Master Thesis
Software Engineering
Thesis no: MSE-2003-09
June 2003

Optimizing Genetic Algorithms for Time Critical Problems

Christian Johansson
Gustav Evertsson

Department of
Software Engineering and Computer Science
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby
Sweden

This thesis is submitted to the Department of Software Engineering and Computer Science at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Software Engineering. The thesis is equivalent to 2 x 20 weeks of full time studies.

Contact Information:
Author(s):
Christian Johansson
Address:
Lindblomsvägen 97
372 33 Ronneby
E-mail: christian.johansson@norrkoping.se

Gustav Evertsson
Address:
Lindblomsvägen 82
372 33 Ronneby
E-mail: me@guzzzt.com

University advisor(s):
Stefan Johansson
E-mail: sjo@bth.se
Department of Software Engineering and Computer Science

Department of
Software Engineering and Computer Science
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby
Sweden

Internet : www.bth.se/ipd
Phone : +46 457 38 50 00
Fax : + 46 457 271 25

ABSTRACT

Genetic algorithms have a lot of properties that makes it a good choice when one needs to solve very complicated problems. The performance of genetic algorithms is affected by the parameters that are used. Optimization of the parameters for the genetic algorithm is one of the most popular research fields of genetic algorithms. One of the reasons for this is because of the complicated relation between the parameters and factors such as the complexity of the problem. This thesis describes what happens when time constraints are added to this problem. One of the most important parameters is population size and we have found by testing a well known set of optimization benchmark problems that the optimal population size is not the same when time constraints were involved.

Keywords: genetic algorithms, population size, real-time systems, Optimization.

Contents

Abstract **i**

1 Introduction **1**
1.1 History . . . 1
1.2 Genetic Algorithms . . . 2
1.2.1 The genetic information . . . 3
1.2.2 Fitness function . . . 3
1.2.3 Selection . . . 3
1.2.4 Reproduction . . . 4
1.2.5 Crossover . . . 4
1.2.6 Mutation . . . 6
1.2.7 Other operations . . . 6
1.3 Where are genetic algorithms used? . . . 6
1.4 Optimization of the population size . . . 7
1.4.1 Optimization based on generations . . . 7
1.4.2 Optimization based on time constraint . . . 7
1.5 Hypothesis . . . 8
1.6 Methodology . . . 8
1.7 Outline of the thesis . . . 8

2 Problem domain **9**
2.1 Controlling and Designing a time critical system . . . 10
2.2 Genetic algorithms to solve time critical problems . . . 10

3 Genetic Algorithm Parameters **12**
3.1 Exploitation vs. exploration . . . 12
3.1.1 Mutation rate . . . 12
3.1.2 Crossover rate . . . 13
3.1.3 Adaptive genetic algorithms . . . 13
3.2 Population size . . . 14

4 Experimental setup **16**
4.1 Choice of genetic algorithm . . . 16
4.1.1 Selection . . . 16
4.1.2 Mutation and Crossover . . . 16
4.1.3 Parameter to binary representation . . . 16
4.1.4 Random . . . 17
4.2 De Jong test functions . . . 17

4.2.1 Test function 1 or Sphere . . . 18
4.2.2 Test function 2 or Rosenbrock's Saddle . . . 18
4.2.3 Test function 3 or Step . . . 18
4.2.4 Test function 4 or Quartic . . . 18
4.2.5 Test function 5 or Shekel's Foxholes . . . 19
4.3 Execution of the test . . . 20
4.3.1 Log file generation . . . 20
4.3.2 Graph transformation . . . 20
4.3.3 Maximum fitness generation . . . 20

5 Analysis of the results **22**
5.1 Explanation of the figures . . . 22
5.2 Analysis of the results . . . 22
5.3 Time and Performance measuring . . . 25
5.4 Comparison to others studies . . . 34
5.5 Function to find optimal population size . . . 34
5.6 De Jong 5 . . . 34
5.7 Functions characteristics . . . 37

6 Discussion **38**

7 Conclusions and future work **39**
7.1 Future Work . . . 39

Bibliography **40**

Chapter 1

Introduction

Genetic algorithms have a lot of attributes that makes its a good choice when one needs to solve very complicated problems. The simplicity and robustness of the algorithm has made it popular among developers.

1.1 History

In 1859 Charles Darwin formulated the fundamental principle of natural selection. The principle explains that organisms who are well adapted in their environment will have a greater chance of surviving and reproducing than other organisms (who are less adapted). (Darwin, 1859)

Later in 1865 Gregor Mendel formulated the principle of genetics which explains how organisms evolve from one generation to the next. (Mendel, 1886). The two theories were unlinked until 1920's when it was proved that genetics and natural selection were not in contradiction to each other and the combination of those theories led to the ideas to the modern theory of evolutionary.

Evolution Computing started as a fraction of computer science in the 1950s and 1960s as an idea to solve optimization problems. Bremermann was the first to start investigating how genetic algorithms could be used (Bremermann, 1962). It was not until Holland developed the theory by trying to use ideas from nature and transform them into computer science that the algorithms became popular. This was the foundation of the genetic algorithms that are used today. He introduced the algorithm that evolved from one population to a new population with selection, crossover, mutation and inversion operators (Holland, 1975). He also presented in 1968 the Schema Theory in which he tried to prove that genetic algorithms work in theory. Goldberg took the first big step to develop this theory (Goldberg, 1989). His hypothesis, *The Building Block Hypothesis* states that the performance is most affected by crossover. This is a big difference from Hollands original theory that is focused on how mutation and crossover are destructive in their nature. In the 1990s people started to criticize and debate the usefulness of the schema theory (Dianati et al., 2002) (Mitchell and Forrest, 1994).

Parallel to the development of the genetic algorithm other researchers have proposed alternative evolutionary algorithms. Examples of these are evolutionary strategies (Back et al., 1991), evolutionary programming and genetic

programming (Koza, 1989). One thing they all have in common is that they evolve populations. The differences are how the population is represented and how the operators are used to make the individuals better.

1.2 Genetic Algorithms

There exists many different implementations of Genetic Algorithms. Because of the complexity in deriving good parameters the researchers tend to prefer to use a robust traditional algorithm rather than implement their own algorithm to each new problem. Davis states that even the simplest Genetic Algorithm for a particular problem must have a set of the following five components to work properly (Davis, 1991):

- A genetic representation for potential solutions to the problem
- A way to create an initial population of potential solutions
- An evaluation function that plays the role of the environment, rating solutions in terms of their fitness to the environment
- Genetic operators that alter individuals for the composition of children
- Values for various parameters that the Genetic Algorithm uses (Population size, probabilities of applying genetic operators, etc.)

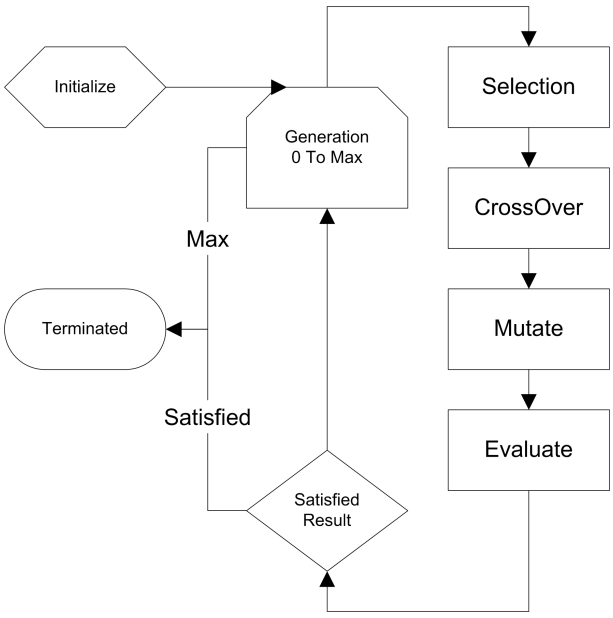


Figure 1.1: A basic flowchart of a genetic algorithm.

1.2.1 The genetic information

The genetic information in a genetic algorithm corresponds to a DNA chain in an individual. The DNA chain describes the solution or more correctly the parameters of the solution. The genetic information is built by chromosomes and is in general coded in a binary form (but can also be in another form of data). A solution is generally called an individual and the set of individuals that are present in a generation of a genetic algorithm are referred to as the population.

1.2.2 Fitness function

The fitness function, also called evaluation function, is the genetic algorithm component that rates a potential solution by calculating how good they are relative to the current problem domain.

1.2.3 Selection

Selection is a way for the genetic algorithm to move towards promising regions in the search space. The individuals with high fitness are selected and they will have a higher probability of survival to the next generation. With too much selection pressure, genetic search will terminate prematurely; while with too little selection pressure, evolutionary progress will be slower than necessary. A lower selection pressure is recommended in the start of the genetic search; while a higher selection pressure is recommended at the end in order to narrow the search space. (Spears, 1995)

A common used selection algorithm is roulette wheel selection. The roulette wheel selection contains the following steps: (See equations 1.1)

1. Calculate the total fitness of the population
2. Calculate the probability of selection p_i for each individual v_i
3. Calculate the cumulative probability q_i for each individual v_i
4. Generate a random (float) number r from the range [0..1] if(r $< q_1$) select individual v_1, otherwise select the i:th individual v_i (2$\leq$ i$\leq$ Population Size) such that $q_{i-1} <$ r $\leq q_i$

$$F = \sum_{i=1}^{PopulationSize} evaluate(v_i) \tag{1.1a}$$

$$p_i = \frac{evaluate(v_i)}{F} \tag{1.1b}$$

$$q_i = \sum_{j=1}^{i} p_j \tag{1.1c}$$

1.2.3.1 Stochastic Sampling

The selection phase determines the actual number of copies from each chromosome based on its probability to survive. The selection phase has two parts: determine the chromosomes expected values and convert the values to numbers of offspring. The expected value of a chromosome is a real number indicating the average number of offspring. The sampling procedure is used to convert the real expected value to the number of offspring. Roulette wheel selection is one of the most used stochastic selection techniques. We imagine a roulette wheel with slots sized according to the individuals' fitness. For each spinning on the roulette wheel an individual is selected to the new population. There is a chance that one individual is selected more than once but this is accepted according to the Schema Theorem; The best individuals grow in number, the average stay even, and the worst have the largest chance to die off (Michalewicz, 1994). See equations 1.1 for the mathematical implementation.

1.2.3.2 Deterministic Sampling

Deterministic sampling is a deterministic procedure which selects the best chromosomes from parents and offspring. One deterministic selection technique that is often used together with a stochastic selection technique is elitism. In general selection there is a risk that the best member of the population fails to survive to the next generation. The elitist strategy may increase the speed of domination of a population with a super individual, but on the other side it may improve the Genetic Algorithm performance in some domains. This is made by selecting the fittest member or members of the population and they are passed on to the next generation without being altered by any genetic operators. Using elitism ensures that the maximum fitness of the population can never be reduced from one generation to the next. Elitism usually brings about a more rapid convergence of the population. In some applications elitism improves the chances of locating an optimal individual, while in others it reduces it (Goldberg, 1989).

1.2.3.3 Mixed Sampling

Mixed sampling contains both random and deterministic features simultaneously.

1.2.4 Reproduction

One thing that all evolution algorithms have in common is that they have a population that in one way or another must evolve and become better and better. In genetic algorithms this is done primarily using two operations - mutation and crossover.

1.2.5 Crossover

In biology, crossover is a complicated process, typically exchanging chromosome portions of genetic material. The result is that a large variety of genetic combinations will be produced. In most genetic algorithms, recombination is implemented by the means of a crossover operator which operate on pairs of chromosomes to produce new offspring by exchanging segments of the genetic

material from the parent's. In other words, crossover takes two of the fittest genetic strings in a population and combines the two of them to generate new genetic strings.

1.2.5.1 1 point crossover

Traditionally, Genetic Algorithms have relied on the simplest form of crossover, using a 1 point crossover. 1 point crossover works in the following way. A point is randomized in the DNA and the new outcome has the DNA from the first parent DNA to the point and the DNA after the point from its second parent. See figure 1.2 for an example. The 1 point crossover may easily be generalized to an arbitrary number of crossover points. Studies have shown that a 2-point crossover is the least disruptive of the crossover techniques (Spears, 1995).

Parent 1 DNA:	01010110
Parent 2 DNA:	11100001
Crossover point:	3
Child 1 DNA:	01000001
Child 2 DNA:	11110110

Figure 1.2: Example on single point crossover

1.2.5.2 Uniform crossover

Many empirical studies have shown the benefit of higher numbers of crossover points (Spears, 1995). Instead of choosing a random point for crossover a template is used. The template is random bit values. The values are shifted on the positions that is 0. See Figure 1.3 for an example. Some work has focused on uniform crossover, which involves the average L/2 crossover points for a string of length L. Studies have also proven that this case is not general and only optimal for a given problem. Uniform crossover is the most disruptive of the crossover techniques. (Spears, 1995)

Parent 1 DNA:	01010110
Parent 2 DNA:	11100001
Template:	01010111
Child 1 DNA:	11110111
Child 2 DNA:	01000001

Figure 1.3: Example on uniform crossover

1.2.5.3 Adaptive crossover

Adaptive crossover is a Genetic Algorithm that decides, as it runs, which form of crossover that is optimal. This can be done in 2 ways, locally and globally. Locally this is done by adding one more chromosome to the DNA string. If the chromosome is zero (0) on both parents then the 2-point crossover is used, if the chromosome is one (1) on both parents then the uniform crossover is used. And if the case is that the parents have different chromosomes, then random is

used to decide which crossover form that would be used. Globally this is done by looking at the whole population. If the population has more zeros (0) in the chromosome, that decides which crossover technique that will be used, then 2-points crossover will be used on the whole population. If the population has more ones (1) in the chromosome that decides which crossover technique that will be used then uniformed crossover will be used on the whole population. All genetic operators are allowed to manipulate this extra chromosome including crossover and mutation, in both local and global ways. (Spears, 1995)

1.2.5.4 Guided Crossover

Guided crossover is an alternative to the normal crossover with some advantages. The guided crossover operator is made to raise the quality of the result so the genetic algorithm has a bigger change of coming closer to the global optima. First step in the guided crossover is to select two individuals from the population. The offspring of the crossover operation is then randomly taken from the line joining the two parents closest to the one with the highest fitness. (Rasheed, 1999)

1.2.6 Mutation

Mutation is a genetic operator that changes one or more gene values in a chromosome. Mutation is an important part of the genetic search, which helps to prevent the population from stagnating at any local optima. Mutation means that you take one or more random chromosomes from the DNA string and change it (e.g. 0 become 1 and vice versa).

1.2.7 Other operations

The above mentioned operators are the most common operations in a genetic algorithm. There exists more operations that do not have big impact on the Genetic Algorithm, but can be applied on some problems to generate a more suitable solution. Some of these operators are sexual reproduction, duplication and deletion etc.

1.2.7.1 Inversion and Reordering

Inversion is another way for children to be different from their parents. The Inversion technique inverts two randomly chosen chromosomes to make a new outcome and focus only on a single DNA string. It is inspired by nature but has not often been seen useful in genetic algorithms and are therefore rarely used. (Davis, 1991)

1.3 Where are genetic algorithms used?

Genetic algorithms can be used in a wide range of problem domains. Mitchell and Forrest identifies nine different types of problems where genetic algorithms are used (Mitchell and Forrest, 1994):

Optimization An area where genetic algorithms have been used as a solution to a wide range of optimization problems, from numerical models (such as the De Jong functions (Goldberg, 1989)) to scheduling and hardware design.

Automated Programming Genetic algorithms can be used to develop computer programs by evolving different solutions. It can also be used to evolve different types of computer structures.

Machine and robot learning Genetic algorithms can be used for both classifier and predicting purposes. It can be used to design neural networks and classifier systems.

Economic models Genetic algorithms have been used to make models of the economic market.

Immune system models Genetic algorithms can be used to make models of various types, including mutation and and other natural phenomenons.

Ecological models These types of models include host-parasite co-evolution and symbioses.

Population genetics models Genetic algorithms can be used to make models in the field of population genetics.

Interactions between evolution and learning This include studies on how learning and species evolutions affect one another.

Models of social systems A popular problem that has been studied is how ants work together in a social structure (Dorigo and Gambardella, 1997). This category also include research on cooperation and communication.

1.4 Optimization of the population size

One of the most important parameters in a genetic algorithm is population size. To get the best fitness after a predefined number of generations, the population size has to be set correctly (Jong, 1975).

1.4.1 Optimization based on generations

If the population size is set too low the genetic algorithm will have too few possible ways to alter new individuals so the fitness will be low. If the population size is set too high the population will have too much bad genetic material to provide a good fitness.

1.4.2 Optimization based on time constraint

If the genetic algorithm has a predefined amount of time during which it can be run, the population size has to be optimized depending on the amount of time instead of the number of generations. If the population size is set too low the genetic algorithm will have too few possible ways to alter new individuals so the fitness will be low. If the population size is set too high the genetic algorithm

will take longer time to run each generation, therefore the genetic algorithm does not have so many generations to alter the population, so the fitness will be low.

1.5 Hypothesis

The size of the population will differ if the size is optimized based on the amount of time instead of the number of generations.

1.6 Methodology

To confirm or reject our hypothesis we will test a well known set of optimization benchmark problems in a genetic algorithm and explore the genetic algorithms parameter space in the chosen set of problems. This is done by a simulation experiment. To gain knowledge of current research in the area of parameter settings for genetic algorithms we will do a literature survey.

1.7 Outline of the thesis

We began this thesis by explaining, what a genetic algorithm is and how it is working. We will continue to explain what we mean with a genetic algorithm in a time critical context. After this we present others research on how to setup the different parameters in a genetic algorithm. Later, we describe the experimental setup and the results from the experiment. Finally, our conclusions and future work will be covered at the end of the report.

Chapter 2

Problem domain

We have in this report focused on the time critical problems domain. The time that the application uses to solve the problem is a concern. There exists a wide range of terms that describe these types of systems. The term 'real-time system' is widely used for systems that have a deadline before the systems fail. The term 'embedded system' is also used to describe applications that are embedded into an environment where they have to process some information within a given time. One definition of a real-time system is:

> A real-time system is a system that is required to react to stimuli from the environment (including the passage of physical time) within time intervals dictated by the environment.
>
> (Burns and Wellings, 1989)

This definition covers a very wide variety of different systems and we may speak about 'soft' and 'hard' real-time systems. A deadline exists in soft real-time systems but the system is still defined as functioning even if it is missed occasionally. Many applications found in a normal workstation are of this type; the user can in most cases wait a few seconds more for the answer. Soft real-time systems can also be of the type 'firm' if it is okay that the system does not give an answer as long as it is infrequently but the system has no use of a late answer. Hard real-time systems is the opposite where a correct answer must be given before the deadline and a late answer is a failure in the same way as no answer at all or an incorrect answer. An example of this type is embedded systems within vehicles where late answers can be catastrophic. Many systems are not as polarized as these but have a combination of both soft and hard deadlines. One example is that the system has a soft deadline after one second, where a more exact result is calculated and if this requirement is not fulfilled there is a hard deadline after three seconds, where an approximated answer is given.

We have found that the current research on time critical systems can be divided into different areas depending on their focus. The first area is the research that works on how to get control over the problem and how to design a system and know that it will manage the time limits. The second area comprises research on making the algorithms more efficient. How the improvements are done depends on the nature of the problem and the design of the solution.

2.1 Controlling and Designing a time critical system

As real-time systems become larger and more complex, challenges of creating the control system have raised. Musliner et al. identifies some of the problems that a developer of a real-time system must take into consideration (Musliner et al., 1994). The application must be able to sense the surrounding environment and be able to adapt to the output from it. In normal real-time systems an example of this is that an embedded flight control system can change mode from normal flight to landing when the plane comes close enough to the landing field. It is no use to have all the parts of the system activated all the time. Artificial intelligence comes in handy when the boundaries between different phases are not that clear any more. The system in this situation will adapt easier to the environment.

Musliner et al. also divide the solutions into three different categories (Musliner et al., 1994):

1. Embedding artificial intelligence into a real-time system - The artificial intelligence subsystem must meet the deadline but does not have to care about the rest of the problems associated with real-time systems.

2. Embedding real-time reactions into an artificial intelligence system - This is the opposite idea where the artificial intelligence system must implement all the real-time functionalities.

3. Coupling artificial intelligence and real-time subsystems as parallel, cooperating components - This type tries to take the best from both subsystems and combine them.

Hamidzadeh and Shekhar propose a methodology that can be used during design to get a formal specification of the real-time problem when artificial intelligence is used in the solution. They split the time of the artificial intelligence algorithm into two phases. The first is the planning phase where the artificial intelligence algorithm search for a solution that will be used during the second phase; the execution phase. They discuss and compared alternative algorithms to come up with which of them that is most suited for different problems. (Hamidzadeh and Shekhar, 1993)

2.2 Genetic algorithms to solve time critical problems

The genetic algorithm can at any time be interrupted and it will have an answer. This differs from many other algorithms that must finish before they can present an answer at all. The normal genetic algorithm can in theory have a very bad worst case response. When the answer is used in a hard real-time system, a genetic algorithm will not be the best solution for this type of system because of the insecure answer. Genetic algorithms fits a soft real-time system.

Dasgupta has studied real-time systems where the optimum changes over time. This is often the case in real-time systems because of changes in the

environment around the systems. The solution that Dasgupta proposes is to use a structured genetic algorithm. The algorithm keeps more of the genetic material which makes it less sensitive to changes of the optimum. Dasgupta compares the structured genetic algorithm to a standard genetic algorithm with a high mutation rate. The high mutation rate compensates for the problem that a standard genetic algorithm has to find the shifting optimum. (Dasgupta, 1993)

Chapter 3

Genetic Algorithm Parameters

Optimization is one of the most popular research fields of genetic algorithms (Alander, 1999). One of the reasons for this is because the parameters can not be optimized one by one. The relation between them makes this impossible. The parameters must also be optimized to the problem domain where it is used. A normal genetic algorithm has the following parameters: max generations, population size, crossover rate and mutation rate.

3.1 Exploitation vs. exploration

The best way to evolve a population depends on finding the optimal trade-off between exploitation and exploration of the genetic material. The extreme on the one side is to have no mutation at all and only crossover. With only crossover there is no exploration and it is a big chance that some good genetic material will be missed. On the other side, mutation is the only operation that is used. This will create no exploitation and superior DNA strings will be destroyed. If the superior DNA is lost the algorithm will have more in common with random search algorithms than with evolutionary algorithms.

The conclusion is made because mutation has no mechanism in its nature for protecting genetic material with high survival rate. This comes from the fact that standard mutation does not consider the genetic information before modifying it in the same way as crossover does. If the mutation operator is implemented in a way so it uses the genetic information in the algorithm it can receive a higher survival rate. In this way it can come closer to crossover and the differences become harder to notice. The conclusion from this is that mutation and crossover simply are two cases of a more general operation that can handle all the exploration (Spears, 1993).

3.1.1 Mutation rate

As with much of the research in the genetic algorithm field, a good start to estimate the mutation and crossover rates is to see how nature does it. Studies have shown that crossover is the major reproduction method and mutation

is only used very rarely (Goldberg, 1989). Other studies point out that the mutation rate has a close relationship with the error thresholds. The error threshold is the point where higher mutation rate will destroy the DNA material more frequently than it is constructed. Viruses have shown to live near the error thresholds. (Ochoa et al., 1999)

One conclusion can be drawn from the above and that is that the mutation rate must be set very low, Goldberg stated a general guideline for the mutation rate to be one mutation per thousand bits (Goldberg, 1989). More resent research agrees to this and state that the mutation rate of 0.001 is the optimal if recombination is used (Ochoa et al., 1999). With too high mutation rate the search will turn into a primitive random search.

With a time critical situation the result can be different. An empirical test (Ochoa et al., 1999) has shown that a higher mutation rate is better in the beginning but that the lower wins in the long run. They tested with 0.01 which gave a better result and if the algorithm does not run for too long it can be an improvement.

3.1.2 Crossover rate

When the mutation rate is set too low it is the opposite for the crossover rate. It has been tested that for the De Jong test functions it is recommended to set the crossover rate to 0.6 (Jong, 1975).

3.1.3 Adaptive genetic algorithms

Goldberg stated that crossover is the most important parameter in a genetic algorithm (Dianati et al., 2002). It is not a coincidence that the research has focused on both optimizing the operation and the crossover rate.

The probability of crossover and mutation depend much on the problem domain and how the problem landscape is shaped. Problems with a high risk to get caught in a local optimum need a higher mutation rate to prevent this. Crossover and mutation rate also depend on where in the process the genetic algorithm is. If one mutation rate is optimal in the beginning of the process, and there is much chaos in the population, this does not mean that is the optimal later on when the algorithm is closer to the optimum.

Hinterding et al. classified the types of adaptation into four different categories: static, dynamic deterministic, dynamic adaptive and dynamic self-adaptive (Hinterding et al., 1997).

3.1.3.1 Static genetic algorithms

Static is the normal algorithm with fixed probability for crossover and mutation throughout the evolution. The developer will have to make a balanced decision of the parameters. The decision can consume a lot of time when designing to save time when the algorithm runs, therefore is it common that developers pick a well known parameter setting that may not be perfect for the particular problem (Ochoa et al., 1999).

3.1.3.2 Dynamic deterministic adoption

With Dynamic deterministic the parameters will be changed based on some rule that does not use any feedback from the population. An example of this is to set the mutation rate high and throughout the evolution set the mutation rate lower and lower. See equation 3.1 for an example. (Hinterding et al., 1997)

$$Rate_{mutation} = \frac{0.1}{log(g)} \tag{3.1}$$

3.1.3.3 Dynamic adaptive

With adoptive genetic algorithms the mutation and crossover rate are dependent on the fitness value. When solving maximizing problems, a low fitness value will result in high probability for mutation and crossover. In the same way a high fitness value results in lower mutation and crossover rate. Empirical studies have shown that the adaptive genetic algorithm outperforms the static genetic algorithm in most cases. (Srinivas and Patnaik, 1994)

There exist many versions of dynamic adaptive genetic algorithms. One other example is developed by Ho et al. and called the Probabilistic Rule-based Adaptive Model (Ho et al., 1999). This model divides the population into at least three groups co-evolving. The evolution is separated into different epochs. The parameters are decided in the beginning of each epoch. The feedback from the different parameter setting is then used to set the parameters for the next epoch.

3.1.3.4 Dynamic self-adaptive

With dynamic self-adaptive algorithms the parameters are encoded into the DNA string and evaluated in the same way as the rest of the DNA information. The idea behind this is to let the better individuals have a bigger chance of producing better children. (Hinterding et al., 1997)

3.2 Population size

Population size is one of the most important parameters (Arabas et al., 1994). If it is too low the genetic algorithm will have to few solutions to alter a good result; if it is too high the performance impact may be too great so its takes longer time to reach a good solution.

As with most parts of genetic algorithms the researchers have looked at nature to see how things work there. Studies have shown that a high population size is an advantage in nature. The bigger population has a superior change of surviving a natural catastrophe such as a virus attack and will also have a more stable evolution because it is more resistant to mutations of the DNA. (Alander, 1992)

Different methods have been used to come up with the answer to what the best population size is. One early recommendation about how to set the population is that size should be set to a 'moderate size'. With the De Jong functions this means a recommended population size around 30. (Jong, 1975)

Goldberg has made a theoretical analysis of how to set the population size. He states that more empirical studies must be performed to investigate how accurate his predictions are. (Alander, 1992)

Grefenstette used a genetic algorithm to find the optimal parameter settings for six control parameters. He came to the conclusion that the optimal population size is between 30 and 100. He used the five De Jong test functions in his experiment. (Grefenstette, 1986)

Odetayo tested different population size on a multi-output, unstable, dynamic system (Odetayo, 1993). They compared their results to Grefenstette's and the conclusion was that the population size must be adjusted for different problems. They recommend a population size of 300 for the problem they used in their study.

Alander comes to the same conclusion as Odetayo but they present it as a function of the problem complexity (Alander, 1992). Their conclusion is that problems with higher complexity also needs larger population size.

As an alternative to the different recommendations on how to set the population size, Arabas et al. proposed an algorithm with varying population size (Arabas et al., 1994). The idea behind it is the same as the adaptive genetic algorithms described above, to let the algorithm adopt the parameters to the current situation.

Chapter 4

Experimental setup

4.1 Choice of genetic algorithm

There exists many different implementation and techniques of genetic algorithms. There is no way to test them all, therefore the experiments are only executed by one type of genetic algorithm. The implementation of the genetic algorithm that we used as starting point for the experiment is originally developed by Denis Cormier (Department of Industrial Engineering, North Carolina State University) and is in our opinion, regarding to literature studies a commonly used genetic algorithm (Michalewicz, 1994).

4.1.1 Selection

We wanted the algorithm to be as general as possible so the result of the experiments can be used elsewhere. For this reason we have chosen to use the standard roulette wheel selection.

4.1.2 Mutation and Crossover

We have chosen to follow the recommendations from De Jong and have used a 1-point crossover and normal mutation with a crossover rate of 0.6 and mutation rate of 0.001 to be used in the experiment. The rate is calculated in the experiments so it is a rate of individuals and not bits as the original rates. With the rate is it possible that more than one chromosome is mutated for each individual. We have limit it so the individuals can at most have one mutated chromosome per generation. The formula used is:

$$Rate_{individual} = parameters * sizeof_{parameter} * Rate_{bits} \tag{4.1}$$

4.1.3 Parameter to binary representation

We represent the DNA string in binary form and it is converted into its real value when the fitness value is calculated. The values are saved in the C type unsigned long long. We then convert this to a double with the formula 4.2. Gen_{min} and Gen_{max} are the interval for the De Jong function that are used

and $Max_{ulonglong}$ is the max value for a unsigned long long.

$$Value_{double} = (Value_{ulonglong} * (Gen_{max} - Gen_{min})/Max_{ulonglong}) + Gen_{min} \tag{4.2}$$

4.1.4 Random

We used a pseudo-random number generator called 'ran2' in our experiment (Press et al., 1993). It has a period of more than 10^{18} numbers, so it is more then enough for the experiment. We have used the implementation from Galib (Wall, 2003). We chose not to use the system supplied rand() function because we needed the numbers to be more random than this function could give us. The normal ANSI C implementation standard only requires a minimum period of 32767. The advantage this function has is that it is very fast, but that was not a requirement from our side.

To generate seeds to the random number generator, we have used true random numbers from random.org (Haahr, 1998). This is done with atmospheric noise from a radio, that is then used to generate the numbers.

We have chosen this combination of a pseudo-random number generator together with true random numbers as the seed to get the best combination of performance and randomness.

4.2 De Jong test functions

In 1971 De Jong created a number of mathematical functions for testing evolutionary algorithms with. These test functions are continuous and are considered by many to be a minimum standard performance comparison for evolutionary algorithms, which genetic algorithms are a part of. The test suite has become a classic set and contains a set of 5 mathematical functions that isolate the common difficulties found in many optimization problems (Goldberg, 1989). The functions characteristics include:

- continuous/discontinuous
- convex/non-convex
- unimodal/multi-modal
- quadratic/non-quadratic
- low-dimensionality/high-dimensionality
- deterministic/stochastic

When a function is discontinuous, not all points are connected. A function is convex if there for every two points in the funcion, all the points between them are also inside of the function. A function that is unimodal has only one optimum, but a multimodal has different local optima as well. If a function is quadric, a single parameter can have two roots. The function is stochastic if the result is random based.

We have chosen to work with the De Jong test functions because they are well tested, so we can compare our result to others. Because of the different

characteristics that De Jong's functions has are we able to test our hypotheses over a large set of mathematical functions.

The functions were originally made as minimization problems, but the functions can be easily translated into maximization problems. It is the later version that has been used in this experiment.

4.2.1 Test function 1 or Sphere

This three-dimensional function is a simple, non-linear, convex, unimodal, symmetric function that poses very little difficulty for most optimization methods. The sphere is intended to be a performance measure of the general efficiency of an algorithm.

$$f_1 = \sum_{i=1}^{3} x_i^2, x_i \epsilon [-5.12, 5.12] \tag{4.3}$$

4.2.2 Test function 2 or Rosenbrock's Saddle

This two-dimensional function is a standard test function in optimization literature and was first proposed by Rosenbrock in 1960 (Jong, 1975). The function is smooth, unimodal and can be quite difficult for algorithms that are unable to identify promising search directions with little information. The difficulty lies in transiting a sharp, narrow ridge that runs along the top of the saddle in the shape of a parabola.

$$f_2 = 100(x_1^2 - x_2)^2 + (1 - x_1)^2, x_i \epsilon [-2.048, 2.048] \tag{4.4}$$

4.2.3 Test function 3 or Step

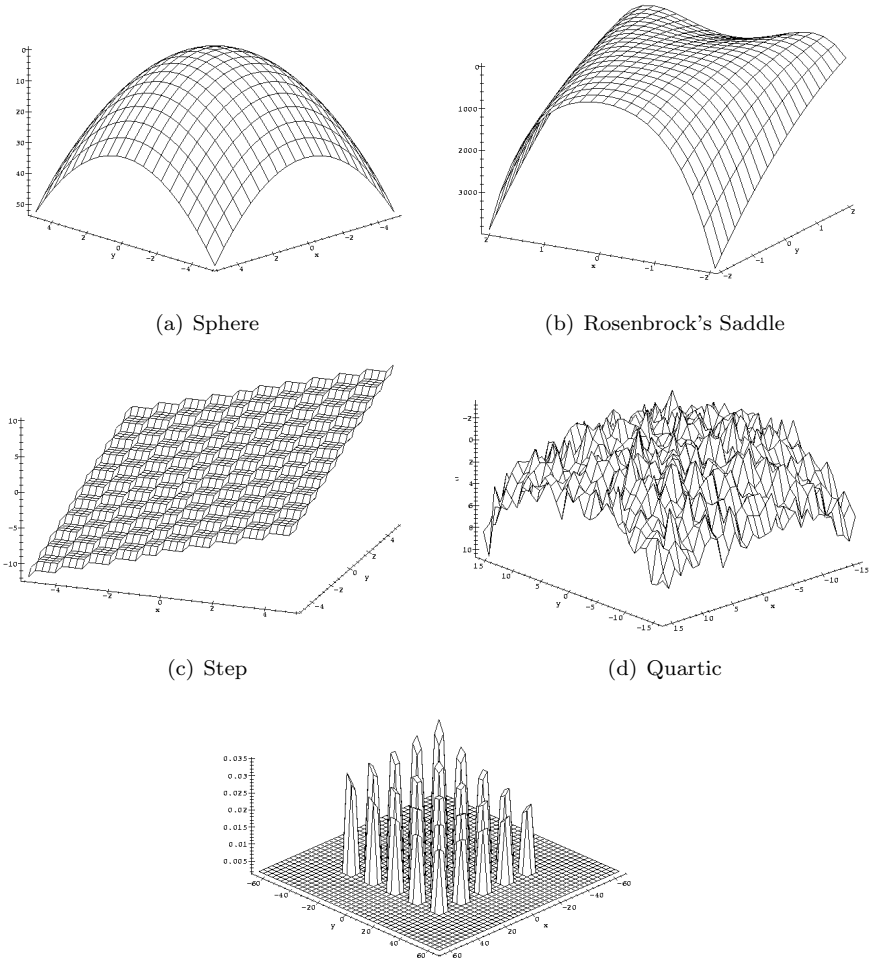
This five-dimensional function that is discontinues, consists of many flat plateaus with uniform, steep ridges. This function poses considerable difficulty for algorithms that require gradient information to determine a search direction and it is easy for ordinary optimization algorithms to get stuck on one of the flat plateaus.

$$f_3 = \sum_{i=1}^{5} floor(x_i), x_i \epsilon [-5.12, 5.12] \tag{4.5}$$

4.2.4 Test function 4 or Quartic

This 30-dimensional function includes a random noise variable that ensures the function never evaluates to exactly the same value for the same solution vector. Thus, the quartic function tests an algorithm's ability to locate the global optimum for a simple unimodal function that is padded heavily with Gaussian noise.

$$f_4 = \sum_{i=1}^{30} (ix_i^4 + Gauss(0, 1)), x_i \epsilon [-1.28, 1.28] \tag{4.6}$$



(a) Sphere

(b) Rosenbrock's Saddle

(c) Step

(d) Quartic

(e) Shekel's Foxholes

Figure 4.1: 2-dimensional versions of De Jong's test functions.

4.2.5 Test function 5 or Shekel's Foxholes

This two-dimensional function contains many (i.e., 25) foxholes of varying depth surrounded by a relatively flat surface. Many algorithms will become stuck in the first foxhole they fall into.

$$a = \begin{bmatrix} -32 & -16 & 0 & 16 & 32 & -32 & -16 & \dots & 0 & 16 & 32 \\ -32 & -32 & -32 & -32 & -32 & -16 & -16 & \dots & 32 & 32 & 32 \end{bmatrix}$$

$$f_5 = 0.002 + \sum_{j=1}^{25} \frac{1}{j + \sum_{i=1}^{2} (x_i - a_{ij})^6}, x_i \epsilon [-65.356, 65.356] \tag{4.7}$$

4.3 Execution of the test

The test exists in three parts. First we generate a log file with test data from our genetic algorithm and translated it to graphs where the tests shows how the genetic algorithm performed over generations. The second part of the test was to transform the test data so it shows how the genetic algorithm performed in time instead of generations. The last part we calculated the optimal population that is generating the maximum fitness for each graphs.

4.3.1 Log file generation

Every De Jong function is executed by the developed genetic algorithm. The simulation data is stored in a log file containing all the necessary information such as population, generation, time and fitness for every logging point. The randomness in a genetic algorithm can effect the result so it suddenly gets a better or worse result then it does in general, to partially eliminate this source of error every test has an average from 10 tests. This data is later translated to three-dimensional graphs. One graph is showing how long time it takes to generate a certain generation on a specified population. The second graph shows the fitness for a certain generation on a specified population.

4.3.2 Graph transformation

To see how the genetic algorithms change the fitness in time rather than in generations a transformation of the data is made. Because the data does not have all the data for every point an interpolation has to be made. This is done by the nearest neighbor technique, see equation 4.8 and figure 4.2 for clarification.

$$f(p,\hat{t}) = f(g,p), when \mid \hat{t} - t(g,p) \mid is \min \tag{4.8}$$

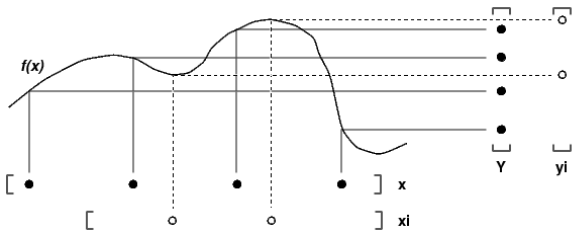


Figure 4.2: Interpolation, the image is taken from the Matlab documentation. The interpolation of the table [x,Y] looks up the elements of xi in x, and, based upon their locations, returns values yi interpolated within the elements of Y.

4.3.3 Maximum fitness generation

Because the Genetic Algorithm is based on randomness, a solution can be better or worse then it is in general, action has to be made to ensure that the conclusion is believable. The maximum population for a specified time or generation is

calculated by a formula. If the maximum exists on a certain population then the adjacent population will also be good. The formula for which population that generates the maximum fitness can be seen in equation 4.9 and the formula for which generation that generates the maximum fitness can be seen in equation 4.10. Because these formulas include the adjacent population this source of error is minimized because if a certain population is good then the adjacent population should be almost as good.

The calculation for the maximum fitness for each generation or for each time frame $\hat{f}$ is done by altering the population

$$\hat{f}(p,t) =$$

$$\max_{p \epsilon P} \left(\frac{f(p-2,t) + 2 * f(p-1,t) + 4 * f(p,t) + 2 * f(p+1,t) + f(p+2,t)}{10} \right) \tag{4.9}$$

$$\hat{f}(p,g) =$$

$$\max_{p \epsilon P} \left(\frac{f(p-2,g) + 2 * f(p-1,g) + 4 * f(p,g) + 2 * f(p+1,g) + f(p+2,g)}{10} \right) \tag{4.10}$$

Chapter 5

Analysis of the results

5.1 Explanation of the figures

Figures 5.1-5.5 show the development of the fitness value over generations. The X axle is the population Size and the Y axle is the generation. From these figures we could then find the optimal population size over generation that is presented in figures 5.6-5.10. We then used interpolation to replace generations with the execution time for the experiments. This is shown in figures 5.16-5.20. From these figures we could then in the same way as above find the optimal population size but in this case over time instead of generations. This is shown in figures 5.11-5.15. We needed to have the executions times for the interpolation and these are presented in figures 5.21-5.25.

We extracted some static data from figures 5.11-5.15 and this is put together in table 5.1. We did the same with figures 5.11-5.15 and this is put together in table 5.2.

5.2 Analysis of the results

It is not easy to describe the optimal population as a parametric function because of the complexity of the relations between the different parameters. We do not know what this function looks like so it is as a black box function for us. Alander is among those who have tried to come up with such a function (Alander, 1992). The calculation for a such function is not easier when time is added as an extra dimension to the problem.

To start with we can see in Table 5.1 that the optimal population depends much on the problem that is optimized. We draw this conclusion because we have used the same implementation and parameter setting for all test functions.

As we can see in the Figures 5.1-5.4 and 5.6-5.9, a high population is best in the beginning and then landing in a lower population. This is not so strange because if there is a big population the search become more stochastic and there is a higher probability that a good solution is found. However later it is more important to narrow the search to assume existence of in general more good genetic material. This is not the case when we look at the time rather than generations, as we can see in Figures 5.16-5.19 and 5.11-5.14 the population is

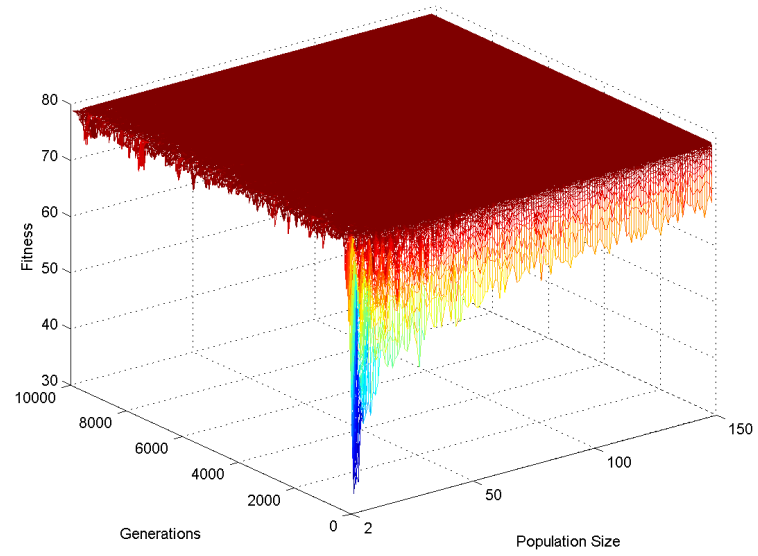


Figure 5.1: De Jong function 1 developing over generations.

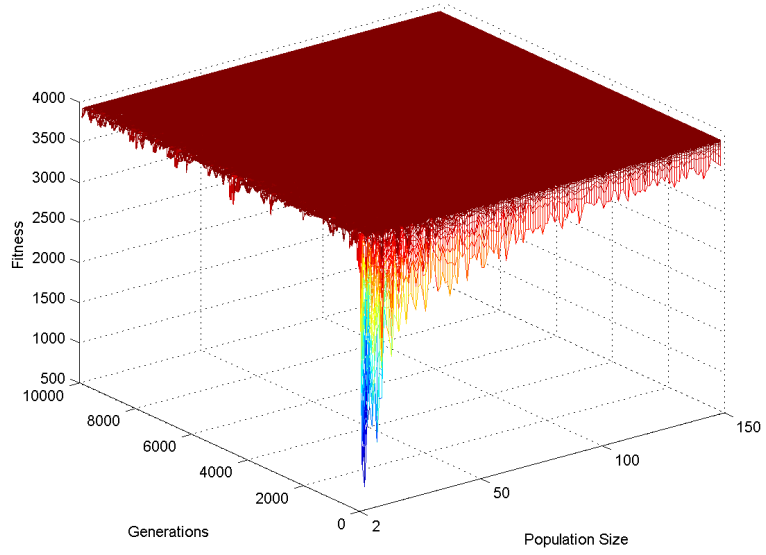


Figure 5.2: De Jong function 2 developing over generations.

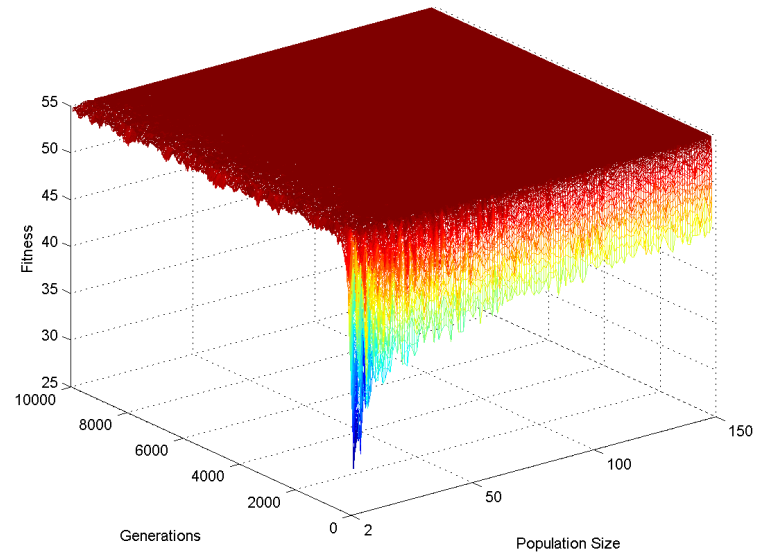


Figure 5.3: De Jong function 3 developing over generations.

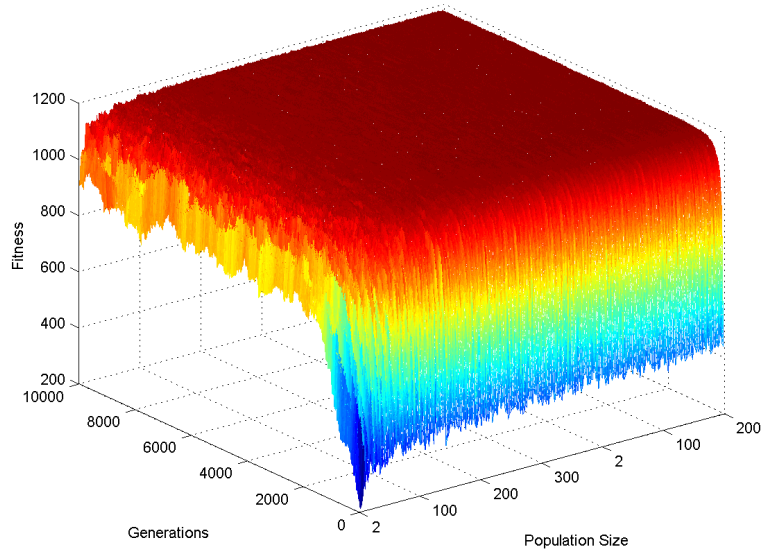


Figure 5.4: De Jong function 4 developing over generations.

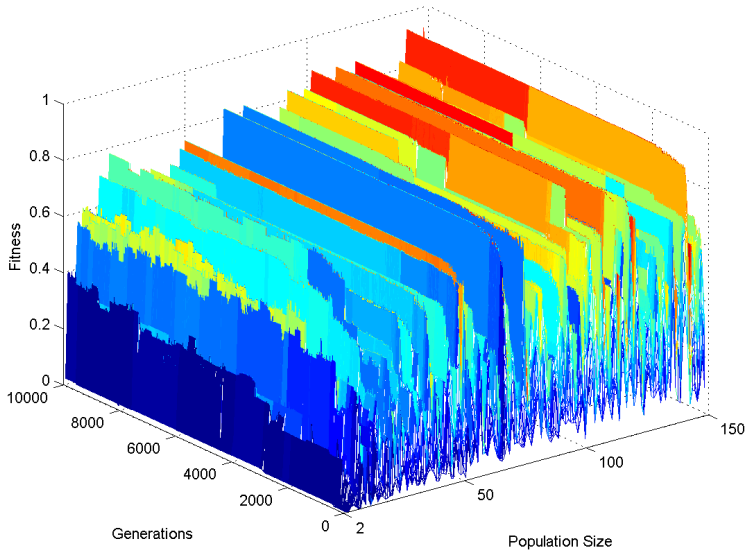


Figure 5.5: De Jong function 5 developing over generations.

not so high at the beginning. This is because with a high population a higher initialization time is needed.

De Jong	1	2	3	4	5
min	3	3	6	3	3
max	43	34	54	52	143
mean	25.49	16.06	20.7	14.02	132
median	26.5	16	21.5	9	142
std	7.03	4.784	7.774	9.777	30.59

Table 5.1: Result from time critical test.

5.3 Time and Performance measuring

In the Figures 5.21-5.24 we can see that for each population, the time is direct proportional to generation and the difference in time by each population is in the form $T(p) = ap^2 + bp + c$, this agreed with our predication when we calculated the time by looking at the code.

To get the optimal fitness even when the algorithm has stabilized, the result is calculated in a long timescale and therefore the graphs flattened.

De Jong divided the method for performance measuring into two definitions, on-line and off-line performance. On-line performance is used to measure the situations where the genetic algorithm dynamically adjusts the system. Off-line measures the systems where the genetic algorithm works independently from the rest of the system. (Jong, 1975)

De Jong	1	2	3	4	5
min	9	3	9	144	122
max	150	150	150	300	143
mean	28.07	17.09	25.74	275.1	110
median	26	16	22	283	104
std	13.65	10.54	11.44	24.01	14.07

Table 5.2: Result from non-time critical test.

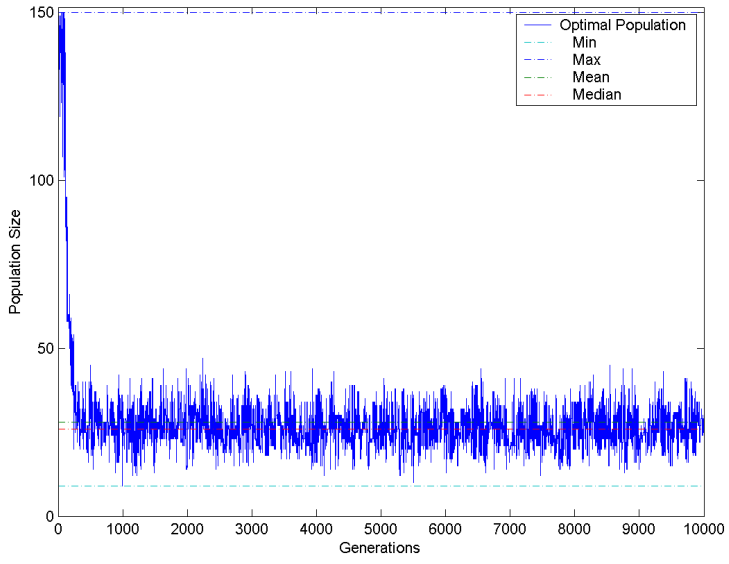


Figure 5.6: The optimal population size for De Jong function 1 over generations.

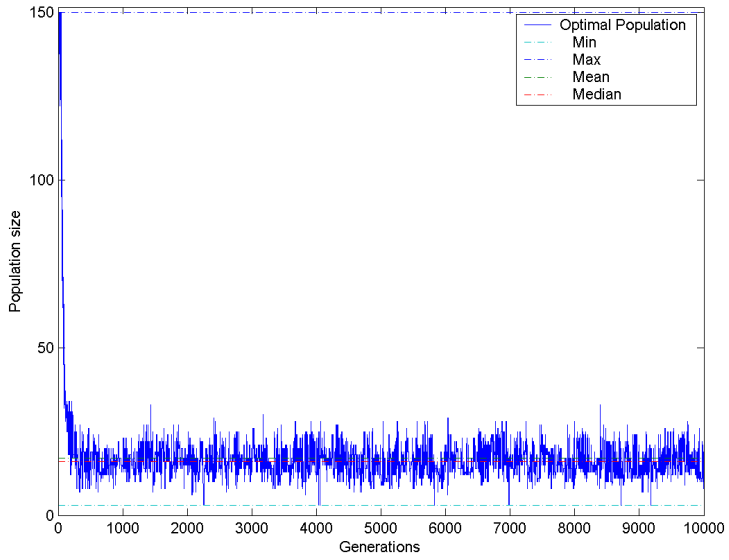


Figure 5.7: The optimal population size for De Jong function 2 over generations.

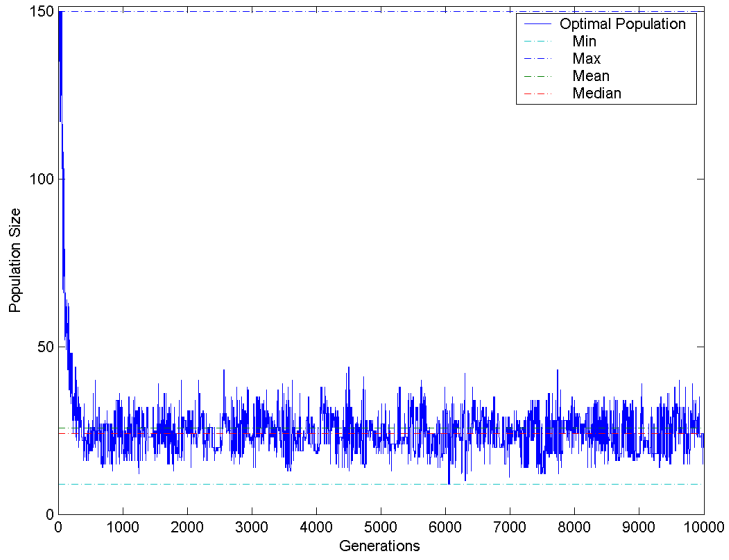


Figure 5.8: The optimal population size for De Jong function 3 over generations.

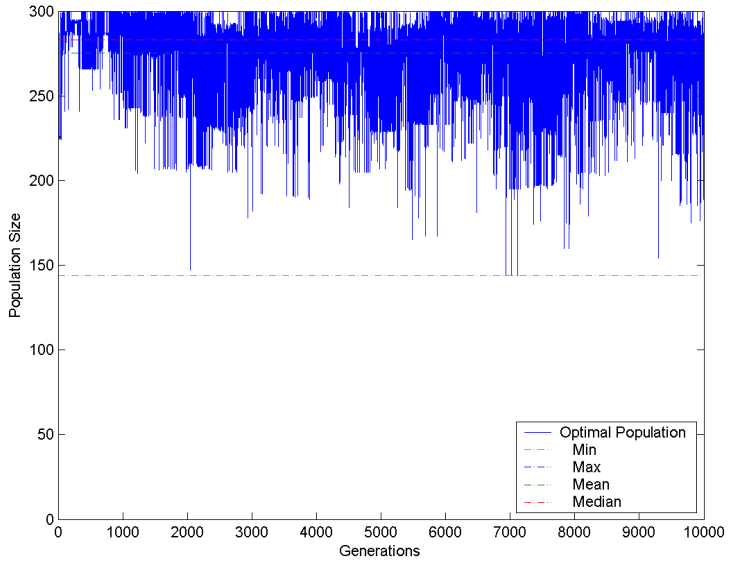


Figure 5.9: The optimal population size for De Jong function 4 over generations.

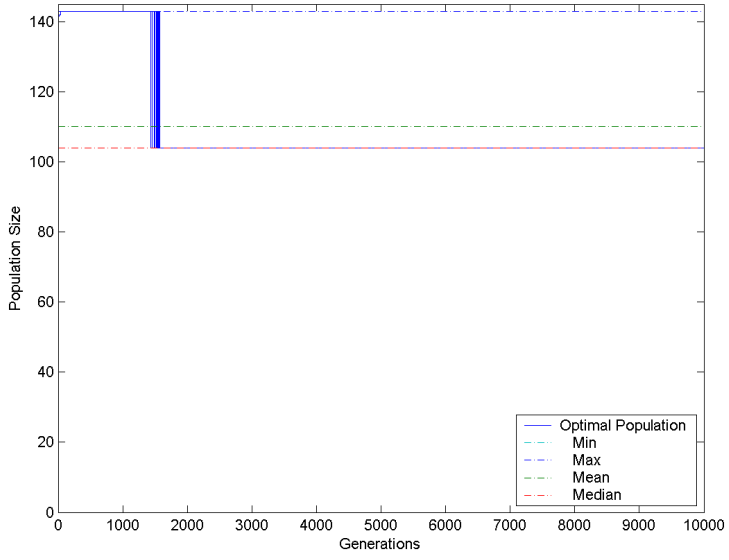


Figure 5.10: The optimal population size for De Jong function 5 over generations.

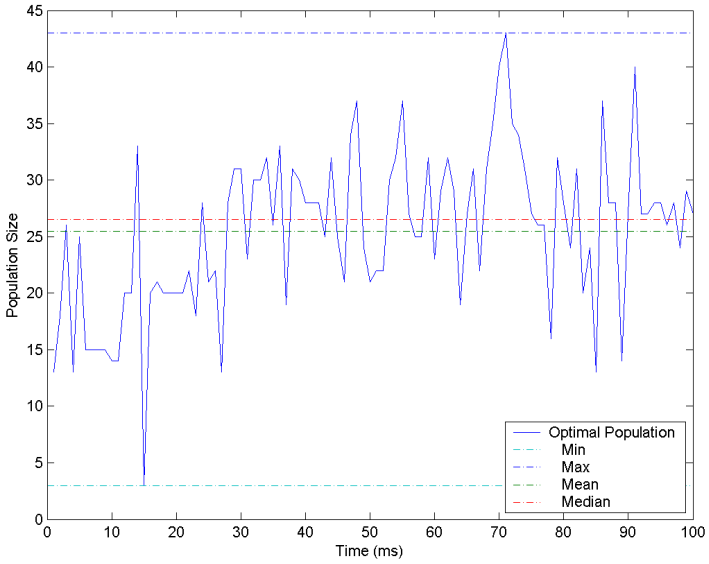


Figure 5.11: The optimal population size for De Jong function 1 over time.

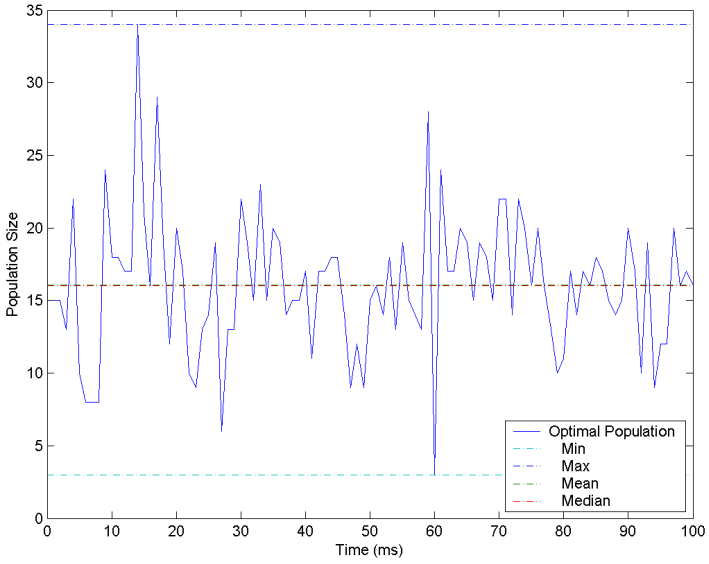


Figure 5.12: The optimal population size for De Jong function 2 over time.

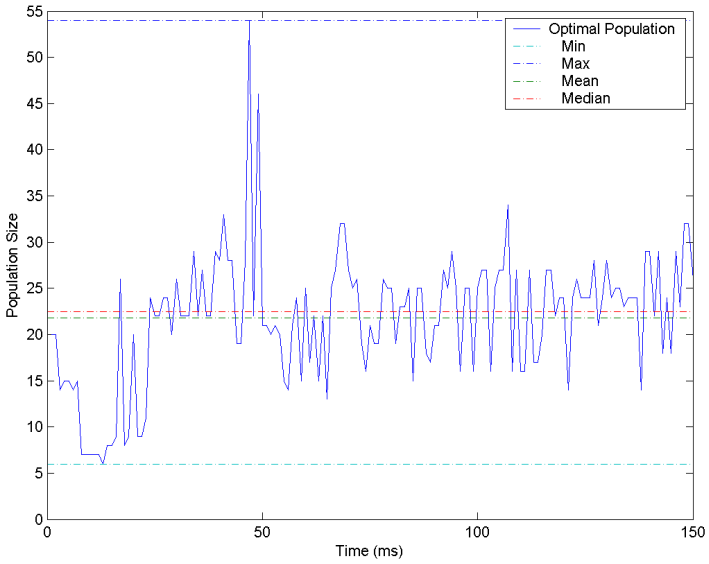


Figure 5.13: The optimal population size for De Jong function 3 over time.

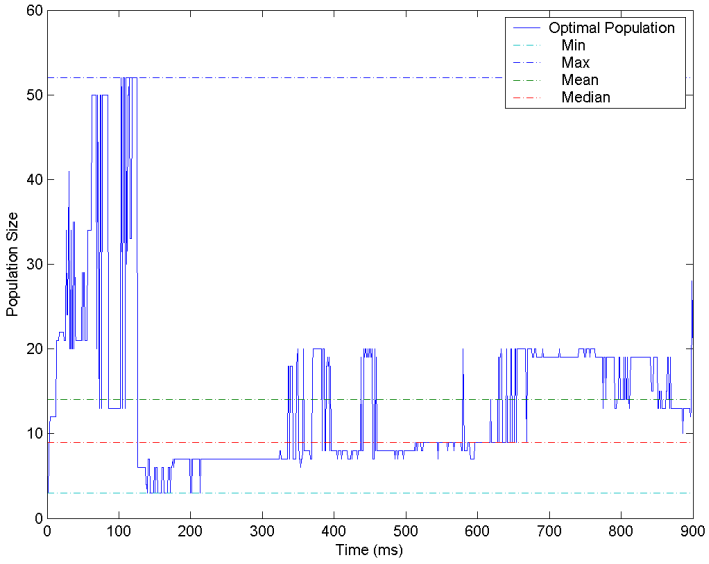


Figure 5.14: The optimal population size for De Jong function 4 over time.

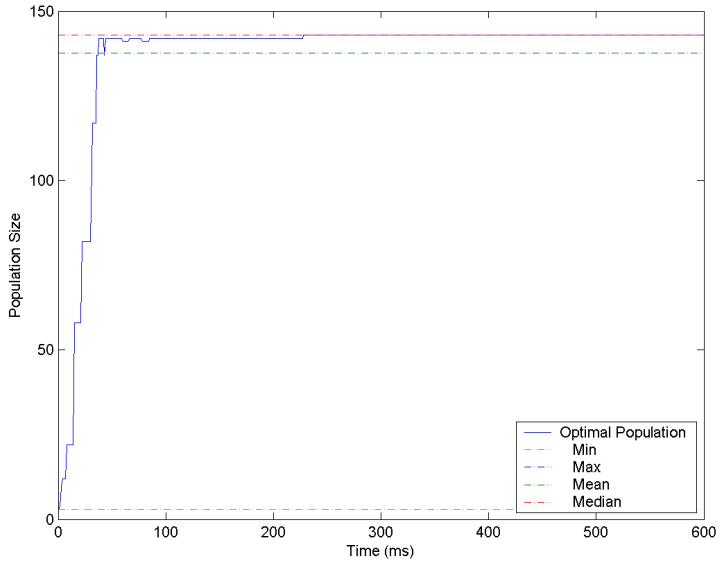


Figure 5.15: The optimal population size for De Jong function 5 over time.

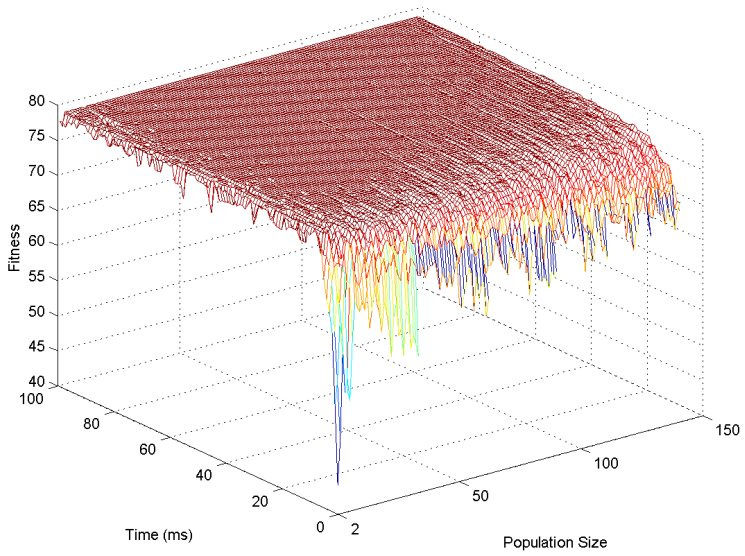


Figure 5.16: De Jong function 1 developing over time.

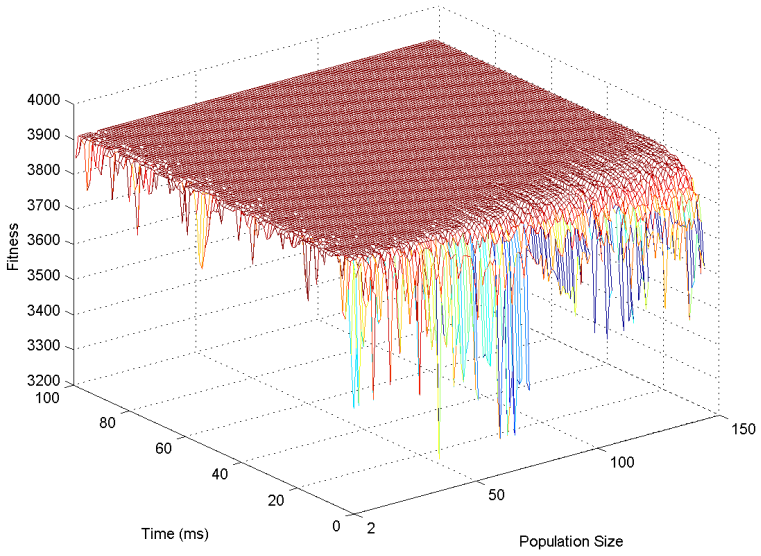


Figure 5.17: De Jong function 2 developing over time.

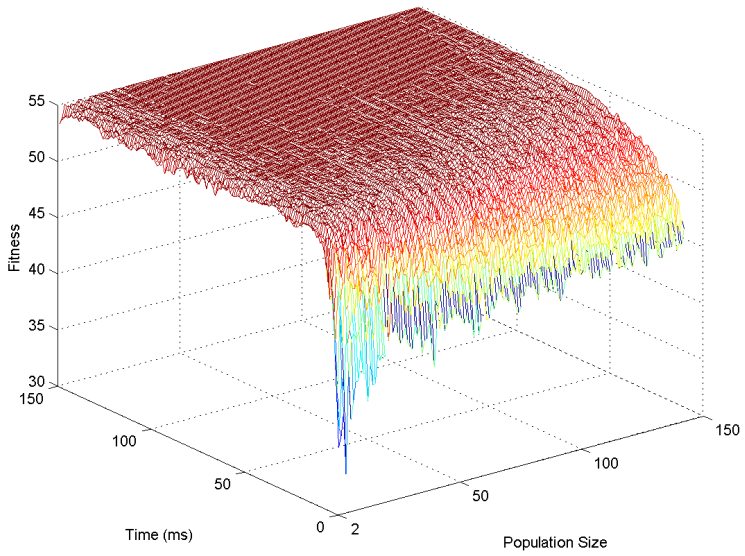


Figure 5.18: De Jong function 3 developing over time.

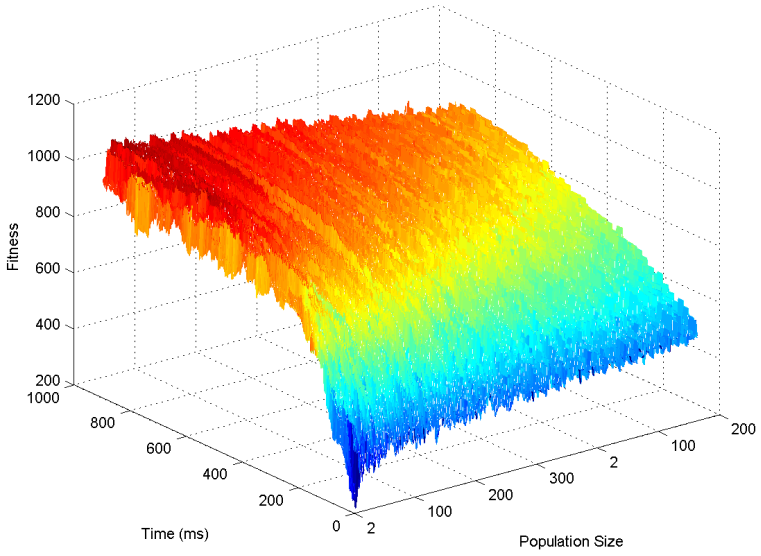


Figure 5.19: De Jong function 4 developing over time.

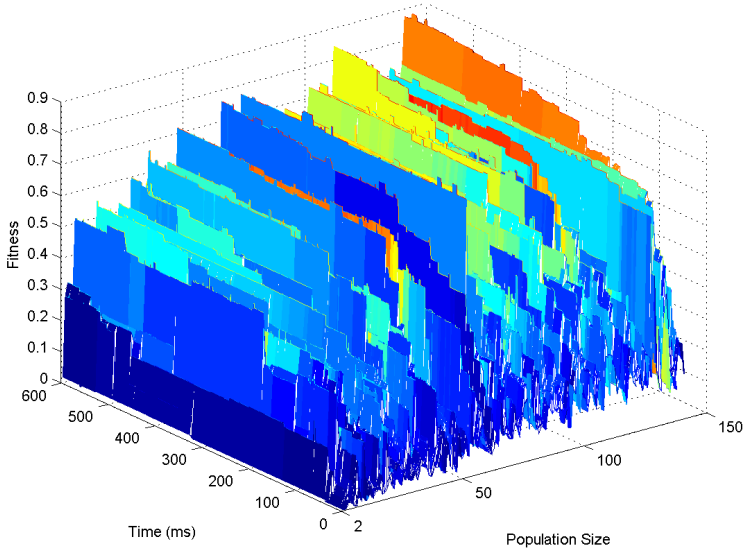


Figure 5.20: De Jong function 5 developing over time.

5.4 Comparison to others studies

Alander comes to the same conclusion as we have, that the optimal population depends on the complexity of the problem. (Alander, 1992)

De Jong tested with population sizes 50, 100 and 200 for test function 1. He found that 50 works best for on-line performance and 200 for off-line performance. (Jong, 1975)

We have made a more complete experiment of population size in time critical systems than De Jong. We have focused on trying to test a wide range of population sizes for all five test functions. When we compare De Jong's result with ours we can see that even if the result shows the same thing (lower population is better for online performance) the optimal population size is not the same. Why the optimal population size is not the same is depending on several things. De Jong used only three cases of population size, 50, 100 and 200, but we used the whole scope between 2 and 150. De Jong had also a small difference in his implementation, where he used generation gap as one more parameter. The generation gap means that individuals can survive through generations without being changed.

As said earlier in this report, Grefenstette used an alternative way to find the optimal parameter settings for the De Jong test functions (Grefenstette, 1986). With a genetic algorithm he came to the same conclusion that we did that to get a good on-line performance it is better with a smaller population size than off-line performance. He recommends 30 to 100 for on-line and 60 to 110 for off-line performance.

We found it hard to compare our research method with the one that Grefenstette used because of the differences in the studies. But his recommendations are in line with the one that we have found with our experiment and the one that De Jong stated.

5.5 Function to find optimal population size

If the optimal population can be described as a black box function, the function for finding it when time is involved should also be possible to describe as a black box function. The difference is that $T(p)$ is added into the function. It means that a higher population will result in fewer generations, so the genetic operators such as crossover, mutation and selection will not be executed as much as with a low population. We can see the affect this has on the result in Table 5.1 and 5.2. The optimal population is in general a bit lower when time is added in the function. Alander comes to the conclusion that this is because a larger population responsed more slowly but it is better for a long term goal because of the reduced rate of allele loss (Alander, 1992).

5.6 De Jong 5

In our analysis of the result we have disregarded De Jong 5. The nature of De Jong 5, seen in 4.1(e), makes the result too random to make any good conclusions. Between two simulations the best population size can shift in a range of 100 populations. This will not effect our hypothesis because we notice

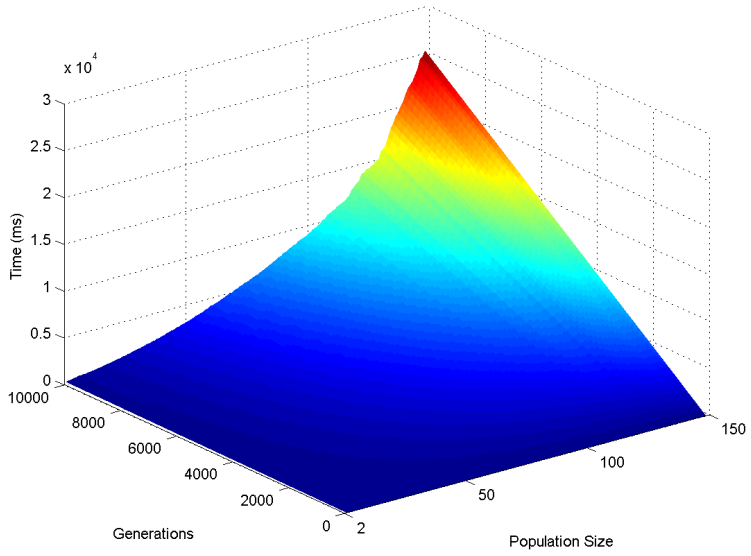


Figure 5.21: The execution time for De Jong function 1.

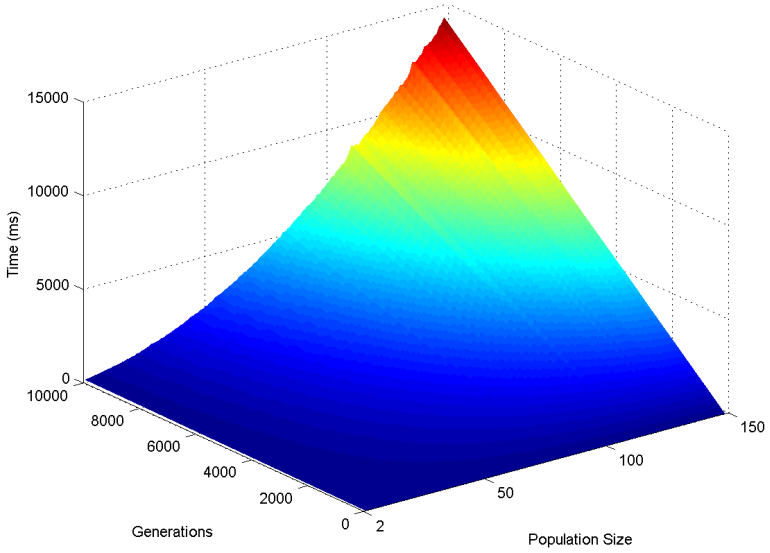


Figure 5.22: The execution time for De Jong function 2.

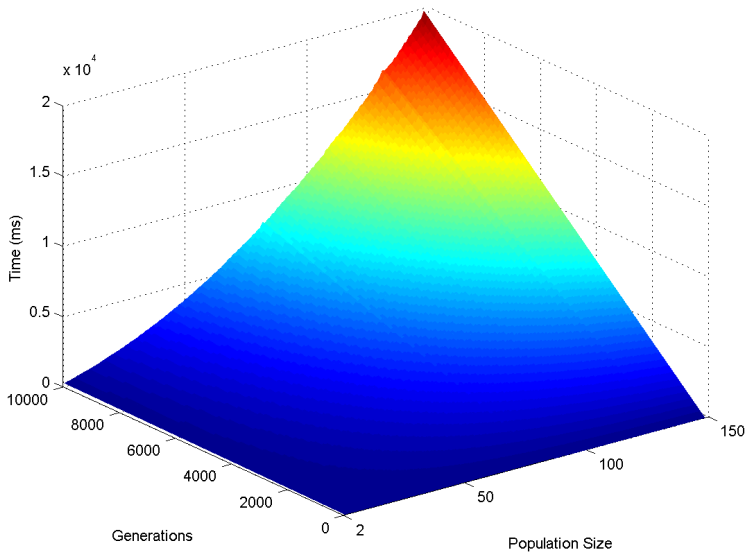


Figure 5.23: The execution time for De Jong function 3.

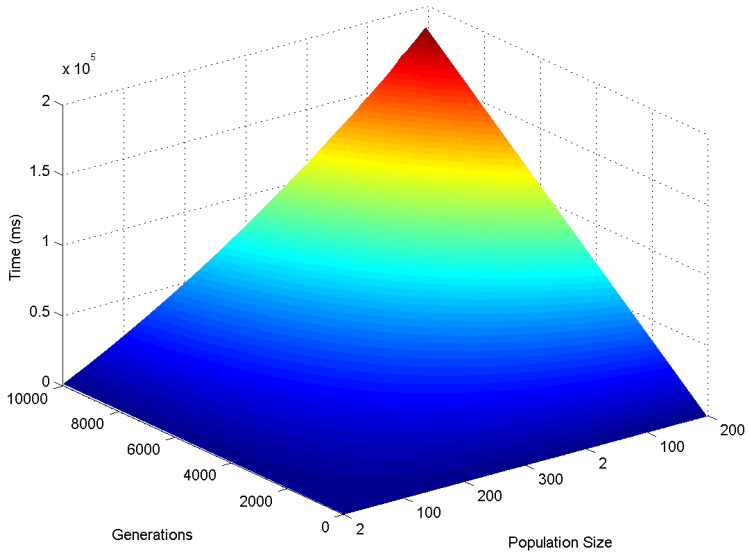


Figure 5.24: The execution time for De Jong function 4.

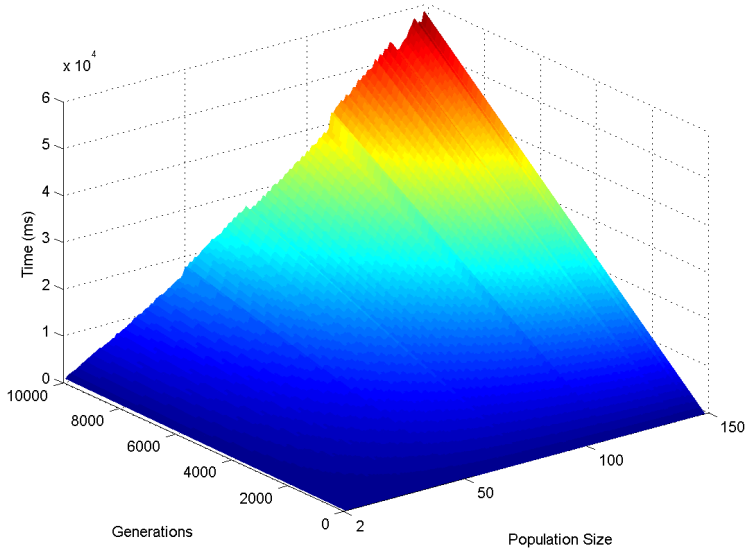


Figure 5.25: The execution time for De Jong function 5.

that the optimal population size differs between if the problem concerning time or generation in other functions.

5.7 Functions characteristics

As we can see if we compare the result from Tables 5.1, 5.2 with 5.3 we can draw some conclusions. First of all we can see that the test function 4 who has high dimensions and is stochastic and test function 5 that is multimodal needs a much higher population size.

We can also see if we compare Table 5.3 with the Figures 5.18 and 5.19 that test function 3 and 4 need more time to come to a good solution. From the tables we can see that the characteristics that is special from these two functions is that the test function 3 is non-continuous and the test function 4 is high-dimension.

De Jong	1	2	3	4	5
Continuous	Y	Y	N	Y	Y
Convex	Y	N	N	Y	N
Unimodal	Y	Y	Y	Y	N
Quadratic	Y	Y	N	Y	N
Dimensions	3	2	5	30	2
Deterministic	Y	Y	Y	N	Y

Table 5.3: The characteristics of the test functions.

Chapter 6

Discussion

From the experiment we noticed that genetic algorithms should be configured differently in a time critical context then in a context where the time is not a quality attribute.

From the literature survey we saw that many reports have referenced De Jong's PhD thesis (Jong, 1975) and also John J Grefenstette's work (Grefenstette, 1986). After the experiments were performed we were able to find these works and saw that they have focused on the same problem. We also noticed that they drew just about the same conclusion as we did.

During our studies we have tried to make a fair study of the population size on real-time problems. However, we had some problems deciding on how this should be done. We tested different time measures and ways to plot the data. We finally measured the time with help of the cycle counter in the processor, to get a calculation as exact as possible and run all the tests on equivalent computers. The exactness of the time measurement was effected by other processes that were running on equivalent computers, but not so big that they will effect the result in a noticeable different way. This problem can be seen in Figures 5.21-5.25, where the curves are not even.

One possibility or in the case a problem with genetic algorithms is that crossover, mutation and selection etcetera can be combined in any way. It is hard to know how different combinations can affect the result. This can make it hard to compare the result from different studies. This also makes it harder to move our result to other genetic algorithms and get the same result.

Chapter 7

Conclusions and future work

We began this thesis by introducing what a genetic algorithm is and how it works. We also made a literature survey to see different approaches to genetic algorithms and how others have calculated the different parameters. Except to gather material from previous work, we tried to see how a genetic algorithm behaves in a time critical context.

Because there is a difference between the optimal population size when the problem focus on time instead of generation. Our hypotheses is true for the functions that we have tested. However we have not been able to prove this statement for test function 5.

We found that our hypothesis should always be in consider when designing a genetic algorithm in time critical systems. The analysis of the result from the simulations show that the results we get are different if time is concluded. From the literature survey we noticed that the population size is very often calculated regarding to the generations and not taking any consideration to the time. This conveys that the calculation can not be used in a context which involve time.

We found that the population size may be smaller when time constraints are present, but need more tests to ensure that this is always the case. This is because it is affected by more factors such as the complexity of the problem.

7.1 Future Work

When we made this report we came across some new questions but we did not have the time to investigate them further.

When a genetic algorithm is in a time critical context it can be better to run it several times in short periods of time instead of running it once for a long time period. Such an approach can be better if the characteristic of the problem resemble De Jong 5 function 4.1(e).

We only had time to investigate how the population size differs in real-time systems. However, we believe that the other parameters that a genetic algorithm has also affect the performance in such a way that they must be optimized for time critical problems before the genetic algorithm itself can be said to be completely optimized.

Bibliography

Jarmo T. Alander. On optimal population size of genetic algorithms. In *Proceedings of the CompEuro 92. Computer Systems and Software Engineering*, pages 65–70. IEEE Comput. Soc. Press, 1992.

Jarmo T. Alander. Genetic algorithms - an introduction. Technical report, University of Vaasa, Department of Information Technology and Production Economics, Finland, 1999.

Jaroslaw Arabas, Zbigniew Michalewicz and Jan Mulawka. Gavaps - a genetic algorithm with varying population size. In *Proceedings of the First IEEE Conference on Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence.*, volume 1, pages 73–78. IEEE, 1994.

Thomas Back, Frank Hoffmeister and Hans-Paul Schwefel. A survey of evolution strategies. In Morgan Kaufmann, editor, *Proceedings of the 4th International Conference on Genetic Algorithms*, pages 2–9. IEEE Press, San Diego, CA, USA, 1991.

H. J. Bremermann. Optimization through evolution and recombination. In M.C. Yovits, G. T. Jacobi and G.D. Goldstine, editors, *Proceedings Conferance on Self-organizing Systems*, pages 93–106. Spartan Books, Washington, DC, 1962.

Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages*. Addison-Wesly, second edition, 1989.

Charles Darwin. *On the origin of species*. John Murray, London, 1859.

Dipankar Dasgupta. Optimisation in time-varying environments using structured genetic algorithms. Technical Report IKBS-17-93, University of Strathclyde, Department of Compputer Science, Glasgow U.K., 1993.

Lawrence Davis. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, New York, 1991.

Mehrdad Dianati, Insop Song and Mark Treiber. An introduction to genetic algorithms and evolution strategies. 2002.

Marco Dorigo and Luca Maria Gambardella. Ant colony system: A cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1(1):53–66, 1997.

David E. Goldberg. *Genetic Algorithms, in Search, Optimization & Machine Learning*. Addison-Wesley, 1989.

John J Grefenstette. Optimization of control parameters for genetic algorithms. *IEEE Transactions on Systems, Man, and Cybernetics*, 16(1):122–128, 1986.

Mads Haahr. random.org. 1998. Www.random.org, last checked june 2003.

Babak Hamidzadeh and Shashi Shekhar. Specification and analysis of real-time problem solvers. In *IEEE Transactions on Software Engineering, Man and Cybernetics*, volume 19, pages 788–802. IEE/IEEE, 1993.

Hinterding, Michalewicz and Eiben. Adaptation in evolutionary computation: A survey. In *IEEECEP: Proceedings of The IEEE Conference on Evolutionary Computation, IEEE World Congress on Computational Intelligence*. 1997.

C.W. Ho, K.H. Lee and K.S. Leung. A genetic algorithm based on mutation and crossover with adaptive probabilities. In *Proceedings of the 1999 Congress on Evolutionary Computation, CEC 99.*. IEEE, 1999. ISBN 0780355369.

John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. MIT Press, 1975.

Kenneth Allan De Jong. *An Analysis of the Behavior of a class of genetic adaptive systems*. Ph.D. thesis, The University of Michigan, 1975.

J. R. Koza. Hierarchical genetic algorithms operating on populations of computer programs. In N. S. Sridharan, editor, *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence IJCAI-89*, volume 1, pages 768–774. Morgan Kaufmann, 1989.

Gregor Mendel. Versuche ber pflanzen-hybriden. *Verhandlungen des naturforschenden Vereins in Brnns*, 1886. Availible at http://www.mendelweb.org/MWGerText.html.

Zbigniew Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag, second, extended edition, 1994. ISBN 3-540-58090-5.

Melanie Mitchell and Stephanie Forrest. Genetic algorithms and artificial life. *Artificial Life*, 1(3):267–289, 1994.

David John Musliner, James Hendler, Ashok K. Agrawala, Edmund H. Durfee, Jay K. Strosnider and C. J. Paul. The challenges of real-time AI. Technical Report CS-TR-3290, University of Maryland Institute for Advanced Computer Studies, 1994.

Gabriela Ochoa, Inman Harvey and Hilary Buxton. On recombination and optimal mutation rates. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 1, pages 488–495. Morgan Kaufmann, Orlando, Florida, USA, 1999. ISBN 1-55860-611-4.

Michael O. Odetayo. Optimal population size for genetic algorithms: an investigation. *IEE Colloquium on Genetic Algorithms for Control Systems Engineering*, 1993.

William H. Press, Saul A. Teukolsky, William T. Vetterling and Brian P. Flannery. *Numerical Recipes in C*. Cambridge University Press, Cambridge, UK, 1993.

Khaled Rasheed. Guided crossover: A new operator for genetic algorithm based optimization. In Peter J. Angeline, Zbyszek Michalewicz, Marc Schoenauer, Xin Yao and Ali Zalzala, editors, *Proceedings of the Congress on Evolutionary Computation*, volume 2, pages 1535–1541. IEEE Press, Mayflower Hotel, Washington D.C., USA, 1999. ISBN 0-7803-5537-7 (Microfiche).

William M. Spears. Crossover or mutation? In L. Darrell Whitley, editor, *Foundations of Genetic Algorithms 2*, pages 221–237. Morgan Kaufmann, San Mateo, CA, 1993.

William M. Spears. Adapting crossover in a genetic algorithm. In J. R. McDonnell, R. G. Reynolds and D. B. Fogel, editors, *Proc. of the Fourth Annual Conference on Evolutionary Programming*, pages 367–384. MIT Press, Cambridge, MA, 1995.

M. Srinivas and L.M. Patnaik. Adaptive probabilities of crossover and mutation in genetic algorithms. In *IEEE Transactions on Systems, Man and Cybernetics*, volume 24, pages 656–667. IEE/IEEE, 1994.

Matthew Wall. Galib 2.4.2. 2003. Lancet.mit.edu/ga/, last checked june 2003.