

NEAR EAST UNIVERSITY

Faculty of Engineering

Department of Computer Engineering

GENETIC ALGORITHM BASED ON OPTIMIZATION

**Graduation Project
COM- 400**

Student: Shahid Islam Butt (970944)

Supervisor: Asst. Prof. Dr Rahib Abiyev

Lefkoşa — 2001

ACKNOWLEDGEMENTS

First of all I am happy that Allah Taa'la the Almighty Supreme Being and Hazrat Muhammad (peace be upon him) has provided me with the strength and courage to complete the task, and also I am grateful especially to Mr. Ghulam Ahmad in my life who have supported me, advised me, taught me and who have always encouraged me to follow my dreams and ambitions. This Project is dedicated to My Dearest Parents, My younger brother and youngest sisters. They have taught me that no dream is unachieveable.

I wish to thank my advisor. Assistant Professor Dr. Rahib Abiyev for intellectual support, encouragement, and enthusiasm, which made this project possible, and his patience for correcting both my stylistic and scientific errors.

My sincerest thanks must go to my friends especially to Mr. Jamal Ashiq Qureshi , Mr. Ammar Ali, Mr. Naveed Mustafa, Mr. Babar Rehman, Mr. Hafiz Zulfiqar Ali and Mr. Awais Janjua who shared their suggestions and evaluations throughout the completion of the project and in my graduation.

I thank Allah Taa'la for giving me courage and strength to achieve aims and objectives of the life.

TABLE OF CONTENTS

ACKNOWLEDGEMENT	i
TABLE OF CONTENTS	ii
ABSTRACT	iv
INTRODUCTION OF GENETIC ALGORITHMS	v
1. State of understanding genetic algorithms for solving optimization problem.	1
1.1 – Introduction	1
1.1.1 – First words	1
1.1.2 – History	1
1.2 – Chromosome	1
1.2.1 - Reproduction	2
1.3 – Search space	2
1.3.1 – Search space	2
1.3.2 – NP – Hard Problem	3
1.4 – Basic Genetic Algorithms	4
1.4.1 – Basic of Genetic Algorithm	4
1.5 – The Genetic Operators	6
1.6 – Genetic Algorithms based on Optimization	14
1.6.1 – Optimization based on Genetic Algorithm	14
1.6.2 – Genetic Algorithm Structural Optimization	15
1.7 – Genetic Algorithm	16
1.7.1 – Basic Description	16
1.7.2 – Outline of the Basic Genetic Algorithm	17
1.8 – Operators of GA	18
1.8.1– Encoding of a chromosome	18
1.8.2 – Crossover	19
1.8.3– Mutation	20
1.9 – Parameters of Genetic Algorithm	20
1.9.1 – Crossover and Mutation Probability	20
1.9.2 – Other Parameters	21
1.10 – Selection	21
1.10.1 – Roulette Wheel Selection	21
1.10.2 – Rank Selection	22
1.10.3 – Steady-State Selection	23
1.10.4 – Elitism	24
1.11 – Encoding	24
1.11.1 – Binary encoding	24
1.11.2 – Permutation encoding	25
1.11.3 – Value encoding	26
1.11.4 – Tree encoding	26
1.12 – Crossover and Mutation	27

1.12.1 - Binary Encoding	28
1.12.1.1 - Crossover	28
1.12.1.2 - Mutation	29
1.12.2 - Permutation Encoding	29
1.12.2.1 - Crossover	29
1.12.2.2 - Mutation	29
1.12.3 - Value Encoding	30
1.12.3.1 -Crossover	30
1.12.3.2 - Mutation	30
1.12.4 - Tree Encoding	30
1.12.4.1 - Crossover	30
1.12.4.2 - Mutation	30
1.13 - Traveling Salesman Problem	31
1.13.1 - About the problem	31
1.13.2 - Implementation	31
1.13.3 - Crossover	31
1.13.4 - Mutation	31
1.13.5 - Encoding	31
1.14 - Recommendations	32
1.14.1 - Parameters of GA	32
1.14.2 - Applications of GA	33
2. Genetic Algorithms from Models	35
2.1 - Introduction	35
2.2 - The main ingredients of simple trading models	36
2.3 - Trading model indicators	37
2.4 - Operations on indicators	40
2.5 - Risk- sensitive performance measure	41
2.6 - Trading model optimization	42
2.7 - The genetic algorithm to find and optimize simple trading model	43
2.8 - Genetic algorithms with sharing scheme for multi-modal functions	45
2.9 - Modified sharing function for robust optimization	48
3. Non-linear Optimization	51
3.1 - Non-linear optimization	51
3.1.1 - The Sequential Quadratic Programming Algorithm	53
3.1.2 - Augmented Langrangian Algorithm	57
3.1.3 - Reduced Gradient Algorithm	58
3.1.4 - Feasible Sequential Quadratic Programming Algorithm	60
CONCLUSION	61
REFERENCES	62
APPENDIX	64

ABSTRACT

Due to the complexity of the processes, it has become very difficult to control them on the base of traditional methods. In such condition it is necessary to use modern methods for solving these problems. One of such method is global optimization algorithm based on mechanics of natural selection and natural genetics, which is called Genetic Algorithms. In this project the application problems of genetic algorithms for optimization problems, its specific characters and structures are given. The Basic Genetic operations, Selection, Reproduction, Crossover, Encoding and Mutation operations are widely described. The effectivity of genetic algorithms for solving the genetic algorithms are shown in the following chapters. After the representation of optimization problems, structural optimization and the finding of optimal solution of quadratic equation are given.

The practical application for selection, reproduction, crossover, and mutation operation are shown. The functional implementation of GA based optimization in MATLAB Toolbox is considered.

INTRODUCTION

The GENETIC ALGORITHMS is a model of machine learning, which derives its behavior from a metaphor of the processes of EVOLUTION in nature. This is done by the creation within a machine of a POPULATION of INDIVIDUALS represented by CHROMOSOMES, in essence of a set of character strings that are analogous to the base-4 chromosomes that we see in our own DNA. The individuals in the population then go through a process of evolution.

Genetic algorithms (GA) seek to solve optimization problems using the methods of evolution, specifically survival of the fittest. In a typical optimization problem, there are a number of variables, which control the process, and a formula or algorithm, which combines the variables to fully model the process. The problem is then to find the values of the variables, which optimize the model in some way. If the model is a formula, then we will usually be seeking the maximum or minimum value of the formula. There are many mathematical methods, which can optimize problems of this nature (and very quickly) for fairly "well-behaved" problems. These traditional methods tend to break down when the problem is not so "well-behaved." We should note that EVOLUTION (in nature or anywhere else) is not a purposive or directed process. That is, there is no evidence to support the assertion that the goal of evolution is to produce Mankind. Indeed, the processes of nature seem to boil down to different Individuals competing for resources in the ENVIRONMENT. Some are better than others, those that are better are more likely to survive and propagate their genetic material. In nature, we see that the encoding for our genetic information (GENOME) is done in a way that admits asexual REPRODUCTION (such as by budding) typically results in OFFSPRING that are genetically identical to the PARENT. Sexual REPRODUCTION allows the creation of genetically radically different offspring that are still having the same general flavor (SPECIES). At the molecular level what occurs (wild over simplification alert) is that a pair of Chromosomes bump into one another, exchange chunks of genetic information and drift apart. This is the RECOMBINATION operation, which GA errors generally

refer to as CROSSOVER because of the way that genetic material crosses over from one chromosome to another,

The CROSSOVER operation happens in an ENVIRONMENT where the ELECTION of who gets to mate is a function of the FITNESS of the INDIVIDUAL, i.e. how good the individual is at competing in its environment. Some GENETIC ALGORITHMS use a simple function of the fitness measure to select individuals (probabilistically) to undergo genetic operations such as crossover or asexual REPRODUCTION (the propagation of genetic material unaltered). This is fitness-proportionate selection. Other implementations use a model in which certain randomly selected individuals in a subgroup compete and the fittest is selected. This is called tournament selection and is the form of selection we see in nature when stags rut to vie for the privilege of mating with a herd of hinds. The two processes that most contribute to evolution are crossover and fitness based on reproduction.

As it turns out, there are mathematical proofs that indicate that the process of FITNESS proportionate REPRODUCTION is, in fact, near optimal in some senses, MUTATION also plays a role in this process, although how important its role is continues to be a matter of debate (some refer to it as a background operator, while others view it as playing the dominant role in the evolutionary process). It cannot be stressed too strongly that the GENETIC ALGORITHM (as a SIMULATION of a genetic process) is not a random search for a solution to a problem (highly fit INDIVIDUAL). The genetic algorithm uses stochastic processes, but the result is distinctly non-random (better than random) GENETIC ALGORITHMS are used for a number of different application areas.

An example of this would be multidimensional OPTIMIZATION problems in which the character string of the CHROMOSOME can be used to encode the values for the different parameters being optimized. In practice, therefore, we can implement this genetic model of computation by having arrays of bits or characters to represent the CHROMOSOME. Simple bit manipulation operations allow the implementation of CROSSOVER, MUTATION and other operations. Although a substantial amount of research has been performed on variable-length strings and other structures, the majority

of work with GENETIC ALGORITHM is focused on fixed-length character strings. We should focus on both this aspect of fixed-lengthness and the need to encode the representation of the solution being sought as a character string, since these are crucial aspects that distinguish GENETIC PROGRAMMING, which does not have a fixed length representation and there is typically no encoding of the problem,

When the GENETIC ALGORITHM is implemented it is usually done in a manner that involves the following cycle: Evaluate the FITNESS of all of the INDIVIDUALS in the POPULATION. Create a new population by performing operations such as CROSSOVER, fitness-proportionate REPRODUCTION and MUTATION on the individuals whose fitness has just been measured. Discard the old population and iterate using the new population.

One iteration of this loop is referred to as a GENERATION. There is no theoretical reason for this as an implementation model. Indeed, we do not see this punctuated behavior in POPULATIONS in nature as a whole, but it is a convenient implementation model.

The first GENERATION (generation 0) of this process operates on a POPULATION of randomly generated INDIVIDUALS. From there on, the genetic operations, in concert with the FITNESS measure, operate to improve the population.

Description:

Genetic algorithms use a vocabulary borrowed from natural genetics, a candidate solution is called an individual. Quite often this individual called also truing or chromosome. This might be a little bit misleading; each cell of every organism of a given species carries a certain number of chromosomes, however, we talk about one-chromosome individuals only. Chromosomes are made of units genes arranged in linear succession; every gene controls the inheritance of one or several characters

Each gene can assume a finite number of values, called alleys (feature values). In binary representation chromosome is a vector, consisting of the bits succession, i.e. the

succession of zeroes and ones. A set of chromosomes makes a population. A number of chromosomes in population define a population size. The genetic algorithm evaluates a population and generates a new one iteratively, with each successive population referred to as a generation. The population undergoes a simulated evolution, at each generation the relatively "good" solutions reproduce while the relatively "bad" solutions die. To distinguish between different solutions we use an objective (evaluation) function, which plays the role of an environment. Quite often the objective function is called also fitness function.

CHAPTER ONE

STATE OF ART UNDERSTANDING GENETIC ALGORITHMS FOR SOLVING OPTIMIZATION PROBLEMS

1.1.1 - First Words

Genetic algorithms are a part of evolutionary computing, which is a rapidly growing area of artificial intelligence. As you can guess, genetic algorithms are inspired by Darwin's theory about evolution. Simply said, solution to a problem solved by genetic algorithms is evolved.

1.1.2 - History

Idea of evolutionary computing was introduced in the 1960s by I. Rechenberg in his work "Evolution strategies" (Evolution strategy in original). His idea was then developed by other researchers. Genetic Algorithms (GA) were invented by John Holland and developed by him and his students and colleagues. This lead to Holland's book "Adaption in Natural and Artificial Systems" published in 1975.

In 1992 John Koza has used genetic algorithm to evolve programs to perform certain tasks. He called his method "genetic programming" (GP). LISP programs were used, because programs in this language can expressed in the form of a "parse tree", which is the object the GA works on.

1.2 – Chromosome

All living organisms consist of cells. In each cell there is the same set of chromosomes. Chromosomes are strings of DNA and serves as a model for the whole organism. A chromosome consist of genes, blocks of DNA. Each gene encodes a particular protein. Basically can be said, that each gene encodes a trait, for example color of eyes.

Possible settings for a trait (e.g. blue, brown) are called alleles. Each gene has its own position in the chromosome. This position is called locus.

Complete set of genetic material (all chromosomes) is called genome. Particular set of genes in genome is called genotype. The genotype is with later development after birth base for the organism's phenotype, its physical and mental characteristics, such as eye color, intelligence etc.

1.2.1 – Reproduction

During reproduction, first occurs recombination (or crossover). Genes from parents form in some way the whole new chromosome. The new created offspring can then be mutated. Mutation means, that the elements of DNA are a bit changed. This changes are mainly caused by errors in copying genes from parents.

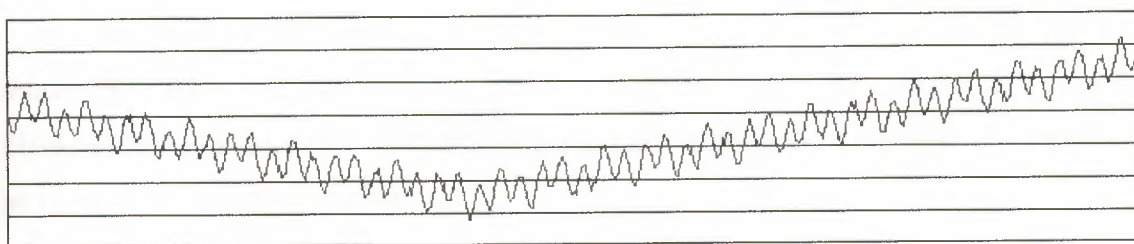
The fitness of an organism is measured by success of the organism in its life.

1.3 – Search Space

1.3.1 – Search Space

If we are solving some problem, we are usually looking for some solution, which will be the best among others. The space of all feasible solutions (it means objects among those the desired solution is) is called search space (also state space). Each point in the search space represents one feasible solution. Each feasible solution can be "marked" by its value or fitness for the problem. We are looking for our solution, which is one point (or more) among feasible solutions, that is one point in the search space.

The looking for a solution is then equal to a looking for some extreme (minimum or maximum) in the search space. The search space can be whole known by the time of solving a problem, but usually we know only a few points from it and we are generating other points as the process of finding solution continues.



Example of a search space

The problem is that the search can be very complicated. One does not know where to look for the solution and where to start. There are many methods, how to find some suitable solution (i.e. not necessarily the best solution), for example hill climbing, tabu search, simulated annealing and genetic algorithm. The solution found by these methods is often considered as a good solution, because it is not often possible to prove what is the real optimum.

1.3.2 – NP-hard Problems

Examples of difficult problems, which cannot be solved in the “traditional” way, are NP problems.

There are many tasks for which we know fast (polynomial) algorithms. There are also some problems that are not possible to be solved algorithmically. For some problems was proved that they are not solvable in polynomial time.

But there are many important tasks, for which it is very difficult to find a solution, but once we have it, it is easy to check the solution. This fact led to NP-complete problems. NP stands for no deterministic polynomial and it means that it is possible to “guess” the solution (by some no deterministic algorithm) and then check it, both in polynomial time. If we had a machine that can guess, we would be able to find a solution in some reasonable time.

Studying of NP-complete problems is for simplicity restricted to the problems, where the answer can be yes or no. Because there are tasks with complicated outputs, a class of problems called NP-hard problems has been introduced. This class is not as limited as class of NP-complete problems.

For NP-problems is characteristic that some simple algorithm to find a solution is obvious at a first sight just trying all possible solutions. But this algorithm is very slow (usually $O(2^n)$) and even for a bit bigger instances of the problems it is not usable at all.

Today nobody knows if some faster exact algorithm exists. Proving or disproving this remains as a big task for new researchers (and maybe you). Today many people think, that such an algorithm does not exist and so they are looking for some alternative methods, example of these methods are genetic algorithms.

Examples of the NP problems are satisfiability problem, traveling salesman problem or knapsack problem. Compendium of NP problems is available.

1.4 - Basic Genetic Algorithms

1.4.1 - Basics Of Genetic Algorithms

The three most important aspects of using genetic algorithms are:

- (1) Definition of the objective function.
- (2) Definition and implementation of the genetic representation.
- (3) Definition and implementation of the genetic operators. Once these three have been defined.

The generic genetic algorithm should work fairly well. Beyond that you can try many different variations to improve performance, find multiple optima (species -if they exist, or parallels the algorithms.

Algorithm GA is

// Start with an initial time

T: =0;

// Initialize a usually random population of individuals

Initpopulation P (t);

// Evaluate fitness of all initial individuals of population

Evaluate P (t);

// Test for termination criterion (time, fitness, etc.)

While not done do

// Increase the time counter

T: = t+ 1,

// Select a sub-population for offspring production

P': = select parents P (t);

// Recombine the "genes" of selected parents

Recombine P' (t);

// Perturb the mated population stochastically

Mutate P' (t),

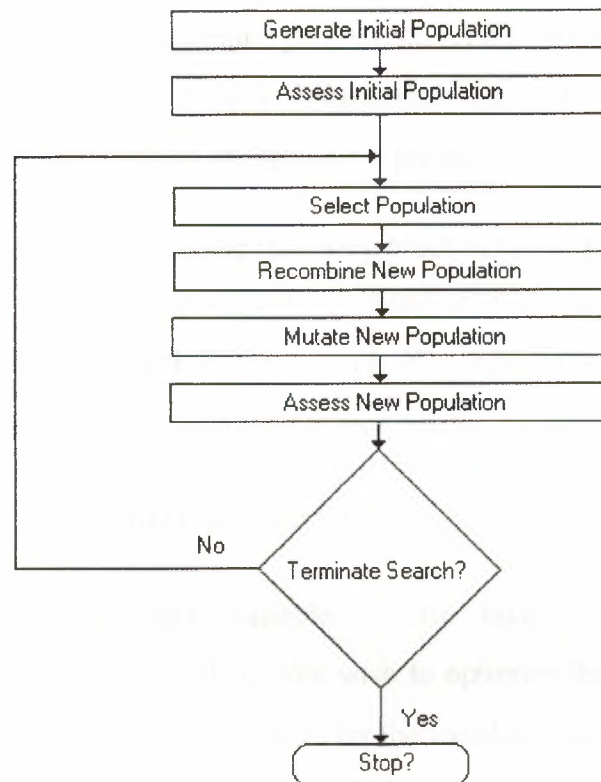
// Evaluate it's new fitness evaluate P' (t);

// Select the survivors from actual fitness

$P := \text{survive } P, P'(t);$

od

End GA.



1.5 -The Genetic Operators

The initial population is chosen at random. GA simulates genetic evolution of a population of tentative solutions (individuals) by means of selection and survival of the fittest, crossover and mutations. Every individual is typically represented as a bit sequence, which makes up its "genetic code". The function to be optimized provides "fitness" values. The structure of a simple genetic algorithm is the same as the structure of any convolution program. During iteration t , a genetic algorithm maintains a population of potential

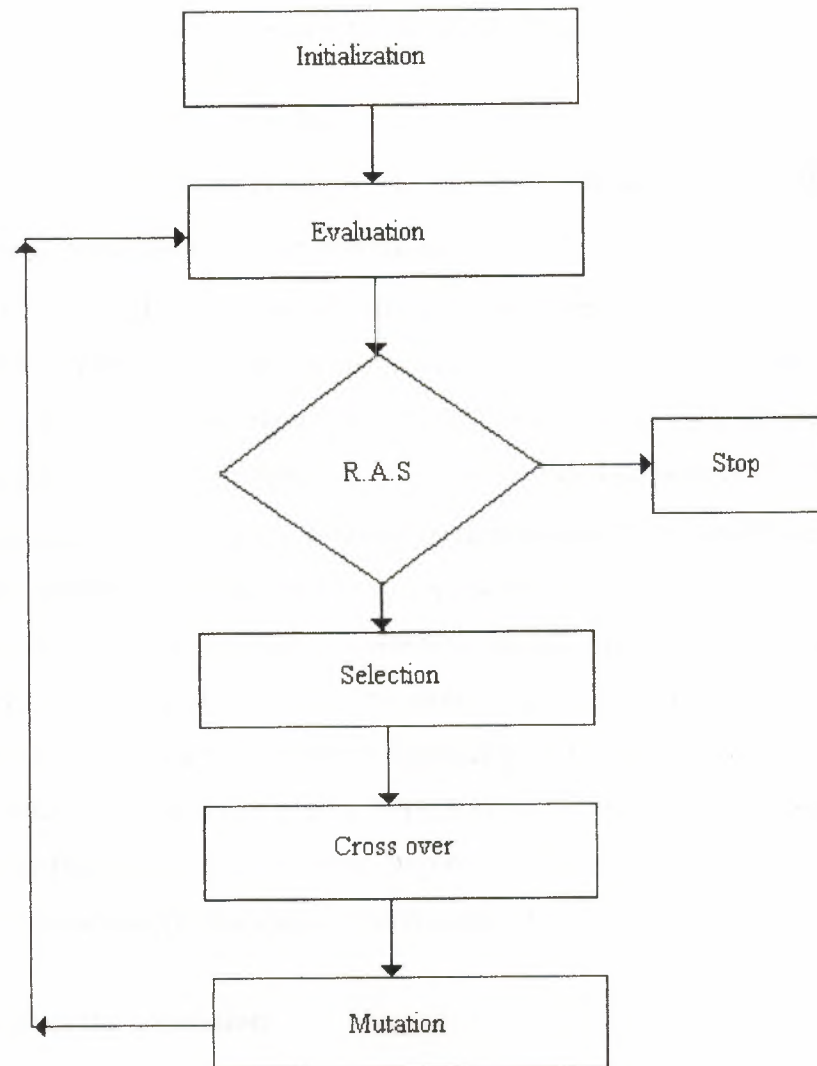
solutions (chromosomes, vectors), $G(t) = \{x'_1, \dots, x'_n\}$, each solution x'_i is evaluated to give some measure of its "fitness". Then, a new population (iteration $t+1$) is formed by selecting the more fit individuals. Some members of this new population undergo reproduction by means of crossover and mutation, to form new solutions. Crossover combines the features of two parent chromosomes to form two similar offspring by swapping corresponding segments of the parents- For example, if the parents are represented by five-dimensional vectors $(a_1, b_1, c_1, d_1, e_1)$ and $(a_2, b_2, c_2, d_2, e_2)$, then crossing the chromosomes after the second gene would produce the offspring $(a_1, b_1, c_2, d_2, e_2)$ and $(a_2, b_2, c_1, d_1, e_1)$. Mutation arbitrarily after one or more genes of a selected chromosome, by a random change with a probability equal to the mutation rate.

For concrete problem GA has the following block-schema. We discuss the actions of a genetic algorithm for a simple parameter optimization problem. Now suppose we wish to maximize a function of k variables, $f(x_1, \dots, x_k): R^k \rightarrow R$. If the optimization problem is to minimize a function f , this is equivalent to maximizing a function g , where $g = -f$, i.e.

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

Suppose further that each variable x can take values from a domain $D_i = [a_i, b_i] \subseteq \mathbb{R}$ and $f(x) \neq 0$ for all x_i . We wish to optimize the function f with some required precision; suppose six decimal places for the variables values are desirable.

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



STRUCTURE OF SIMPLE GENETIC ALGORITHM

potential solution) is represented by a binary string of length $m = \sum_{i=1}^k m_i$, the first m_1 bits map into a value from the range $[a_1, b_1]$, the next group of m_2 bits map into a value from the range $[a_2, b_2]$ and so on, the last group of m_k bits map into a value from the range $[a_k, b_k]$. To initialize a population, we can simply set some p_s number of chromosomes randomly in a bit wise fashion. However, if we have some knowledge about the distribution of potential optima, we may use such information in arranging the set of initial (potential) solutions. The rest of the algorithm is straightforward, in each generation we evaluate each chromosome (using the function f on the decoded sequences of variables), select new population with respect to the probability distribution based on fitness values, and recombine the chromosomes in the new population by mutation and crossover operators. After some number of generations, when no further improvement is observed, the best chromosome represents an (possibly the global) optimal solution. Often we stop the algorithm after a fixed number of iterations depending on speed and resource criteria. For the selection process (selection of a new population with respect to the probability distribution based on fitness values), we must implement the following actions at first, Calculate the fitness value $eval(v_i)$ for each chromosome $v_i (i = 1, \dots, p_s)$.

- Find the total fitness of the population

$$F = \sum_{i=1}^{p_s} eval(v_i)$$

- Calculate the probability of a selection p_n^2 for each chromosome $v_i (i = 1, \dots, p_s)$:

$$p_n^i = eval(v_i) / F$$

- Calculate a cumulative probability p_{cum}^i for each chromosome $v_i (i = 1, \dots, p_s)$:

$$p_{cum}^i = \sum_{j=1}^i p_n^j$$

The selection process is implemented p_s times; each time we select a single chromosome for a new population in the following way:

- Generate a random (float) number r from the range $[0,1]$,
- If $r < p_{cum}^i$ then select the first chromosome (v_1); otherwise select the $I - t_j$ chromosome v_i ($2 \leq i \leq p_s$) such that

$$p_{cum}^{i-1} < r < p_{cum}^i$$

Obviously, some chromosomes would be selected more than once; the best chromosomes get more copies; the average stay even, and the worst die off. Now we are ready to apply the first recombination operator, crossover, to the individuals in the new population. One of the parameters of a genetic system is probability of crossover p_c . This probability gives us the expected number $p_c p_s$ of chromosomes which undergo the crossover operation. We proceed in the following way:

For each chromosome in the (new) population:

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

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

$$(b_1 b_2 \dots b_{pos} b_{pos+1} \dots b_m)$$

$$(c_1 c_2 \dots c_{pos} c_{pos+1} \dots c_m)$$

are replaced by a pair of their offspring:

$$(b_1 b_2 \dots b_{pos} c_{pos+1} \dots c_m)$$

$$(c_1 c_2 \dots c_{pos} b_{pos+1} \dots b_m)$$

The intuition behind the applicability of the crossover operator is information exchange between different potential solutions.

The next recombination operator, mutation, is performed on a bit-by-bit basis.

Another parameter of the genetic system, probability of mutation p_m , gives us the expected number of mutated bits $p_m \cdot m \cdot p_s$. Every bit (in all chromosomes in the whole population) has an equal chance to undergo mutation i.e. change from 0 to 1d of vice versa. So we proceed in the following way.

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

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

The intuition behind the mutation operator is the introduction of some extra variability into the population.

Following selection, crossover, and mutation, the new population is ready for its next evaluation. This evaluation is used to build the probability distribution (for the next selection process). The rest of evolution is just cyclic repetition of the above steps.

However, as it frequently occurs, in earlier generations the fitness values of some chromosomes are better than the value of the best chromosome after a finite number of generations.

It is necessary to note, that classical GA may employ roulette wheel method for selection, which is a stochastic version of the survival of the fittest mechanism. In this method of selection, candidate strings from the current generation $G(t)$ are selected to survive to the next generation $G(t+1)$ by designing a roulette wheel where each string in the population is represented on the wheel in proportion to its fitness value. Thus those strings, which have a high fitness, are given a large share of the wheel, while those strings with low fitness are given a relatively small portion of the roulette wheel. Finally, spinning the roulette wheel p_s times and accepting as candidates those strings, which are indicated at the completion of the spin, make selections,

Example 1.5: As an example, Suppose $p_s=5$, and consider the following initial population of strings;

$G(0) = \{(10110), (11000), (11110), (01001), (00110)\}$, For each string v_i , in the population, the fitness may be evaluated: $eval(v_i)$. The appropriate share of the roulette wheel to allot the i -th string is obtained by dividing the fitness of the i -th string by the sum of the fitnesses of the entire population:

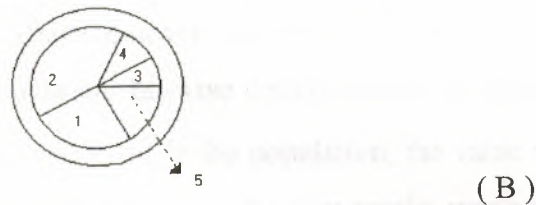
$$\frac{eval(v_i)}{\sum_{i=1}^{p_s} eval(v_i)}$$

Figure (1.5) shows a listing of the population with associated fitness values and the corresponding roulette wheel.

To compute the next population of strings, the roulette wheel is spun five times [3]. The strings chosen by this method of selection, though, are only candidate strings for the next population. Before actually being copied into the new population, these strings must undergo crossover and mutation.

String Fitness Relative (A)

V_i	$eval(v_i)$	Fitness
V_1	10110 2.23	0.14
V_2	11000 7.27	0.47
V_3	11110 1.05	0.07
V_4	01001 3.35	0.21
V_5	00110 1.69	0.11



In figure (A) listing of the five-string population and the associated fitness values,

(b) Corresponding roulette wheel for string selection.

The integers shown on the roulette wheel correspond to string labels,

110101	110101	110100
100100	100100	100101

- (a) (b) (c)

An example (figure) of a crossover for two 6-bit strings,

- (a) Two strings are selected for crossover.
- (b) A crossover site is selected at random. In this case, $k=4$,
- (c) Now swap the two strings after the k -th bit.

Pairs of the p_s (assume p_s even) candidate strings, which have survived selection, are next chosen for crossover, which is a recombination mechanism. The probability that the crossover operator is applied will be denoted by p_c . Pairs of string are selected randomly from $G(t)$, without replacement, for crossover. A random integer k , called the crossing site, is chosen from $\{1, 2, \dots, m-1\}$, and then the bits from the two chosen strings are swapped after the k -th bit with a probability p_c . This process is repeated until $G(t)$ is empty. For example, Figure 11.3. Illustrates a crossover for two 6-bit strings. In this case, the crossing site k is 4, so the bits from the two strings are swapped after the fourth bit.

Finally, after crossover, mutation is applied to the candidate strings. The mutation operator is a stochastic bit-wise complementation applied with uniform probability p_m . That is, for each single bit in the population, the value of the bit is nipped from 0 to 1 or from 1 to 0 with probability p_m . As an example, suppose $p_m=0.1$, and the string $v=11100$ is to undergo mutation. The easiest way to determine which bits, if any, to flip is to choose a uniform random number $r \in [0,1]$ for each bit in the string. If $r \leq p_m$, then the bit is flipped; otherwise, no action is taken. For the string v above, suppose the random numbers (0.91, 0.43, 0.03, 0.67, 0.29) were generated, and then the resulting mutation is shown below. In this case, the third bit was flipped.

Before mutation: 11100

After mutation: 11000

After mutation, the candidate strings are copied into the new population of strings $G(t+1)$, and the whole process is repeated [4].

1.6 - Genetic Algorithms based on Optimization

1.6.1 - Optimization based on Genetic Algorithms

Genetic algorithms were formally introduced in the United States in the 1970s by John Holland at University of Michigan. The continuing price/performance improvements of computational systems have made them attractive for some types of optimization. In particular, genetic algorithms work very well on mixed (continuous *and* discrete), combinatorial problems. They are less susceptible to getting 'stuck' at local optima than gradient search methods. But they tend to be computationally expensive.

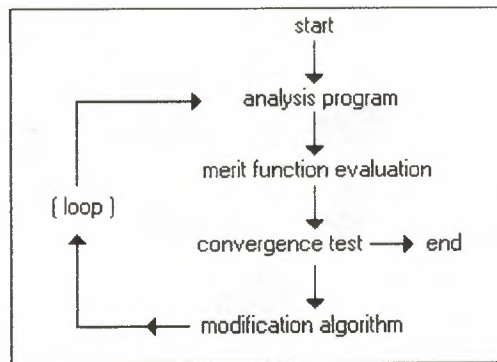
To use a genetic algorithm, you must represent a solution to your problem as a genome (or chromosome). The genetic algorithm then creates a population of solutions and applies genetic operators such as mutation and crossover to evolve the solutions in order to find the best one.

This presentation outlines some of the basics of genetic algorithms. The three most important aspects of using genetic algorithms are:

- (1) Definition of the objective function.
- (2) Definition and implementation of the genetic representation.
- (3) Definition and implementation of the genetic operators, Once these three have been defined, the generic genetic algorithm should work fairly well. Beyond that you can try many different variations to improve performance, find multiple optima (species - if they exist), or parallelism the algorithms.

Genetic algorithm (GA) uses the principles of evolution, natural selection, and genetics from natural biological systems in a computer algorithm to simulate evolution.

Essentially, the genetic algorithm is an optimization technique that performs a parallel, stochastic, but directed search to evolve the most fit population. In this section we will introduce the genetic algorithm and explain how it can be used for design of fuzzy systems. The genetic algorithm borrows ideas from and attempts to simulate Darwin's theory on natural selection and Mendel's work in genetics on inheritance. The genetic algorithm is an optimization technique that evaluates more than one area of the search space and can discover more than one solution to a problem. In particular, it provides a stochastic optimization method where if it "gets stuck" at a local optimum, it tries to simultaneously find other parts of the search space and jump out" of the local optimum to a global one.



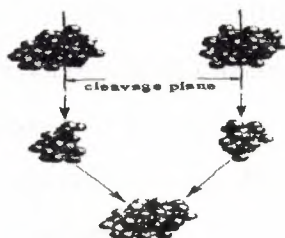
1.6.1 Characteristics common to all optimizers

1.6.2 - Genetic Algorithm Structural Optimization

Atomistic models of materials can provide accurate total energies. For problems where the structures are not known, however, discovering the lowest energy geometry is difficult. This is particularly true for atomic clusters, whose structure may vary dramatically with a small change in the number of atoms. For this type of problem, the number of possible stable structures increases exponentially fast with the number of atoms. Furthermore, there is considerable experimental difficulty in determining the structure of an atomic cluster. We have been able to address this problem using a novel approach to applying genetic algorithms. The Darwinian evolution process inspires these algorithms. A

population of structures is maintained, and "mating" structures and selecting out the lowest energy geometries produce new generations,

The key to a successful genetic algorithm is to design a mating process that allows for the good parts of the parent structures to be inherited by the next generation. Such a process allows for efficient searching of the possible stable structures. A poor mating algorithm is no better than a random search. We have designed a new mating process, depicted at left. Two structures are chosen as "parent" structures, Each one is divided into two halves by a cleavage plane. A new structure is generated by connecting half of each parent into a new cluster, followed by atomic relaxation to a local minimum. We have successfully applied our "cut and paste" approach to a number of challenging problems, including: (12).



1.7 – Genetic Algorithm

1.7.1 – Basic Description

Genetic algorithms are inspired by Darwin's theory about evolution. Solution to a problem solved by genetic algorithms is evolved.

Algorithm is started with a set of solutions (represented by chromosomes) called population. Solutions from one population are taken and used to form a new population. This is motivated by a hope, that the new population will be better than the old one. Solutions which are selected to form new solutions (offspring) are selected according to their fitness, the more suitable they are the more chances they have to reproduce.

This is repeated until some condition (for example number of populations or improvement of the best solution) is satisfied.

1.7.2 – Outline of the Basic Genetic Algorithm

1. [Start] Generate random population of n chromosomes (suitable solutions for the problem)
2. [Fitness] Evaluate the fitness $f(x)$ of each chromosome x in the population
3. [New population] Create a new population by repeating following steps until the new population is complete
 - a. [Selection] Select two parent chromosomes from a population according to their fitness (the better fitness, the bigger chance to be selected)
 - b. [Crossover] With a crossover probability cross over the parents to form a new offspring (children). If no crossover was performed, offspring is an exact copy of parents.
 - c. [Mutation] With a mutation probability mutate new offspring at each locus (position in chromosome).
 - d. [Accepting] Place new offspring in a new population
4. [Replace] Use new generated population for a further run of algorithm
5. [Test] If the end condition is satisfied, stop, and return the best solution in current population
6. [Loop] Go to step 2

Some Comments:

As you can see, the outline of Basic GA is very general. There are many things that can be implemented differently in various problems.

First question is how to create chromosomes, what type of encoding choose. With this is connected crossover and mutation; the two basic operators of GA. Encoding, crossover and mutation are introduced in next chapter.

Next questions are how to select parents for crossover. This can be done in many ways, but the main idea is to select the better parents (in hope that the better parents will produce better offspring). Also you may think, that making new population only by new offspring can cause lost of the best chromosome from the last population. This is true, so so called elitism is often used. This means, that at least one best solution is copied without changes to a new population, so the best solution found can survive to end of run.

Maybe you are wandering, why genetic algorithms do work. It can be partially explained by Schema Theorem (Holland), however, this theorem has been criticized in recent time. If you want to know more, check other resources.

1.8 - Operators of GA

As you can see from the genetic algorithm, the crossover and mutation are the most important part of the genetic algorithm. The performance is influenced mainly by these two operators. Before we can explain more about crossover and mutation, some information about chromosomes will be given.

1.8.1 - Encoding of a Chromosome

The chromosome should in some way contain information about solution that it represents. The most used way of encoding is a binary string. The chromosome then could look like this:

Chromosome 1	1101100100110110
Chromosome 2	1101111000011110

Each chromosome has one binary string. Each bit in this string can represent some characteristic of the solution. Or the whole string can represent a number - this has been used in the basic GA.

Of course, there are many other ways of encoding. This depends mainly on the solved problem. For example, one can encode directly integer or real numbers, sometimes it is useful to encode some permutations and so on.

1.8.2 - Crossover

After we have decided what encoding we will use, we can make a step to crossover. Crossover selects genes from parent chromosomes and creates a new offspring. The simplest way how to do this is to choose randomly some crossover point and everything before this point copy from a first parent and then everything after a crossover point copy from the second parent.

Crossover can then look like this (| is the crossover point):

Chromosome 1	11011		00100110110
Chromosome 2	11011		11000011110
Offspring 1	11011		11000011110
Offspring 2	11011		00100110110

There are other ways to make crossover, for example we can choose more crossover points. Crossover can be rather complicated and very depends on encoding of the encoding of chromosome. Specific crossover made for a specific problem can improve performance of the genetic algorithm.

1.8.3 - Mutation

After a crossover is performed, mutation takes place. This is to prevent falling all solutions in population into a local optimum of solved problem. Mutation changes randomly the new offspring. For binary encoding we can switch a few randomly chosen bits from 1 to 0 or from 0 to 1. Mutation can then be following:

Original offspring 1	1101111000011110
Original offspring 2	1101100100110110
Mutated offspring 1	1100111000011110
Mutated offspring 2	1101101100110110

The mutation depends on the encoding as well as the crossover. For example when we are encoding permutations, mutation could be exchanging two genes.

1.9 - Parameters of Genetic Algorithms

1.9.1 - Crossover and Mutation Probability

There are two basic parameters of GA - crossover probability and mutation probability.

Crossover probability says how often will be crossover performed. If there is no crossover, offspring is exact copy of parents. If there is a crossover, offspring is made from parts of parents' chromosome. If crossover probability is 100%, then all offspring is made by crossover. If it is 0%, whole new generation is made from exact copies of chromosomes from old population (but this does not mean that the new generation is the same!). Crossover is made in hope that new chromosomes will have good parts of old chromosomes and maybe the new chromosomes will be better. However it is good to leave some part of population survive to next generation.

Mutation probability says how often will be parts of chromosome mutated. If there is no mutation, offspring is taken after crossover (or copy) without any change. If mutation is performed, part of chromosome is changed. If mutation probability is 100%, whole

chromosome is changed, if it is 0%, nothing is changed. Mutation is made to prevent falling GA into local extreme, but it should not occur very often, because then GA will in fact change to random search.

1.9.2 - Other Parameters

There are also some other parameters of GA. One also important parameter is population size.

Population size says how many chromosomes are in population (in one generation). If there are too few chromosomes, GA have a few possibilities to perform crossover and only a small part of search space is explored. On the other hand, if there are too many chromosomes, GA slows down. Research shows that after some limit (which depends mainly on encoding and the problem) it is not useful to increase population size, because it does not make solving the problem faster.

1.10 - Selection

As you already know from the Genetic algorithm outline, chromosomes are selected from the population to be parents to crossover. The problem is how to select these chromosomes. According to Darwin's evolution theory the best ones should survive and create new offspring. There are many methods how to select the best chromosomes, for example roulette wheel selection, Boltzman selection, tournament selection, rank selection, steady state selection and some others.

1.10.1 - Roulette Wheel Selection

Parents are selected according to their fitness. The better the chromosomes are, the more chances to be selected they have. Imagine a roulette wheel where are placed all chromosomes in the population, every has its place big accordingly to its fitness function, like on the following picture.



Then a marble is thrown there and selects the chromosome. Chromosome with biggest fitness will be selected more times.

Following algorithm can simulate this.

- [Sum] Calculate sum of all chromosome fitness's in population - sum S .
- [Select] Generate random number from interval $(0, S) - r$.
- [Loop] Go through the population and sum fitness's from 0 - sum s . When the sum s is greater than r , stop and return the chromosome where you are.

Of course, step 1 is performed only once for each population.

1.10.2 - Rank Selection

The previous selection will have problems when the fitness's differs very much. For example, if the best chromosome fitness is 90% of the entire roulette wheel then the other chromosomes will have very few chances to be selected.

Rank selection first ranks the population and then every chromosome receives fitness from this ranking. The worst will have fitness 1, second worst 2 etc. and the best will have fitness N (number of chromosomes in population).

You can see in following picture, how the situation changes after changing fitness to order number.





Situation before ranking (graph of fitness's)



Situation after ranking (graph of order numbers)

After this all the chromosomes have a chance to be selected. But this method can lead to slower convergence, because the best chromosomes do not differ so much from other ones.

1.10.3 - Steady-State Selection

This is not particular method of selecting parents. Main idea of this selection is that big part of chromosomes should survive to next generation.

GA then works in a following way. In every generation is selected a few (good - with high fitness) chromosomes for creating a new offspring. Then some (bad - with low

fitness) chromosomes are removed and the new offspring is placed in their place. The rest of population survives to new generation.

1.10.4 - Elitism

Idea of elitism has been already introduced. When creating new population by crossover and mutation, we have a big chance, that we will loose the best chromosome.

Elitism is name of method, which first copies the best chromosome (or a few best chromosomes) to new population. The rest is done in classical way. Elitism can very rapidly increase performance of GA, because it prevents losing the best found solution.

1.11 - Encoding

Encoding of chromosomes is one of the problems, when you are starting to solve problem with GA. Encoding very depends on the problem. In this section will be introduced some encodings, which have been already used with some success.

1.11.1 - Binary Encoding

Binary encoding is the most common, mainly because first works about GA used this type of encoding.

In binary encoding every chromosome is a string of bits, 0 or 1.

Chromosome A	101100101100101011100101
Chromosome B	111111100000110000011111

Example of chromosomes with binary encoding

Binary encoding gives many possible chromosomes even with a small number of alleles. On the other hand, this encoding is often not natural for many problems and sometimes corrections must be made after crossover and/or mutation.

Example of Problem: Knapsack problem

The problem: There are things with given value and size. The knapsack has given capacity. Select things to maximize the value of things in knapsack, but do not extend knapsack capacity.

Encoding: Each bit says, if the corresponding thing is in knapsack.

1.11.2 - Permutation Encoding

Permutation encoding can be used in ordering problems, such as travelling salesman problem or task ordering problem.

In permutation encoding, every chromosome is a string of numbers, which represents number in a sequence.

Chromosome A	1 5 3 2 6 4 7 9 8
Chromosome B	8 5 6 7 2 3 1 4 9

Example of chromosomes with permutation encoding

Permutation encoding is only useful for ordering problems. Even for this problems for some types of crossover and mutation corrections must be made to leave the chromosome consistent (i.e. have real sequence in it).

Example of Problem: Travelling salesman problem (TSP)

The problem: There are cities and given distances between them. Travelling salesman has to visit all of them, but he does not to travel very much. Find a

sequence of cities to minimize travelled distance.

Encoding: Chromosome says order of cities, in which salesman will visit them.

1.11.3 - Value Encoding

Direct value encoding can be used in problems, where some complicated values, such as real numbers, are used. Use of binary encoding for this type of problems would be very difficult.

In value encoding, every chromosome is a string of some values. Values can be anything connected to problem, form numbers, real numbers or chars to some complicated objects.

Chromosome A	1.2324 5.3243 0.4556 2.3293 2.4545
Chromosome B	ABDJEIFJDHDIERJFDLDFLFEGT
Chromosome C	(back), (back), (right), (forward), (left)

Example of chromosomes with value encoding

Value encoding is very good for some special problems. On the other hand, for this encoding is often necessary to develop some new crossover and mutation specific for the problem.

Example of Problem: Finding weights for neural network

the problem: There is some neural network with given architecture. Find weights for inputs of neurons to train the network for wanted output.

Encoding: Real values in chromosomes represent corresponding weights for inputs.

1.11.4 - Tree Encoding

Tree encoding is used mainly for evolving programs or expressions, for genetic programming.

In tree encoding every chromosome is a tree of some objects, such as functions or commands in programming language.

Chromosome A	Chromosome B
<pre> graph TD A((+)) --- B((x)) A --- C((/)) C --- D((5)) C --- E((y)) </pre>	<pre> graph TD A[do until] --- B[step] A --- C[wall] </pre>
$(+ x (/ 5 y))$	$(\text{do until step wall})$

Example of chromosomes with tree encoding.

Tree encoding is good for evolving programs. Programming language LISP is often used to this, because programs in it are represented in this form and can be easily parsed as a tree, so the crossover and mutation can be done relatively easily.

Example of Problem: Finding a function from given values
the problem: Some input and output values are given. Task is to find a function, which will give the best (closest to wanted) output to all inputs.

Encoding: Chromosome is a function represented in a tree.

1.12 - Crossover and Mutation

Crossover and mutation are two basic operators of GA. Performance of GA very depends on them. Type and implementation of operators depends on encoding and also on a problem. There are many ways how to do crossover and mutation. There are only some examples and suggestions how to do it for several encoding.

1.12.1 - Binary Encoding

1.12.1.1 - Crossover

Single point crossover - one crossover point is selected, binary string from beginning of chromosome to the crossover point is copied from one parent, the rest is copied from the second parent .



$$11001011 + 11011111 = 11001111$$

Two point crossover - two crossover point are selected, binary string from beginning of chromosome to the first crossover point is copied from one parent, the part from the first to the second crossover point is copied from the second parent and the rest is copied from the first parent



$$11001011 + 11011111 = 11011111$$

Uniform crossover - bits are randomly copied from the first or from the second parent



$$11001011 + 11011101 = 11011111$$

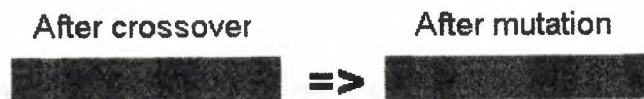
Arithmetic crossover - some arithmetic operation is performed to make a new offspring



$$11001011 + 11011111 = 11001001 \text{ (AND)}$$

1.12.1.2 - Mutation

Bit inversion - selected bits are inverted



$$11001001 \Rightarrow 10001001$$

1.12.2 - Permutation Encoding

1.12.2.1 - Crossover

Single point crossover - one crossover point is selected, till this point the permutation is copied from the first parent, then the second parent is scanned and if the number is not yet in the offspring it is added

Note: there are more ways how to produce the rest after crossover point

$$(1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9) + (4\ 5\ 3\ 6\ 8\ 9\ 7\ 2\ 1) = (1\ 2\ 3\ 4\ 5\ 6\ 8\ 9\ 7)$$

1.12.2.2 - Mutation

Order changing - two numbers are selected and exchanged

(1 2 3 4 5 6 **8** 9 7) => (1 **8** 3 4 5 6 **2** 9 7)

1.12.3 - Value Encoding

1.12.3.1 - Crossover

All crossovers from binary encoding can be used

1.12.3.2 - Mutation

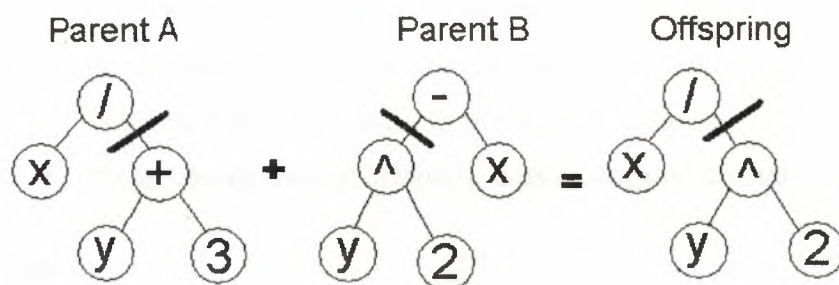
Adding a small number (for real value encoding) - to selected values is added (or subtracted) a small number

(1.29 5.68 **2.86** **4.11** 5.55) => (1.29 5.68 **2.73** **4.22** 5.55)

1.12.4 - Tree Encoding

1.12.4.1 - Crossover

Tree crossover - in both parent one crossover point is selected, parents are divided in that point and exchange part below crossover point to produce new offspring



1.12.4.2 - Mutation

Changing operator, number - selected nodes are changed

1.13 - Traveling Salesman Problem

1.13.1 - About the Problem

Travelling salesman problem (TSP) has been already mentioned in one of the previous chapters. To repeat it, there are cities and given distances between them. Travelling salesman has to visit all of them, but he does not to travel very much. Task is to find a sequence of cities to minimize travelled distance. In other words, find a minimal Hamiltonian tour in a complete graph of N nodes.

1.13.2 - Implementation

Population of 16 chromosomes is used. For encoding these chromosome permutation encoding is used about encoding you can find, how to encode permutation of cities for TSP. TSP is solved on complete graph (i.e. each node is connected to each other) with euclidian distances. Note that after adding and deleting city it is necessary to create new chromosomes and restart whole genetic algorithm.

You can select crossover and mutation type. I will describe what they mean.

1.13.3 - Crossover

- One point - part of the first parent is copied and the rest is taken in the same order as in the second parent
- Two point - two parts of the first parent are copied and the rest between is taken in the same order as in the second parent
- None - no crossover, offspring is exact copy of parents

1.13.4 - Mutation

- Normal random - a few cities are chosen and exchanged
- Random, only improving - a few cities are randomly chosen and exchanged only if they improve solution (increase fitness)

- Systematic, only improving - cities are systematically chosen and exchanged only if they improve solution (increase fitness)
- Random improving - the same as "random, only improving", but before this is "normal random" mutation performed
- Systematic improving - the same as "systematic, only improving", but before this is "normal random" mutation performed
- None - no mutation

1.13.5 - Encoding

Chromosomes are functions represented in a tree.

1.14 - Recommendations

1.14.1 - Parameters of GA

This chapter should give you some basic recommendations if you have decided to implement your genetic algorithm. These recommendations are very general. Probably you will want to experiment with your own GA for specific problem, because today there is no general theory which would describe parameters of GA for *any* problem.

Recommendations are often results of some empiric studies of GAs, which were often performed only on binary encoding.

✓ **Cross-over-rate**

Crossover rate generally should be high, about 80%-95%. (However some results show that for some problems crossover rate about 60% is the best.)

✓ **Mutation-rate**

On the other side, mutation rate should be very low. Best rates reported are about 0.5%-1%.

✓ **Population-size**

It may be surprising, that very big population size usually does not improve

performance of GA (in meaning of speed of finding solution). Good population size is about 20-30, however sometimes sizes 50-100 are reported as best. Some research also shows, that best population size depends on encoding, on size of encoded string. It means, if you have chromosome with 32 bits, the population should be say 32, but surely two times more than the best population size for chromosome with 16 bits.

✓ **Selection**

Basic roulette wheel selection can be used, but sometimes rank selection can be better. There are also some more sophisticated method, which changes parameters of selection during run of GA. Basically they behaves like simulated annealing. But surely elitism should be used (if you do not use other method for saving the best found solution). You can also try steady state selection.

✓ **Encoding**

Encoding depends on the problem and also on the size of instance of the problem. Operators depend on encoding and on the problem.

1.14.2 - Applications of GA

Genetic algorithms have been used for difficult problems (such as NP-hard problems), for machine learning and also for evolving simple programs. They have been also used for some art, for evolving pictures and music.

Advantage of GAs is in their parallelism. GA is travelling in a search space with more individuals (and with genotype rather than phenotype) so they are less likely to get stuck in a local extreme like some other methods.

They are also easy to implement. Once you have some GA, you just have to write new chromosome (just one object) to solve another problem. With the same encoding you just change the fitness function and it is all. On the other hand, choosing encoding and fitness function can be difficult.

Disadvantage of GAs is in their computational time. They can be slower than some other methods. But with today's computers it is not so big problem.

To get an idea about problems solved by GA, here is a short list of some applications:

- ❑ Nonlinear dynamical systems - predicting, data analysis
- ❑ Designing neural networks, both architecture and weights
- ❑ Robot trajectory
- ❑ Evolving LISP programs (genetic programming)
- ❑ Strategy planning
- ❑ Finding shape of protein molecules
- ❑ TSP and sequence scheduling
- ❑ Functions for creating images

CHAPTER TWO

GENETIC ALGORITHMS FROM MODELS

2.1 - Introduction

Trading models are algorithms proposing trading recommendations for financial assets. In our approach we limit this definition to a set of rules based on past financial data. The financial data, which are typically series of prices, enter the trading model in the form of indicators corresponding to various kinds of averages. Although progress has been made in understanding financial markets, there is no definitive prescription on how to build a successful trading model and how to define the indicators. Automatic search and optimization techniques can be considered when addressing this problem.

However, optimizing trading models for financial assets without overfitting is a very difficult task because the scientific understanding of financial markets is still very limited. Overfitting means building the indicators to fit a set of past data so well that they are no longer of general value: instead of modeling the principles underlying the price movements, they model the specific movements observed during a particular time period. Such a model usually exhibits a different behavior (or may fail to trade successfully at all) when tested out-of-sample. This difficulty is related to the fact that many financial time series do not show stability of their statistical behavior over time especially when they are analyzed intra-daily.

To minimize over fitting during optimization, the optimization process must include the following important ingredients:

- a good measure of the trading model performance,
- indicator evaluation for different time series,
- large data samples,
- a robust optimization technique,

- last but not least strict testing procedures.

The new element we want to present in this paper is a way to automatize the search for improved trading models. Genetic algorithms offer a promising approach for addressing such problems. Genetic algorithms consider a population of possible solutions to a given problem and evolve it according to mechanisms borrowed from natural genetic evolution: reproduction and selection. The criterion for selecting an individual is based on its fitness to the environment, or more precisely, to the quality of the solution it bears. A possible solution is coded as a gene, which is formally the data structure containing the values of the quantities characterizing the solutions.


In the framework of the present application, a gene will contain the indicator parameters, for example time horizons and a weighting function for the past, and also the type of operations used to combine them. The fitness function will be based on the return obtained when following the recommendations of a given trading model.

2.2 - The main ingredients of simple trading models

In this section, we review the different ingredients that constitute the basis of a trading model and reformulate them in terms of simple quantities that can be used in conjunction with a genetic algorithm. Real trading models can be quite complicated and may require many different rules that also depend on the model's own trading history. Here we limit ourselves to simple models that depend essentially on a set of indicators that are pure functions of the price history or of the current return. The purpose of this simplification is to make model coding and representation issues easier so we can study indicator behavior.

The basic rule of a simple trend-following trading model is

$$\text{IF } |I| > K \text{ THEN } G: = \text{sign}(I) \text{ ELSE } G: = 0$$

Where I is an indicator whose sign and value give the direction and strength of the current trend. The constant K is a break level and G , called the gearing, is the recommended position of the model such that, $long = +1$, $short = -1$ and $neutral = 0$. 

In this study we investigate how to construct and select good indicators or combinations of indicators in order to make such simple trading models robust (i.e. so that they perform well out-of-sample).

More complex models can be developed afterwards combining such simple trading models or using more complex rules. For instance to introduce a simple contrarian strategy the previous rule can be modified as

$$\text{IF } |I| > K \text{ THEN } G: = \text{sign}(I) * \text{sign}(L - S) \text{ ELSE } G: = 0$$

Where S is an indicator that gives the strategy (trend following if $S < L$ or contrarian if $(S > L)$, and the constant L is the overbought/oversold break level.

2.3 - Trading model indicators

Indicators are variables of the trading system algorithm whose values, together with the system rules, determine the trading decision process. In various papers we gave different descriptions of indicators that have been used in conjunction with trading models. Here, we focus on some abstract classification of these indicators in order to combine them sensibly with the genetic algorithm. For the time being, the indicators we use are function of the time series itself. In a later stage, we can envisage using as indicators for a particular time series a function of other time series, like, for instance, interest rate functions for studying FX-rates.

First, we define two general classes of indicators. The symmetric I_s and the antisymmetric I_a indicators:

$$I_s(X) = I_s(-X) \text{ and } I_a(X) = -I_a(-X) \quad (2.1)$$

Where X is the basic variable used to define the indicator. This variable is generally a function the logarithm of price X but can be also a function the current return of the model T_c . A typical antisymmetric indicator is a momentum of the logarithm of price itself ($I_a(X) = X$ where $X = x$). A typical symmetric indicator would be a measure of the volatility. The simplest one is the absolute price change ($I_s(X) = |X|$ where $X = x(t_j) - x(t_j - \Delta t)$). The two classes will be used differently in the trading model. The antisymmetric indicators are the ones that provide the dealing signal while the symmetric indicators *modulate* it. For instance, they may forbid the model to trade or may modulate the threshold values. An indicator using the current return can be used for programming stop losses or stop profit, or to compute the risk of an open position.

Indicators are characterized by several parameters. We first describe the parameters that are common to both classes of indicators. Any indicator of the sort described here is composed of moving averages (MA) of different types so the first parameter is the *range* Δt_r of the moving average. The second is the weighting function of the past. As linear combinations of repeated applications of EMAs have much useful properties we use here a weighting function with two parameters defined as

$$MA_{X,j,n}(\Delta t_r, t) \equiv EMA_X^{(j,n)}(\Delta t_r, t) = \frac{1}{n+1-j} \sum_{i=j}^n EMA_X^{(i)}(\Delta t_r, t) \quad (2.2)$$

Where we have two additional parameters: j and n , with $1 \leq j \leq n$.

The quantity $EMA_X^{(i)}$ is the i th application of the EMA operator, i.e.

$$EMA_X^{\{i\}} = EMA(EMA_X^{(i-1)}(\Delta t_r, t)) \quad (2.3)$$

Where $EMA_X^{(1)}$ is computed with the formula

$$\text{EMA}_X(\Delta t_r, t) = \frac{1}{\Delta t_r} \int_{-\infty}^t X(t') e^{-\frac{t-t'}{\Delta t_r}} dt' \quad (2.4)$$

And $\text{EMA}_X^{(0)} \equiv X$. With this definition, $\text{MA}_{X,j,n}$ can model a wide variety of moving averages of the series $X(t)$.

In addition to a moving average, an indicator may be a momentum of various orders. The simple momentum (of order 0) is defined as:

$$m_{X,j,n}(t) \equiv X(t) - \text{MA}_{X,j,n}(t), \quad (2.5)$$

The concept of momentum can be extended to the *first* momentum which is a difference of two moving averages with different ranges, and the *second* momentum which is a linear combination of three moving averages with different ranges with the property that this combination is equal to zero for a straight line; it indicates the overall curvature of the series for a certain depth in the past. To avoid the introduction of too many additional parameters in the indicators, we restrict the possible variation of the *order* of the momentum parameter to only three possible values 0, 1 and 2.

Because of the construction of some input variable X , mainly used in the computation of symmetric indicators, the number of parameters in the problem may be extended. For instance, the volatility has two parameters, the price change time interval and the sample period on which the volatility is computed.

In order to be able to combine different indicators and obtain similar results for different FX rate time series we need to normalize each indicator. To obtain this normalization we divide the value of the indicator by the square root of a long term moving average of the squared values of the indicator. Such normalization is also very useful in order to make the indicators more adaptive to the market changes.

2.4 - Operations on indicators

The operations on the indicators are essentially the four mathematical operations: $+$, $-$, $*$, $/$. In order to generate sensible trading models these operations must be performed following a set of rules:

- In section 2, we saw that the trading model takes a position according to an antisymmetric indicator I and a symmetric or antisymmetric strategy indicator S . To limit the size of the function space and prevent the generation of very complex functions, the operations $*$ and $/$ are restricted to operate between a symmetric and an antisymmetric indicator and the operations $+$ and $-$ are restricted to operate between indicators of the same type.
- For all the operations the problem of normalization is present. In the case of multiplications or divisions scaling is not necessary because we have already constructed the indicators so that they are of order one. In the case of additions and subtractions the resulting indicator need to be renormalized.
- Division should only be used with a modified value of the indicator in order to avoid division by zero. One would never divide by I but by $I + \text{sign}(I) * \varepsilon$ where ε is a small positive constant. This constant can be chosen to be always the same if the indicators are well normalized, for instance 0.0001.
- We shall assume that the *number of operations* allowed is limited. We limit this number to three at first.

We have deliberately left out a large number of possible operations; square roots, power functions or any log or exp functions. This is for the sake of simplicity and also because we think that the present set is already wide enough to produce interesting results. These additional operations can be included at a later stage.

2.5 - Risk-sensitive performance measure

We wrote in the introduction that optimizing trading models is difficult because of the noise present in the data and the risk of overfitting. The challenge is to find indicators that are robust in the sense of being smooth and giving consistent results out-of-sample.

The first step is to define a value describing trading model performance in order to minimize the overfitting in the in-sample period and allowing us to compare different trading models against each other. The profit made by the model could be this fitness function but such a measure does not take into account the risk assumed by the model. As risk is a major concern of investors it is necessary to add a risk component to the fitness function. We use a risk-sensitive performance measure of the trading model that was developed for the optimization of FX real-time trading models. This performance measure, called X_{eff} , is defined as:

$$X_{eff} = \bar{R} - \frac{C}{2} \sigma^2 \quad (2.6)$$

Where R is the annualized average total return, $C(C>0)$ is a risk aversion constant and σ^2 is the variance of the the total return curve against time, where a steady linear growth of the total return represents the zero variance case. Because the variance $\sigma^2 \geq 0$, we have $X_{eff} \leq \bar{R}$. While the total return may mask considerable risk introduced by a high return volatility, the effective return is risk-sensitive: the higher the volatility of return the lower the effective return. In other words, high effective returns indicate highly stable returns.

A good approach to obtain more robust indicators is to test each new indicator simultaneously on different exchange rate time series (since our experience has shown that trading models are robust if they work simultaneously on different rates without changing parameters). This increases the number of possible situations tested by the model in sample. The performance measure of the model is then given by the average of the

X_{eff} performance measure obtained for each time series corrected by the variance of the different X_{eff} values.

$$X'_{eff} = \bar{X}_{eff} - \frac{\sigma X_{eff}}{3} \quad (2.7)$$

This correction decreases the probability of obtaining indicators that have a good average overall but which vary strongly from one time series to another.

2.6 -Trading model optimization

To optimize and test our trading models we split the available historical data into three different periods. The first period is used to build-up the indicators, the second one is for the optimization of the trading model parameters and the third is for selecting the best trading models. The build-up period generally contains more than ten years of daily data, which is used to update long-term indicators. The end of this period is 12 December 1986 for all the exchange rates. As many exchange rate time series do not show stability of their statistical behavior over time, the rest of the historical data, from 1 January 1987 to 30 June 1994, is divided into different alternating periods of in-sample and out-of-sample data. By this subdivision of the in-sample period we may test a larger variety of statistical behaviors and we increase the probability of getting more robust and up-to-date parameters. The size of each in-sample period must be large enough to obtain statistically valid performance measures. Here we use in-sample periods with a size of one and a half years. The performance measure of the model is then given by the X_{eff} performance measure obtained from all in-sample periods.

The optimization process involves thousands of simulation runs through the in-sample data. To obtain results in a reasonable amount of time we extract and use only hourly data (equally spaced in the business time scale). This choice of data sampling produces trading models with results, which are very similar to the ones obtained with the full high frequency time series. To select the best parameter set we need an algorithm

which explores the parameter space in an efficient way and chooses solutions that correspond not only to the largest average effective return but also lie on broad peaks or high flat regions of the effective return function. Such a requirement will ensure that small variations in the model parameters keep the system in the high performance region in terms of X_{eff} .

Genetic algorithms (GA) have been shown to be useful in the optimization of multimodel functions in highly complex landscapes, especially when the function does not have an analytic description and is noisy or discontinuous. The usefulness of genetic algorithms for such problems comes from their evolutionary, adaptive capabilities. When given a measure of the fitness (performance) of a particular solution and a population of adequately coded feasible solutions, the GA is able to search many regions of the parameter space simultaneously. In the GA, better than average solutions are sampled more frequently and thus, through the genetic operations of crossover and mutation, new promising solutions are generated and the average fitness of the whole population improves over time. Although GAs is not guaranteed to find the global optimum, they tend to converge towards good regions of high fitness. This is all we need, since global optima may be of little use in our application. In the next section we will first describe a "naive" genetic algorithm approach to our problem, pointing out its drawbacks and how they can be circumvented.

2.7 - The genetic algorithm to find and optimize simple trading models

For our first attempt, we used a classical genetic algorithm with the following features:

- Each gene is represented as an array of real numbers. In such a representation, each element of the array can be used to store parameters of different types (Boolean, integer and real values). Except the Boolean types that can take only the values 0 or 1, all the other parameters can be selected from a given list of possible values or varied in a given range.

- The population of a few tens of individuals is initialized at random, although seeding the initial population with reselected individuals is also possible. The selection of individuals for reproduction is fitness-proportionate, sometimes called roulette wheel selection. We use here a two-point crossover and mutation. For the mutation, we pick randomly a value inside the allowed range (list). In the crossover, the corresponding elements of the two genes to be modified are either simply exchanged or computed by linear interpolation. The probabilities of crossover and mutation are respectively 0.9 and 0.08.
- We use generational replacement with elitism and no duplicates. At each new generation the major part of the population of parents is replaced by their offspring's. Only a limited number of the best individuals (the elite) are kept unchanged. The elite rate is generally of the order of 5% of the population size. We also eliminate all duplicate individuals to maintain population diversity and to avoid useless and time-consuming fitness evaluations.

The trading model evaluation program does the fitness evaluation. This is a lengthy process since each gene (trading model) runs through a long time series of prices. In fact, this phase accounts for most of the total computing time. Because the different parameters are directly stored in the gene, the evaluation program is able to translate straightforwardly the content of the gene into the trading model to test. If a given gene corresponds to an invalid trading model the evaluation program returns the minimum possible fitness. Such a gene is eliminated in the construction of the next generation. The number of invalid individuals is usually very small.

To speed up this process, all genes of a given generation are evaluated in parallel over several machines of a workstation network. This has been done so far with a special-purpose network job queuing system. In future versions, we plan to use the more portable PVM (Parallel Virtual Machine) system, which will allow us to take advantage of several different platforms in a transparent way. In this distributed computing approach special attention must be paid to fault-recovery and check pointing issues.

When running the above genetic algorithm, the performance of the selected model on the data set used in the learning process (in-sample) was excellent. The behavior of these models on the test data set (out-of-sample) was, however, not satisfactory. This is a very common phenomenon known as overfitting that plagues most real-world data-driven processes. As explained in the introduction, there are many techniques to try to avoid overfitting. In this case, it seems that the GA described above is itself partly responsible for the poor generalization capabilities. In fact, the population invariably converges after while on one high-fitness, but often unstable, peak. In the next section we present modifications to the standard genetic algorithms that allow simultaneous searching for different high-fitness solutions. It will show that a judicious selection among the "best" solutions helps reducing the overfitting problem.

2.8 - Genetic algorithms with sharing scheme for multi-modal functions

In the context of genetic algorithms, optimizing multi-modal functions has been investigated using methods inspired from the natural notions of niche and species. The general goal is to be able to create and maintain several subpopulations, ideally one for each major peak of the fitness function, instead of having the whole population converge to a single global optimum.

Goldberg and Richardson proposed one of the best methods. The idea is that the GA perception of the fitness function is changed in such a way that when individuals tend to concentrate around a high peak, the fitness there is reduced by a factor proportional to the number of individuals in the region. This has the effect of diminishing the attractiveness of the peak and allowing parts of the population to concentrate on other regions. This effective fitness of an individual i , called the shared fitness S_f is given by

$$S_f(i) = \frac{f(i)}{m(i)} \quad (2.8)$$

Where $f(i)$ is the original fitness and $m(i)$ is called the niche count. For an individual i , the quantity $m(i)$ is calculated by summing the sharing function values sh contributed by all N individuals of the population:

$$m(i) = \sum_{j=1}^N sh(d_{ij}) \quad (2.9)$$

Where d_{ij} is the distance between two individuals i and j and

$$sh(d_{ij}) = \begin{cases} 1 - \left(\frac{d_{ij}}{\sigma_n} \right)^\alpha & \text{if } d_{ij} < \sigma_n, \\ 0 & \text{otherwise} \end{cases} \quad (2.10)$$

The quantities α and σ_s are constants.

A difficulty of this method is to choose an adequate value of σ_s as this requires prior knowledge about the number of peaks in the solution space. In our economic applications as well as in many realistic problems, this information is not readily available.

A new method is proposed in based on a different sharing scheme and using an adaptive cluster methodology. The authors show that this method is effective at revealing unknown multi-modal function structures and is able to maintain subpopulation diversity. This method establishes analogies between clusters and niches in the following way: the GA population is divided, by McQueen's adaptive KMEAN clustering algorithm, into K clusters of individuals that correspond to K niches. The shared fitness calculation is the same as in the classical sharing method, but the niche count $m(i)$ is no longer associated with σ_s . In this case the number of individuals in the cluster to which the individual i belongs plays a central role in the niche count calculation. As the number of clusters is associated with the number of niches (peaks), the individuals are put into a single partition of K clusters, where K is not fixed a prior i , but determined by the algorithm itself.

Therefore no a priori knowledge about the number of peaks of the fitness function is required as in the classical sharing method. The niche count $m(i)$ is computed as:

$$m(i) = N_c - N_c * \left(\frac{d_{ic}}{2D_{\max}} \right)^\alpha \quad x_i \in C_c \quad (2.11)$$

Where N_c is the number of individuals in the cluster c , α is a constant, d_{ic} is the distance between the individual i and the centroid of its niche. The algorithm requires a distance metric in order to compute the distance between two clusters and the distance between one individual and one cluster. Two clusters are merged if the distance between their centroids is smaller than a threshold parameter D_{\min} . Moreover, when an individual is further away than a maximum distance D_{\max} from all existing cluster centroids, a new cluster is formed with this individual as a member. The efficiency of the algorithm is improved by sorting the population in descending order according to the individual's fitness before the application of the clustering.

Such a standard genetic algorithm with sharing and clustering has been applied to standard multi-modal and continuous fitness functions with good results. One example of a more complex application is the determination of the optimum parameters of the *business* time scale that is used for analyzing price history and computing indicators. In this example, the optimization is quite difficult because we have to optimize simultaneously 17 parameters and the function to optimize is non-linear in some of the parameters. To solve this problem it was necessary to normalize the parameter space for the genetic algorithm, i.e. each parameter is only allowed to vary in the range [0,1]. In simple problems the two clustering parameters are generally set to $D_{\min} = 0.15$ and $D_{\max} = 0.15$. But here, because of the high dimensionality of the parameter space, the values of the clustering parameters D_{\min} and D_{\max} must be much larger. In this case, the two parameters are multiplied by \sqrt{n} where n is the number of parameters to be optimized. The results obtained with this genetic algorithm are very good and the sharing and clustering methods clearly increased

the speed of convergence compared to the simple genetic algorithm described in the previous section.

When applied to the indicator optimization problem the genetic algorithm with sharing and clustering runs into difficulties. If the fitness landscape contains too many sharp peaks of high fitness all the selected clusters concentrate around these peaks and the genetic algorithm is unable to find robust solutions. In the next section, we propose some modifications to the genetic algorithm to detect clusters in the parameter space that correspond to more general and robust solutions.

2.9 - Modified sharing function for robust optimizations

We need to find a new genetic algorithm that avoids the concentration of many individuals around sharp peaks of high fitness but detects broad regions of the parameter space that contain a group of individuals with a high average fitness level and a small variance of the individual fitness values.

To solve this problem we propose a new sharing function that penalizes clusters with a large variance of the individual fitness values and also penalizes clusters with too many solutions concentrated inside too small a region. The distance metric considered here is the Euclidean distance computed in the real parameter space (phenotypic sharing). In the proposed sharing scheme all the individuals that belong to a given cluster c will share the same fitness value, i.e.

$$s_f(i) = \bar{f}_c - \left(\frac{N_c}{N_{av}} + \frac{1-r_d}{r_d} \right) \sigma(f_c) \quad \forall x_i \in C_c \quad (2.12)$$

Where N_c is the number of genes in the cluster c and where the average fitness value

\bar{f}_c And standard deviation of the individual fitness values $\sigma(f_c)$ as defined as usual by

$$\bar{f}_c = \frac{1}{N_c} \sum_{i=1}^{N_c} f(i) \quad \text{And} \quad \sigma(f_c) = \sqrt{\frac{1}{N_c - 1} \sum_{i=1}^{N_c} (f(i) - \bar{f}_c)^2} \quad (2.13)$$

As the method is based on the distribution of gene fitness inside each cluster, we keep only the clusters that contain at least a minimum number of members. We use here a minimum cluster size of two individuals. As we also need to keep enough clusters of reasonable size, we have to limit the size of the largest clusters. The term N_c / N_{av} of the equation (2.5) is used to control the number of genes inside each cluster. If N_c is smaller than the expected average number of genes inside each cluster N_{av} the correction is reduced, otherwise it is increased. In this study the constant N_{av} is chosen as the population size divide by the (pre-configured) expected number of clusters to be kept.

The second term $(1 - r_d) / r_d$ of equation (2.5) is used to penalize clusters with a concentration of genes around their centroid that is too high. The value r_d is defined as:

$$r_d = \sqrt{\frac{1}{N_c} \sum_{i=1}^{N_c} \left(\frac{d_{ic}}{D_{\max}} \right)} \quad (2.14)$$

Where d_{ic} is the distance of gene i to the centroid of the corresponding cluster c . Here the square root is used to avoid too large a correction for an average concentration of genes.

To keep the cluster's space as large as possible, we also have to minimize the overlap between different clusters. To reduce this overlap, the clustering parameter D_{\min} must be quite large and here we use $D_{\min} = D_{\max}$. In order to have reasonable clustering parameters for a parameter space of high dimensionality, the values of the two clustering parameters D_{\min} and D_{\max} are multiplied by \sqrt{n} where n is the number of parameters to be optimized.

With this new sharing scheme, the selection pressure is no longer specific to each individual, as in a standard GA, but is the same for all genes present in a given cluster. This gives us a selection mechanism, which tries to find subpopulations of solutions with an average high quality instead of the best individual solution. Of course, the overall convergence speed is a little reduced.

The selection pressure towards good solutions is still present through the adaptive cluster methodology, which tends to create clusters around a group of good individuals and through the reproduction technique that uses elitism and mating restriction inside each cluster. Moreover, to keep variety in the population, all the individuals who do not really belong to any clusters (i.e. which are further away than the maximum distance D_{\max} from all existing cluster centroids) will have an unmodified fitness value. During the reproduction phase these individuals will have no mating restriction and generally a slightly higher selection probability.

To speed up the full process, the result of each different gene is stored and not recomputed when this gene appears again in future generations. Moreover, the information of all the previously computed solutions can be used at the end to assess the reasonableness of the optimum solution.

When finished, the algorithm selects for each cluster the best solution that is not further away than $D_{\max} / 2$ from the cluster centroid. The final solution selected from the cluster is the solution that has the greatest average fitness after correcting for variance, i.e. the maximum value of $\bar{f}_c - \sigma(f_c)$.

CHAPTER THREE

NON-LINEAR OPTIMIZATION

3.1 - Non-linear optimization

The general constrained optimization problem is to minimize a nonlinear function subject to nonlinear constraints. Two equivalent formulations of this problem are useful for describing algorithms. They are

$$\min\{f(x): c_i(x) \leq 0, i \in T, c_i(x) = 0, i \in \varepsilon\} \quad (3.1)$$

Where each c_i is a mapping from \mathcal{R}^n to \mathcal{R} , and T and ε are index sets for inequality and equality constraints, respectively; and

$$\min\{f(x); c(x) = 0, l \leq x \leq u\} \quad (3.2)$$

Where c maps \mathcal{R}^n to \mathcal{R}^m , and the lower- and upper-bound vectors, l and u , may contain some infinite components.

The main techniques that have been proposed for solving constrained optimization problems are reduced-gradient methods, sequential linear and quadratic programming methods, and methods based on augmented Lagrangians and exact penalty functions. Fundamental to the understanding of these algorithms is the Lagrangian function, which for formulation (3.1) is defined as:

$$(x, \lambda) L = f(x) + \sum_{i \in T \cup \varepsilon} \lambda_i c_i(x)$$

The Lagrange is used to express first-order and second-order conditions for a local minimizer. We simplify matters by stating just first-order necessary and second-order sufficiency conditions without trying to make the weakest possible assumptions.

The first-order necessary conditions for the existence of a local minimizer x^* of the constrained optimization problem (3.1) require the existence of Lagrange multipliers λ_i^* , such that

$$\nabla_x L(x^*, \lambda^*) = \nabla f(x^*) + \sum_{i \in A^*} \lambda_i^* \nabla c_i(x^*) = 0$$

Where,

$$A^* = \{i \in T : c_i(x^*) = 0\} \cup \in$$

Is the *active set* at x^* , and $\lambda_i^* \geq 0$ if $i \in A^* \cap I$. This result requires a constraint qualification to ensure that the geometry of the feasible set is adequately captured by a linearization of the constraints about x^* . A standard constraint qualification requires the constraint normal, $\nabla c_i(x^*)$ for $i \in A^*$, to be linearly independent.

The second-order sufficiency condition requires that (x^*, λ^*) satisfies the first-order condition and that the Hessian of the Lagrangian

$$\nabla^2_{xx} L(x^*, \lambda^*) = \nabla^2 f(x^*) + \sum_{i \in A^*} \lambda_i^* \nabla^2 c_i(x^*)$$

Satisfies the condition

$$w^T \nabla^2_{xx} L(x^*, \lambda^*) w > 0$$

For all nonzero w in the set

$$\{w \in \mathbb{R}^n : \nabla c_i(x^*)^T w = 0, i \in T^* \cup \varepsilon_1 \nabla c_i(x^*)^T w \leq 0, i \in T_0^*\}$$

Where

$$T_+^* = \{i \in A^* \cap T : \lambda_i^* > 0\}, T_0^* = \{i \in A^* \cap T : \lambda_i^* = 0\}$$

The previous condition guarantees that the optimization problem is well behaved near x^* ; in particular, if the second-order sufficiency condition holds, then x^* is a strict local minimizer of the constrained problem. An important ingredient in the convergence analysis of a constrained algorithm is its behavior in the vicinity of a point (x^*, λ^*) that satisfies the second-order sufficiency condition.

3.1.1 - The Sequential Quadratic Programming Algorithm

It is a generalization of Newton's method for unconstrained optimization in that it finds a step away from the current point by minimizing a quadratic model of the problem. A number of packages, including NPSOL, NLPQL, OPSYC, OPTIMA, MATLAB, and SQP, are founded on this approach. In its purest form, the sequential QP algorithm replaces the objective function with the quadratic approximation

$$q_k(d) = \nabla f(x_k)^T d + \frac{1}{2} d^T \nabla^2_{xx} L(x_k, \lambda_k) d$$

and replaces the constraint functions by linear approximations. For the formulation (3.1), the step d_k is calculated by solving the quadratic subprogram

$$\min \{ q_k(d) : c_i(x_k) + \nabla c_i(x_k)^T d \leq 0, i \in T \}$$

$$c_i(x_k) + \nabla c_i(x_k)^T d = 0, i \in \varepsilon \} \quad (3.3)$$

The local convergence properties of the sequential QP approach are well understood when (x^*, λ^*) satisfies the second-order sufficiency conditions. If the starting point x_0 is sufficiently close to x^* , and the Lagrange multiplier estimates $\{\lambda_k\}$ remain sufficiently close to λ^* , then the sequence generated by setting $x_{k+1} = x_k + d_k$ converges to x^* at a second-order rate. These assurances cannot be made in other cases. Indeed, codes

based on this approach must modify the sub-problem (3.3) when the quadratic q_k is unbounded below on the feasible set or when the feasible region is empty.

The Lagrange multiplier estimates that are needed to set up the second-order term in q_k can be obtained by solving an auxiliary problem or by simply using the optimal multipliers for the quadratic sub-problem at the previous iteration. Although the first approach can lead to more accurate estimates, most codes use the second approach.

The strategy based on (3.3) makes the decision about which of the inequality constraints appear to be active at the solution internally during the solution of the quadratic program. A somewhat different algorithm is obtained by making this decision prior to formulating the quadratic program. This variant explicitly maintains a working set W_k of apparently active indices and solves the quadratic programming problem

$$\min \{ q_k(d) : c_i(x_k) + \nabla c_i(x_k)^T d = 0, i \in W_k \} \quad (3.4)$$

To find the step d_k . The contents of W_k updated at each iteration by examining the Lagrange multipliers for the sub-problem (3.4) and by examining the values of $c_i(x_k + 1)$ at the new iterate $x_k + 1$ for $i \notin W_k$. This approach is usually called the EQP (equality-based QP) variant of sequential QP, to distinguish it from the IQP (inequality-based QP) variant described above.

The sequential QP approach outlined above requires the computation of $\nabla^2_{xx} L(x_k, \lambda_k)$. Most codes replace this matrix with the BFGS approximation B_k , which is updated at each iteration. An obvious update strategy (consistent with the BFGS update for unconstrained optimization) would be to define

$$S_k = x_k + 1 - x_k, \quad Y_k = \nabla_x L(x_k + 1, \lambda_k) - \nabla_x L(x_k, \lambda_k)$$

and update the matrix B_k by using the BFGS formula

$$B_{k+1} = B_k - \frac{B_k \delta_k \delta_k^T B_k}{\delta_k^T B_k \delta_k} + \frac{Y_k Y_k^T}{Y_k^T \delta_k}$$

However, one of the properties that make Broyden-class methods appealing for unconstrained problems-its maintenance of positive definiteness in B_k is no longer assured, since $\nabla_{xx}^2 L(x^*, \lambda^*)$ is usually positive definite only in a subspace. This difficulty may be overcome by modifying Y_k . Whenever $Y_k^T \delta_k$ is not sufficiently positive, Y_k is reset to

$$Y_k \leftarrow \theta_k Y_k + (1 - \theta_k) B_k \delta_k,$$

Where $\theta_k \in [0,1]$ is the number closest to 1 such that $Y_k^T \delta_k \geq \sigma \delta_k^T B_k \delta_k$ for some $\sigma \in (0,1)$. The SQP and NLPQL codes use an approach of this type.

The convergence properties of the basic sequential QP algorithm can be improved by using a line search. The choice of distance to move along the direction generated by the sub-problem is not as clear as in the unconstrained case, where we simply choose a step length that approximately minimizes f along the search direction. For constrained problems we would like the next iterate not only to decrease f but also to come closer to satisfying the constraints. Often these two aims conflict, so it is necessary to weigh their relative importance and define a merit or penalty function, which we can use as a criterion for determining whether or not one point is better than another. The l_1 merit function

$$P_1(x; v) = f(x) + \sum_{i \in \mathcal{E}} v_i |c_i(x)| + \sum_{i \in \mathcal{T}} v_i \max(c_i(x), 0), \quad (3.5)$$

Where $v_i > 0$ are penalty parameters, is used in the NLPQL, MATLAB, and SQP codes, while the augmented Lagrangian merit function

$$L_A(x, \lambda; v) = f(x) + \sum_{i \in \mathcal{E}} \lambda_i c_i(x) + \frac{1}{2} \sum_{i \in \mathcal{E}} v_i c_i^2(x) + \frac{1}{2} \sum_{i \in \mathcal{T}} \psi_i(x, \lambda; v),$$

Where

$$\psi_i(x, \lambda; v) = \frac{1}{v_i} \{ \max\{0, \lambda_i + v_i c_i(x)\}^2 - \lambda_i^2 \},$$

Is used in the NLPQL, NPSOL, and OPTIMA codes. The OPSYC code for equality-constrained problems (for which $T = \Phi$) uses the merit function

$$f(x) + \sum_{i \in \mathcal{E}} \lambda_i c_{i(x)} + \left(\sum_{i \in \mathcal{E}} v_i c_i^2(x) \right)^{\frac{1}{2}}$$

Which combines features of P_1 and L_A .

An important property of the l_1 merit function is that if (x^*, λ^*) satisfies the second-order sufficiency condition, then x^* is a local minimizer of P_1 , provided the penalty parameters are chosen so that $v_i \gg |\lambda_i^*|$. Although this is an attractive property, the use of P_1 requires care. The main difficulty is that P_1 is not differentiable at any x with $c_i(x) = 0$. Another difficulty is that although x^* is a local minimizer of P_1 , it is still possible for the function to be unbounded below. Thus, minimizing P_1 does not always lead to a solution of the constrained problem.

The merit function L_A has similar properties. If (x^*, λ^*) satisfies the second-order sufficiency condition and $\lambda = \lambda^*$, then x is a local minimizer of P_1 , provided the penalty parameters v_i are sufficiently large. If $\lambda \neq \lambda^*$, then we can say only that L_A has a minimizer $x(\lambda)$ near x^* and that $x(\lambda)$ approaches x^* as λ converges to λ^* . Note that in contrast to P_1 , the merit function L_A is differentiable. The Hessian matrix of L_A is discontinuous at any x with $\lambda_i + v_i c_i(x) = 0$ for $i \in T$, but, at least in the case $T_0^* = \Phi$, these points tend to occur far from the solution.

The use of these merit functions by NLPQL is typical of other codes. Given an iterate x_k and the search direction d_k , NLPQL sets $x_{k+1} = x_k + \alpha_k d_k$, where the step length α_k approximately minimizes $(x_k + \alpha d_k; v)$. If the merit function L_A is selected, the step length α_k is chosen to approximately minimize $L_A(x_k + \alpha d_k, \lambda_k + \alpha(\lambda_{k+1} - \lambda_k); v)$, where d_k is a solution of the quadratic programming sub-problem (3.3) and λ_{k+1} is the associated Lagrange multiplier.

3.1.2 - Augmented Lagrangian Algorithm

These are based on successive minimization of the augmented Lagrangian L_A with respect to x , with updates of λ and possibly v occurring between iterations. An augmented Lagrangian algorithm for the constrained optimization problem computes x_{k+1} as an approximate minimizer of the sub-problem

$$\min\{L_A(x, \lambda_k; v_k) : l \leq x \leq u\},$$

Where
$$L_A(x, \lambda; v) = f(x) + \sum_{i \in \mathcal{E}} \lambda_i c_i(x) + \frac{1}{2} \sum_{i \in \mathcal{E}} v_i c_i^2(x)$$

Includes only the equality constraints. Updating of the multipliers usually takes the form

$$\lambda_i \leftarrow \lambda_i + v_i c_i(x_k)$$

This approach is relatively easy to implement because the main computational operation at each iteration is minimization of the smooth function L_A with respect to x , subject only to bound constraints. A large-scale implementation of the augmented Lagrangian approach can be found in the LANCELOT package, which solves the bound-constrained sub-problem by using special data structures to exploit the (group partially separable) structure of the underlying problem. The OPTIMA and OPTPACK libraries also contain augmented Lagrangian codes.

3.1.3 - Reduced-Gradient Algorithm

These avoid the use of penalty parameters by searching along curves that stay near the feasible set. Essentially, these methods take the formulation (3.2) and use the equality constraints to eliminate a subset of the variables, thereby reducing the original problem to a bound-constrained problem in the space of the remaining variables. If x_B is the vector of eliminated or *basic* variables, and x_N is the vector of non basic variables, then

$$x_B = h(x_N),$$

Where the mapping h is defined implicitly by the equation

$$c[h(x_N), x_N] = 0$$

(We have assumed that the components of x have been arranged so that the basic variables come first.) In practice,

$$x_B = h(x_N)$$

Can be recalculated using Newton's method whenever x_N changes. Each Newton iteration has the form

$$x_B \leftarrow x_B - \partial_B c(x_B, x_N)^{-1} c(x_B, x_N),$$

Where $\partial_B c$ is the Jacobian matrix of c with respect to the basic variables. The original constrained problem is now transformed into the bound-constrained problem.

$$\min \{ f(h(x_N), x_N) : l_N \leq x_N \leq u_N \}$$

Algorithms for this reduced sub-problem subdivide the non basic variables into two categories. These are the *fixed* variables x_F , which usually include most of the variables that are at either their upper or lower bounds and that are to be held constant on the current iteration, and the *super basic* variables x_S , which are free to move on this iteration. The

standard reduced-gradient algorithm, implemented in CONOPT, searches along the steepest-descent direction in the super basic variables. The generalized reduced-gradient codes GRG2 and LSGRG2 use more sophisticated approaches. They either maintain a dense BFGS approximation of the Hessian of f with respect to x_S or use limited-memory conjugate gradient techniques. MINOS also uses a dense approximation to the super basic Hessian matrix. The main difference between MINOS and the other three codes is that MINOS does not apply the reduced-gradient algorithm directly to problem (3.1), but rather uses it to solve a linearly constrained sub-problem to find the next step. The overall technique is known as a projected augmented Lagrangian algorithm.

Operations involving the inverse of $\partial_B c(x_B, x_N)$ are frequently required in reduced-gradient algorithms. These operations are facilitated by an LU factorization of the matrix. GRG2 performs a dense factorization, while CONOPT, MINOS, and LSGRG2 use sparse factorization techniques, making them more suitable for large-scale problems.

When some of the components of the constraint functions are linear, most algorithms aim to retain feasibility of all iterates with respect to these constraints. The optimization problem becomes easier in the sense that there is no curvature term corresponding to these constraints that must be accounted for and, because of feasibility, these constraints make no contribution to the merit function. Numerous codes, such as NPSOL, MINOS and some routines from the NAG (NAG Fortran or NAG C) library, are able to take advantage of linearity in the constraint set. Other codes, such as those in the IMSL, PORT 3, and PROC NLP libraries, are specifically designed for linearly constrained problems. The IMSL codes are based on a sequential quadratic programming algorithm that combines features of the EQP and IQP variants. At each iteration, this algorithm determines a set N_k of near-active indices defined by:

$$N_k = \{i \in T : c_i(x_k) \geq -r_i\},$$

Where the tolerances r_i tend to decrease on later iterations. The step d_K is obtained by solving the sub-problem.

$$\min\{q_K(d) : c_i(x_K + d_K) = 0, i \in \mathcal{E}, c_i(x_K + d_K) \leq c_i(x_K), i \in N_K\},$$

Where

$$q_K(d) = \nabla f(x_K)^T d + \frac{1}{2} d^T B_K d,$$

And B_K is a BFGS approximation to $\nabla^2 f(x_K)$. This algorithm is designed to avoid the short steps that EQP methods sometimes produce, without taking many unnecessary constraints into account, as IQP methods do.

3.1.4 - Feasible Sequential Quadratic Programming Algorithm

Finally, we mention feasible sequential quadratic programming algorithms, which, as their name suggests, constrain all iterates to be feasible. They are more expensive than standard sequential QP algorithms, but they are useful when the objective function f is difficult or impossible to calculate outside the feasible set, or when termination of the algorithm at an infeasible point (which may happen with most algorithms) is undesirable. The code FSQP solves problems of the form

$$\min\{f(x) : c(x) \leq 0, Ax = b\}$$

In this algorithm, the step is defined as a combination of the sequential QP direction, a strictly feasible direction (which points into the interior of the feasible set) and, possibly, a second-order correction direction. This mix of directions is adjusted to ensure feasibility while retaining fast local convergence properties. Feasible algorithms have the additional advantage that the objective function f can be used as a merit function, since, by definition, the constraints are always satisfied. FSQP also solves problems in which f is not itself smooth, but is rather the maximum of a finite set of smooth functions.

$$f_i : \mathbb{R}^n \rightarrow \mathbb{R}$$

CONCLUSION

First of all in this project the state of understanding the art of genetic algorithms for solving genetic algorithms is considered. The Crossover and Mutation are the two basic parameters of genetic algorithms. In Encoding the crossover these parameters are used.

The other parameters of genetic algorithms are Selection, Encoding and Population-size.

The applications of genetic algorithms are used to solve the NP-hard problems for machine learning and also for evolving simple problems. The main features of the genetic algorithms are based on optimization and they are represented by Multimodal functions and Simple trading models.

The problems are also solved by using the Non-linear optimization. Solving the non-linear optimization the algorithms were used that are as follows:

- a) The Sequential Quadratic programming algorithm.
- b) Augmented Lagrangian algorithm.
- c) Reduced-gradient algorithm.
- d) Feasible Sequential Quadratic Programming algorithm.

The main techniques used for solving constrained optimization problems are written above. By using these algorithms we solved the optimization problems.

The Appendices were used in these algorithms that are as follows:

- 1) Conopt
- 2) GrG2
- 3) Lancelot
- 4) Mat-lab Optimization Toolbox
- 5) Minos etc.

REFERENCES

- [1] Davis.L *Handbook of Genetic Algorithms*, Van Nostrand, New York, 1991.
- [2] Wolfgang Banzhaf, Colin Reeves, col Reeves *Foundations of Genetic Algorithms*, Morgan Kaufmann Publishers, New York, 1999.
- [3] David E. Goldberg *Eddison, Optimization and Machine Learning* Wesley Pub Co;ISBN:0201157675, January 1989.
- [4] Randy L. Haupt Sue Ellen Haupt (January 1998) John Willy & sons; ISBN: 0471188735
- [5] Mitsuo Gen, Runwei Cheng (2000) *Genetic Algorithms and Engineering Optimization*. John Willy & Sons, New York: ISBN: 047315311
- [6] Nirwan Ansari, Edwin hold (April 1997) *Computational Intelligence For Optimization* Kluwer Academic Publications; ISBN: 0792398386
- [7] David E.Goldberg and J.Richardson, (1987), *Genetic Algorithms With Sharing Formulation Modal function Optimization*. J.J, Hillsdale, NJ.
- [8] Csc Report on Scientific Computing 1997, 1998, Cray T3E, User's Guide CSC User's Guide.
- [9] Juha Hattaja, May 30- June 3, 1994, *Using Genetic Algorithms For Optimization*, Center For Scientific Computing Finland.
- [10] David H. Ackley, 1985, *Genetic Algorithm and their Application*, Lawrence Erlbaum Associates. N.J.
- [11] Thomas Gruninger and David Wallace, Department Of Mechanical Engineering, Stuttgart University Of Massachusetts Avenue, Massachusetts Institute of Technology.
- [12] Thomas And Hand-Paul Schwefel, 1993, *An Overview Of Evolutionary Algorithms For Parameter Optimization*, NY.

Web references

- 1) www.aic.nrl.navy.mil/galist/
- 2) www.systemtechnik.tu-ilmenau.de/~pohlheim/ga_toolbox/
- 3) <http://ee.usyd.edu.au/~dong/semvar/>
- 4) <http://bullwinkle.as.utexas.edu/travis/ga>
- 5) www.cyen.com/pbc/eurospacetech/emoga
- 6) www.anl.gov/labdb/current/ext/m010
- 7) rosowww.epfl.ch/ism97/ism97_abs_1348
- 8) www.ieee.org
- 9) www.computer.org/abstract
- 10) www.csc.fi/math_topics/opt
- 11) www.csc.fi/oppaat/gam/
- 12) www.csc.fi/oppaat/synb/
- 13) <http://cadlab.mit.edu/>
- 14) [http:// www.netlib.org](http://www.netlib.org)
- 15) [http:cne.gmu.edu/modules/ga/](http://cne.gmu.edu/modules/ga/)

APPENDICES

CONOPT

General non-linear programming models with sparse non-linear constraints

The algorithm in CONOPT is based on the generalized reduced gradient (GRG) algorithm. All matrix operations are implemented by using sparse matrix techniques to allow very large models. Without compromising the reliability of the GRG approach, the overhead of the GRG algorithm is minimized by, for example, using dynamic feasibility tolerances, reusing Jacobians whenever possible, and using an efficient re-inversion routine. The algorithm uses many dynamically set tolerances and therefore runs, in most cases, with default parameters.

CONOPT is available as a subroutine library and as a subsystem under the modeling systems AIMMS, AMPL, GAMS, and LINGO. CONOPT is available for PCs and most workstations. All versions are distributed in compiled form. The system is continuously being updated, mainly to improve reliability and efficiency on large models. The latest additions are options for SLP and steepest edge.

GRG2

Non-linear programming

GRG2 uses an implementation of the generalized reduced gradient (GRG) algorithm. It seeks a feasible solution first (if one is not provided) and then retains feasibility as the objective is improved. It uses a robust implementation of the BFGS quasi-Newton algorithm as its default choice for determining a search direction. A limited-memory conjugate gradient method is also available, permitting solutions of problems with hundreds or thousands of variables. The problem Jacobian is stored and manipulated as a dense matrix, so the effective size limit is one to two hundred active constraints (excluding simple bounds on the variables, which are handled implicitly).

The GRG2 software may be used as a stand-alone system or called as a subroutine. The user is not required to supply code for first partial derivatives of problem functions; forward or central difference approximations may be used instead. Documentation includes a 60-page user's guide, in-line documentation for the subroutine interface, and complete installation instructions.

GRG2 is written in ANSI FORTRAN. A C version is also available. Machine dependencies are relegated to the subroutine INITLZ, which defines three machine-dependent constants.

Lancelot

Unconstrained Optimization Problem

The LANCELOT package uses an augmented Lagrangian approach to handle all constraints other than simple bounds. The bounds are dealt with explicitly at the level of an outer-iteration sub problem, where a bound-constrained nonlinear optimization problem is approximately solved at each iteration.

The algorithm for solving the bounded problem combines a trust region approach adapted to handle the bound constraints, projected gradient techniques, and special data structures to exploit the (group partially separable) structure of the underlying problem.

The software additionally provides direct and iterative linear solvers (for Newton equations), a variety of preconditioning and scaling algorithms for more difficult problems, quasi-Newton and Newton methods, provision for analytical and finite-difference gradients, and an automatic decoder capable of reading problems expressed in Standard Input Format (SIF).

LANCELOT A is written in standard ANSI Fortran 77. Single- and double-precision versions are available. Machine dependencies are isolated and easily adaptable. Automatic installation procedures are available for DEC VMS, DEC ULTRIX, Sun UNIX, Cray UNICOS, IBM VM/CMS, and IBM AIX.

Mat-lab Optimization Toolbox

Linear programming, quadratic programming, unconstrained and constrained optimization of nonlinear functions, nonlinear equations, nonlinear least squares, minimax, multi objective optimization, semi-infinite programming.

Linear programming --- a variant of the simplex method. An initial phase is needed to identify a feasible point

Quadratic programming --- an active set method. A linear programming problem is solved to determine an initial feasible point.

Unconstrained minimization --- two routines are supplied. One implements a quasi-Newton algorithm, using either DFP or BFGS to update an approximate inverse Hessian, according to a switch selected by the user. Gradients may be supplied by the user; if they are not, finite differencing is used. The second routine uses the Nelder-Mead simplex algorithm, for which derivatives are not needed.

Constrained minimization --- sequential quadratic programming. The BFGS formula is used to maintain an approximation to the Hessian. Han's merit function is used to determine the step length at each iteration.

Nonlinear equations --- Newton's method and the Levenberg-Marquardt algorithm are supplied. The user chooses the algorithm by setting a switch.

Nonlinear least squares --- the Gauss-Newton method and the Levenberg-Marquardt method are supplied. The user makes the choice.

Minimax --- these problems can be formulated as constrained optimization problems, and a sequential quadratic programming algorithm is used to solve them here. Advantage is taken of the structure of the problem in the choice of approximate Hessian.

Multi objective optimization --- The problem is formulated as one of decreasing a number of objective functions below a certain threshold simultaneously, so it is viewed as a constrained optimization problem. Again, sequential quadratic programming is used to solve it.

Semi-infinite programming --- Cubic and quadratic interpolation is used to locate peaks in the infinite constraint set and therefore to reduce the problem to a constrained optimization problem.

Minos

Linear programming, unconstrained and constrained nonlinear optimization

Linear programming: A primal simplex method is used. A sparse LU factorization of the basis is maintained, using a Markowitz ordering scheme and Bartels-Golub updates as implemented in the LUSOL package of Gill, Murray, Saunders, and Wright.

Nonlinear objective, linear constraints: A reduced-gradient algorithm is used. This is an active-set method (a natural extension of the simplex method). The variables are classified as-, super basic, and non-basic, with the number of super basics indicating the effective non-linearity of the objective. The constraints are satisfied before the objective is evaluated. Feasibility is maintained thereafter. Search directions are generated using a quasi-Newton approximation to the reduced Hessian.

Nonlinear constraints: A projected augmented Lagrangian algorithm is used. As in Robinson's method, each major iteration solves a linearly constrained sub problem to generate a search direction. The sub problem objective is an augmented Lagrangian function. The sub problem constraints are the true linear constraints plus linearizations of the nonlinear constraints. Convergence is usually achieved, although the step length choice is heuristic. (A reliable merit function is not yet known.)

MINOS is designed to handle thousands of constraints and variables. Constraint data may be input from MPS files or via subroutine parameters. Non-linearities are specified by Fortran subroutines. (Ideally these should provide both functions and gradients. Missing gradients are estimated by finite differences.) The GAMS and AMPL systems may be used as alternative user interfaces. See their entries for details.

MINOS is distributed on floppy disk. Fortran 77 source code is provided, along with test problems and makes files for Unix, VMS and DOS systems.

Nag C Library

Linear programming, quadratic programming, minimization of a nonlinear function (unconstrained, bound-constrained, linearly constrained, and nonlinearly constrained), and minimization of a sum of squares.

For problems with nonlinear constraints, a sequential QP algorithm is used. For unconstrained problems and problems with simple bounds, quasi-Newton and conjugate gradient methods are provided. The Nelder-Mead simplex method is provided for unconstrained problems. For minimizing a sum of squares, a Gauss-Newton method is used. The LP and QP routines use a numerically stable active-set strategy.

An option-setting mechanism is provided in all routines, in order to keep the basic parameter-list to a minimum, while allowing a large degree of flexibility in controlling the algorithm. The routines have the ability to print the solution, as well as various amounts of intermediate output to monitor the computation.

Service routines are provided for checking user-supplied routines for first derivatives and for computing a covariance matrix for nonlinear least squares problems.

The NAG C Library is available in tested, compiled form for several hardware/software-computing environments.

Nag Fortran Library

Linear programming, mixed-integer linear programming, quadratic programming, minimization of a nonlinear function (unconstrained, bound constrained, linearly constrained, and nonlinearly constrained), and minimization of a sum of squares (unconstrained, bound constrained, linearly constrained, and nonlinearly constrained).

For problems with nonlinear constraints, a sequential QP algorithm is used. For unconstrained problems and problems with simple bounds, quasi-Newton, modified Newton, and conjugate gradient methods are provided. The Nelder-Mead simplex method is provided for unconstrained problems. For minimizing a sum of squares, a Gauss-Newton method is used. The LP and QP routines use a numerically stable active set strategy in which the linear constraint matrix may be dense or sparse. Problem data may be supplied in MPSX format.

An option-setting mechanism is provided in the more recent routines, in order to keep the basic parameter list to a minimum, while allowing a large degree of flexibility in controlling the algorithm. These routines also have the ability to print the solution, as well as various amounts of intermediate output to monitor the computation.

Service routines are provided for approximating first or second derivatives by finite differences, for checking user-supplied routines for first or second derivatives, and for computing a covariance matrix for nonlinear least squares problems.

The NAG Fortran Library is available in tested, compiled form for a large number of different hardware and software computing environments.

NLPQL

Smooth nonlinear programming with equality and inequality constraints

NLPQL solves smooth nonlinear programming problems, i.e. minimizes a nonlinear objective function subject to nonlinear equality and inequality constraints. It is assumed that all model functions are continuously differentiable.

The internal algorithm is a sequential quadratic programming (SQP) method. Proceeding from a quadratic approximation of the Lagrangian function and a linearization of the constraints, a quadratic sub problem is formulated and solved to get a search direction. Subsequently a line search is performed with respect to two alternative merit functions, and the Hessian approximation is updated by the modified BFGS formula.

Special features of NLPQL are

- Separate handling of upper and lower bounds on the variables,
- Reverse communication,
- Internal scaling,
- Initial multiplier and Hessian approximation,
- Feasibility with respect to bounds and linear constraints,
- Full documentation by initial comments.

NLPQL is written in double-precision Fortran 77 and organized in the form of a subroutine. Nonlinear problem functions and analytical gradients must be provided by the user within special subroutines or the calling program.

NPSOL

Minimization of smooth nonlinear functions

NPSOL is a Fortran package designed to solve the nonlinear programming problem: the minimization of a smooth nonlinear function subject to a set of constraints on the variables. The problem is assumed to be stated in the following form:

$$\begin{array}{ll} \text{minimize} & f(x) \\ x \in \mathcal{R} & \\ \\ \text{subject} & l \leq \begin{Bmatrix} x \\ Ax \\ c(x) \end{Bmatrix} \leq u, \end{array}$$

Where $f(x)$ is a smooth nonlinear function, A is an m_L -matrix, and $c(x)$ is an m_N -vector of smooth nonlinear constraint functions.

The user must supply an initial estimate of the solution of the problem, subroutines that evaluate $f(x)$, $c(x)$, and as many first partial derivatives as possible. Unspecified derivatives are approximated by finite differences.

If the problem is large and sparse, the MINOS package should be used, since NPSOL treats all matrices as dense.

NPSOL is a sequential quadratic programming method incorporating an augmented Lagrangian merit function and a BFGS quasi-Newton approximation to the Hessian of the Lagrangian. If there are no nonlinear constraints, the gradients of the bound and linear constraints are never recomputed, and NPSOL will function as a specialized algorithm for linearly constrained optimization.

It can be arranged that the problem functions are evaluated only at points that are feasible with respect to the bounds and linear constraints.

NPSOL uses subroutines from the LSSOL constrained linear least squares package, which is distributed together with NPSOL.

Optima Library

Unconstrained optimization, constrained optimization, sensitivity analysis

OPVM --- Unconstrained optimization, unstructured objective function; suitable for small problems.

OPVMB --- Optimization subject to simple bounds.

OPLS --- Unconstrained nonlinear least squares.

OPNL --- Nonlinear equations, by minimizing sum of squares of the residuals.

OPCG --- Nonlinear conjugate gradient method.

OPODEU --- Unconstrained optimization problems by tracing the solution curve of a system of ODEs (homotopy method).

OPTNHP --- Unconstrained optimization using the truncated Newton method; no Hessian storage or calculation required.

OPRQP --- Sequential quadratic programming, but superseded by the OPXRQP routine.

OPXRQP --- A more efficient implementation of sequential quadratic programming; uses the EQP variant.

OPSQP --- Another implementation of sequential quadratic programming, but uses the IQP variant, which gives rise to inequality-constrained sub problems.

OPALQP --- Similar to OPSQP, but uses an augmented Lagrangian line search function.

OPSMT --- Nonlinear programming, using a SUMT technique.

OPIPF --- Sequential minimization of a sequence of augmented Lagrangians.

OPODEC --- Homotopy method: traces the solution curve of a system of ODEs.

OPSEN --- Tests the sensitivity of the objective function around the optimal point of an unconstrained problem.

OPSEC --- Like OPSEN, but for the solution of a constrained problem.

Software is written in Fortran 77.

Optpack

Unconstrained optimization and nonlinear constrained optimization with special software to handle bound constraints, linear equality constraints, and general nonlinear constraints

Unconstrained optimization is performed using the conjugate gradient algorithm. Constrained optimization is performed using a new scheme that combines multiplier methods with preconditioning and linearization techniques to accelerate convergence.

The software is written in double precision Fortran. The code is documented by internal comments. Research reports providing the theoretical basis for the algorithms are available on request. User feedback is much appreciated.

Port

General minimization, nonlinear least squares, separable nonlinear least squares, linear inequalities, linear programming, and quadratic programming.

The nonlinear optimizers have unconstrained and bound-constrained variants, and use trust region algorithms. Gradients and Jacobians can be provided by the caller or approximated automatically by finite differences. The general minimization routines use either a quasi-Newton approximation to the Hessian matrix or a Hessian provided by the

caller; the nonlinear least squares routines adaptively switch between the Gauss-Newton Hessian approximation and an "augmented" approximation that uses a quasi-Newton update. Function and, if necessary, gradient values may be provided either by subroutines or by reverse communication.

There is a special separable nonlinear least squares solver for the case of one nonlinear variable; it uses Brent's one-dimensional minimization algorithm for the nonlinear variable. Brent's algorithm is also available by itself, as is an implementation of the Nelder-Mead simplex method.

The feasible point (linear inequalities) and linear and quadratic programming routines start by taking steps through the interior and end with an active set strategy. The quadratic programming solvers use the Bunch-Kaufman factorization and thus can find local minimizers of indefinite problems.

None of the solvers is meant for large numbers of variables. When there are n variables and m equations (where $m = 1$ for general minimization), the nonlinear solvers require $O(n^2 m)$ or $O(n^3)$ arithmetic operations per iteration. The linear and quadratic solvers use dense-matrix techniques.

Software is written in ANSI Fortran 77, with single- and double-precision versions of all solvers. Machine-dependent constants are provided by subroutines I1MACH, R1MACH, and D1MACH.

PROC NLP (SAS/OR Software)

General and specialized nonlinear optimization

The NLP procedure offers a set of optimization techniques for minimizing or maximizing a continuous nonlinear function $f(x)$ of n decision variables with boundary, general linear, and nonlinear equality and inequality constraints. PROC NLP supports a number of algorithms for solving this problem that take advantage of the special structure

of the objective and constraint functions. Two algorithms are especially designed for quadratic optimization problems, and two other algorithms are provided for the efficient solution of nonlinear least-squares problems.

PROC NLP is part of SAS/OR Software, a fully integrated component of the SAS System. Along with its programming statements, PROC NLP uses SAS data sets (proprietary format) for input and for output. By taking advantage of the SAS System's Multiple Engine Architecture, PROC NLP can in effect read from and write to over fifty different database formats.

In addition to producing output SAS data sets, PROC NLP can print text output detailing the initial decision variable values, the search for an initial feasible solution, the optimization history, and the values of decision variables, derivatives, and covariance matrices at optimality.

Input Data

- Objective function and the constraints are specified using the programming statements of PROC NLP
- Additional data sets can be used to generate constraints and objectives
 - DATA= data set specifies an objective function that is a combination of n other functions
 - INQUAD= data set (sparse or dense format) specifies the objective of a quadratic programming problem
 - INEST= or INVAR= data set specifies initial values for the decision variables, the values of constants that are referred to in the program statements, as well as simple boundary and general linear constraints
 - MODEL= data set specifies a model saved from a previous execution of the NLP procedure

Output Data

- OUT= output data set contains variables generated in the program statements defining the objective function (and perhaps derivatives) plus any variables used in a DATA= input data set
- OUTEST= data set contains values of decision variables, derivatives, and covariance matrices at optimality, and can be used in subsequent PROC NLP calls as an INEST= input data set
- OUTMOD= data set contains the programming statements and can be used in subsequent PROC NLP calls as a MODEL= input data set

Optimizers

The following algorithms are available via PROC NLP for use with these categories of nonlinear programs:

- **Nonlinear min/maximization with linear constraints**
 - A trust-region algorithm (Dennis, Gay, & Welch, 1981, Gay, 1983, and Moré and Sorensen, 1983)
 - Two different Newton-Raphson algorithms using line search or ridging
 - Quasi-Newton algorithms updating either an approximation of the inverse Hessian or the Cholesky factor of an approximate Hessian
 - A double dogleg algorithm (Gay, 1983 and Dennis and Mei, 1979)
 - Various conjugate gradient algorithms with the Powell and Beale automatic restart update (Powell, 1977, and Beale, 1972), Fletcher-Reeves update, Poliak-Ribiere update, or conjugant-descent update (Fletcher, 1987)
 - The Nelder-Mead simplex algorithm with a modification of Powell's COBYLA implementation (Powell, 1992)

- **Nonlinear min/maximization, unconstrained or with boundary constraints**
 - Any of the algorithms listed above, substituting the original Nelder-Mead simplex algorithm for the COBYLA version
- **Nonlinear min/maximization with nonlinear constraints**
 - A quasi-Newton algorithm that is a modification of Powell's Variable Metric Constrained Watch Dog (VMCWD) algorithm (Powell, 1978, 1982)
 - The Nelder-Mead simplex algorithm with a modification of Powell's COBYLA implementation (Powell, 1992)
- **Nonlinear least squares with linear constraints**
 - The Levenberg-Marquardt algorithm (Moré, 1978)
 - a hybrid quasi-Newton algorithm (Fletcher & Xu, 1987, Lindstrom & Wedin, 1984, and Al-Baali & Fletcher, 1986)
- **Quadratic min/maximization with linear or boundary constraints**
 - Solve as a linear complementarily problem (if the symmetric matrix is positive/negative semi-definite for a min/maximization and the variables are restricted to be positive)
 - Use a general quadratic optimization active set algorithm (Gill, Murray, Saunders, & Wright, 1984)

Derivatives

PROC NLP may require derivatives of the objective function and the constraints. These can be obtained

- Analytically (using a special derivative compiler), the default method
- Via finite difference approximations
- Via user-supplied exact or approximate numerical functions

Problem Size Limitations

The size of a problem that PROC NLP can solve depends on the host platform, the available memory, and the available space for utility data sets. PROC NLP does not place any additional limits on problem size.

Available Platforms

The SAS System is supported on all major personal computer, workstation, and mainframe operating systems.

SQP

Nonlinear programming

SQP uses an implementation of Powell's successive quadratic programming algorithm and is aimed specifically at large, sparse nonlinear programs. It solves the quadratic programming sub-problems by using a sparsest-exploiting reduced gradient method. Sparse data structures are used for the constraint Jacobian, and there is an option to represent the approximate Hessian as a small set of vectors using a limited memory-updating scheme.

SQP requires the same user-supplied subroutines as GRG2 and has similar subroutine and data file interfaces. The entry describing GRG2 contains more details.

SQP is written in ANSI FORTRAN. Machine dependencies are relegated to the subroutine INITLZ, which defines three machine-dependent constants.