    end if;
  Error_Coeff := 0;
```

If the individual is healthy it gets a distance of less than 99999.99. Then the individual joins in the reproduction and the length of the route is added to the sum of distances.

```
    if (Calc_Array(t) /= 99999.99) then
      Sum_Distance := Sum_Distance + Calc_Array(t);
      end if;
    end loop;
```

If the individual could not be repaired it gets the fitness of 0.

```
  for t in Population'Range loop
    if (Calc_Array(t) = 99999.99) then
      Fitness(t) := 0.0;
    else
      Fitness(t) := (Calc_Array(t) / Sum_Distance) * 100.0;
      end if;
    end loop;
  end Check_Fitness;
```

The Main Program

As can be seen the main program is very simple again. It starts with the elementary initialization of the population. This is done with the random number generator of the package. After this is done the program enters the loop as in the first example.

Some more lines are included to the main structure to produce additional output and information for the user. This is a good hint for everybody who wants to program genetic algorithms himself. Additional informations are necessary to find were the problems in the genetic system occur.

```
begin
  Init_Population(Population);
  best_Way := Way(1);
```

```
loop
  Check_Fitness(Population, Way, Calc_Array, Fitness);
  Generation := Generation + 1;
  put("Generation : "); put(Generation); new_line;
```

The next lines select the output: If a better route is found, it is remembered.

```
    for t in Population'Range loop
      if (best_Value > Calc_Array(t)) then
        best_Value := Calc_Array(t);
        best_Way   := Way(t);
        best_Generation := Generation;
        end if;
```

All practical routes are displayed. This lines should be cancelled if the displayed information hides the important information of the system.

```
      if (Fitness(t) > 0.0) then
        Show_Way(Way(t)); put(integer(Calc_Array(t))); new_line;
        end if;
      end loop;
```

The best route found so far is displayed, too.

```
    put("best Value : Generation : "); put(best_Generation); new_line;
    Show_Way(best_Way); put(integer(best_Value)); new_line;
```

If there are not enough individuals for the next generation, the roulette wheel reproduction is cancelled. This might be nescessary in some cases.

```
    Roulette := false;
```

```
      for t in Fitness'Range loop
        if (Fitness(t) > 0.0) then
          Roulette := true;
          end if;
        end loop;
      if (Roulette = true) then
        Roulette_Wheel(Population,Fitness,0);
        end if;
```

The rest of the procedure look quite familiar.

```
      Crossover(Population,haploid,randomize,Crossing_Dummy,0,0);
      Mutation(Population,Mutation_Quote,0);
      end loop;
  end Traveling_Salesman;
```

4.4 Additional Comments

This system is, of course, a very small one. But working with a bigger system would expand the code and would be very hard to understand, too. Surely not all problems could be solved with this package, there are problems which require a two dimensional system of genetic algorithms. Also there may be problems which require different procedures. Genetic algorithms are in the beginning of the use outside of the labors and the number of researched problems is not very big. Most discussed problems are of a very theoretical nature and a lot of the real world problems may look different.

This procedure is a hint how such a system may look alike, as written above there is no rule how a genetic algorithms has to be.

A full description of the interface to the different packages is in the appendix, as well as a full listing of all of the used packages.æ

Chapter 5

Two Dimensional GAs

A very new development in the area of genetic algorithms is the introduction of two dimensional genetic algorithms. This chapter will show the reasons which leads to the development of these kind of algorithms and the way these are working. The last part of this chapter is a brief discription of a system developed to solve some problems with two dimensional genetic algorithms. The theory of genetic algorithms is not discussed in the public, so this chapter is a short discription of the research which is done so far.

5.1 The Theory of Two Dimensional Genetic Algorithms

The idea of the use of two dimensional genetic algorithms was born by trying to optimize a system where the subject of the optimization could easily be transformed into a matrix. But transforming this into a string looked quite more complicated. In this specific problem the task was to optimize a structure from the graph theory.

As known a graph could be represented by a matrix, where a conection between the mth and the n knot is represented by a '1' at the mth row and the nth column and a '1' at the nth row and the mth column of this matrix. This matrix is called the adjacenz matrix of the graph.

This representation is very simple and has its fundaments in the mathematical graph theory. A different representation into a binary string will be much more complicated.

If it is possible to prove the equivalence between an one dimensional and a two dimensional individual it should be possible to define the basic operations of genetic algorithms this way that they are suitable for two dimensions.

5.1.1 Equivalence between 1-D- and 2-D-Individuals

The equivalence between the two different types of individuals can easily be shown by defining a function which projects an arbitrary $m \times n$-matrix on a $k \times 1$-matrix. The size k of the second matrix is clearly defined by the multiplication of m and $n : k = m \cdot n$.

The second matrix can be interpreted as an binary string. This is exactly the form needed to work with genetic algorthms. If the dimensions of the matrix are fixed[1] the inversion is defined, too.

So the first way of handling two-dimensional structures would be the following algorithm :

1. check the fitness of the individual
2. transform the individual into a string
3. use the rules for selection, crossover and mutation as known

[1]as they are usually

4. inverse the transformation
5. go to the first step

This algorithm works properly, the only reason to think about another possibility is of philosophical nature.

Is there a satisfactional reason to transform the two-dimensional individual ? And are there matrices where a transformation could destroy the structure of the individual ? By taking the example above it could be seen, that there cannot be any function to transform this individual which should be prefered to use. With other words: how should this transformation be made ? Is it nescessary to concat the rows of the matrix to construct the string, or should it be a concatination of the columns, or should it be a quite different function ?

It is possible to pass this questions, if equivalent functions for selection, crossover and mutation can be found. There would not be any reason to think about this problems and the algorithms could be normally used.

5.1.2 Equivalence of the Operations

It is very simple to prove equivalence between the operations *selection*, because it is not nescessary to define an operation different from the one-dimensional case. The structure of the individual is not important at this stage of the reproduction.

There is no problem at the mutation, too. This operation only changes single genes. This is independent from the structure of the individual. Only the number of existing genes is important, so that an expansion to higher dimensions will not change the behaviour of this operation.

A real difference can be found at the operation *crossover*.Here the structure of the individual is important and there has to be an expansion of the definition for one-dimensional individuals.

A proper definition of the two dimensional crossover would be the following one
:

Additional to the point of crossover, which is an intersection of either the rows or the coloumns it is useful to define a second point which intersects the other dimension of the individual. By this it is possible to define a cut in the rows and the columns. The result of this is a matrix divided into four parts or quadrants. By exchanging the first and the third quadrant between the partners of the crossover it is possible to expand this operation to the new kind of individuals.

Experiments with the new defined operations[**?**] show, that they are working and producing sufficent results.

5.2 Stability

A theoretical term I did not refer so far is the *stability* of a string.

Fundamentically an individual is a binary string A, consisting of k single elements a_1 to a_k. The order of these elements[2] is not important. To work with the operations of the genentic algorithms we need a full population. This population is called $\mathbf{A}(t)$, where t is the index for the actual generation. Also a string A' which is a part of the string A. This string A' consists of k' genes.

Now the probability that the string A' is distroyed while the crossover is $\frac{k'-1}{k}$. The possibility that the point of crossover is outside of the string A' is defined by the following formula

$$p_{string} = \frac{(k-1)-(k'-1)}{k-1} + c = \frac{k-k'-2}{k-1} + c$$

(by [Gol89]). Additionally it has to be calculated that the string A' is distroyed by the mutation. This is the additional factor c in the formula above. Explicit

[2]or genes

this factor c is defined by

$$c = (k - k') \cdot p_{mut}$$

To calculate the same values for two dimensional individuals it is necessary to add a second part to this formula. The probability that a submatrix is distroyed by the reproduction is defined by

$$p_{matrix} = p_{column} \cdot p_{row}$$

A submatrix M' is a $k' \times k''$-Matrix inside of the matrix M. Because the two points of crossover are selected randomally I can multiply the probabilities for each dimension. The full formula is now given by

$$p_{matrix} = \frac{k - k' - 2}{k - 1} + (k - k') \cdot p_{mut} \cdot \frac{k - k'' - 2}{k - 1} + (k - k'') \cdot p_{mut}$$

Because both parts of this formula are always smaller then 1 it could be said that a submatrix of n elements is more stabil then a substring of n elements.

5.3 Demonstation of the Use of a 2D-GA

This last section shows, how a genetic algorithm is used to optimize a graph[Gri91a]. Because the use of two dimensional individuals is new I do not have enough literature to refer to. The results presented here were made by doing own researches. The used package is documented in the appendix, it is mainly based on the AGA package.

A graph is a mathematical structure. In this specific case it is represented by an adjacence matrix. This binary matrix is by definition a $n\ timesn$ matrix, where n is the number of used knots in the graph. A definition of this matrix is given in the section above.

A fitness function for graphs can be defined on many different ways. One of the possible fitness functions is the *average path length*. The *average path length* is a value which discripes the number of steps in a tree from the root of the tree to a certain knot.

If the *average path length* is small a used search algorithms only needs a few steps to reach the wanted knot. A full description of the theory is given by KNUTH[**?**].

Expanding this definition for arbitrary graphs gave a very good fitness function and some experiments should show, how a graph will respond to this fitness function.

The surprising result of the tests was, that commonly the graph developed a very compact structure so that the distance between single knots was very small. This development was done in only a few generations, a fitness close to ninety percent of the possible optimum was reached within 30 generations[3].

The verify the results the experiments were repeated with a changed fitness function. This time a function was used which could be called a *cost function*. The function gives each possible connection between two knots a value. This value is called the *cost* of the connection. The accumulated costs of the system should be minimal.

This problem is typical for a class of problems occuring in the field of operation research. It discripes the problem of finding the cheapest connection between A and B, for example the cheapest transport between two towns.

The results of the tests done here are simular to the results of the first test series. The shape of the resulting graph look different from the first one, it is not compact but more like line connecting the root to the edge with the cheapest path.

Other experiments use 2D-GAs for indexing documents within an information retreival system[Loe92]. Results here show again, that this way of using genetic

[3]based on the experimental environment, using between 20 and 200 knots

algorithms work properly and produces quite good results.

It is important to say that these experiments are not final results of a research with the aim to establish 2D-GAs into the theory, but single, summerized results. It is nescessary to define experiments which confirm these results.

5.4 Ideas for a further usage of 2D-GA

The results made by the very first experiments are not satisfactionary enough to think on serious use of this kind of genetic algorithms. But if the results are confirmed by others they could be used in a wide area of subjects.

For example the optimization of neuronal networks can be done by using 2D-GAs. At the California Institute of Technologie (CALTECH) experiments are made to optimize those networks by genetic algorithms, but how these algorithms are programmed is not known now.

A problem in the area of operations research is the transport of goods in a factory such that the costs are minimal. These systems are very complex and the algorithms to solve this problems are complicated. The time used to produce plans for the transport can be reduced by using genetic algorithms.æ

Appendix A

The Package for the SGA

This is the package for the use of simple genetic algorithms. It is absolutly different from the PASCAL program developed by Goldberg[Gol89]. It tries to optimize the processes of the elements of genetic algorithms and also should be a skeleton for the development of other, more specific genetic algorithms. I assume that the mechanisms of the programming language ADA are known, so I do not explain the syntax of it.

A.1 The Header of the Package

The following program is the header of the package. To use the package a programmer only needs to know this part of it. Because it is so essential it is better documented than the body.

```
pragma LIST(ON);

with text_io; use text_io;
with random; use random;
with float_io; use float_io;
with integer_io; use integer_io;
```

These first used package are part of standart ADA environment. It is possible, that in some implementations the names are different. I used the names of the ADA implementation for the Apollo workstations. The package `random` is part of this report and not a standart of ADA.

```
generic
```

The following parameters must be defined by instantiation of the package :

- `MaxAllele`: The number of genes[1] in an individual
- `MaxPopSize`: The number of individuals in the population

```
   MaxAllele            : integer;
   MaxPopSize           : integer;
```

```
package Simple_Genetic_Algorithms is
```

The first definition in the package are the most important types for genetic algorithms. This type build the interface of the program and the SGA package. The used names find their equivalent in the book of Goldberg[Gol89].

```
  type Individuum       is array (1..MaxAllele) of integer range 0..1;
  type Population_Array is array (1..MaxPopSize) of Individuum;
  type Fitness_Array    is array (1..MaxPopSize) of float;
```

[1]or alleles

The first function initializes the population. Because there is no special rule an individual has to be initialized the procedure can be handled as a black box, creating individuals without information how this is done.

```
procedure Init_Population (Population : in out Population_Array);
```

Next is the definition of the elementary functions of SGAs :

1. The roulette wheel,
2. the crossover and
3. the mutation. First defined function is the `roulette_wheel`. This function produces the roulette wheel selection. Parameters of this function are :

- `Population` is the sum of all individuals.
- `Fitness` is an array in which for all individuals is their specific fitness.

```
procedure Roulette_Wheel (Population : in out Population_Array;
                           Fitness    : in     Fitness_Array);
```

The procedure `Crossover` works on the population and returns the new generation after the crossover step.

```
procedure Crossover      (Population : in out Population_Array);
```

The last of the three steps is the mutation. The procedure `Mutation` is an implementation of this step. The parameters of this procedure is again the population and the quote of mutation, which is a floating point variable < 1.

```
procedure Mutation       (Population : in out Population_Array;
                           Mutationquote: in   float);
```

The last procedure is a simple output procedure. This procedure can be replaced by the programmer without changing the core of the genetic algorithms. Its output is a tabular with four columns :

1. the individuals,
2. its percentage of fitness,
3. the integer value the individual does represent and
4. the result of the calculation done with the individual

```
  procedure Simple_Display(Generation : integer;
                           Population : in Population_Array;
                           Fitness    : in Fitness_Array;
                           Org_Value  : in Fitness_Array;
                           Calc_Value : in Fitness_Array);

end Simple_Genetic_Algorithms;
```

This ends the header of this package.

A.2 The Body of the Package

The package body is normally closed as a secret. The mechanisms of ADA do not allow, that anybody can inspect the following lines or uses the variables defined inside.

```
package body Simple_Genetic_Algorithms is

  Best_Org_Value        : float := 0.0;
  Best_Calc_Value       : float := 0.0;
```

```
Best_Generation       : integer := 0;
Last_Gen_Accum_Org    : float := 0.0;
Last_Gen_Accum_Calc   : float := 0.0;
```

The Initialization of the Population

```
procedure Init_Population (Population : in out Population_Array) is
  t, x                : integer;
  Single_Individuum   : Individuum;
begin
  for t in Population'Range loop
    for x in Population(t)'Range loop
      Single_Individuum(x) := random_ND_integer(65535) mod 2;
      if (Single_Individuum(x) > 1) then
        Single_Individuum(x) := 1;
      end if;
    end loop;
    Population(t) := Single_Individuum;
  end loop;
end Init_Population;
```

The Roulette Wheel Selection

As seen above the first procedure is the roulette wheel selection.

```
procedure Roulette_Wheel (Population : in out Population_Array;
                          Fitness    : in     Fitness_Array) is
  New_Population : Population_Array;
  Wheel_Length   : constant := 100;
  Wheel          : array (1..Wheel_Length) of integer;
```

```
    Position       : integer := 1;
    t              : integer;
    x              : integer;
    field          : integer;

  begin
```

This is the initialization of the roulette wheel. Every individual gets fields according its fitness, which must be than better then 1 % of the accumulated fitness in the system. The size of the roulette wheel will be less or equal 100 fields.

```
    for t in Population'Range loop
      if (Fitness(t) >= 1.0) then
        for x in Position..(Position+integer(Fitness(t))) loop
          if (x <= 100) then
            Wheel(x):=t;
          end if;
        end loop;
        Position := Position + integer(Fitness(t));
      end if;
    end loop;
```

Start of the selection of the new generation.

```
    for t in Population'Range loop
      field := 101;
      while (field > 100) loop
        field := (random_ND_integer(65535) mod Position) + 1;
      end loop;
      New_Population(t) := Population(Wheel(field));
    end loop;
```

Let the new generation take the place of the old one.

```
    for t in Population'Range loop
      Population(t) := New_Population(t);
    end loop;
```

This finishes the procedure.

```
  end Roulette_Wheel;
```

The Crossover

The crossover is the longest and most difficult part of the package.

```
  procedure Crossover (Population : in out Population_Array) is
    New_Population     : Population_Array;
    Partner            : array (Population'Range) of integer;
    x,y,z              : integer;
    filled             : boolean;
    Selected_Partner   : integer;
    Point_of_Crossover : integer;
    New_1st_Individuum : Individuum;
    New_2nd_Individuum : Individuum;
    Dummy              : Individuum;

  begin
```

First it is necessary to initialize the fields for the relationships. This fields are reserved for the information about the partners.

```
    for x in Partner'Range loop
      Partner(x) := Population'Length + 1;
    end loop;
```

Each individual gets its partner.

```
    for x in 1..(Partner'Last / 2) loop
      Selected_Partner := (random_ND_integer(65535) mod (Population'Length / 2))
                          + (Population'Length / 2) + 1;
      loop
        filled := false;
        for y in 1..x loop
          if (Partner(y) = Selected_Partner) then
            filled := true;
            Selected_Partner :=(random_ND_integer(65535) mod (Population'Length
                                  / 2)) +(Population'Length / 2);
            end if;
          end loop;
        exit when (filled = false);
      end loop;
      Partner(x) := Selected_Partner;
    end loop;
```

Now the partners are selected. Now the procedure finds the point of crossover for each pair and starts the change of the genetic information.

```
    for x in 1..(Partner'Last / 2) loop
      New_1st_Individuum := Population(x);
      New_2nd_Individuum := Population(Partner(x));
      Point_of_Crossover := (random_ND_integer(65535) mod Population(x)'Length);
      if (Point_of_Crossover > 0) then
```

```
        Dummy := New_1st_Individuum;
        for z in 1..Point_of_Crossover loop
          New_1st_Individuum(z) := New_2nd_Individuum(z);
          New_2nd_Individuum(z) := Dummy(z);
        end loop;
      end if;
      Population(x) := New_1st_Individuum;
      Population(Partner(x)) := New_2nd_Individuum;
    end loop;
  end Crossover;
```

This ends this procedure. Important for all procedures is the fact that they work independent from the size of the individuals and the population.

The Mutation

The mutation is a very short and simple procedure. It has only to check, if there are any mutations and than choose the genes which must be changed.

```
  procedure Mutation       (Population : in out Population_Array;
                            Mutationquote: in   float) is
    Number_of_Mutations : integer;
    Selected            : integer;
    Point_of_Mutation   : integer;
    Mutant              : Individuum;
    Quote               : float;
    Correction          : float;

    begin
      Quote := Mutationquote;
```

If the mutation quote is too small or too big a message for the user is printed. The system also chooses a different medium quote.

```
if (Mutationquote < 0.001) then
  put_line("Quote of Mutation too small ! Assumed new quote : 0.005 ");
  Quote := 0.005;
elsif (Mutationquote > 0.999) then
  put_line("Quote of Mutation too big ! Assumed new quote : 0.005 ");
  Quote := 0.005;
end if;
```

The next step calculates the number of genes which have to mutate.

```
Correction := 1000.0 / float(MaxPopSize);
Number_of_Mutations := integer(Quote * float(MaxPopSize) * Correction);
```

If the number of mutations is greater then 0, the individuals and their genes are selected. Than the mutation is executed.

```
if (Number_of_Mutations > 0) then
  Number_of_Mutations := random_ND_integer(65535) mod Number_of_Mutations;
end if;
if (Number_of_Mutations > 0) then
  for x in 1..Number_of_Mutations loop
    Selected := (random_ND_integer(65535) mod Population'Length) + 1;
    Point_of_Mutation := (random_ND_integer(65535) mod
                           Population(Selected)'Length) + 1;
    Mutant := Population(Selected);
    if (Mutant(Point_of_Mutation) = 0) then
      Mutant(Point_of_Mutation) := 1;
    else
```

```
          Mutant(Point_of_Mutation) := 0;
        end if;
      Population(Selected) := Mutant;
      end loop;
    end if;
  end Mutation;
```

The Display Function

The display function is not documentated.

```
procedure Simple_Display(Generation : integer;
                         Population : in Population_Array;
                         Fitness    : in Fitness_Array;
                         Org_Value  : in Fitness_Array;
                         Calc_Value : in Fitness_Array) is
Single_Allele         : integer;
Element               : Individuum;
This_Gen_Accum_Org    : float;
This_Gen_Accum_Calc   : float;
Change_of_Accum_Calc  : float;

begin
  new_page;
  put ("Genetic Algorithms Experimental Environment: Simple Display ");
  put_line ("Procedure");
  new_line;
  put ("Generation :");
  put (Generation);
  new_line;
```

```
    for T in Population'Range loop
      Element := Population(T);
      for X in Element'Range loop
        Single_Allele := Element(X);
        put(Single_Allele,1);
      end loop;
    put(" | "); put(integer(Fitness(T)),4);
    put(" | "); put(integer(Org_Value(T)),4);
    This_Gen_Accum_Org := This_Gen_Accum_Org + Org_Value(T);
    put(" | "); put(integer(Calc_Value(T)),4);
    This_Gen_Accum_Calc := This_Gen_Accum_Calc + Calc_Value(T);
    if (Best_Calc_Value < Calc_Value(T)) then
      Best_Calc_Value := Calc_Value(T);
      Best_Generation := Generation;
    end if;
    new_line;
    end loop;
    Change_of_Accum_Calc := This_Gen_Accum_Calc - Last_Gen_Accum_Calc;
    put("Best Fitness :  "); put(integer(Best_Calc_Value));
    put(" in Generation : "); put(Best_Generation); new_line;
    put("Accum. Fitness: "); put(integer(This_Gen_Accum_Calc));
    put(" Last Generation : "); put(integer(Last_Gen_Accum_Calc));
    put(" Change : "); put(integer(Change_of_Accum_Calc)); new_line;
    Last_Gen_Accum_Calc := This_Gen_Accum_Calc;
  end Simple_Display;

end Simple_Genetic_Algorithms;
pragma LIST(OFF);
```

æ

Appendix B

The Package for the AGA

```
pragma LIST(ON);

with text_io; use text_io;
with random; use random;
with fio; use fio;
with iio; use iio;

generic
```

The definition of the variables which must be set before the instantiation looks very simulary to the definition of the SGA package. The only additional variable included is called `MaxCrossPoints`. It is the controlling variable for the fixed point crossover procedure. The programmer has to define the number of points in the string where a crossover is allowed.

```
   MaxAllele            : integer;
   MaxPopSize           : integer;
   MaxCrossPoints       : integer;
```

```
package Advanced_Genetic_Algorithms is
```

First comes a number of definitions of the most important types for the advanced genetic algorithms. These types are :

1. `Sex`: prepared for the implementation of sexuality in the crossover procedure, but in the moment it is not used.
2. `Crossing`: a new type with the possible values `haploid` and `diploid`. The meaning of these values is explained later.
3. `Haploid_Method`: is a type with possible values : `haploid`, if the system has to perform a haploid crossover and `diploid` otherwize.
4. The types `Individuum`, `Population_Array` and `Fitness_Array` are identical to the SGA package.
5. `Sex_Array` containts information of the sex of each individual.
6. `Crossing_Points` containts the possible points of crossover, if the method of crossover is haploid. In the other case it containts the list of dominant genes.

```
  type Sex              is (male,female);
  type Crossing         is (haploid, diploid);
  type Haploid_Method   is (fixed, randomize);
  type Individuum       is array (1..MaxAllele) of integer range 0..1;
  type Population_Array is array (1..MaxPopSize) of Individuum;
  type Fitness_Array    is array (1..MaxPopSize) of float;
  type Sex_Array        is array (1..MaxPopSize) of Sex;
  type Crossing_Points  is array (1..MaxCrossPoints) of integer;
```

The definition of the procedure to initialize the population is identical to the definition in the SGA package.

```
procedure Init_Population (Population : in out Population_Array);
```

The following lines are the definitions of the functions of the advanced genetic algorithms. Most of the parameters are identical to the parameters of the SGAs.

```
procedure Roulette_Wheel (Population     : in out Population_Array;
                          Fitness        : in     Fitness_Array;
                          Generation_Gap : in     integer);
```

The additional parameter at the roulette wheel procedure is the value for the size of the generation gap. If the parameter is equal to 0 the procedure is identical to the roulette wheel procedure of the SGA package.

The crossover procedure looks really different from the one in the SGA package. It contains many more parameters which have to be explained. The parameters of this procedure are :

1. `Population`: the population of the system,
2. `Method`: the method of the crossover. Values of this variable are `haploid` and `diploid`.
3. `Selection`: gives the possibilities for the haploid crossover methods. Possible values for this parameter are `fixed` and `randomized`. If this parameter has the value `randomized` the point of crossover is selected randomly. Otherwise the array `crossing` containts informations about the possible point where a crossover is allowed.
4. `Crossing`: depending on the value of `Method` this variable has two different meanings. The use at a hapoid crossover is explained above. By using the diploid crossover this array contains informations about the dominant gene.
5. `No_of_Crossing_Points`: this variable should contain the same information as the variable `MaxCrossPoints`. But different to the variable

`MaxCrossPoints` this variable does not only reserve the space for the table the pointes of crossover are stored in but gives the real number of points for the crossover.

6. `Generation_Gap`: the generation gap, see above.

```
procedure Crossover      (Population            : in out Population_Array;
                          Method                : in     Crossing;
                          Selection             : in     Haploid_Method;
                          Crossing              : in     Crossing_Points;
                          No_of_Crossing_Points : in     integer;
                          Generation_Gap        : in     integer);
```

The procedure for mutation has the same expansions as the procedure for the roulette wheel has. Therefor it need no more explaination.

```
procedure Mutation       (Population     : in out Population_Array;
                          Mutationquote  : in     float;
                          Generation_Gap : in     integer);
```

The following procedure is identical to the function in the SGA package.

```
procedure Simple_Display (Generation     :        integer;
                          Population     : in     Population_Array;
                          Fitness        : in     Fitness_Array;
                          Org_Value      : in     Fitness_Array;
                          Calc_Value     : in     Fitness_Array);
```

```
end Advanced_Genetic_Algorithms;
```

B.1 The Package Body

The next lines contains the package body. As written in the chapter before it is normally closed as a secret and should not be used but for expansion and improvement.

```
package body Advanced_Genetic_Algorithms is


  Best_Org_Value        : float := 0.0;
  Best_Calc_Value       : float := 0.0;
  Best_Generation       : integer := 0;
  Last_Gen_Accum_Org    : float := 0.0;
  Last_Gen_Accum_Calc   : float := 0.0;
```

The Initialisation Procedure

The following procedure is the procedure to initialize the individuals. The procedure is identical to the procedure in the SGA package.

```
  procedure Init_Population (Population : in out Population_Array) is
    t, x                : integer;
    Single_Individuum   : Individuum;
  begin
    for t in Population'Range loop
      for x in Population(t)'Range loop
        Single_Individuum(x) := random_ND_integer(65535) mod 2;
        if (Single_Individuum(x) > 1) then
          Single_Individuum(x) := 1;
        end if;
      end loop;
      Population(t) := Single_Individuum;
    end loop;
  end Init_Population;
```

The Selection Procedure

Additional to the first roulette wheel implementation this system containts also the mechanism of the generation gap. The individuals of with the best fitness are transfered into the generation gap. After the roulette wheel has finished, the individuals are transfered back into the population. Because the possition of these inidividuals in the population is specified it is very simple for the other procedures to continue with the gap.
\begin{verbatim}

```
  procedure Roulette_Wheel (Population     : in out Population_Array;
                            Fitness        : in     Fitness_Array;
                            Generation_Gap : in     integer) is
    New_Population : Population_Array;
    Gap            : Population_Array;
    Gap_Array      : array (1..MaxPopSize) of integer;
    Wheel_Length   : constant := 100;
    Wheel          : array (1..Wheel_Length) of integer;
    Position       : integer := 1;
    t              : integer;
    x              : integer;
    field          : integer;
    NettoPopSize   : integer;
    Dummy          : integer;

  begin
    NettoPopSize := MaxPopSize - Generation_Gap;
```

The first step of the procedure is to calculate the size of the population without the individuals in the gap.

```
for t in Population'Range loop
  if (Fitness(t) >= 1.0) then
    for x in Position..(Position+integer(Fitness(t))) loop
      if (x <= 100) then
        Wheel(x):=t;
      end if;
    end loop;
    Position := Position + integer(Fitness(t));
  end if;
end loop;
```

The next one transferes the best individuals into the gap. Therefor the system has to sort the fitness.

```
for t in Population'Range loop
  Gap_Array(t) := t;
end loop;
for t in Population'Range loop
  for x in t..MaxPopSize loop
    if (Fitness(Gap_Array(t)) > Fitness(Gap_Array(x))) then
      Dummy        := Gap_Array(t);
      Gap_Array(t) := Gap_Array(x);
      Gap_Array(x) := Dummy;
    end if;
  end loop;
end loop;
for t in 1..Generation_Gap loop
```

```
      Population(NettoPopSize + t) := Population(Gap_Array(t));
    end loop;
```

Start of the selection of the new generation.

```
    for t in 1..NettoPopSize loop
      field := 101;
      while (field > 100) loop
        field := (random_ND_integer(65535) mod Position) + 1;
      end loop;
      New_Population(t) := Population(Wheel(field));
    end loop;
```

Let the new generation take the place of the old one.

```
    for t in 1..NettoPopSize loop
      Population(t) := New_Population(t);
    end loop;
```

This finishes the reproduction phase.

```
  end Roulette_Wheel;
```

The Crossover Procedure

The crossover procedure is the expanded version of the SGA procedure. The interface of this procedure is already explained in the section above.

```
  procedure Crossover (Population            : in out Population_Array;
                       Method                : in     Crossing;
                       Selection             : in     Haploid_Method;
```

```
                   Crossing              : in     Crossing_Points;
                   No_of_Crossing_Points : in     integer;
                   Generation_Gap        : in     integer) is
  New_Population          : Population_Array;
  Partner                 : array (Population'Range) of integer;
  x,y,z                   : integer := 0;
  filled                  : boolean;
  Selected_Partner        : integer := 0;
  Point_of_Crossover      : integer := 0;
  New_1st_Individuum      : Individuum;
  New_2nd_Individuum      : Individuum;
  Dummy                   : Individuum;
  Number_of_Partners      : integer := 0;
  Number_of_Partners_half : integer := 0;
```

This internal function returns the dominant gene. It is only used, if the crossover works in the diploid mode.

```
 function Dominant (x,y : in integer) return integer is
   Position_x : integer := 0;
   Position_y : integer := 0;
   begin
   for t in Crossing'Range loop
     if (Crossing(t) = x) then
       Position_x := t;
     end if;
     if (Crossing(t) = y) then
       Position_y := t;
     end if;
   end loop;
```

```
    if (Position_x > Position_y) then
      return Position_x;
    else
      return Position_y;
    end if;
    end Dominant;

  begin
    Number_of_Partners := MaxPopSize - Generation_Gap;
    Number_of_Partners_half := Number_of_Partners / 2;
-- Initialize the fields for the relationship
    for x in 1..Number_of_Partners loop
      Partner(x) := Population'Length + 1;
    end loop;
-- Select the partners
    for x in 1..Number_of_Partners_half loop
      Selected_Partner := (random_ND_integer(65535) mod Number_of_Partners_half)
                          + Number_of_Partners_half + 1;
      loop
        filled := false;
        for y in 1..x loop
          if (Partner(y) = Selected_Partner) then
            filled := true;
            Selected_Partner :=(random_ND_integer(65535) mod
                                                    Number_of_Partners_half)
                                  + Number_of_Partners_half;
          end if;
        end loop;
        exit when (filled = false);
      end loop;
      Partner(x) := Selected_Partner;
    end loop;
```

If the crossover method is the hapoid crossover, then the procedure has to find point of crossover and start the change of the genetic information. If fixed points for the crossover are defined, then the procedure hat to use these points.

```
  if (Method = haploid) then
    for x in 1..Number_of_Partners_half loop
      New_1st_Individuum := Population(x);
      New_2nd_Individuum := Population(Partner(x));
      if (Selection = randomize) then
        Point_of_Crossover := (random_ND_integer(65535) mod
                               Population(x)'Length);
      else
        Point_of_Crossover := Crossing(random_ND_integer(65535) mod
                               (No_of_Crossing_Points + 1));
      end if;
      if (Point_of_Crossover > 0) then
        Dummy := New_1st_Individuum;
        for z in 1..Point_of_Crossover loop
          New_1st_Individuum(z) := New_2nd_Individuum(z);
          New_2nd_Individuum(z) := Dummy(z);
        end loop;
      end if;
      Population(x) := New_1st_Individuum;
      Population(Partner(x)) := New_2nd_Individuum;
    end loop;
```

Else if the crossover method is diploid, the system has to use to priority table

```
    else
      for x in 1..Number_of_Partners_half loop
        New_1st_Individuum := Population(x);
        New_2nd_Individuum := Population(Partner(x));
        for z in New_1st_Individuum'Range loop
          Dummy(z) := Dominant(New_1st_Individuum(z),New_2nd_Individuum(z));
        end loop;
        Population(x)           := Dummy;
        Population(Partner(x))  := Dummy;
      end loop;
    end if;
```

Final procedures of this phase

```
  end Crossover;
```

The Mutation Procedure

With the exception of the generation gap, this procedure works identically to the function of the SGA. It is important to notify, that most changes in the AGA package belong to the crossover phase. This is an index for the fact, that this phase still gives a lot work to do and that the best algorithm is still not found.

```
  procedure Mutation       (Population     : in out Population_Array;
                            Mutationquote  : in     float;
                            Generation_Gap : in     integer) is
    Number_of_Mutations   : integer := 0;
    Selected              : integer := 0;
    Point_of_Mutation     : integer := 0;
```

```
Mutated_Gene          : integer := 0;
Individuums_to_Mutate : integer := 0;
Mutant                : Individuum;
Quote                 : float;
Correction            : float;

begin
  Individuums_to_Mutate := MaxPopSize - Generation_Gap;
  Quote := Mutationquote;
  if (Mutationquote < 0.001) then
    put_line("Quote of Mutation too small ! Assumed new quote : 0.005 ");
    Quote := 0.005;
  elsif (Mutationquote > 0.999) then
    put_line("Quote of Mutation too big ! Assumed new quote : 0.005 ");
    Quote := 0.005;
  end if;
  Correction := 1000.0 / float(MaxPopSize);
  Number_of_Mutations := integer(Quote * float(MaxPopSize) * Correction);
  if (Number_of_Mutations > 0) then
    Number_of_Mutations := random_ND_integer(65535) mod Number_of_Mutations;
  end if;
  if (Number_of_Mutations > 0) then
    for x in 1..Number_of_Mutations loop
      Selected := (random_ND_integer(65535) mod Individuums_to_Mutate) + 1;
      Point_of_Mutation := (random_ND_integer(65535) mod
                               Population(Selected)'Length) + 1;
      Mutant := Population(Selected);
      Mutated_Gene := Mutant(Point_of_Mutation);
      while (Mutated_Gene = Mutant(Point_of_Mutation)) loop
```

```
            Mutated_Gene := random_ND_integer(65535) mod 2;
          end loop;
          Mutant(Point_of_Mutation) := Mutated_Gene;
          Population(Selected) := Mutant;
        end loop;
      end if;
    end Mutation;
```

The SGA Simple Display Procedure

I wanted to replace this function by a more sophisticating output procedure, but there exists no common ADA graphics interface. To keep the procedure portable I decided to keep the SGA output in use.

```
  procedure Simple_Display(Generation : integer;
                           Population : in Population_Array;
                           Fitness    : in Fitness_Array;
                           Org_Value  : in Fitness_Array;
                           Calc_Value : in Fitness_Array) is
  Single_Allele         : integer;
  Element               : Individuum;
  This_Gen_Accum_Org    : float;
  This_Gen_Accum_Calc   : float;
  Change_of_Accum_Calc  : float;

  begin
    new_page;
    put ("Genetic Algorithms Experimental Environment: Simple Display ");
    put_line ("Procedure");
    new_line;
    put ("Generation :");
    put (Generation);
    new_line;
    for T in Population'Range loop
      Element := Population(T);
      for X in Element'Range loop
        Single_Allele := Element(X);
        put(Single_Allele,1);
      end loop;
    put(" | "); put(integer(Fitness(T)),4);
    put(" | "); put(integer(Org_Value(T)),4);
    This_Gen_Accum_Org := This_Gen_Accum_Org + Org_Value(T);
    put(" | "); put(integer(Calc_Value(T)),4);
    This_Gen_Accum_Calc := This_Gen_Accum_Calc + Calc_Value(T);
    if (Best_Calc_Value < Calc_Value(T)) then
      Best_Calc_Value := Calc_Value(T);
      Best_Generation := Generation;
    end if;
    new_line;
    end loop;
    Change_of_Accum_Calc := This_Gen_Accum_Calc - Last_Gen_Accum_Calc;
    put("Best Fitness :  "); put(integer(Best_Calc_Value));
    put(" in Generation : "); put(Best_Generation); new_line;
    put("Accum. Fitness: "); put(integer(This_Gen_Accum_Calc));
    put(" Last Generation : "); put(integer(Last_Gen_Accum_Calc));
    put(" Change : "); put(integer(Change_of_Accum_Calc)); new_line;
    Last_Gen_Accum_Calc := This_Gen_Accum_Calc;
  end Simple_Display;
```

```
end Advanced_Genetic_Algorithms;
pragma LIST(OFF);
```

This summery of procedures is just a skeleton of possible programming. I am sure, that the efficancy of most of the procedures can be improved. So I want to encourage the user of these packages to do so and use the possibilities of ADA for this work. æ

Appendix C

The Random Number Generator

The programming language has no random number generator included. But this generator is vital for the use of genetic algorithms. The following package is developed to include such a generator into the used packages. It depends on a chapter in [?]. There a full documentation of the functions could be found, the names used here are mostly identical to the names used in this book.

The following module generates random integers and floating point numbers, which are normal and exponential deviated.

```
-- Random number generators in Ada
-- Version 1.0
-- see: W.H.Press, B.P.Flannery, S.A.Teukolsky, W.T.Vetterling
--      Numerical Recipes in C
--      Cambidge, 1988
--      ISBN 0-521-35465-X
--      Chapter 7, Pages 204 - 241

with calendar;
with math_lib;

package random is
-- Function for normal (Gaussian) deviates
  function random_ND return float;
  function random_ND_integer(reach : integer) return integer;
-- Function for Exponential deviates
  function random_ED return float;
  function random_ED_integer(reach : integer) return integer;
end random;

package body random is
-- static variables for the function "ran1"
  M1            : integer := 259200;
  M2            : integer := 134456;
  M3            : integer := 243000;
  IA1           : integer := 7141;
  IA2           : integer := 8121;
  IA3           : integer := 4561;
  IC1           : integer := 54773;
  IC2           : integer := 28411;
  IC3           : integer := 51349;
  RM1           : float   := 0.0000038502;
  RM2           : float   := 0.0000074377;
  x             : integer := 737;
  ix1, ix2, ix3 : integer;
  r             : array (0..98) of float;
  iff           : integer := 0;
  initialized   : boolean := false;
  init_value    : integer := -31415;
-- static variables for the function "ran2"
  M             : integer := 714025;
  IA            : integer := 1366;
  IC            : integer := 150889;
-- static variables for the function "ran3"
  MBIG          : integer := 1000000000;
  MSEED         : integer := 161803398;
  MZ            : integer := 0;
  FAC           : float;
-- static variables for the function "random_ND"
  iset          : integer := 0;
  gset          : float   := 0.0;

  function ran1 return float is
    temp : float;
    j    : integer;
    begin
      if (initialized  = false) then
        ix1 := (IC1 - init_value) mod M1;
        ix1 := (IA1 * ix1 + IC1) mod M1;
        ix2 := ix1 mod M2;
        ix1 := (IA1 * ix1 + IC1) mod M2;
        for j in 1..97 loop
          ix1 := (IA1 * ix1 + IC1) mod M1;
          ix2 := (IA2 * ix2 + IC2) mod M2;
          r(j) := float(ix1 + ix2 * integer(RM2)) * RM1;
        end loop;
        initialized := true;
        end if;
      ix1 := (IA1 * ix1 + IC1) mod M1;
      ix2 := (IA2 * ix2 + IC2) mod M2;
      ix3 := (IA3 * ix3 + IC3) mod M3;
      j   := 1 + ((97 * ix3) / M3);
      temp := r(j);
      r(j) := float(ix1 + ix2 * integer(RM2)) * RM1;
      return temp;
    end ran1;

    function random_ND return float is
      use math_lib;
      fac, r, v1, v2    : float;
      temp              : float;
      begin
      if (iset = 0) then
        loop
          v1 := 2.0 * ran1 -1.0;
          v2 := 2.0 * ran1 -1.0;
          r  := v1 * v1 + v2 * v2;
          exit when (r >= 1.0);
        end loop;
        temp := ln(r)/r;
        if (temp < 0.0) then
          fac  := sqrt(-2.0 * temp);
        else
          fac  := sqrt(2.0 * temp);
        end if;
        gset := v1 * fac;
        iset := 1;
        temp := v2 * fac;
        return temp;
      else
        iset := 0;
        return gset;
      end if;
      end random_ND;

      function random_ND_integer(reach : integer) return integer is
        value           : integer := 0;
        minus_one       : integer := -1;
      begin
        value := integer(random_ND * float(reach));
        if (value < 0) then
          value := (value * minus_one);
        end if;
        return value;
      end random_ND_integer;

      function random_ED return float is
      use math_lib;
        dummy : float := 0.0;
      begin
        dummy := -1.0 * (ln(ran1)/ln(10.0));
```

```
        return dummy;
      end random_ED;

      function random_ED_integer(reach : integer) return integer is
        value           : integer := 0;
        minus_one       : integer := -1;
      begin
        value := integer(random_ED * float(reach));
        if (value < 0) then
          value := (value * minus_one);
        end if;
        return value;
      end random_ED_integer;

-- Initialization of package
use calendar;
begin
  init_value := integer(seconds(clock));
  if (init_value > 0) then
    init_value := -1 * init_value;
  end if;
end random;
```

æ

Bibliography

[Bra91] T.Bratke, J.Gramatzki, T.Wagner, R.Welker: Entwicklung eines Programms zur Stundenplanbelegung mit Hilfe von Genetischen Algorithmen
in: [Joc91]

[Col91] Collins Road Atlas Britian
Edinburgh, 1991

[Dav91] L.Davis: Handbook of Genetic Algorithms
New York, 1991

[Dew85] A.K.Dewdney: Exploring the field of genetic algorithms in a primordial computer sea full of flips
in: Scientific American, November 1985

[Gol89] D.E.Goldberg: Genetic Algorithms in Search, Optimization and Machine Learning
Ann Arbor, 1989

[Gri90] A.Grimm: Genetische Algorithmen
in: [Joc91]

[Gri91a] A.Grimm: Untersuchungen über den Einsatz von Genetischen Algorithmen bei topologischen Optimierungen
in: [Joc91]

[Gri91b] A.Grimm: Experimentelle Untersuchungen über den Einsatz von Genetischen Algorithmen bei Selbstorganisierenden Datenstrukturen
Cologne Polytechnic
Dep. of Computer Science (FB20)
Gummersbach, 1991

[Gue91] C.Günthner: Modifiziertes Tourenproblem am Beispiel einer Telefonkette
in: [Joc91]

[Hof85] D.R.Hofstadter: Gödel, Escher, Bach: ein endlos geflochtenes Band
Stuttgart, 1985

[Hol68] J.H.Holland: Hierarchical description of universal spaces and adaptive systems
in: Technical Report ORA Projects 01252 and 08226
University of Michigan
Department of Computer and Communication Sciences
Ann Arbor, 1968

[Hop88] J.E.Hopcroft, J.D.Ullman: Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie
Bonn, 1988

[Joc91] F.Jochum (Editor): Genetische Algorithmen - Prinzipien, Anwendungen, Experimente
Cologne Polytechnic
Dep. of Computer Science
Gummersbach, 1991

[Kie90] B.Kiesswetter: Experimentelle Untersuchung und Anwendung Genetischer Algorithmen in den Bereichen maschinelles Lernen und Optimierung
Cologne Polytechnic
Dep. of Computer Science
Gummersbach, 1990

[Loe92] K.H.Loeber: Indexierung eines Dokumentenbestandes mit Hilfe Genetischer Algorithmen
Cologne Polytechnic
Dep. of Computer Science
Gummersbach, 1992

[Luc91] C.B.Lucansius, M.J.J.Blommers, L.M.C.Buydens, G.Kateman: A Genetic Algorithm for Conformational Analysis of DNA
in: [Dav91], Kap.18, S.251 ff.

[Moo74] A.M.Mood, F.A.Graybill, D.C.Boes: Introduction to the Theory of Statistics
Auckland, 1974

[Pre89] W.H.Press, B.P.Flannery, et al.:Numerical Recipes
The Art of Scientific Computing
(C Version)
Cambridge, 1989

[Wir85] N.Wirth, K.Jensen: Pascal
User Manual and Report
New York, 1985

æ