TESINA FINAL DE CARRERA

Títol

Metaheuristics and their application in engineering optimization

Autor/a

David Alan López Martínez

Tutor/a

Antonio Rodríguez Ferran

Departament

Matemàtica aplicada

Intensificació

Mètodes numèrics

Data

Octubre 2012

Abstract

Title: Metaheuristics and their application in engineering optimization

Author: López Martínez, David Alan

Tutor: Rodríguez Ferran, Antonio

Every engineering design problem aims for the optimal solution that, for instance, maximizes efficiency while minimizing cost. The use of optimization techniques in order to achieve such goal is known as engineering optimization. In the last few years this subject has drawn the attention of many engineers to optimization algorithms.

The present document studies the possibility of using metaheuristics, a recently devised type of computational methods, in engineering optimization. Metaheuristics are nature inspired optimization algorithms that have already been applied in many areas. To be able to conclude if these algorithms perform well when facing engineering design problems, several classic optimization methods and two metaheuristics, namely Genetic Algorithm and Simulated Annealing, have been implemented. The different methods have then been used to solve optimization problems related to engineering, such as structural design or logistics.

According to the results of the simulations carried out, Genetic Algorithm and Simulated Annealing outperform classic methods in all the problems proposed. Genetic Algorithms excel in unconstrained continuous optimization and combinatorial optimization while Simulated Annealing is best at constrained optimization. Most of the time the metaheuristics studied are capable of reducing the error obtained by classic methods even for simulation times under one second.

The conclusion extracted from this dissertation is that metaheuristics can be successfully applied to engineering optimization. With the advances in computer science more complex metaheuristics can be implemented and run in reasonable time lapses. Therefore, these optimization algorithms should be considered as a powerful tool for engineers of any area.

Resumen

Título: Metaheurísticas y su aplicación en optimización ingenieril

Autor: López Martínez, David Alan

Tutor: Rodríguez Ferran, Antonio

Todo problema de diseño de ingeniería trata de encontrar la solución óptima que, por ejemplo, maximiza la eficiencia mientras minimiza el coste. El uso de técnicas de optimización para lograr dicho objetivo se conoce como optimización ingenieril. En los últimos años este tema ha provocado que muchos ingenieros se fijen en los algoritmos de optimización.

El presente documento estudia la posibilidad de usar metaheurísticas, un tipo de métodos computacionales ideados recientemente, en optimización ingenieril. Las metaheurísticas son algoritmos de optimización inspirados en la naturaleza que han sido ya aplicados en muchas áreas. Para poder concluir si estos algoritmos funcionan bien cuando hacen frente a problemas de diseño de ingeniería, varios métodos clásicos de optimización y dos metaheurísticas, concretamente Algoritmo Genético y Recocido Simulado, han sido implementados. Los diferentes métodos han sido utilizados luego para resolver problemas de optimización relacionados con la ingeniería, tales como diseño estructural o logística.

De acuerdo con los resultados de las simulacions llevadas a cabo, el Algoritmo Genético y el Recocido Simulado superan a los métodos clásicos en todos los problemas propuestos. Los Algoritmos Genéticos sobresalen en optimización continua sin restricciones y optimización combinatoria mientras el Recocido Simulado es mejor en optimización con restricciones. En la mayoría de los casos las metaheurísticas estudiadas son capaces de reducir el error obtenido por los métodos clásicos incluso para tiempos de simulación de menos de un segundo.

La conclusión extraída de esta tesina es que las metaheurísticas se pueden aplicar satisfactoriamente en optimización ingenieril. Con los avances en las ciencias de la computación metaheurísticas más complejas pueden ser implementadas y ejecutadas en lapsos de tiempo razonables. Por lo tanto, estos algoritmos de optimización deberían ser comnsiderados como una herramienta potente para ingenieros de cualquier ámbito.

Contents

1 **Introduction** 1

2 **State of the art** 1

2.1 Heuristics and Metaheuristics . . . 2

3 **Engineering Optimization** 3

3.1 Optimization . . . 3

3.2 Type of Optimization . . . 4

3.3 Optimization Algorithms . . . 6

3.4 Metaheuristics . . . 7

4 **Classic Algorithms** 8

4.1 Newton's Method . . . 8

4.2 Penalty Method . . . 9

4.3 Exhaustive Search . . . 11

4.4 Pure Random Search . . . 11

5 **Metaheuristics** 11

5.1 Genetic Algorithm . . . 12

5.1.1 Background to GA . . . 12

5.1.2 Algorithm . . . 13

5.1.3 Operators . . . 14

5.1.4 Parameters . . . 18

5.2 Simulated Annealing . . . 19

5.2.1 Background to SA . . . 20

5.2.2 Algorithm . . . 20

5.2.3 Operators . . . 21

5.2.4 Parameters . . . 24

6 Benchmark Problems **25**

6.1 Unconstrained Continuous Optimization 26

6.1.1 De Jong's Function 26

6.1.2 Rastrigin's Function 27

6.1.3 Six-Hump Camel Back Function 27

6.2 Constrained Continuous Optimization 28

6.2.1 Schmit Structure 28

6.3 Combinatorial Optimization 31

6.3.1 Travelling Salesman Problem 31

7 Implementation and Testing **32**

7.1 Newton's Method 32

7.1.1 Implementation 33

7.1.2 Testing 34

7.2 Exhaustive Search 44

7.2.1 Implementation 45

7.2.2 Testing 46

7.3 Pure Random Search 50

7.3.1 Implementation 51

7.3.2 Testing 52

7.4 Genetic Algorithm 55

7.4.1 Implementation 55

7.4.2 Testing 61

7.5 Simulated Annealing 72

7.5.1 Implementation 72

7.5.2 Testing 74

8 Simulation Results and Discussion **83**

8.1 Unconstrained Continuous Optimization 85

8.1.1 De Jong's Function 85

8.1.2 Rastrigin's Function 87

8.1.3 Six-Hump Camel Back Function 88

8.2 Constrained Continuous Optimization 89

8.2.1 Schmit Structure 90

8.3 Combinatorial Optimization 92

8.3.1 Travelling Salesman Problem 92

9 Conclusions **95**

List of Figures

1 Classification of optimization problems (source: [20]) 5

2 Classification of algorithms (source: [20]) 7

3 GA algorithm flow chart diagram (source: self made) 14

4 Graphical representation of the Roulette Wheel Selection (source: self made) 16

5 Graphical representation of Uniform Crossover (source: self made) 16

6 Graphical representation of Single Point Crossover (source: self made) 17

7 SA algorithm flow chart diagram (source: self made) 21

8 Graphical representation of De Jong's function in 2D (source: [13]) 26

9 Graphical representation Rastrigin's function in 2D (source: [13]) 27

10 Graphical representation Six-hump camel back function (source: [13]) 28

11 Schmit structure (source: [14]) 29

12 Schmit structure isostatic quasi-optimal solution (source: [14]) 30

13 Schmit structure hyperstatic optimal solution (source: [14]) 30

14 Travelling Salesman Problem illustration (source: wikipedia) 31

15 Graphical representation of Rastrigin's function in 1D (source: self made) 35

16 Graphical representation of Rastrigin's function first derivative in 1D 36

17 Graphical representation of the behaviour of solution f(x) with initial approximation (source: self made) 37

18 Zoomed graphical representation of the behaviour of solution f(x) with initial approximation (source: self made) 37

19 Graphical representation of the behaviour of solution x with initial approximation (source: self made) 38

20 Zoomed graphical representation of the behaviour of solution x with initial approximation (source: self made) 39

21 Graphical representation of the behaviour of cost with initial approximation (source: self made) 40

22 Zoomed graphical representation of the behaviour of cost with initial approximation (source: self made) . . . 41

23 Graphical representation of the behaviour of solution f(x) with tolerance (source:self made) . . . 42

24 Graphical representation of the behaviour of solution x with tolerance (source: self made) 43

25 Graphical representation of the behaviour of iterations with tolerance (source: self made) 44

26 Graphical representation of the behaviour of solution f(x) with grid size (source: self made) . . . 47

27 Graphical representation of the behaviour of solution x with grid size (source: self made) 48

28 Graphical representation of the behaviour of nº of iterations until best solution is found with grid size (source: self made) . . . 49

29 Graphical representation of the behaviour of simulation time with grid size (source: self made) . . . 50

30 Graphical representation of avg. solution f(x) with number of iterations (source: self made) . . . 52

31 Graphical representation of avg. solution x with number of iterations (source: self made) 53

32 Graphical representation of the behaviour of avg. number of iterations until best solution is found with number of iterations (source: self made) . . . 54

33 Graphical representation of the behaviour of avg. simulation time with number of iterations (source: self made) . . . 55

34 Graphical representation of the performance of the GA models with the 1st set of parameters (source: self made) . . . 62

35 Graphical representation of the performance of the GA models with the 2nd set of parameters (source: self made) . . . 63

36 Graphical representation of the performance of the GA models with the 3rd set of parameters (source: self made) . . . 64

37 Graphical representation of the performance of the GA models with the 4th set of parameters (source: self made) . . . 64

38 Graphical representation of the performance of the GA models with the 5th set of parameters (source: self made) . . . 65

39 Graphical representation of avg. solution f(x) with initial population (source: self made) 66

40 Graphical representation of avg. solution x with initial population (source: self made) . 66

41 Graphical representation of avg. number of iterations until best solution is found with initial population (source: self made) . . . 67

42 Zoomed graphical representation of avg. number of iterations until best solution is found with initial population (source: self made) . . . 67

43 Graphical representation of avg. solution f(x) with crossover rate (source: self made) . . 68

44 Graphical representation of avg. number of iterations until best solution is found with crossover rate (source: self made) . . . 69

45 Graphical representation of avg. solution f(x) with mutation rate (source: self made) . . 69

46 Graphical representation of avg. number of iterations until best solution is found with mutation rate (source: self made) . . . 70

47 Graphical representation of avg. solution f(x) with number of generations (source: self made) . . . 71

48 Graphical representation of avg. number of iterations until best solution is found with number of generations (source: self made) . . . 71

49 Graphical representation of avg. solution f(x) with search diameter (source: self made) . 74

50 Zoomed graphical representation of avg. solution f(x) with search diameter (source: self made) . . . 75

51 Graphical representation of avg. solution x with search diameter (source: self made) . . 76

52 Zoomed graphical representation of avg. solution x with search diameter (source: self made) . . . 76

53 Graphical representation of number of iterations until best solution is found with search diameter (source: self made) . . . 77

54 Graphical representation of avg. solution f(x) with initial temperature (source: self made) 78

55 Graphical representation of avg. solution x with initial temperature (source: self made) 78

56 Graphical representation of avg. number of iterations until best solution is found with initial temperature (source: self made) . . . 79

57 Graphical representation of avg. solution f(x) with number of iterations (source: self made) . . . 80

58 Zoomd graphical representation of avg. solution f(x) with number of iterations (source: self made) . . . 80

59 Graphical representation of avg. solution x with number of iterations (source: self made) 81

60 Zoomed graphical representation of avg. solution x with number of iterations (source: self made) 82

61 Graphical representation of avg. number of iterations until best solution is found with number of iterations (source: self made) 83

62 Graphical representation of De Jong's function simulations results (source: self made) . 86

63 Graphical representation of Rastrigin's function simulations results (source: self made) . 87

64 Graphical representation of Six-Hump Camel Back function simulations results (source: self made) 89

65 Graphical representation of Schmit structure simulations results (source: self made) . . 91

66 TSP distribution of cities (source: self made) 93

67 Graphical representation of TSP Exhaustive Search simulation results (source: self made) 94

68 Graphical representation of TSP simulations results (source: self made) 94

List of Tables

1 Operators for each GA model (source: self made) 62

2 Parameters used in the simulations for each method 84

3 Isostatic and hyperstatic solutions for schimt structure (source: self made) 90

4 TSP Exhaustive Search simulation results (source: self made) 93

5 Summary of simulations results for unconstrained continuous optimization (source: self made) 97

6 Summary of simulations results for constrained continuous optimization (source: self made) 97

7 Summary of simulations results for combinatorial optimization (source: self made) . . . 98

1 Introduction

Nowadays optimization is found everywhere, from professional fields to personal decisions. Any enterprise intends to maximize its profits or performance while minimizing costs. Engineering design, planning, logistics or finances all involve some kind of optimization. Even in our daily activities people tend to optimize time spent and enjoyment, for instance. In fact, we are constantly searching for the optimal solutions to every problem we meet, though we are not necessarily able to find such solutions.

Metaheuristics are a relatively new family of computational methods which can be used to solve optimization problems. Most metaheuristic algorithms are nature inspired as they have been developed based on some abstraction of nature. Nature has evolved over millions of years and has found perfect solutions to almost all the problems she met. We can thus learn the success of problem solving from nature and develop nature inspired heuristic and metaheuristic algorithms. More specifically, some nature inspired algorithms are inspired by Darwin's evolutionary theory. Consequently, they are said to be biology inspired.

Many classical methods have intrinsic limitations on their performance due to their essential mechanics while, usually, metaheuristics are only limited by computational cost. Recent advances in computer science involve an exponential increase in computation power and therefore the limitations on metaheuristics' performance are continuously being dramatically reduced. This fact has drawn attention on this kind of methods in the last years.

The aim of this study is to conclude whether metaheuristics can be successfully applied to engineering optimization. A number of classical and metaheuristic optimization methods are implemented and a series of simulations performed in order to compare the behaviour of the studied algorithms. The simulations consist in using the implemented algorithms to solve several proposed benchmark problems related to engineering. For the objective set, the present is a qualitative research rather than a quantitative research.

Three types of optimization problems are studied, namely unconstrained continuous optimization, constrained continuous optimization and combinatorial optimization. These encompass most of the habitual optimization problems found. An effort has been put into finding engineering optimization problems, for instance, structural design and logistics.

The document starts by introducing optimization in terms of the type of problems and solving methods. A more thorough explanation of the algorithms studied and the benchmark problems used follows. Then the implementation is presented in detail together with the testing done for the parameters of each method. Finally, the simulations are performed that will allow to make a judgement on the feasibility of using metaheuristics in engineering optimization.

2 State of the art

Even thought mathematical optimization is a relatively new branch of mathematics, optimization in the broadest sense has always been present in any human decision making processes. Although subconsciously, when facing any given choice we tend to search for the best solution possible. Thus, throughout human history, optimzation has always been present.

Before mathematics was even considered a modern science, many mathematicians solved optimization problems. In ancient Egypt, for instance, optimization was used in the construction of the piramids. In ancient Greek, many optimization problems were proposed and subsequently solved. For example, Euclid proved that a square encloses the greatest area amongst all the possible rectangles with the same perimeter.

Up to the twentieth century, as mentioned above, many mathematicians studied a number of optimization problems. Usually the optimal solutions were found by exhaustively studying the search space or using brute force approaches. In such attempts of solving optimization problems the effort was put into finding the actual optimal solution rather than devising an algorithm that would allow to find the optimum in different situations and problems. Some problems remain unsolved and are still being studied today. An example to this is the Travelling Salesman Problem for a large number of cities, a benchmark used in this dissertation, which was proposed as it is known today around the year 1930 but was introduced for the first time in the eighteenth century.

Many well known mathematicians and physicists have also contributed to the study of optimization problems. Just to mention a few, Fermat and Lagrange found calculus-based formulas for identifying optima while Newton and Gauss proposed iterative methods for moving towards an optimum. The iterative method proposed by Newton was the widely used Newton's Method or Newton-Raphson Method, included in this dissertation. Another example is Cauchy, who proposed a general iterative method for solving systems of equations that leads to two other notorious methods: gradient method and steepest descent.

In the twentieth century, an algorithm for linear programming was developed and used in economics. Linear programming was historically the first term for optimization. Effective methods for finding solutions using linear programming were found. These advancements were made during the first half of the century.

The development of the transistor in the early 1950s revolutionized the field of electronics and paved the way for computers. With the evolution of computers and the consequent accessibility to computation power, mathematical optimization has been greatly reinvigorated. A large number of computational optimization methods and algorithms have been devised since the 1960s and even more appear every day. The reason behind this is that the number of mathematical operations computers can do nowadays permit the execution of algorithms that were unthinkable before. Obviously this is also accompanied by the fact that today we are living in the era of the information and thus it is very easy to have access to ongoing studies as they are being carried out. In the same way as many other fields, optimization is growing exponentially in the last decades. Such are the advancements in optimization since the existance of computers that it would take a whole book to write even a brief history on optimization after 1960.

2.1 Heuristics and Metaheuristics

In the last few decades a new group of optimization methods has appeared taking advantage of the aforementioned increase in computational power, namely heuristics and metaheuristics. The term heuristic makes reference to the fact that the method uses experience as a tool to find the best solution to an optimization problem. These methods are usually based on trial-and error. While heuristic methods don't find the exact solution to a problem, like calculus-based methods do, they excel at producing acceptable solutions in a reasonably practical time. There is no guarantee at all that these

methods will find the optimal solution. Despite this fact, they are usually efficient in finding nearly optimal solutions in an acceptable time lapse. Heuristics are usually designed to solve one specific problem, on the other hand, metaheuristics are problem-independent methods that use some kind of heuristic together with some stochastic component.

The first to use heuristic algorithms was probably Alan Turing when trying to decipher german encoded messages during the Second World War. Turing described his method as a heuristic algorithm since there was no guarantee to find the correct solution but it could be expected to work most of the time. It was a tremendous success.

Further studies resulted in the development of evolutionary algorithms. Probably the most notable algorithms of this type are the Genetic Algorithms (GA), inspired by Darwin's natural selection. Genetic Algorithms were devised by John Holland and his collaborators in the 1960s and 1970s. Holland published his findings in the groundbreaking book "Adaptation in Natural and Artificial Systems" in 1975. Other evolutionary algorithms were developed around the same time, for instance Evolution Strategy or Evolutionary Programming.

The next big step was the development of Simulated Annealing (SA) by S. Kirkpatrick et al. in 1983, inspired by the annealing process of metals. Both GA and SA are studied in this dissertation and will be explained in detail later in the document.

Many more heuristic and metaheuristic methods have been devised to date. These include Tabu Search, Ant Colony Optimization, Harmony Search, Artificial Bee Colony and Firefly Algorithm amongst others.

For a more thorough description of the history of optimization the book "Engineering Optimization: An Introduction with Metaheuristic Application" by the author Xin-She Yang can be consulted. This book mentions some important contributors to optimization from the ancient times up to today. A very good overview of the evolution of optimization can be found there, together with a brief description of the algorithms cited above.

3 Engineering Optimization

Mathematical optimization includes a number of different problems with the objective of finding the best possible solution. The wide range of problems and searching methods included make it difficult to classify optimization in terms of the type of problem or the type of searching algorithm. There is no consensus in literature on the way this classification must be done, therefore one specific approach is presented in this dissertation.

3.1 Optimization

It is possible to write most optimization problems in the generic form

$$\begin{aligned} \underset{\boldsymbol{x}\epsilon\Re^n}{minimize} \quad & f_i(\boldsymbol{x}), \quad (i = 1, 2, ..., M), & (1)\\ subject\,to \quad & \phi_j(\boldsymbol{x}) = 0, \quad (j = 1, 2, ..., J), & (2)\\ & \psi_k(\boldsymbol{x}) \leq 0, \quad (k = 1, 2, ..., K), & (3) \end{aligned}$$

where $f_i(\boldsymbol{x})$, $\phi_j(\boldsymbol{x})$ and $\psi_k(\boldsymbol{x})$ are functions of the design vector

$$\boldsymbol{x} = (x_1, x_2, ..., x_n)^T. \tag{4}$$

In the above formulation, the components x_i of $\boldsymbol{x}$ are called design or decision variables. The decision variables can be continuous, discrete or anything in between and belong to the search space $\Re^n$. The functions $f_i(\boldsymbol{x})$ are called the objective functions. The objective function is also known as the cost function or energy function. The equalities ϕ_j and ψ_k inequalities are called constraints. The optimization problem can also be formulated as a maximization problem.

Somtimes an optimization problem can have no objective function and consist only in constraints. In this case the problem is known as a feasibility problem since any feasible solution is an optimal.

The possibility exists that the objective function can not be written in the explicit form since it is too complex or it can not be expressed mathematically. In such cases some sort of simulations have to be carried out to determine whether a solution is optimal or not. In this dissertation the problems studied all can be formulated through the explicit form of the objective function.

In the following sections, the types of optimization problems and optimization search algorithms are presented. As mentioned before, only one specific classification is included amongst the many available in literature.

3.2 Type of Optimization

The huge variety of existing optimization problems makes it difficult to establish a certain classification of these. There is no consensus in the literature on how this should be done, therefore the classification here is aimed at presenting the optimization problem types in a clear manner so the reader can get a grasp of the basic types. In general terms, the optimization problems can be classified by the number of objectives, number of constraints, landscape of the objective function, function form, type of design variables and determinacy (see Figure 1).

According to the number of objectives, an optimization problem can be single objective or multiobjective. A single objective problem is that in which only one objective function exists that represents the one value to be maximized or minimized. For example, the design of a structure that must support a certain weight with the lowest cost possible. In this example the one objective is to minimize the structure cost. On the other hand, multiobjective problems must optimize a number of different objectives or objective functions. Therefore, an example would be the same structure as before if,

besides minimizing the cost, we try to maximize the weight it can support. In such case, two different objective functions exist corresponding to the cost and maximum weight supported. Most engineering optimization problems are multiobjective.

When calssifying the optimization problems in terms of the number of constraints there are obviously those which have no constraints, unconstrained problems, and those that do have a number of constraints, constrained optimization problems. The constraints in the second type can be both equalities and inequalities. Furthermore, any equality constraint can be rewritten as two inequality constraints and therefore sometimes in the literature only inequality constraints are considered.

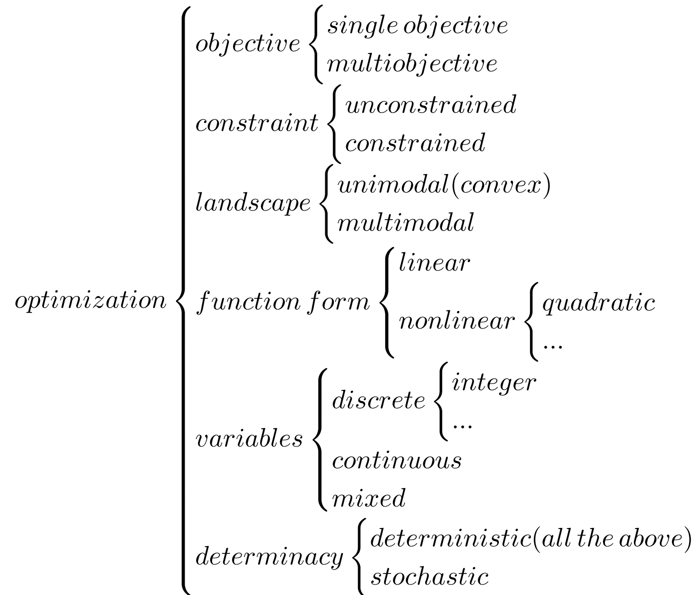

Figure 1: Classification of optimization problems (source: [20])

Another way of classifying the optimization problems is by the landscape of the objective functions. Such can functions can may have a single valley or peak with a unique global optimum or they might, on the other hand, have a number of valleys and peaks with different local optima and therefore one or more global optima. In the first case we have a unimodal landscape, which can in turn be convex, while in the second case we have a multimodal landscape. Obviously, multimodal optimization problems are harder to solve and are found more often in engineering optimization. An example of a unimodal landscape is De Jong's Function, meanwhile Rastrigin's Function is an example of a multimodal landscape. Both examples are studied in this dissertation.

The optimization problems can also be classified according to the function form. If the objective function and the constraints are linear then it is called a linear programming problem. If only the constraints are linear, but the objective function is not, then it becomes a linearly constrained problem. However, if the objective function and the constraints are nonlinear, the usual case, then it is a nonlinear optimization problem. When this happens, the problems can be classified again according to the type of function form once more. For example, the function form can be nonlinear with a quadratic form.

The type of variables in optimization problems can also be used in the classification of these. When the variables are discrete then we talk about discrete optimization. Particular cases of discrete optimization

are those in which the problem variables are integers or natural numbers. Discrete optimization is also known as combinatorial optimization sometimes. The problem variables can also be continuous, like in the case of real numbers, in such case the problem belongs to continuous optimization. These two types of optimization problems have been studied in the present work. Many design problems can present both discrete and continuous variables, resulting in a mixed optimization problem.

All the above classifications belong to the case in which any set of variables can be evaluated by means of the objective function and constraints in an exact way. Therefore, the optimization problem can be called deterministic in this sense since the values for the objective function and constraints can be determined without uncertainty. Nonetheless, not every optimization problem is deterministic. It is common to find engineering optimization problems where some objective function or constraint can not be determined exactly since there is some kind of inherent uncertainty associated to the physical parameter studied. In such cases we have stochastic optimization problems.

3.3 Optimization Algorithms

In the preceding section, the different types of optimization problems are presented according to one possible classification. This section proceeds the same way with the different existing optimization algorithms used to solve such optimization problems.

Suppose that we have an optimization problem that consists of finding a certain object that is located in the centre of a room. Loosely speaking, there are two opposite ways in which to perform the search of the object. The first option consists in entering the room and randomly walking around until the object is found, supposing that you can not see the object from afar. The second option is based on the knowledge beforehand that the object is in the center of the room and consists in entering the room and walking directly to the center hoping to find the object there. In this analogy, the first option is a pure stochastic or random search while the second is known as a hill-climbing technique. Every optimization algorithm can be understood as an instance of one of the options above or something in between.

In general terms, optimization algorithms can be divided into two main groups: deterministic and stochastic. A deterministic algorithm follows a series of steps based on exact calculations and therefore, no matter how many times one specific deterministic algorithms is run, it always finds the same solution. On the other hand, stochastic algorithms have a random behaviour in at least one of the steps followed to search for the optimum. Thus a stochastic algorithm gives a different solution, or at least follows a different path to such solution, for every simulation run.

Deterministic algorithms are mainly classic calculus-based optimization algorithms. These range from linear and nonlinear programming to gradient-based or gradient-free algorithms. Some examples of deterministic algorithms are Linear Programming, Newton's Method and Gradient Descent. The first is obviously used to solve linear problems while the other two methods belong to the gradient-based algorithms. Many deterministic optimization algorithms were devised before the twentieth century and therefore are considered classic or conventional. As mentioned above, these methods are repeatable and always follow the same path towards the solution. The only way of varying the result obtained by means of a deterministic method is changing the starting search point. Some algorithms are deterministic by nature but include a small random component. These hybrid algorithms are classified as stochastic in the literature.

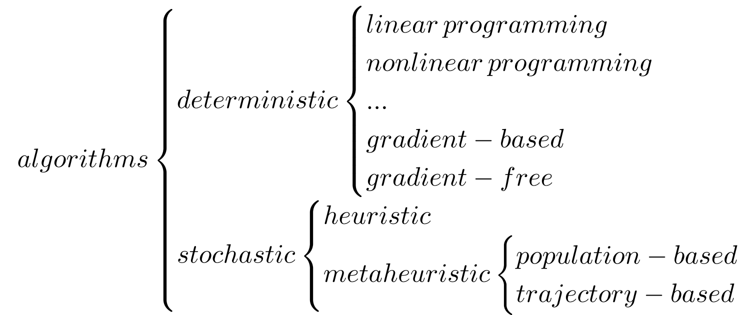

Figure 2: Classification of algorithms (source: [20])

Most conventional or classic algorithms are deterministic. For example, the Simplex method in linear programming is deterministic. Some deterministic optimization algorithms used the gradient information, they are called gradient-based algorithms. For example, the well-known Newton-Raphson algorithm is gradient-based as it uses the function values and their derivatives, and it works extremely well for smooth unimodal problems. However, if there is some discontinuity in the objective function, it does not work well. In this case, a non-gradient algorithm is preferred. Non-gradient-based or gradient-free algorithms do not use any derivative, but only the function values. Hooke-Jeeves pattern search and Nelder-Mead downhill simplex are examples of gradient-free algorithms.

For stochastic algorithms, we have in general two types: heuristic and metaheuristic, though their difference is small. Loosely speaking, heuristic means 'to find' or 'to discover by trial and error'. Quality solutions to a tough optimization problem can be found in a reasonable amount of time, but there is no guarantee that optimal solutions are reached. It is expected that these algorithms work most of the time, but not all the time. This is usually good enough when we do not necessarily want the best solutions but rather good solutions which are easily reachable.

Further development over the heuristic algorithms is the so-called metaheuristic algorithms. Here meta- means 'beyond' or 'higher level', and they generally perform better than simple heuristics. In addition, all metaheuristic algorithms use certain tradeoff of randomization and local search. It is worth pointing out that no agreed definitions of heuristics and metaheuristics exist in literature, some use 'heuristics' and 'metaheuristics' interchangeably. However, recent trends tend to name all stochastic algorithms with randomization and local search as metaheuristic. Here we will also use this convention. Randomization provides a good way to move away from local search to the search on the global scale. Therefore, almost all metaheuristic algorithms intend to be suitable for global optimization.

3.4 Metaheuristics

Most metaheuristic algorithms are nature-inspired as they have been developed based on some abstraction of nature. Nature has evolved over millions of years and has found perfect solutions to almost all the problems she met. We can thus learn the success of problem-solving from nature and develop nature-inspired heuristic and/or metaheuristic algorithms. More specifically, some nature-inspired algorithms are inspired by Darwin's evolutionary theory. Consequently, they are said to be biology-inspired or simply bio-inspired.

Two major components of any metaheuristic algorithms are: selection of the best solutions and randomization. The selection of the best ensures that the solutions will converge to the optimality, while the randomization avoids the solutions being trapped at local optima and, at the same, increase the diversity of the solutions. The good combination of these two components will usually ensure that the global optimality is achievable.

Metaheuristic algorithms can be classified in many ways. One way is to classify them as: population-based and trajectory-based. For example, genetic algorithms are population-based as they use a set of strings, so is the particle swarm optimization (PSO) which uses multiple agents or particles. PSO is also referred to as agent-based algorithms.

On the other hand, simulated annealing uses a single agent or solution which moves through the design space or search space in a piecewise style. A better move or solution is always accepted, while a not-so-good move can be accepted with certain probability. The steps or moves trace a trajectory in the search space, with a non-zero probability that this trajectory can reach the global optimum.

Two major modern metaheuristic methods are studied in this dissertation: genetic algorithms (GA) and simulated annealing (SA).

The efficiency of an algorithm is largely determined by the complexity of the algorithm. The algorithm complexity is often denoted by the order notation, which will be introduced below.

4 Classic Algorithms

This section contains a brief description of a few well known classical optimization algorithms, namely Newton's Method, Exhaustive Search and Pure Random Search. The first is a calculus based method, the second is a brute force method and the third is a pure stochastic method.

4.1 Newton's Method

Newton's method, also known as Newton–Raphson method, is a root-finding algorithm, but it can be modified for solving optimization problems. This is because optimization is equivalent to finding the root of the first derivative $f'(\boldsymbol{x})$ of the objective function $f(\boldsymbol{x})$. For a continuously differentiable function $f(\boldsymbol{x})$, we have the Taylor expansion in terms of $\triangle \boldsymbol{x} = \boldsymbol{x} - \boldsymbol{x}_n$ about a fixed point $\boldsymbol{x}_n$

$$f(\boldsymbol{x}) = f(\boldsymbol{x}_n) + (\nabla f(\boldsymbol{x}_n))^{\mathrm{T}} \triangle \boldsymbol{x} + \frac{1}{2} \triangle \boldsymbol{x}^{\mathrm{T}} \nabla^2 f(\boldsymbol{x}_n) \triangle \boldsymbol{x} + ...,$$

whose third term is a quadratic form. Hence $f(\boldsymbol{x})$ is minimized if $\triangle \boldsymbol{x}$ is the solution of the following linear equation

$$\nabla f(\boldsymbol{x_n}) + \nabla^2 f(\boldsymbol{x}_n) \triangle \boldsymbol{x} = 0. \tag{5}$$

This leads to

$$\boldsymbol{x} = \boldsymbol{x}_n - \boldsymbol{H}^{-1}\nabla f(\boldsymbol{x}_n), \tag{6}$$

where $\boldsymbol{H}^{-1}$ is the inverse of the Hessian matrix $\boldsymbol{H} = \nabla^2 f(\boldsymbol{x}_n)$, which is defined as

$$\boldsymbol{H}(\boldsymbol{x}) \equiv \nabla^2 f(\boldsymbol{x}) \equiv \begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \vdots & & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{pmatrix}. \tag{7}$$

This matrix is symmetric due to the fact that

$$\frac{\partial^2 f}{\partial x_i \partial x_j} = \frac{\partial^2 f}{\partial x_j \partial x_i}. \tag{8}$$

If the iteration procedure starts from the initial vector $\boldsymbol{x}^{(0)}$, usually a guessed point in the feasible region, then Newton's formula for the nth iteration becomes

$$\boldsymbol{x}^{(n+1)} = \boldsymbol{x}^{(n)} - \boldsymbol{H}^{-1}(\boldsymbol{x}^{(n)})f(\boldsymbol{x}^{(n)}). \tag{9}$$

It is worth pointing out that if $f(\boldsymbol{x})$ is quadratic, then the solution can be found exactly in a single step. However, this method is not efficient for non-quadratic functions.

In order to be able to use Newton's Method in an optimization problem the first derivative of the objective function must be a continuously differentiable function. Besides this requirement, if the global optimum of the problem is to be found, a good enough initial estimate of the solution has to be set. This is usually one of the main drawbacks of this method.

4.2 Penalty Method

The basic idea in penalty method is to eliminate some or all of the constraints and modify the objective function with a penalty term. These term is used to increase the cost of points that not verify the constraints. The penalty is large if the constraints are far from being verified and small if they are close to being verified. Moreover, one particular constraint might bear more importance than the rest, hence weights associated to each constraint are introduced in the formulation that represent the importance of the verification of such constraint.

The objective function is usually modified with the penalty term in two possible ways. The first way is to use additive form:

$$eval(\boldsymbol{x}) = \begin{cases} f(\boldsymbol{x}), & if\, \boldsymbol{x} \epsilon \mathcal{F} \\ f(\boldsymbol{x}) + p(\boldsymbol{x}), & otherwise \end{cases}$$

where $\mathcal{F}$ is the feasible space and $p(\boldsymbol{x})$ represents a penalty term. For minimization problems $p(\boldsymbol{x})$ is positive while for maximization problems it is negative, this guarantees that the penalty increases the cost.

The second way is to use multiplicative form:

$$eval(\boldsymbol{x}) = \begin{cases} f(\boldsymbol{x}), & if\, \boldsymbol{x} \epsilon \mathcal{F} \\ f(\boldsymbol{x}) \cdot p(\boldsymbol{x}), & otherwise \end{cases}$$

for minimization problems $p(\boldsymbol{x})$ bigger than one while for maximization problems it is smaller than one. The additive penalty is used much more often than the multiplicative in the literature.

As an example, for a simple function optimization problem with equality and inequality constraints

$$\begin{array}{lll} \underset{\boldsymbol{x} \epsilon \Re^n}{minimize} & f(\boldsymbol{x}), & \boldsymbol{x} = (x_1, ..., x_n)^{\mathrm{T}}, \\ subject\, to & \phi_i(\boldsymbol{x}) = 0, & (i = 1, ..., M), \\ & \psi_j(\boldsymbol{x}) \leq 0, & (j = 1, ..., N), \end{array}$$

the idea is to define a penalty function so that the constrained problem is transformed into an unconstrained problem. Now we define $g(\boldsymbol{x}, \mu_i, \nu_j)$

$$g(\boldsymbol{x}, \mu_i, \nu_j) = f(\boldsymbol{x}) + \sum_{i=1}^{M} \mu_i \phi_i^2(\boldsymbol{x}) + \sum_{j=1}^{N} \nu_j \psi_j^2(\boldsymbol{x}), \tag{10}$$

where $\mu_i \gg 1$ and $\nu_j \geq 0$ are the weights used for each constraint.

4.3 Exhaustive Search

Exhaustive Search, which is also known as Brute Force Search, is a trivial but very recurrent search algorithm in literature. This problem solving technique consists in comparing all the possible solutions to the studied problem and checking which one satisfies the problem statement. For example, to find the square root of a given number (x) Exhaustive Search would multiply every number, from 0 to x, by itself and check whether the solution of the multiplication is x. When the multiplication result is equal to x the solution to the initial problem has been found.

The implimentation of Exhaustive Search is very straightforward and will always find a solution if it exists. However the cost of running Exhaustive Search is usually very high compared to the cost of other methods. This is due to the fact that the cost is proportional to the number of possible solutions to the problem being solved. Thus the cost grows very quickly with the size of the problem's search space size. The use of Exhaustive Search is usually limited to problems of small size or when the ease in implementing the algorithm is more important than the cost or speed.

This method is also used often when testing other algorithms such as metaheuristics. In this dissertation Exhaustive Search has been implemented and used with the aforementioned aim.

When facing continuous optimization the method can only be applied through a discretization of the search space in order to be able to evaluate all the number of candidate solutions in a finite time. This is usually done by defining a grid of points in the search space. Therefore a new parameter appears when solving continuous optimization with the exhaustive method, this parameter is the grid size. The grid size can be understood as the distance between one point and its neighbours in the grid, in the hypothesis of an equally spaced grid. The grid size determines the number of evaluations, or computational cost, together with the precision of the solution obtained. Therefore, it can be compared to the tolerance in Newton's Method.

4.4 Pure Random Search

The simplest stochastic global optimization algorithm is the so-called Pure Random Search (PRS), which was introduced in 1958. This algorithm randomly generates candidate solutions in the search space and evaluates the objective function for each candidate. When the termination criterion is met, the best candidate solution generated thus far is returned as the solution to the problem.

Since this search method is completely stochastic and has no directed search the results obtained are far from optimal. Nonetheless, in simple optimization problems Pure Random Search can be applied successfully. In more complex optmization problems Pure Random Search proves to be very inefficient.

5 Metaheuristics

This section presents the metaheuristic algorithms used in this dissertation. Genetic Algorithm and Simulated Annealing have been implemented and applied to the benchmark engineering problems proposed in this study. Both methods are described thoroughly herein. For further information on these two algorithms the reader can consult the bibliography at the end of this document.

5.1 Genetic Algorithm

Genetic Algorithms are metaheuristics inspired by the theory of natural selection, also known as survival of the fittest. Darwin, in his book "On the Origin of Species", proposed the scientific theory that all species of life have descended over time from common ancestors through a process that he called natural selection. Natural selection explains how the fittest members of a population are more likely to survive and reproduce, hence their genes are passed to the next generations. In the same way, Genetic Algorithms evolve solutions to a problem in order to obtain better solutions in the next generations.

In this section an understanding of the functioning of genetic algorithms is developed. A basic background to GA is included in order to give the reader an overall view on the matter. The background mentions early works on GA and the particularities of this metaheuristic. The section continues to introduce the basic algorithm and explain in detail the existing operators and parameters that constitute the genetic algorithm.

For a more detailed explanation on Genetic Algorithms and how they differ from classic optimization methods, the reader can consult the book "Genetic Algorithms in Search, Optimization and Machine Learning" by David Edward Goldberg (1989) which includes a complete summary on GAs.

5.1.1 Background to GA

Genetic Algorithms were first presented in 1975 by John H. Holland in his book "Adaptation in Natural and Artificial Systems". In his work Holland noticed that artificial systems work very similarly to natural systems and therefore devised a way in which natural selection could be formulated mathematically. He succeeded in implementing an early version of a Genetic Algorithm that used selection, crossover and mutation operators to improve solutions to a problem over a number of generations or iterations.

In mathematical terms the genes in the chromosomes of each member in the population are the problem variables for which we are searching the best values. To calculate the fitness of a given member of the population the objective or cost function is evaluated using the member's genes. Using such encoding it is quite straightforward to implement the operators involved in the Genetic Algorithm.

Today, Genetic Algorithms have been used numerous times to solve complex optimization problems successfully. The good performance of GAs can be ascribed to its operators. In the same way as in nature, the selection and crossover operators are very appropiate at exlpoiting the best solutions obtained so far by passing their genes to the next generations. Moreover, the mutation operator is in charge of the exploration of the search space guaranteeing a diversity of genes in any generation. Probably the most noticeable advantage of Genetic Algorithms is their robustness. It is an easy task to ascertain that GAs always tend to improve the solution obtained in successive generations and therefore consistently approach the global optimum solution to the optimization problem being solved.

Since Genetic Algorithms emulate the biological process of evolution, a population of members is used to represent a generation. This fact implies that at any given time the algorithm has a number of candidate solutions instead of having only one candidate solution like many classic methods do. Like most metaheuristics, being one of their main fortes, GAs use an objective or cost function that does not involve the calculation of derivatives and does not need any additional information. Many calculus

based methods, such as Newton's Method, require the derivatives of the function to optimize which is a huge drawback when such derivatives are difficult or even impossible to obtain. Another difference from calculus based methods, common to most metaheuristics, is obvioously the fact that GAs are stochastic algorithms based on some probabilistic rules rather than deterministic ones.

5.1.2 Algorithm

As described before the GA is an attempt to emulate natural selection, or survival of the fittest, in a way that instead of evolving genetic information the algorithm evolves a series of numbers which represent the variables of the optimization problem being solved. In other words, in natural selection chromosomes (DNA) are evolved to be as fit as possible to the environment while, by analogy, in the GA chromosomes (problem variables) are evolved to be as fit as possible to the cost or objective function (problem function to be optimized).

The algorithm of GA is a simplification of natural selection process where several operators are used to simulate evolution. In the same way it happens in nature, a population of individuals is evolved throughout generations by means of selection of the fittest and reproduction. The population represents the current approximations to the solution, thus each population member is usually a series of numbers depending on the encoding used. These approximations that form the population can be interchangeably referred to as approximations, solutions, members and chromosomes by analogy to the GA nature. Each chromosome consists in a series of numbers which are a solution to the problem under study, in other words, the chromosome contains values for the problem variables and therefore the chromosome's length is given by the number of variables in the problem. Each of these numbers are called genes in GA; so a chromosome has a number of genes equal to the number of variables in the problem.

The whole process of natural selection can be simulated using five simple operators, namely Initial Population, Selection, Crossover, Mutation and Replacement. The Initial Population operator is in charge of generating the initial population. Selection evaluates the fitness of the population members and selects the parents for the new generation. Crossover and Reproduction belong to the process of Reproduction and carry out the crossing, or crossover, of the population members and the mutation of the offspring respectively. The result of Reproduction is the new generation of offspring. Finally, the operator Replacement merges the present population with the offspring to generate the new population. The point where the simulation stops or the number of generations simulated needs to be managed by the last operator used, Termination Criteria.

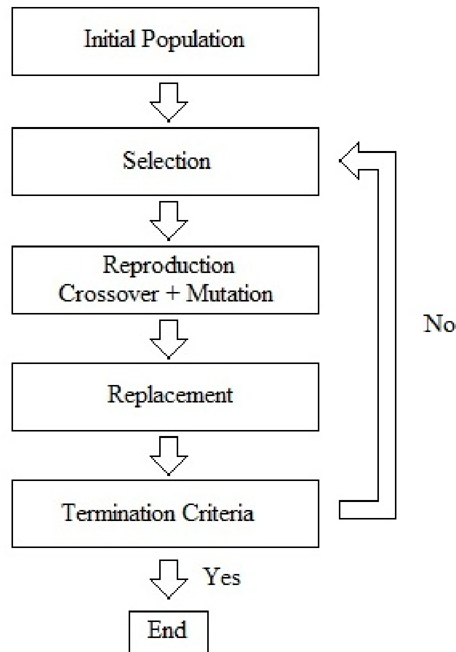

Figure 3: GA algorithm flow chart diagram (source: self made)

Given the fact that the GA is a method designed to be run by a computer its algorithm can be expressed in terms of a number of operators (or functions) and a set of parameters which determine how the method behaves, besides of course the objective problem's variables. The operators are the actors who guarantee that the method obtains better solutions with the number of iterations and set the algorithm flow. The parameters on the other hand are values to be fixed according to the type of optimization problem faced; these vary the behaviour of the algorithm to better adapt to the problem being solved.

5.1.3 Operators

A more thorough explanation of the operators of the GA is presented in this section. The function of each operator is described in detail along with some basic working mechanics. While each operator has a certain role in the algorithm, these roles can be carried out in a number of different ways. An example to this is the operator used to generate the initial population which can be generated with randomly or in a more greedy approach. The general purpose of each operator is presented here, together with the description of the specific operators used in the implementation done for this dissertation.

Initial Population

The purpose of this operator is to generate the initial population of solutions or approximations to the solution. Obviously, this operator is only used in the first generation or iteration for each simulation run. To be able to generate the initial population the operator needs to know the size of the population that must be generated, thus the first parameter of the GA is the size of the initial population (number of members).

Several different ways of generating the first generation of population members exist. The focus of this study is not to exhaustively compare all the possible operators but to compare a general GA algorithm to other methods. Therefore, only a few variations of each operator are studied.

Two variations of the Initial Population operator have been used in this dissertation, namely Random Initial Population and Greedy Initial Population.

Random Initial Population is a trivial variation of the operator that generates new members randomly in the search space given by the problem statement.

The Greedy Initial Population operator generates a population of better than average solutions (members) by, for example, generating a population larger than necessary and then select the best solutions to form the initial population.

Selection

Selection is the stage of a genetic algorithm in which individual members are chosen from the current population for breeding (crossover or recombination). The different variations of the operator define how these members are selected. According to Darwin's evolution theory the best ones should survive and create new offspring.

There are many ways of selecting the best parents, two have been studied here. These are Random Selection and Roulette Wheel Selection. The Random operators have been introduced to make more aparent the differences between the studied metaheuristic methods and more stochastic approaches.

Random Selection is, as expected, the trivial approach where the parents are selected randomly from the population.

In the 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 all chromosomes in the population are placed and the size of the slice of pie each chromosome occupies is determined according to the fitness of the chromosome.

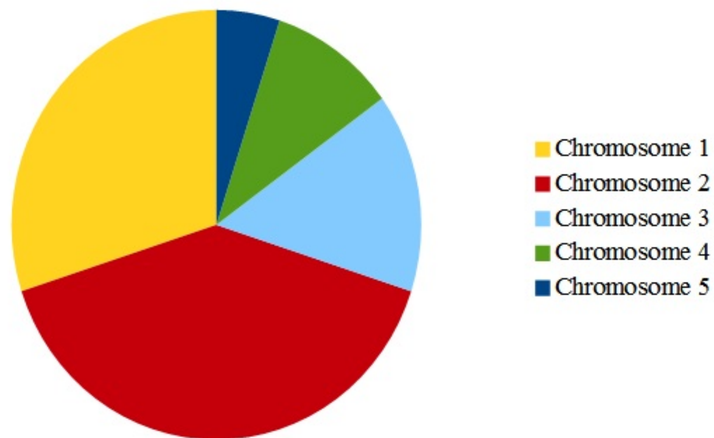

Figure 4: Graphical representation of the Roulette Wheel Selection (source: self made)

Then a marble is thrown into the roulette and selects a new parent. This process must be done twice for each offpsring because the crossover operators used cross two parents to generate one offspring, just like in sexual reproduction. Chromosomes with higher fitness will be selected more times.

Crossover

Crossover is a genetic operator used to vary the chromosomes from one generation to the next. It is analogous to reproduction and biological crossover, upon which genetic algorithms are based. This operator takes more than one parent solution and produces a child solution from them. As with any other operator there are many ways in which this can be done.

Two variations of this operator have been used here as well. These are Uniform Crossover and Single Point Crossover.

In Uniform Crossover genes are randomly copied from the first or from the second parent into the offspring with the same probability.

Figure 5: Graphical representation of Uniform Crossover (source: self made)

On the other hand, in Single Point Crossover one crossover point is selected. Genes before this point are copied from one parent while genes that come after this point are copied from the second parent.

Figure 6: Graphical representation of Single Point Crossover (source: self made)

While the Crossover operator generates one offspring for each pair of parents, this offspring is not always a recombination of the parents genes. Offspring are also generated, with some probability, as an exact copy of one of the parents. The probability with which this happens is set by a new parameter named Crossover Rate. To be more precise, the Crossover Rate is the probability with which parents are crossed to generate the offspring. Therefore the probability of being an exact copy of one of the parents is 1 - Crossover Rate.

Mutation

The Mutation operator introduces randomness into the reproduction by changing randomly some genes in a chromosome. While this prevents the algorithm from falling into local optima, it also increases the exploration capabilities of the method.

The usual approach is to set the probability with which a gene in a chromosome is mutated. This probability is another parameter of GA called Mutation Rate. Mutation is usually understood as completely random so no other variations of this operator have been studied in this dissertation.

Replacement

The Replacement operator is in charge of merging the current population and the generated offspring into the new population. This new population will then be used as starting point for the next generation. Again, there are several different ways of performing this combination.

Like with the other operators, two variations of the Replacement operator are studied in this dissertation: the Random Replacement and the Elitist Replacement.

The Random Replacement operator is again the trivial variation of the Replacement operator. This variation randomly selects a number of members of the current population equal to the number of offspring and replaces these members with the offspring. The result of this replacement is the new population which is used in the next generation as the starting population.

The Elitist Replacement operator favours the replacement of the less fit members of the population in order to introduce the offspring. The members to be replaced are selected using the same Roulette

Wheel strategy used in the selection operator. Although, in this case, the probability with which a certain member is selected is inversely proportional to its fitness. This variation is successful at exploting the best solutions found, or chromosomes, but this is in detriment of the exploration facet of the algorithm.

It is convenient to point out here that the population remains the same size throughout the simulation. Other variations of the Replacement operator exist where this is not true. For example, one possible variation is simply to add the offspring to the population without removing any current members. This causes the population to grow each generation together with the computational cost which can be inconvenient.

Termination Criteria

The generational process in GA is repeated until a termination condition has been reached. Common terminating conditions are: a solution is found that satisfies minimum criteria, fixed number of generations reached and allocated budget (computational cost) reached amongst others.

Since the Termination Criteria operator has no relation to the algorithm's efficiency, the simplest variation has been used that allows a certain control over the simulation time. Such variation is the corresponding to a fixed number of generations reached. The last of the method's parameters is therefore the Number of Generations. When the number of generations given by this parameter is reached the algorithm is stopped.

The higher the Number of Generations the higher the probability of finding a better solution and the higher the computational cost and simulation time.

5.1.4 Parameters

Initial Population

The Initial Population parameter fixes the size of the initial population. The Initial Population operator will generate a number of chromosomes equal to this size. The higher the value of this parameter the more chromosomes must be generated in the first generation, or first iteration of the algorithm, thus the higher the computational cost and the simulation time.

While a low value of this parameter improves the simulation time slightly, it can also act in detriment of the method since the probability of having all the population members converge to a same chromosome increases.

As with any other parameter of the GA, the optimal value for the Initial Population parameter depends on the type of problem studied. Every parameter for each method has been tested and the results presented in the corresponding section in this document.

Crossover Rate

The Crossover Rate parameter sets the probability with which a pair of selected parents will be crossed to generate one offspring. Usually this probability is set below 100% due to the fact that sometimes it is preferable that the selected parents, which should be chromosomes with above average fitness, are passed unmodified to the next generation.

Crossover rate generally should be high, about 80%-95%. However some results show that for some problems crossover rate about 60% is the best. This parameter is thoroughly tested in this study.

Mutation Rate

Mutation Rate sets the probability with which each gene in a offspring chromosome is mutated. The higher the value the more stochastic the algorithm results. A high Mutation Rate ensures the exploration of all the search space. Moreover, a low Mutation Rate favours the exploitation. Usually the Mutation Rate is set to a very low value. Best rates reported are about $0.5\% - 1\%$.

Since the above values depend highly on the way the mutation is implemented, Mutation Rate must also be tested in order to obtain the best values for each problem type studied.

Number of Generations

The Number of Generations can be understood as the length of the simulation. As explained above, the higher the number of generations the better the solution obetained, usually, and the higher the simulation time or computational cost. It seems logical to run as much generations as possible in one simulation but the fact that several simulations must be ran to average the solutions obtained should be considered. Therefore, in order to maintain a feasable simulation time the Number of Generation must be relatively low.

Together with every other parameter, the Number of Generations has been tested for the optimization problems studied in this dissertation. The optimum value for each parameter varies with the type of problem.

5.2 Simulated Annealing

Annealing is a process in metallurgy that consists in heating up a material and then cooling it slowly. After heating up the material, when at high temperature, the atoms move about in great mesure since they have high kinetic energy. At this state the atoms sort themselves in many configurations with different stabilites. As the temperature decreases, the atoms gradually lose the ability to move freely and tend to move only to more stable configurations. In this way, after the whole process is completed, the resulting configuration corresponds to that with the highest stability. The annealing process is commonly used to obtain a more consistent and stable crystal structure which increases the metal's durability.

Simulated Annealing is a metaheuristic optimization algorithm that intends to emulate the process explained above. This method works with one candidate solution at a time. The simulation starts

at a high temperature at which the candidate solution moves around the search space freely. At this stage the algorithm focuses almost exclusively on exploring the search space. In the same way as in the physical process, as the temperature decreases the candidate solution tends to move only to better solutions and thus focuses more on exploitation. This algorithm leads the Simulated Annealing to better solutions in the same way as better internal configurations are found in the annealing process.

In the simulated annealing method (SA), each point of the search space is analogous to a state of some physical system, and the function to be minimized is analogous to the internal energy of the system in that state. The goal is to bring the system, from an arbitrary initial state, to a state with the minimum possible energy.

5.2.1 Background to SA

Simulated annealing was developed in the mid 1970s by Scott Kirkpatrick and several other researchers. It was developed to better optimize the design of integrated circuit chips by simulating the actual process of annealing. An article by S. Kirkpatrick et al. named "Optimization by Simulated Annealing" explaining in more depth the Simulated Annealing can be found in the academic journal Science, Volume 220, Number 4598, 13 May 1983.

Since Simulated Annealing was devised up to today the algorithm has been successfully applied to many optimization problems of different kinds. As an example, in the aforementioned article by S. Kirkpatrick et al., SA is said to be able to solve the Travelling Salesman Problem which is a combinatorial optimization problem.

The algorithm differs from many other metaheuristics in that it uses only one approximation to the solution at a time instead of a number or population of solutions. Compared to other metaheuristics Simulated Annealing has a quite uncomplicated implementation given the relatively low number of operators and parameters used to simulate the annealing process.

5.2.2 Algorithm

By analogy with the physical process each step of the SA algorithm attempts to replace the current solution by a random solution, chosen according to a candidate distribution often constructed from solutions near the current solution. The new solution may then be accepted with a probability that depends both on the difference between the corresponding function values and also on a global parameter T, called the temperature, that is gradually decreased during the process. The dependency is such that the choice between the previous and current solution is almost random when T is large, but increasingly selects the better or "downhill" solution (for a minimization problem) as T goes to zero. The allowance for "uphill" moves potentially saves the method from becoming stuck at local optima, which is the bane of greedier methods.

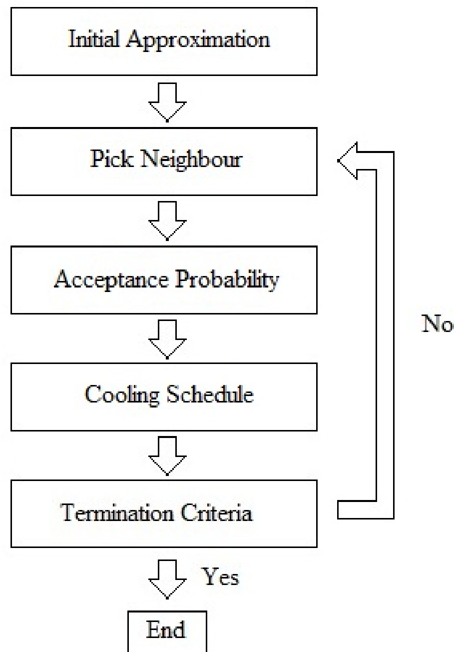

Figure 7: SA algorithm flow chart diagram (source: self made)

In other words, at each step the SA heuristic considers some neighbouring state s' of the current state s, and probabilistically decides between moving the system to state s' or staying in state s. These probabilities ultimately lead the system to move to states of lower energy. Typically this step is repeated until the system reaches a state that is good enough for the application or until a given computation budget has been exhausted, which is introduced through a termination criteria.

In the same way as GA, the algorithm is described in terms of operators (functions) and parameters since it is a computational method and this allows a more straightforward implementation.

5.2.3 Operators

In this section the operators of the Simulated Annealing algorithm are explained in detail. Some of the working mechanics of the operators are introduced here while others appear when the implementation is described or when the simulations are performed and the results to these presented.

Initial Approximation

The Initial Approximation operator works in the same exact way as the Initial Population operator of the GA with the sole difference that in the SA there is only one approximation each iteration or, in other words, the population size here is one. Therefore, the size population parameter is not needed in this algorithm.

The same two variations as in the GA's Initial Population operator are used here, namely Random Initial Approximation and Greedy Initial Approximation. These operators work exactly as their counterparts in GA but, again, only generate one initial approximation.

Pick Neighbour State

The neighbours of a state are new states of the problem that are produced after altering a given state in some particular way. For example, in the traveling salesman problem, each state is typically defined as a particular permutation of the cities to be visited. The neighbours of permutation are the permutations that are produced for example by interchanging a pair of adjacent cities. The action taken to alter the solution in order to find neighbouring solutions is called a "move" and different moves give different neighbours. These moves usually result in minimal alterations of the solution, as the previous example depicts, in order to help an algorithm to optimize the solution to the maximum extent and also to retain the already optimum parts of the solution and affect only the suboptimum parts. In the previous example, the parts of the solution are the city connections.

Searching for neighbours of a state is fundamental to optimization because the final solution will come after a tour of successive neighbours. Simple heuristics move by finding best neighbour after best neighbour and stop when they have reached a solution which has no neighbours that are better solutions. The problem with this approach is that the neighbours of a state are not guaranteed to contain any of the existing better solutions which means that failure to find a better solution among them does not guarantee that no better solution exists. This is why the best solution found by such algorithms is called a local optimum in contrast with the actual best solution which is called a global optimum. Metaheuristics use the neighbours of a state as a way to explore the solutions space and can accept worse solutions in their search in order to accomplish that. This means that the search will not get stuck to a local optimum and if the algorithm is run for an infinite amount of time, the global optimum will be found.

In continuous optimization, the neighbours of a point are all the points in its vecinity. At this point it is necessary to define what the vecinity of a point is then. Here the first parameter of the SA appears, the Search Diameter. This parameter allows to define a different size of the vecinity of a point according to the problem being solved.

In combinatorial optimization, the neighbours are all the possible resutls of performing small changes to the current approximation. For instance, swaping cities in a tour in the Travelling Salesman Problem as explained above.

Acceptance Probability

The probability of making the transition from the current state s to a candidate new state s' is specified by an acceptance probability function $P(e, e', T)$, that depends on the energies $e = E(s)$ and $e' = E(s')$ of the two states, and on a global time-varying parameter T called the temperature. States with a smaller energy are better than those with a greater energy. The probability function P must

be positive even when e' is greater than e. This feature prevents the method from becoming stuck at a local minimum that is worse than the global one.

When T tends to zero, the probability $P(e, e', T)$ must tend to zero if $e' > e$ and to a positive value otherwise. For sufficiently small values of T, the system will then increasingly favor moves that go "downhill" (i.e., to lower energy values), and avoid those that go "uphill." With $T = 0$ the procedure reduces to the greedy algorithm, which makes only the downhill transitions.

In the original description of SA, the probability $P(e, e', T)$ was equal to 1 when $e' < e$, the procedure always moved downhill when it found a way to do so, irrespective of the temperature. Many descriptions and implementations of SA still take this condition as part of the method's definition. However, this condition is not essential for the method to work, and one may argue that it is both counterproductive and contrary to the method's principle.

The P function is usually chosen so that the probability of accepting a move decreases when the difference $e' - e$ increases, that means small uphill moves are more likely than large ones. However, this requirement is not strictly necessary, provided that the above requirements are met.

Given these properties, the temperature T plays a crucial role in controlling the evolution of the state s of the system in relation to its sensitivity to the variations of system energies. To be precise, for a large T, the evolution of s is sensitive to coarser energy variations, while it is sensitive to finer energy variations when T is small.

In the formulation of the method by Kirkpatrick et al., the acceptance probability function $P(e, e', T)$ was defined as 1 if $e' < e$, and $exp((e - e')/T)$ otherwise. This formula was superficially justified by analogy with the transitions of a physical system; it corresponds to the Metropolis-Hastings algorithm, in the case where the proposal distribution of Metropolis-Hastings is symmetric. However, this acceptance probability is often used for simulated annealing even when the pick neighbour function, which is analogous to the proposal distribution in Metropolis-Hastings, is not symmetric, or not probabilistic at all. As a result, the transition probabilities of the simulated annealing algorithm do not correspond to the transitions of the analogous physical system, and the long-term distribution of states at a constant temperature T need not bear any resemblance to the thermodynamic equilibrium distribution over states of that physical system, at any temperature. Nevertheless, most descriptions of SA assume the original acceptance function, which is probably hard-coded in many implementations of SA.

Cooling Schedule

The name and inspiration of the algorithm demand an interesting feature related to the temperature variation to be embedded in the operational characteristics of the algorithm. This necessitates a gradual reduction of the temperature as the simulation proceeds. The algorithm starts initially with T set to a high value (or infinity), and then it is decreased at each step following some cooling schedule, which may be specified by the user but must end with $T = 0$ towards the end of the allotted time budget. In this way, the system is expected to wander initially towards a broad region of the search space containing good solutions, ignoring small features of the energy function; then drift towards low-energy regions that become narrower and narrower; and finally move downhill according to the steepest descent heuristic.

For any given finite problem, the probability that the simulated annealing algorithm terminates with the global optimal solution approaches 1 as the annealing schedule is extended. This theoretical result,

however, is not particularly helpful, since the time required to ensure a significant probability of success will usually exceed the time required for a complete search of the solution space.

The second parameter of the SA is related to the Cooling Schedule. As mentioned above, the algorithm starts initially with a given temperature T. Therefore this initial value must be introduced as a parameter to the algorithm according to the problem studied. This parameter has been named Initial Temperature.

Termination Criteria

In th same ways as in GA the Termination Criteria used is the fixed number of iterations reached. Therefore, here again, a Number of Iterations parameter is used to set the duration of the simulation.

5.2.4 Parameters

In order to apply the SA method to a specific problem, which is given in terms of the search space and the energy function (objective function), one must specify, for the implementation used, the following parameters: search diameter, initial temperature and number of iterations. These choices can have a significant impact on the method's effectiveness. Unfortunately, there are no choices of these parameters that will be good for all problems, and there is no general way to find the best choices for a given problem. Therefore the parameters are tested after the implementation to get a better understanding of how they vary the method's performance for the problems under study.

Search Diameter

Simulated annealing may be modeled as a random walk on a given search space, whose vertices are all possible states, and whose edges are the candidate moves. An essential requirement for the pick neighbour state function is that it must provide a sufficiently short path on this graph from the initial state to any state which may be the global optimum. In other words, the diameter of the search space must be small in terms of the time it takes from any point to get to any other point by picking neighbouring points. In the traveling salesman problem, for instance, the search space for $n = 20$ cities has $n! = 2,432,902,008,176,640,000$ (2.4 quintillion) states; yet the pick neighbour state function that swaps two consecutive cities can get from any state (tour) to any other state in at most $n(n-1)/2 = 190$ steps.

The search diameter permits to determine a subspace of the search space which contains the candidate neighbour states to which the method may jump next. The larger the search diameter, the more number of candidate neighbours and the faster it is to get from any given point to any other point of the search space. If the search diameter is too small it makes it difficult for the method to find the global optimum.

In an euclidean space the search diameter can be undestood like the diameter of the circle containing all the candidate neighbour states and centered in the current state.

Note that this parameter exists only in Continuous Optimization. Thus in Combinatorial Optimization, for example, the neighbours must be picked following another criteria such as swapping two consecutive cities for the travelling salesman problem.

Initial Temperature

This parameter represents the temperature T with which the method starts. Given a certain cooling schedule, the higher the initial temperature value is the more time or iterations it will take to get to a temperature of $T = 0$. This translates in a better solution in detriment of the computational cost. The way in which the temperature descends through the iterations is determined by the cooling schedule therefore the initial temperature must be chosen knowing the cooling schedule and jointly with the number of iterations. These two parameters, initial temperature and number of iterations, together with the cooling schedule operator dictate how the method behaves and how much emphasis the method puts into exploration and exploitation. The final temperature of the method is always $T = 0$.

Number of Iterations

The number of iterations the simulation runs determines the number of times the whole algorithm is executed and thus represents the computational cost or elapsed time and the quality of the solution obtained in the same way the initial temperature does. The number of iterations should be high enough to be able to obtain a acceptable solution and low enough to be able to run the simulation in a reasonable time.

During the simulation, when the temperature drops to $T = 0$, the algorithm becomes exclusively focused on exploitation. At this point, it is usually recommended to allow the SA to run some more iterations with $T = 0$ in order to obtain the best solutions.

This parameter is analogous to the number of generations in the Genetic Algrithm.

6 Benchmark Problems

In this section benchmarks commonly known in the literature are presented, an effort has been put into finding engineering optimization problems.

In order to test the optimization algorithms studied, different types of problems have been used. Specifically three types of optimization problems are utilised to test the algorithm studied, namely Unconstrained Continuous Optimization, Constrained Continuous Optimization and Combinatorial Optimization. Only certain methods can be used for each type of optimization.

Several common benchmark problems are used as representation of Unconstrained Continuous Optimization. These problems are widely used as testing methods in the literature and are therefore included in this dissertation.

The Constrained Continuous Optimization problem studied belongs to the area of structural engineering optimization. The problem consists on the optimization of the structure presented by Schmit (1960). Although the structure consists of a small number of beams its optimization is not different from a larger structure in terms of complexity.

In the last place, the Travelling Salesman Problem is proposed to test the algorithm's behaviour in Combinatorial Optimization. This problem has been very well known and broadly studied up to date. The problem can be considered to belong to the area of transport engineering optimization.

The optimization problems described in this section are used to evaluate the optimization algorithms studied and to compare how metaheuristics perform in relation to classical algorithms.

6.1 Unconstrained Continuous Optimization

6.1.1 De Jong's Function

The so called first function of De Jong's is one of the simplest test benchmark. Function is continuous, convex and unimodal. It has the following general definition

$$f(x) = \sum_{i=1}^{n} x_i^2. \tag{11}$$

Test area is usually restricted to the hypercube $-5.12 \leq x_i \leq 5.12$, $i = 1, ..., n$. Global minimum $f(x) = 0$ is obtainable for $x_i = 0$, $i = 1, ..., n$.

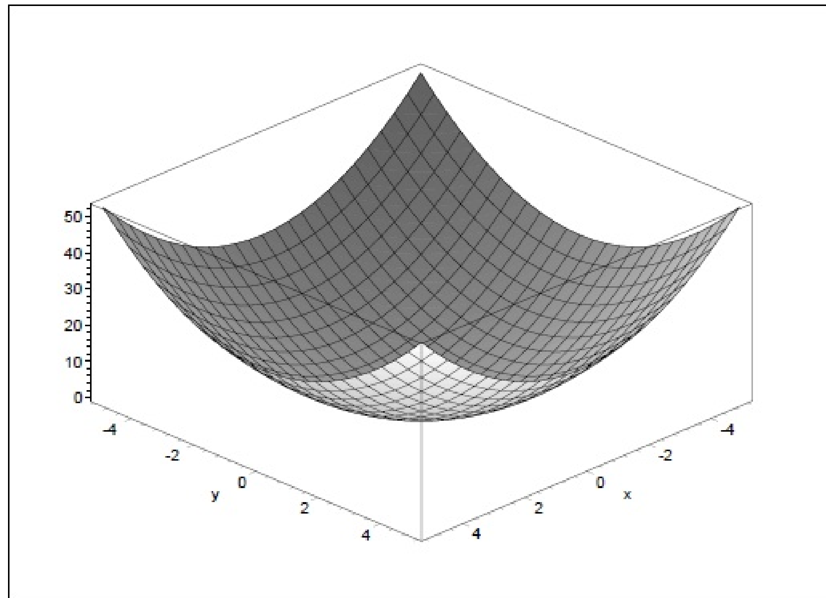

Figure 8: Graphical representation of De Jong's function in 2D (source: [13])

6.1.2 Rastrigin's Function

Rastrigin's function is based on the function of De Jong with the addition of cosine modulation in order to produce frequent local minima. Thus, the test function is highly multimodal. However, the location of the minima are regularly distributed. Function has the following definition

$$f(x) = 10n + \sum_{i=1}^{n} [x_i^2 - 10\cos(2\pi x_i)]. \tag{12}$$

Test area is usually restricted to the hypercube $-5.12 \leq x_i \leq 5.12$, $i = 1, ..., n$. Its global minimum equal $f(x) = 0$ is obtainable for $x_i = 0$, $i = 1, ..., n$.

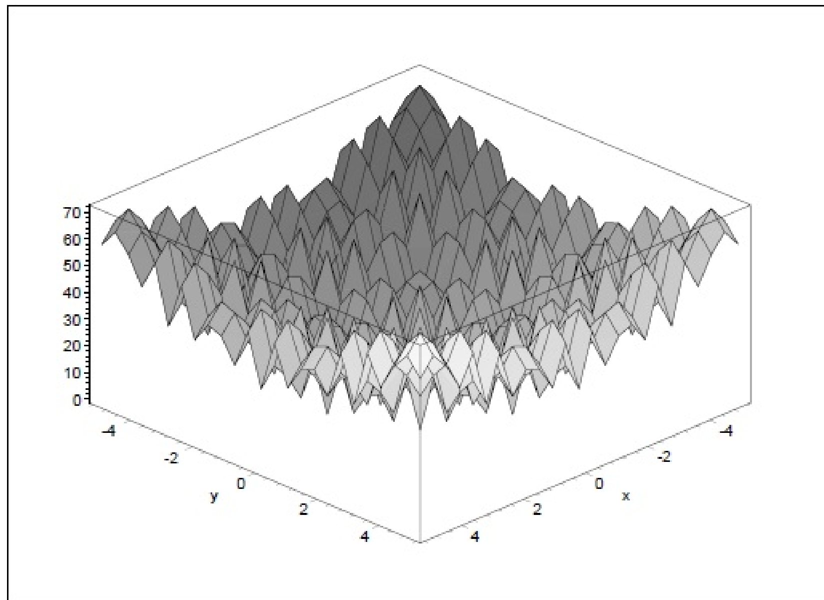

Figure 9: Graphical representation Rastrigin's function in 2D (source: [13])

6.1.3 Six-Hump Camel Back Function

The six-hump camel back function is a global optimization test function. Within the bounded region it owns six local minima, two of them are global ones. Function has only two variables and the following definition

$$f(x_1, x_2) = (4 - 2.1x_1^2 + \frac{x_1^4}{3})x_1^2 + x_1x_2 + (-4 + 4x_2^2)x_2^2. \tag{13}$$

Test area is usually restricted to the rectangle $-3 \leq x_1 \leq 3$, $-2 \leq x_2 \leq 2$. Two global minima equal $f(x) = -1.0316$ are located at $(x_1, x_2) = (-0.0898, 0.7126)$ and $(0.0898, -0.7126)$.

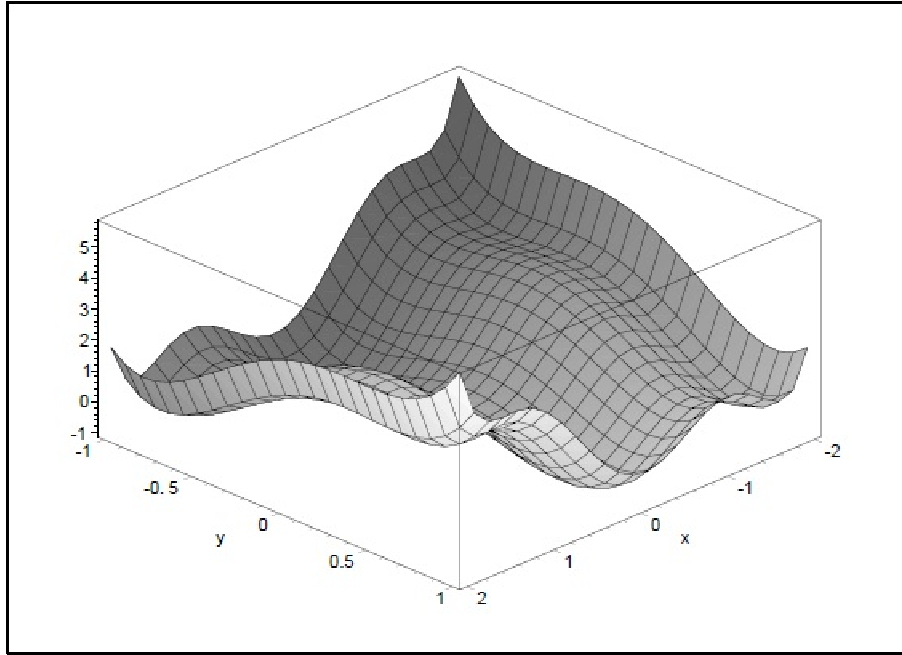

Figure 10: Graphical representation Six-hump camel back function (source: [13])

6.2 Constrained Continuous Optimization

6.2.1 Schmit Structure

Structural optimization is a branch of engineering design optimization that has seen much research during the 1960s thanks to the studies by Schmit in 1960. Schmit noted that the optimization methods at that time were unable to find optimal solutions to structural design problems and often found suboptimal solutions.

He proved that the theories of simultaneous failure modes and maximum stress approach design method failed at finding the optimal of many structure designs since such optimal does not always correspond

to the case where all the failure modes occur simultaneously. In other words, the optimal design for a structure is not always the one in which all the beams are under the maximum stress.

To demonstrate assertion, Schmit presented the simple example shown below.

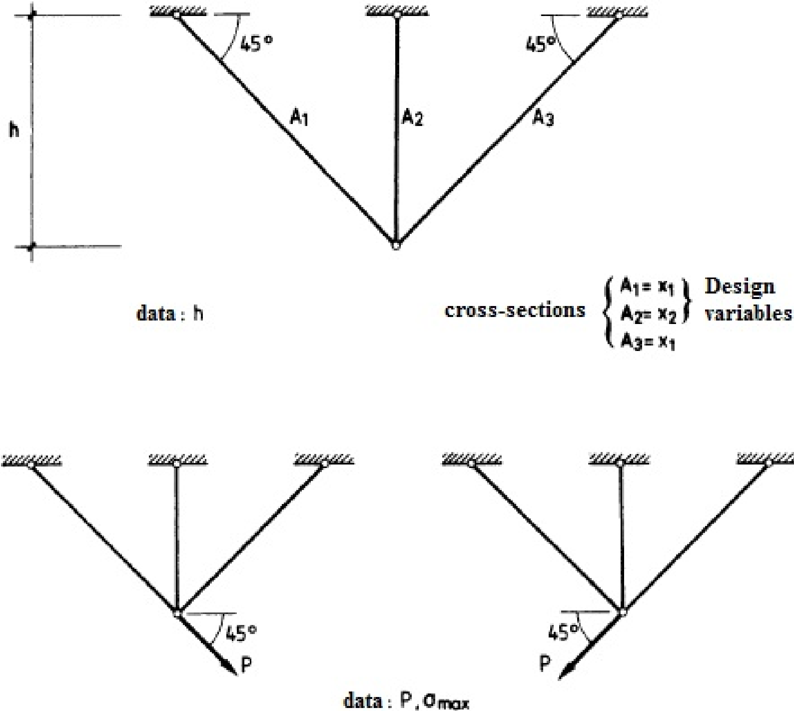

Figure 11: Schmit structure (source: [14])

If the presented articulated structure is designed using the maximum stress approach with the beams' cross sections as optimization variables and the structure total weight as the objective function, considering two different loading scenarios, a quasi-optimal solutions is found where the section of the middle beam is null.

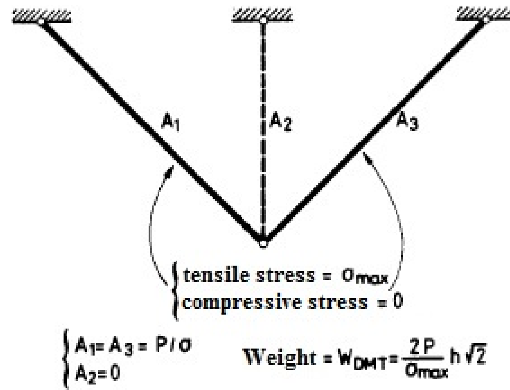

Figure 12: Schmit structure isostatic quasi-optimal solution (source: [14])

However, if the structure is solved by means of other optimization algorithms, a better solution can be found which is in fact the optimum solution for the proposed design. In this new solution the three beams have non-null cross sections, the total weight of the structure is smaller than the one of the quasi-optimal solution and none of the beams are at their maximum stress. This result shows that the theory of simultaneous failure modes and maximum stress approach do give suboptimal solutions in some cases.

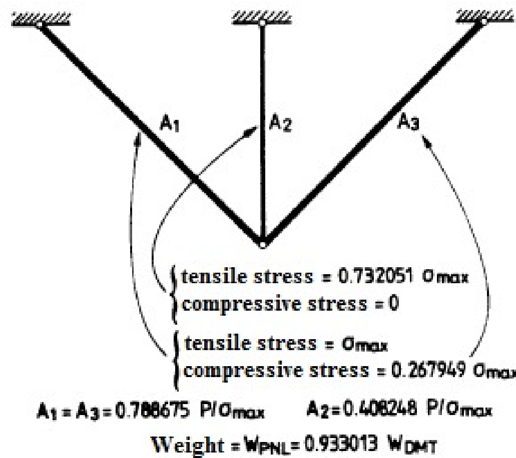

Figure 13: Schmit structure hyperstatic optimal solution (source: [14])

However, by introducing into the design problem the possibility of modifying the geometry of the structure and not only the beam's cross sections, the same mathematical programming techniques lead

to the optimal isostatic solution of different geometry and smaller weight than both of the previous designs.

The problem can be expressed in a mathematical way through an objective function, which is the structure weight, and three constraints that represent the condition that the structure must resist the loads. The variables x_1 and x_2 correspond to the area of the beam's cross-sections. The structural optimization problem can be stated in terms of the objective function (23) and the constraints (24) shown below these lines.

$$\underset{\boldsymbol{x}\epsilon\Re^2}{minimize} \quad f(\boldsymbol{x}) = 2 \cdot \sqrt{2} \cdot x_1 + x_2 \tag{14}$$

$$subject\,to \quad \begin{array}{c} \psi_1(\boldsymbol{x}) = \frac{1-\sqrt{2}}{\frac{x_2}{x_1}+\sqrt{2}} < 1 \\ \psi_2(\boldsymbol{x}) = \frac{\sqrt{2}}{\frac{x_2}{x_1}+\sqrt{2}} < 1 \\ \psi_3(\boldsymbol{x}) = (1 - x_1 \cdot (1 - \frac{\sqrt{2}}{\frac{x_2}{x_1}+\sqrt{2}}))/x_1 < 1 \\ \psi_4(\boldsymbol{x}) = x_1 > 0 \end{array} \tag{15}$$

6.3 Combinatorial Optimization

6.3.1 Travelling Salesman Problem

The Travelling Salesman Problem (TSP) is a classic combinatorial optimization problem used commonly as a benchmark in literature. The problem consists in finding the shortest possible tour that visits a number of cities and returns to the original city. Each city must be visited only once.

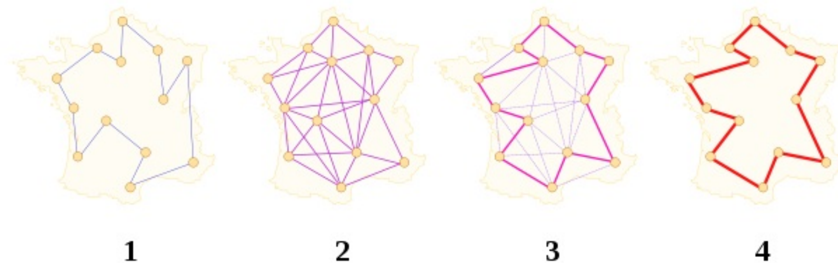

Figure 14: Travelling Salesman Problem illustration (source: wikipedia)

The problem was first formulated as a mathematical problem in 1930 and is one of the most intensively studied problems in optimization. It is used as a benchmark for many optimization methods. Even

though the problem is computationally difficult, a large number of exact methods and heuristics are known, so that some instances with tens of thousands of cities can be solved.

Exhaustive Search can be run to find the exact solution to the Travelling Salesman Problem. Although it does always find the optimum to the problem, running this optimization method is very expensive in terms of computational cost and becomes impractical even for only 20 cities.

Various heuristics and approximation algorithms, which quickly yield good solutions have been devised. Modern methods can find solutions for extremely large problems (millions of cities) within a reasonable time which are with a high probability just $2-3\%$ away from the optimal solution.

The TSP has several applications in many different areas related to engineering, such as planning or logistics. Besides it is used to test new optimization algorithms since it is a good benchmark given its computational complexity.

In the Travelling Salesman Problem the objective function is nothing but the tour length expressed in terms of the coordinates of each city. There are no constraints in this problem, other than the fact that a city can not appear twice in a same tour.

7 Implementation and Testing

This section collects details on the implementation of the various methods studied. Some methods favour the use of a particular programming language that may have built-in functions which make implementation much smoother. Therefore two different programming languages are used, namely MATLAB and C++. In order to make the explanation more comprehensible, the implementation is presented by means of pseudo-code unrelated to any specific programming language. The aim is to present the implementation in such way that it can be used as a guideline by any engineer who might want to do their own implementations. It must be taken into account that, obviously, there are multiple ways in which each algorithm can be successfully implemented.

In addition to the above, the methods are tested to check the implementation and determine the most suited combination of parameters for solving the benchmark problems. Every method is explained separately sorting the information in a more accessible manner.

7.1 Newton's Method

This method, as explained earlier in this dissertation, is used to find the root of the first derivative of the objective function. In this way the method can successfully find the solution to the optimization problem with such objective function. Moreover, to be able to search expressly for a minimum or maximum the method must be further modified. This modification is made at the point of the algorithm where the new approximation is calculated using the current approximation and the gradient. It consists in adding or subtracting the gradient to the current approximation in such way that the value decreases when searching for a minimum and increases when searching for a maximum.

7.1.1 Implementation

The Newton's Method implementation to solve unconstrained continuous optimization problems is quite straightforward. In fact, it is the exact same algorithm as the basic method used for finding a function's roots except the function is replaced by the first derivative of the function. In order words, the function in the formula is replaced by the first derivative of the function (Jacobian matrix) and the first derivative is replaced by the second derivative (Hessian matrix). The program implemented must take as input the function, initial approximation and tolerance. The output is obviously the solution to the optimization problem or, in other words, the optimal value for the variables and function.

Since the method requires the calculation of the Jacobian and Hessian matrices the implementation favours the use of math oriented programming languages such as the MATLAB package. It is possible in MATLAB to take advantage of the built-in functions for calculating the mentioned matrices.

The implementation is presented below these lines by means of pseudo-code related to no programming language in particular. Therefore, the definition of the variables, their type and their memory allocation, for example, are not included.

`Request user to input function equation` (f)

`Request user to input initial approximation` $(initial\,approximation)$

`Request user to input tolerance` $(tolerance)$

`Calculate Jacobian and Hessian matrices` (J, H)

`Set initial approximation as current approximation` $(x = initial\,approximation)$

`Start iteration loop`

`Evaluate Jacobian and Hessian matrices for the current approximation` $(J(x), H(x))$

`Calculate gradient` $(grad = H(x)^{-1} \cdot J(x))$

`Calculate new approximation` $(x_{i+1} = x_i - grad)$

`Check if new approximation is solution` $(\sqrt{\sum grad_i^2} < tolerance)$

`Stop loop if solution found`

`Output solution`

The modified algorithm of the Newton's Method that permits to search expressly for a minimum or maximum separately is only a small variation of the above. The difference resides in the calculation of the new approximation alone. Instead of subtracting the gradient from the current approximation to obtain the new approximation, the gradient is added or subtracted in such way that the resulting new approximation corresponds to a lower function value (minimization) or a higher function value (maximization). Thus each new approximation gets closer to the desired type of optimum and eventually finds the searched solution.

The pseudo-code for this modification is the following:

`Calculate new approximation` $(x_{i+1} = x_i \pm grad\,so\,that\,f(x_{i+1}) < f(x_i)\,in\,minimization\,while\,f(x_{i+1}) > (x_i)\,in\,maximization)$

There are no additional considerations to be made on the implementation but it is worth to point out that a valid input is required so, for instance, the function must be differentiable in the search space.

7.1.2 Testing

The most important input in Newton's Method is the objective function of which we want to find the optimum. It is well known that the behaviour of the method depends in great extent on the form the given function adopts. Since Newton's Method has been used widely there is many information on the literature about its behaviour before specific type of functions.

It is worth to point out once more that in solving optimization problems Newton's Method is used to find the roots of the first derivative of the objective function whose optimal is being searched. Therefore the form of the first derivative dictates the performance of Newton's Method. This method is known to have a quadratic convergence near the solution, or any local optimum, and a linear convergence elsewhere.

In this study, the implemented modified Newton's Method is used to resolve the benchmark problems for unconstrained continuous optimization presented in the corresponding section of the dissertation. Some testing is necessary to be able to understand how the method performs when facing these type of optimization problems and discuss the two parameters that can be varied, which are the initial approximation and the tolerance.

An objective function, representative of the problems under study, must be selected in order to carry out the testing. To be able to extract as many conclusions as possible, a function with several local optima is favoured.

The function used in the tests is Rastrigin's Function in one dimension. Such function has been selected because it is multimodal and a recurrent benchmark in optimization. Moreover, the function has already been introduced since it is one of the problems under study. The testing in this section is done with the one-dimensional version of the function to ease the graphical representation and interpretation of the results obtained. As long as it is possible this same function will be used to test the resting methods.

The graphical representation of the function is the following

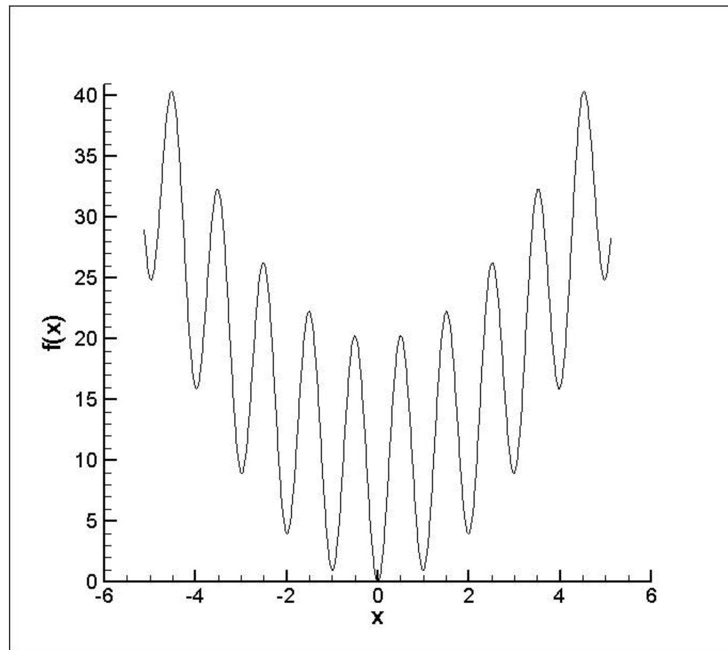

Figure 15: Graphical representation of Rastrigin's function in 1D (source: self made)

As explained previously, Newton's Method solves the optimization problem by finding the roots of the first derivative of the objective function. Therefore it is convenient to include a graphical representation of Rastrigin's Function first derivative in order to make the interpretation of the method's behaviour easier. Newton's Method advances with the slope of the first derivative of the objective function, hence its behaviour is dictated by the form the derivative adopts.

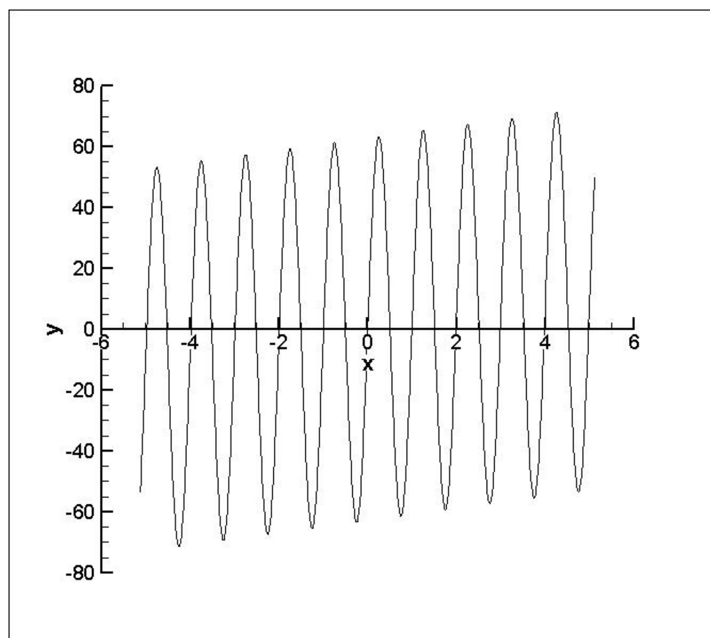

Figure 16: Graphical representation of Rastrigin's function first derivative in 1D

Both parameters are studied separately. Although it is possible that the behaviour of one can be influenced by the behaviour of the other, this interaction can be neglected.

In the first place, the influence of the initial approximation chosen is studied. This is done by using the method to find the minimum of Rastrigin's Function with a number of different initial approximations belonging to the search space for such problem. The search space is given by $-5.12 \leq x \leq 5.12$. Over a hundred initial approximations have been tested. Even though these are not enough to obtain a smooth graphical representation, it is indeed enough to be able to understand the method's behaviour with the initial approximation selected. The tolerance has been fixed to a value of 0.000001 while testing the initial approximation influence. Even though this value may seem very small, if a greater value were to be used then the differences between iterations would be almost non-existent.

The results of testing the initial approximation shows that, as expected, the solution found depends greatly on the initial approximation used. Given the function's first derivative and how Newton's Method works, the algorithm falls into local optima close to the initial approximation. If the initial approximation belongs to a point with a high slope in the first derivative then the solution obtained corresponds to a nearby local minimum. On the other hand, if the initial approximation happens to be a point with a low slope value then the solution obtained is usually far away from the initial approximation. This explains why the method does not strictly find a better solution the closer the initial approximation is to the global optimum. Therefore, choosing an initial approximation close to the global optimum is not always enough. The function's first derivative must also be taken into account, if possible, when selecting the initial approximation.

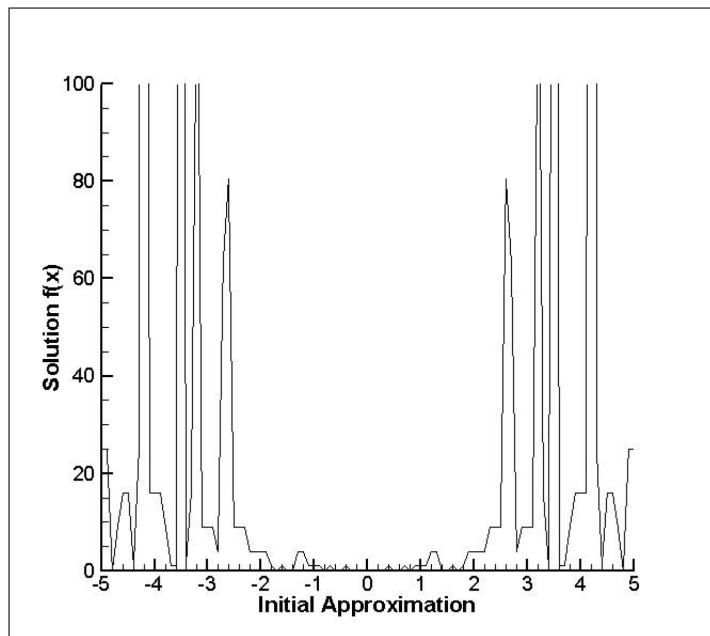

Figure 17: Graphical representation of the behaviour of solution f(x) with initial approximation (source: self made)

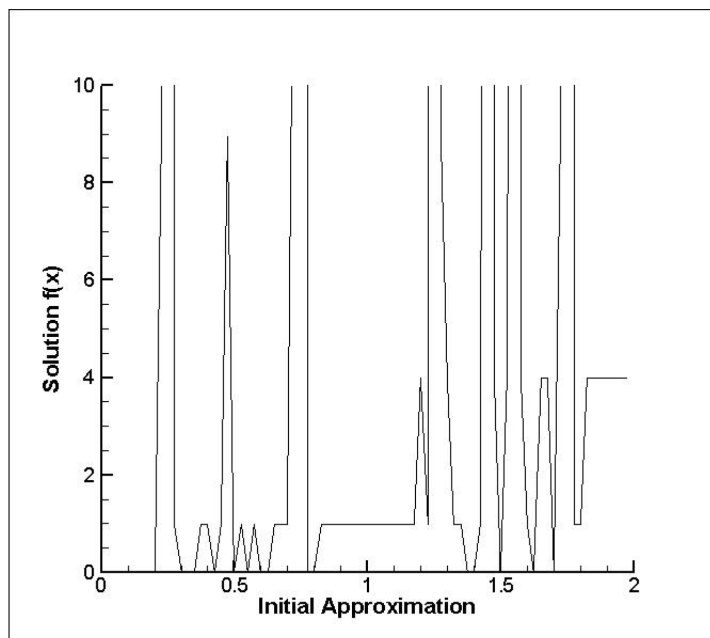

Figure 18: Zoomed graphical representation of the behaviour of solution f(x) with initial approximation (source: self made)

The behaviour of the solution obtained with the initial approximation explained above can be easily appreciated when looking at the figures below. If a positive initial approximation is chosen and the slope of the first derivative of the function in this point happens to be very steep, then the solution found is as well positive. If for the same point the slope happens to be very low then the solution obtained can be a local optimum far away from the initial approximation. For instance, in the figure representing the solution x obtained according to the initial approximation used, it appears clear that with an initial approximation $x = 4.25$ the solution obtained is almost at $x = -30$. This point belongs to a local minimum very far away from the global minimum and, therefore, the solution $f(x)$ at this point is a lot higher than it is at the global minimum, where $x = 0$ and $f(x) = 0$. Thus the solution obtained with such initial approximation is very imprecise.

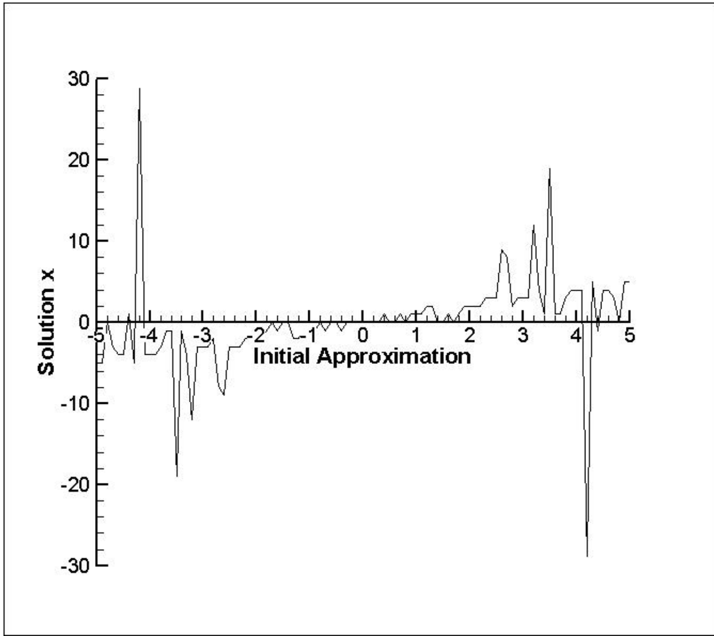

Figure 19: Graphical representation of the behaviour of solution x with initial approximation (source: self made)

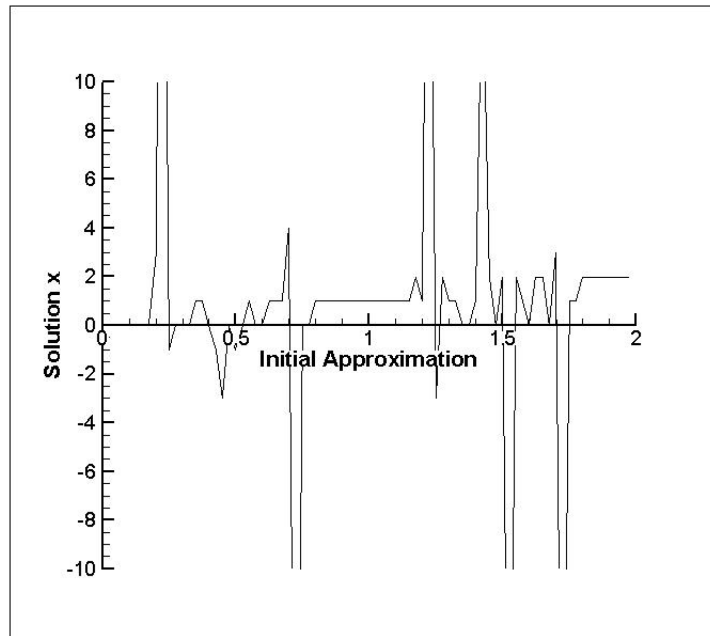

Figure 20: Zoomed graphical representation of the behaviour of solution x with initial approximation (source: self made)

When several computational methods are to be compared to each other in terms of their performance, the solution obtained alone is not enough to conclude what the best method is. The computational cost is key in determining a method's efficiency. Following this approach, the comparison between the methods must be in terms of the solutions obtained for a certain computational cost. Computational cost refers to the number of operations done throughout the simulation but can also be understood as the time elapsed during such simulation.

The time elapsed for every simulation performed using Newton's Method is under one second, so the computational cost is not really an issue in this method. Despite this fact, the number of iterations until the solution has been reached are presented too. Obviously, the closer the initial approximation to a local optimum, the faster the method converges and the less iterations needed. Bear in mind that the initial approximation must be close to the local or global optimum in terms of distance and favourable slope of the first derivative at the initial approximation.

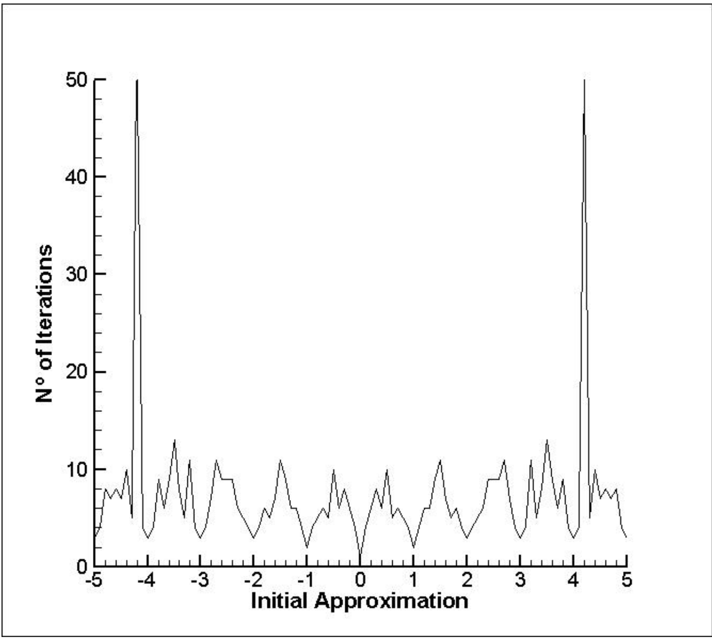

Figure 21: Graphical representation of the behaviour of cost with initial approximation (source: self made)

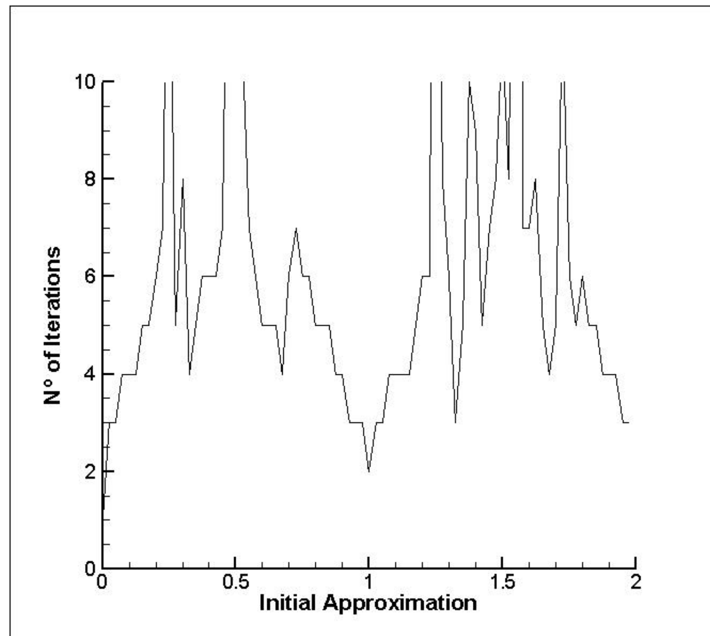

Figure 22: Zoomed graphical representation of the behaviour of cost with initial approximation (source: self made)

The second parameter, the tolerance, is studied in a similar manner. In this case the initial approximation is set to a fixed value of 0.18 and the tolerance is given a series of value ranging from 0 to 1. A reasonable value for the tolerance is strictly determined by the problem under study and the physical attributes represented, if any.

The results of testing the different tolerance values appear below these lines.

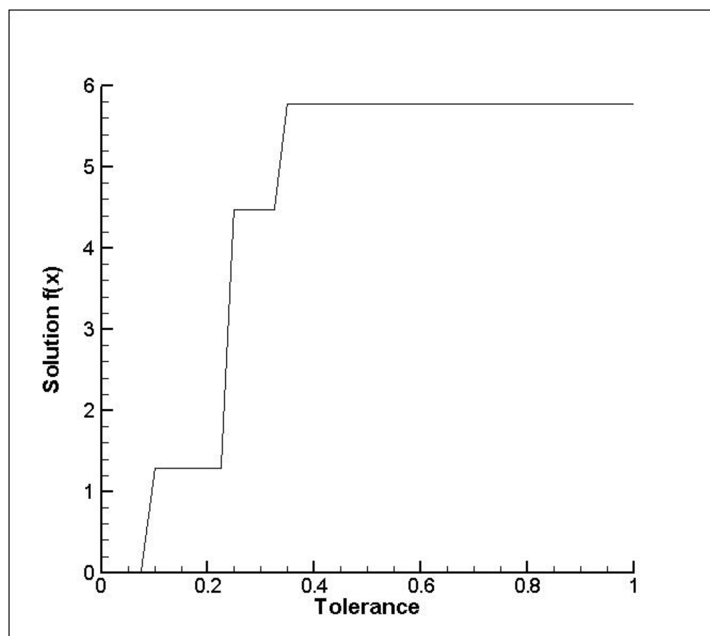

Figure 23: Graphical representation of the behaviour of solution f(x) with tolerance (source:self made)

As is obvious, the smaller the tolerance the smaller the error in the solution obtained and the more precise it will be. The tolerance does not, however, improve the method's behaviour. In other words, the solution obtained will belong to the pit of the same local optimum independently of the tolerance used. The slope of any local optimum is equal to zero, the tolerance can be understood as the maxium slope admitted for the solution. Therefore, the tolerance is a exploitation tool which will allow to obtain more precision in the local optimum found.

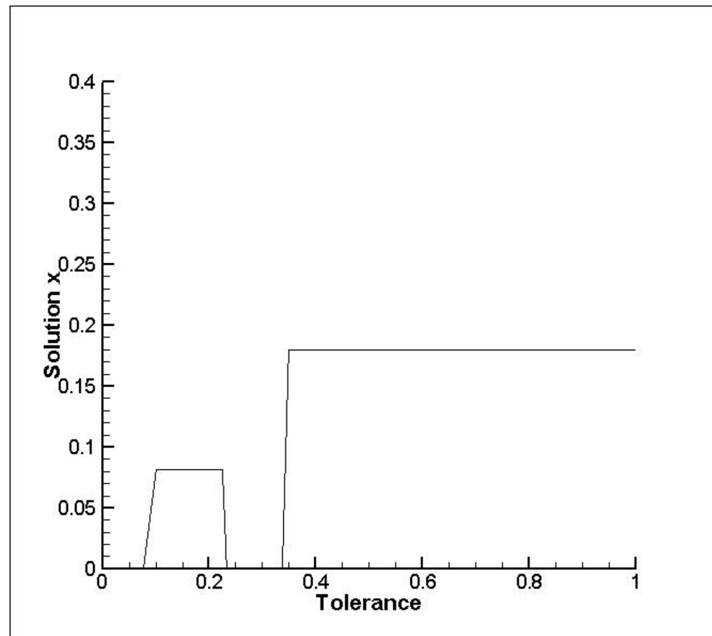

Figure 24: Graphical representation of the behaviour of solution x with tolerance (source: self made)

Although, as mentioned before, the method's computational cost and number of iterations are very low, these have also been studied. The results of the testing show that the number of iterations increases while the tolerance decreases. It must be noted that the number of iterations is not linear with the tolerance.

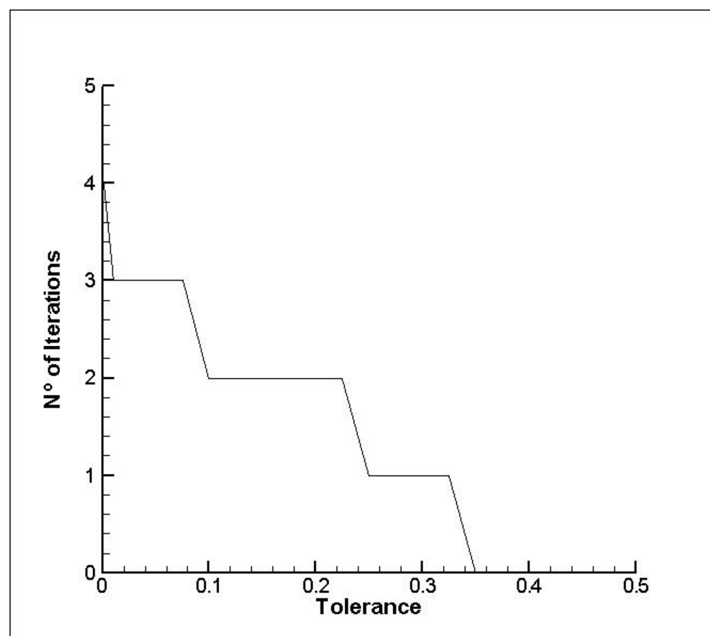

Figure 25: Graphical representation of the behaviour of iterations with tolerance (source: self made)

From the testing performed, several conclusions can be extracted. In the first place, the initial approximation chosen is of utmost importance and must be selected according to the form the function and its first derivative adopt. A suitable initial approximation is one as close as possible to the global optimum and with a considerable slope. Other feasible possibilities for the initial approximation include any point at which the slope of the first derivative of the function studied points towards $x = global\,optimum$, $f(x) = 0$.

Since the computational cost is not really an issue in this method, the tolerance can be set to a very low value in order to obtain the best results.

All in all, while the method excels at exploitation of a certain local optimum, it fails at the exploration of the whole searc space and therefore is not a robust method. Having to choose a suitable initial approximation according to the problem studied, in order not to fall into local optima, is the main drawback of Newton's Method.

7.2 Exhaustive Search

The Exhaustive Search is theoretically a very simple method. The algorithm consists merely on travelling through every point in the search space, evaluating the function and checking whether a best solution has been found. As simple as this might seem at first, when talking in terms of the implementation, it actually bears some complexity. This is due to the fact that it is impossible to travel through every single point in a continuous search space in a finite time.

The method has been implemented to be able to perform well in combinatorial optimization as well as continuous optimization. The difference being that in the first the search space is discrete while in the second the search space must be discretized by means of, for example, the grid size.

It is possible to use this method to solve constraint optimization problems by checking the constraints at each point while evaluating the function and rejecting the solutions that do not verify the constraints. This has also been implemented and it does not add in complexity despite the fact that it does add in computational cost.

This method has one parameter which can be modified to increase the performance in continuous optimization, such parameter is the grid size. This parameter is an indicator of the number of points in the discretized search space and therefore the computational cost of evaluating all of them through the objective function. The values for the grid size which better work with the continuous problems studied in this dissertation are obtained through testing.

In the same way, in combinatorial optimization the number of points in the search space will determine the cost of running the method. Therefore the method is also tested through the Travelling Salesman Problem to be able to understand how does the computational cost, which can also be represented in terms of the number of iterations or simulation time, behave with the increment in the number of points.

7.2.1 Implementation

The only input the Exhaustive Search requires is the objective function to be optimized, the search space and the grid size in the case of continuous optimization. As mentioned before, the algorithm has to travel through every point in the search space, evaluate the function and check if the optimum has been found.

```
for every possible solution
  evaluate objective function with the solution's variables
  if best solution has been found
    store best solution
```

If the algorithm is implemented for a specific problem, hence the number of variables is known beforehand, the implementation consists mainly in a number of nested for loops equal to the number of variables in the problem. In each iteration, or point visited, the function is evaluated and stored if an optimum has been found.

On the other hand, if an implementation of the method is to be made that can handle different problem inputs with different number of variables, then travelling through the search space is no longer trivial to implement. The nested for loops presents in this case the handicap that the number of loops is not known. Therefore, a modification has to be made so that the code includes a variable number of nested for loops depending on the number of variables of the problem. This step is key in the implementation and requires the most attention.

In combinatorial optimization the total number of loops corresponds to the number of possible solutions obtained by combination of the search space points. This can be done because the search space is discrete in this type of optimization and the number of possible solutions is finite. For example, in the Travelling Salesman Problem the number of loops is the number of possible tours.

In continuous optimization the implementation of the algorithm must perform a discretization of the continuous space, with an infinite number of points, into a discrete space, with a finite number of points, to be able to evaluate the solution at every point. This discretization is done by means of the grid size introduced by the user. Any given variable will adopt the values given by discretizing the search space into a set of points separated a distance equal to the grid size from each other. This can be implemented, for one given variable, through a for loop starting at the search space minimum, ending at the search space maximum and with an increase of a value equal to the grid size each iteration. The grid size determines the number of points in the search space, thus it does also determine the number of possible solutions or loops in the algorithm.

In order to search for optima of constrained continuous optimization problems the algorithm is modified to check whether the constraints are verified for each possible solution and to reject the solutions that do not verify.

When facing combinatorial optimization, in particular the Travelling Salesman Problem, the algorithm is analogous to the one above.

```
for every possible tour
  calculate tour's distance
  if shortest tour has been found
    store shortest tour
```

7.2.2 Testing

The grid size is the one parameter that can be modified to improve the method's performance and it does only appear in continuous optimization as explained before. Therefore, this section consists of a number of tests that allow a judgement to be made on what the best values of the grid size are for problems of the type of the studied.

Some conclusions regarding the parameter's behaviour can be made beforehand by means of a theoretical approach. Since the grid size determines the number of possible solutions, the computational cost increases drastically with a decrease in the grid size. A high value of the grid size means the points are more spaced and there is less number of them. In such case the computational cost is low but the precision is too. A low value of the grid size means the points are less spaced and there is more number of them, hence the computational cost will increase together with the precision of the solution obtained.

The intention of the testing is to study the behaviour of the computational cost and precision of the solution with different grid size values.

Rastrigin's Function in one dimension has been used to test the method for the same reasons as in the testing done for Newton's Method. The search space is $-5.12 \leq x \leq 5.12$. The values adopted by the grid size range from 0 to 2.

In general terms, the smaller the grid size the better the solutions obtained. This is due to the fact that the smaller the grid size, the more points on the grid and the higher the precision of the solution. This is specially true when reducing the grid size in several orders of magnitud.

Despite the above being true, there are some exceptions. Given the search space and a certain grid size, the possibility exists that by decreasing slightly the grid size a worse solution than before is obtained. This is due to the fact that, by pure coincidence, the new vertices of the grid are more far away than before from the solution even though the grid size is smaller.

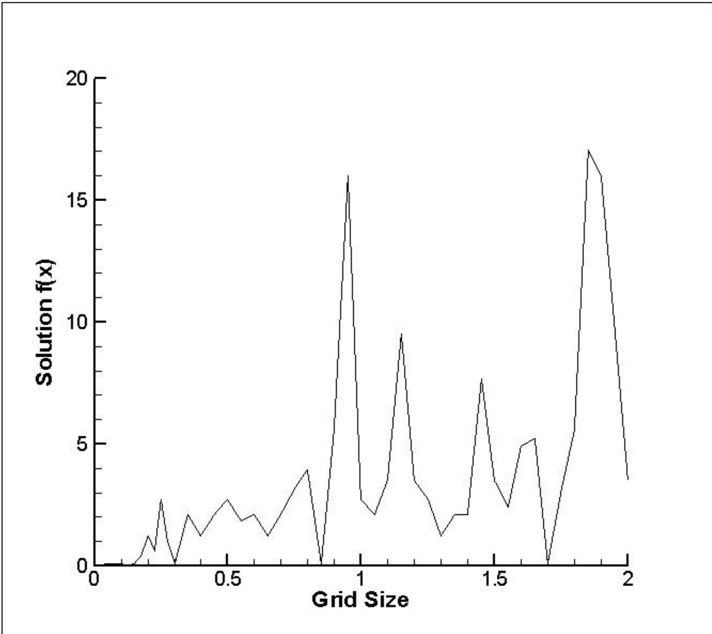

Figure 26: Graphical representation of the behaviour of solution f(x) with grid size (source: self made)

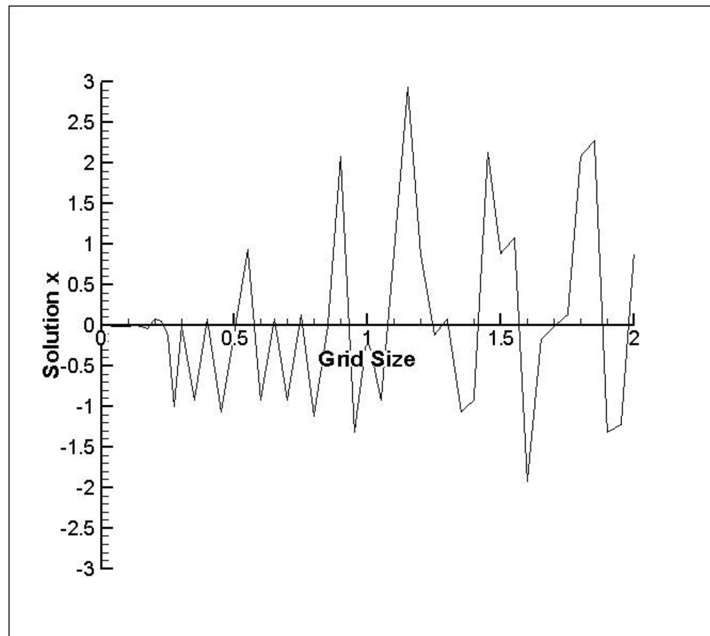

Figure 27: Graphical representation of the behaviour of solution x with grid size (source: self made)

Given the way in which the grid size determines the number of possible solutions, the number of iterations used to evaluate all the solutions grows exponentially with the decrease in grid size. The figures below illustrate this behaviour.

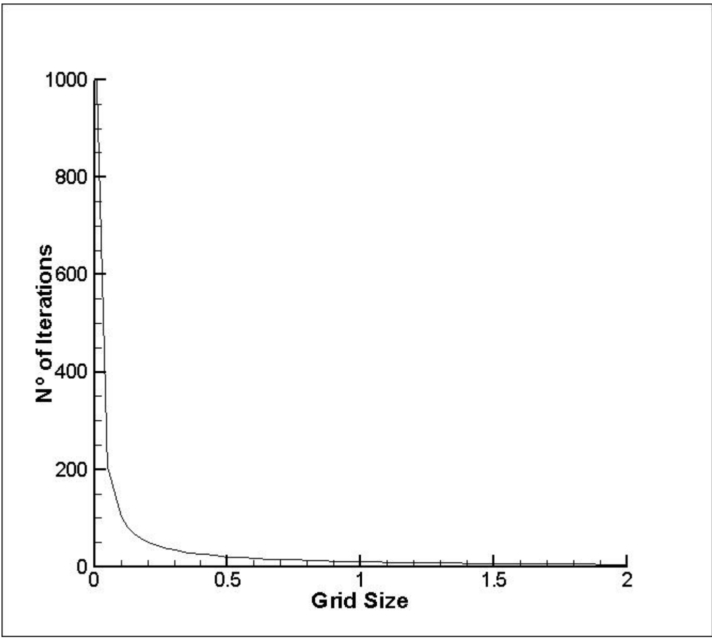

Figure 28: Graphical representation of the behaviour of nº of iterations until best solution is found with grid size (source: self made)

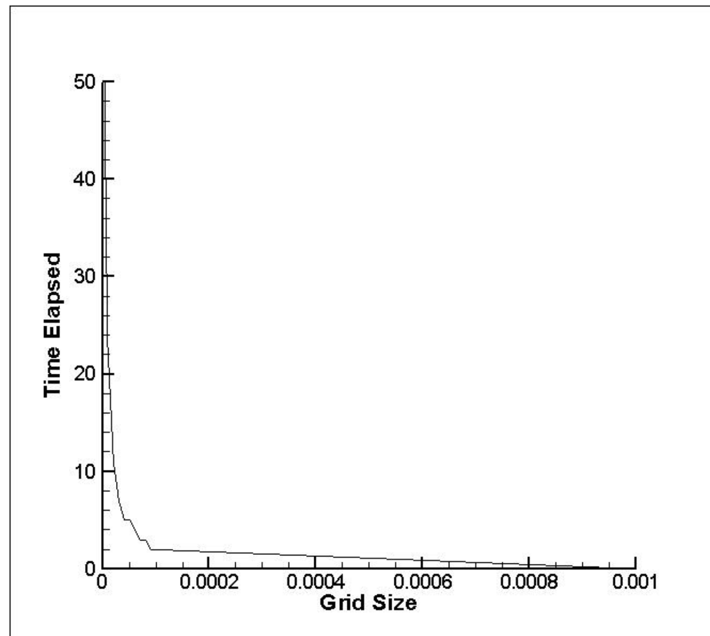

Figure 29: Graphical representation of the behaviour of simulation time with grid size (source: self made)

The above behaviour is comparable to the one expected from the method when facing combinatorial optimization. The grid size determines the number of points in the search space in continuous optimization. Similarly, in combinatorial optimization, the problem statement determines the number of points in the search space. In the assumption that in both cases the number of points is the same then running Exhaustive Search will have approximately the same computational cost. Hence, the conclusions extracted in this section are also applicable to combinatorial optimization.

The testing indicates that the best solutions are obtained for very small grid sizes but these result in high computational costs too. Thus a balance between an acceptable solution and computational cost must be found for each problem.

7.3 Pure Random Search

The implementation of Pure Random Search is very uncomplicated. The algorithm consists in generating random solutions in the search space and checking whether a best solution has been found for a certain number of iterations.

In the same way as the Exhaustive Search, the Pure Random Search can handle constrained optimization by checking if the constrained are verified when checking if a best solution has been found.

The method's only parameter is the number of iterations selected or, in other words, the number of random possible solutions generated and evaluated. As in the Exhaustive Search grid size, this parameter determines how good the solution obtained is and more importantly the cost of running the method. The testing is carried out to better comprehend this behaviour.

7.3.1 Implementation

Most programming languages have built-in random number generators. With the help of these it is trivial to generate a random number in the search space and check if a best solution has been found, for as many iterations as desired. Although this is a true statement in the practice, due to the way in which computers work it is impossible to generate a true random number. Thus the computer usually takes a base number, usually from the internal watch, and performs several mathematical operations to generate an apparently random number.

The pseudo-code for the Pure Random Search is the following:

```
for the desired number of iterations

 for every variable in the solution

  generate random number in the search space for this variable

 evaluate the objective function with the generated solution variables

 if best solution has been found

  store best solution
```

The above algorithm can be also applied to constrained optimization if the unfeasible solutions are never stored or taken into account, which would be sort of a Penalty Method. In such case before evaluating the objective function, the constraints should be checked.

When facing combinatorial optimization the method has to be modified to be able to generate random solutions of a discrete search space which can have additional constraints like the Travelling Salesman Problem does, where every city can only be picked once. For the mentioned problem, studied in this dissertation, the code has been modified so that all the cities are introduced into a pool and then picked one at a time. Therefore a valid tour is obtained after picking all the cities.

```
create a list with all the cities

 for every city in the tour

  pick randomly a city from the remaining cities in the list

  store the city coordenates

  remove the selected city from the list

 calculate the resulting tour distance

 if shortest distance has been found

  store shortest distance
```

7.3.2 Testing

Rastrigin's Function in one dimension is here again the test function used to study the influence of the number of iterations on the method's performance. The test is done in the same conditions as the other methods. The values for the number of iterations range from 1 to 200000.

Since this method's behaviour is purely stochastic, the results obtained through one simulation alone are not representative of the method's performance. Therefore, the method must be run several times and the results presented in terms of the average solution found or average time elapsed. The number of runs is to be selected considering that the number must be high enough to be able to obtain representative values and must be low enough to complete the simulations in a reasonable time. The adopted value for the number of runs in the following simulations is 100.

The graphical representation of the average solution $f(x)$ given the number of iterations draws a logarithmically decreasing function. Hence, there is a threshold beyond which a very big increase in the number of iterations has to be made in order to obtain a slightly better solution. Obviously, the more iterations run the better the solution obtained. The number of iterations must be high enough to obtain a suitable solution but low enough to be able to run the simulation in an acceptable time lapse.

The average solution x is not a reliable indicator because in a search space centered in the origin of coordinates the average of randomly generated numbers is usually close to the same origin of coordinates, which is the actual global optimum of the proble studied here. Therefore, even if the method performed poorly the average solution x expected is always around $x = 0$.

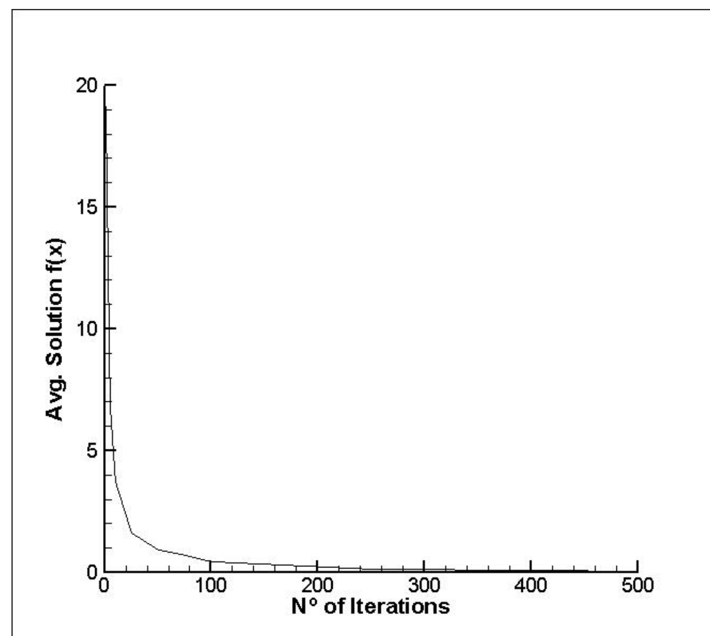

Figure 30: Graphical representation of avg. solution f(x) with number of iterations (source: self made)

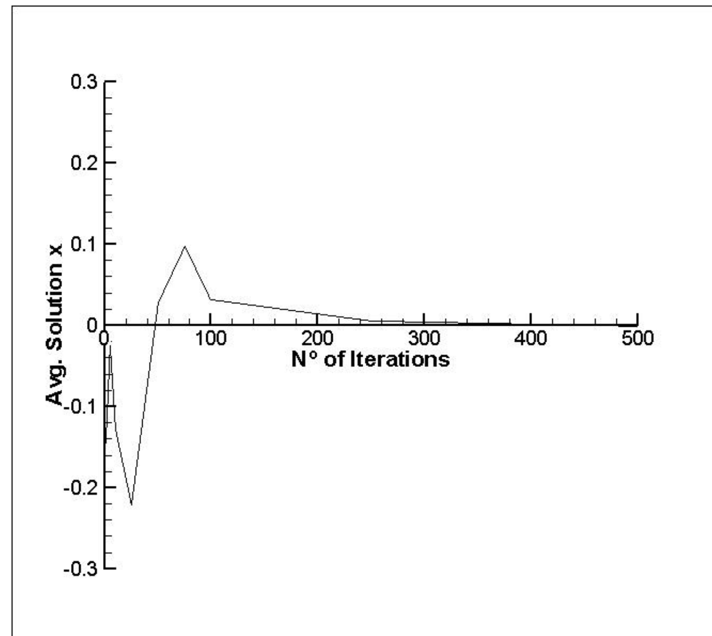

Figure 31: Graphical representation of avg. solution x with number of iterations (source: self made)

The testing done in order to determine the best value for the number of iterations shows that the time elapsed in the simulation has a linear behaviour. Thus, the simulation time increases linearly with the number of generations. Despite this fact, it appears that the solution to the test optimization problem is found after a maximum of around 10000 iterations even if the method is run for 20 times that many iterations.

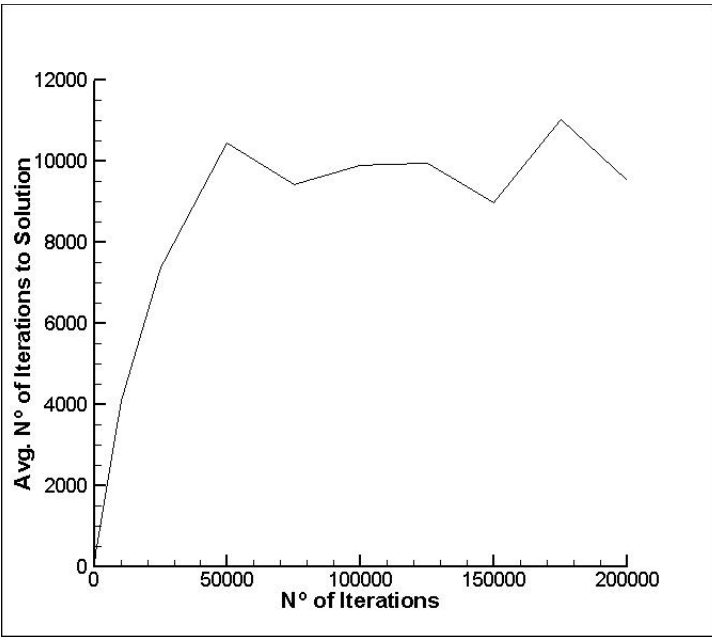

Figure 32: Graphical representation of the behaviour of avg. number of iterations until best solution is found with number of iterations (source: self made)

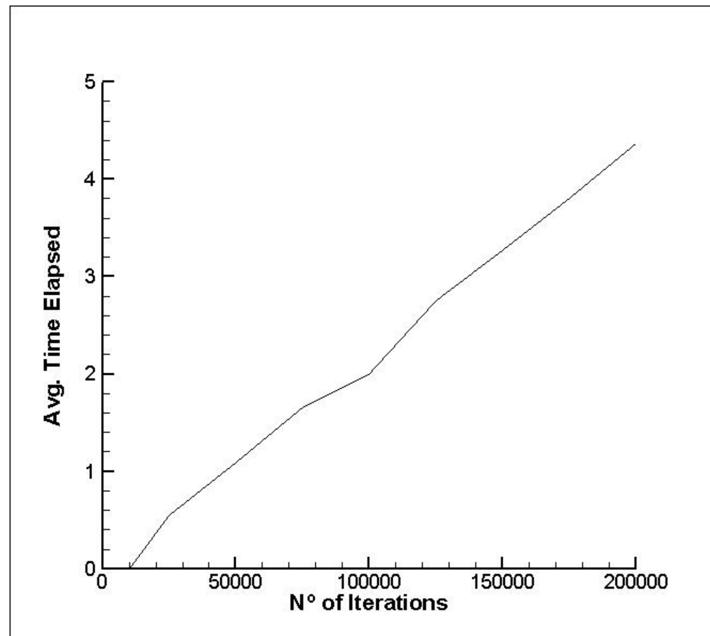

Figure 33: Graphical representation of the behaviour of avg. simulation time with number of iterations (source: self made)

7.4 Genetic Algorithm

The implementation of the GA is quite complex since there are several operators and parameters. As mentioned before, due to the fact that this is a computational method the implementation is structured in a very similar way to the actual algorithm. The implementation for most of the operators is different for Continuous Optimization and Combinatorial Optimization.

The GA can be used in Constrained Optimization by forcing the algorithm to generate only feasable solutions, chromosomes that verify the constraints, or by introducing Penalty Methods such as Death Penalty which consists in removing from the population any member that does not verify the constraints.

Since different variations of the same operators have been implemented, a test on how these perform is due too in order to conclude what set of operators work best. The results for the testing are presented in the second part of this section.

7.4.1 Implementation

The implementation of the GA results in a computer application that takes as an input the problem data, such as objective function and search space, together with the method's parameters to be introduced by the user and after running the simulation outputs the results obtained. Since GA has a

stochastic component the results obtained for one single simulation are not representative, thus the simulation must be run a number of times and the results averaged as explain later in this document.

Even though the implementation of GA is quite straightforward and practically identical to the actual algorithm, special attention must be put into the structure of the implementation's operators or functions. Every operator in the algorithm is implemented as a function and these must be run in the appropiate sequence to simulate each generation. Due to the number of operators and parameters in GA the programming language C++ has been chosen to implement the method since it favours nested classses and functions which will facilitate structuring the implementation.

There are multiple ways in which the implementation of the GA can be structured. In this dissertation, the software design pattern used is a combination of two existing patterns, namely Strategy Pattern and Template Method. The Strategy Pattern is appropiate for structuring the different operators and parameters chosen by the user since this can be understood as strategies that the user selects at runtime. On the other hand, the Template Method allows to implement an abstract class for the general algorithm and concrete classes for the different problem types, such as continuous and combinatorial optimization, which introduce some variations into the algorithm.

As mentioned before, the general flow of the implementation is the same as the GA algorithm. In the first generation the initial population is generated, then for as many generations as fixed the genetic operators selection, crossover, mutation and replacement are applied to the population. Finally, the termination criteria stops the simulation when the desired number of generations has been reached.

The GA algorithm has been implemented as a class and the different operators, which are the steps the algorithm actually does, are functions belonging to this class. The implementation is herein presented introducing separately each operator or function.

Besides the operators used the GA algorithm must input the problem statement, input the operators and parameters chosen by the user, call the operators in the logical sequence and output the results of the simulation. The GA class, where the GA algorithm is implemented, also contains several variables used to control the flow of the algorithm. The current population, selected population and offspring population are stored as lists. Moreover, the chromosomes have been implemented by means of a nested class that defines how a chromosome is for any type of problem. Here, the way in which the problem variables are encoded into the chromosomes has been studied. Along the lines of the present state of the art in GA the problem variables have been encoded using strings of numbers, of the variable type double.

Initial Population

This operator generates the initial population of members by randomly generating numbers for every gene in every chromosome of the population. The most appropiate way of presenting the implementation is to describe it by means of pseudocode. Below these lines the pseudocode for the Initial Population operator is included.

```
for every chromosome in the initial population

  for every gene in the chromosome

    generate random number in the search space for this variable
```

Most programming languages, if not all, include a function that allows to generate random numbers. Even though this is considered true, trully random numbers can not be generated by means of any existing technic. The random function usually takes a known number as a seed, for example the current time, and then performs a number of mathematical operations on this seed to obtain a resulting number that appears to be random.

While the use of the random number generator function is enough to generate an initial population in continuous optimization, in combinatorial optimization some additional code must be written to generate the random initial population.

The pseudocode for generating the initial population in the Travelling Salesman Problem, the combinatorial optimization problem studied, is presented below these lines.

```
create a list with all the cities

for every chromosome in the initial population

  for every two genes in the chromosome

    pick randomly a city from the remaining cities in the list

    store the city x and y coordenates into the two genes

    remove the selected city from the list
```

In the above implementation each city is picked only once for every chromosome, also known as tours in the TSP. Besides, the tour is selected randomly.

As mentioned before in this dissertation, two variations of this operator have been implemented. The Random Initial Population has been described above. The Greedy Initial Population is basically the same as the Random Initial Population, with the only difference that ir generates a population of twice the size of the required and then selects the 50% best chromosomes and uses them to generate the actual initial population for the GA.

Selection

Two variations of this operator have been used in this study. These variations are the Random Selection and the Roulette Wheel Selection operators. The pseudocode for the implementation of the Random Selection is something along the lines of the following:

```
for every offspring chromosome

  randomly select two parents from the current population members
```

On the otherside the Roulette Wheel Selection is a little more complex since it actually selects the parents regarding the chromosomes' fitnesses. The pseudocode for this second variation is included below these lines. It must be taken into account that the parents can not be the same population member and therefore the implementation must repeat the selection process if both parents are, by coincidence, the same member.

```
for every parent needed
```

```
add up the fitness of all the population members

generate a random number between zero and the result of the addition above

declare and initiate variable to store sum (sum = 0)

for every population member

  add the member's fitness to sum

  if sum > random number generated

    select this population member as a parent

    exit the population member loop
```

As explained before, this algorithm favours the selection of the fittest chromosomes. In the above description, the fitness represents how good a solution or chromosome is. In minimization problems the lowest value of the objective function is the best solution, thus if the evaluation of the objective function is called fitness then a lower fitness is better. For this reason, the above algorithm must also be implemented with the multiplicative inverse of the fitness according to the type of optimization problem. For instance, in the TSP the fitness represents the tour length and therefore the multiplicative inverse of the fitness in the code above must be used.

Crossover

The first variation of this operator, the Uniform Crossover, passes down to the offspring each gene randomly from either of the parents with the same probability. The pseudcode is as follows:

```
for every offspring chromosome

  generate random number between zero and one

  if the above random number < Crossover Probability

    for every gene in the offpring

      generate new random number between zero and one

      if new random number < 0.5 then copy gene from first parent

      if new random number > 0.5 then copy gene from second parent
```

The Single Point Crossover selects a point in the chromosome which divides the genetic code passed from each of the parents. The pseudocode for this variation of the operator is included below these lines.

```
for every offspring chromosome

  generate random number between zero and one

  if the above random number < Crossover Probability
```

```
generate random number between one and the number of genes in the chromosome minus one

copy an amount of genes equal to the random number obtained from the first parent starting from the ...

...beginning of the chromosome

copy the rest of the genes of the offspring chromosome from the second parent
```

The above variations of the operator work only in continuous optimization, since in combinatorial optimization the crossover will most likely generate invalid offspring that do not verify the constraints. As mentioned before, in combinatorial optimization the unfeasible offsprings result of the crossover are deleted and a new offpsring is generated through the whole same process. This method is called Death Penalty and belongs to the mentioned Penalty Methods. The problem with this solution is that if the problem is highly constrained then it may take a lot of crossover attempts to obtain every valid offspring. Thus the computational cost can be significantly increased. A more cost efficient approach is to perform the crossover in such a way that it only generates feasible offspring chromosomes.

In the Travelling Salesman Problem, for example, Uniform Crossover can be implemented by copying randomly cities from either parent with the condition that any city can only appear once in the offspring tour. The pseudocode in this case is:

```
for every offspring tour

 generate random number between zero and one

 if the above random number < Crossover Probability

  for every city in the offpring

   generate new random number between zero and one

   if new random number < 0.5

    if first parent's current city is not yet in offspring tour

     copy city from first parent

    if first parent's current city is already in offspring tour

     copy city from second parent

   if new random number > 0.5

    if second parent's current city is not yet in offspring tour

     copy city from second parent

    if second parent's current city is already in offspring tour

     copy city from first parent
```

Mutation

Only one variation for the Mutation operator has been implemented in this dissertation, for the reasons already mentioned. The pseudocode for the implementation of this operator is the following:

```
for every offspring chromosome

  for every gene

    generate random number between zero and one

    if the above random number < Mutation Probability

      mutate the current gene to a random number in the search space
```

Here again, the implementation varies slightly for combinatorial optimization. The pseudocode for the implementation used in the Travelling Salesman Problem is included below these lines.

```
for every offspring tour

  for every city

    generate random number between zero and one

    if the above random number < Mutation Probability

      swap current city with the following city in the tour
```

Although swapping contiguous cities might seem a very small mutation given the number of possible tours for a certain number of cities, it has already been noticed that with a relatively small number of swaps it is possible to get from any starting tour to any other tour possible.

Replacement

The different variations implemented for the Replacement operator are: the Random Replacement and the Elitist Replacement. The pseudocode for the random variation is included below these lines.

```
for every offspring chromosome

  select a random member of the current population

  replace the selected chromosome with the offspring chromosome
```

On the other hand, the implementation of the Elitist Replacement can be explained using the following pseudocode:

```
for every offspring chromosome

  add up the fitness of all the population members

  generate a random number between zero and the result of the addition above

  declare and initiate variable to store sum (sum = 0)
```

```
for every population member
  add the member's fitness to sum
  if sum > random number generated
    replace this population member with the offspring chromosome
    exit the population member loop
```

Here again the fitness is considered proportional to the precision of the solution. Therefore, if the fitness is the result of evaluating the objective function with the chromosome's genes as values for the variables, the above code represents only the case of a maximization problem. When facing minimization problems the fitness value in the pseudocode must be replaced with its multiplicative inverse.

Termination Criteria

The Termination Criteria is a trivial operator that stops the algorithm when the desired number of generations has been reached. The pseudocode is:

```
if generation count > maximum number of generations
  exit the algorithm
else
  add one generation to the generation count
```

The generation count must obviously be set to zero at the start of each simulation.

7.4.2 Testing

Following De Jong's example (De Jong 1975) several models of GA have been studied, each with a different set of operators. This models are then tested with a few sets of parameters in order to find out what models perform best and with what values for the parameters.

Here again, given the stochastic component of the method, a number of runs is required to be able to extract representative solutions averaged from the different runs. The GA in particular has a big stochastic component and therefore the chosen number of simulations to be run is 1000.

This method is not tested properly in a one dimensional problem since the crossover operator would have no function. Therefore the method must be tested at least with a two dimensional problem. Rastrigin's function in two dimensions has been used in the testing. The description of such function can be found in the corresponding section of this dissertation.

The differences between the different GA models tested are the operators used. Below these lines a table is included that shows what operators are used by each model.

Model	Strategies			
	Initial Population	Selection	Crossover	Replacement
GA1	Random	Random	Uniform	Random
GA2	Random	Roulette Wheel	Uniform	Random
GA3	Random	Roulette Wheel	Single Point	Random
GA4	Random	Roulette Wheel	Single Point	Elitist
GA5	Greedy	Roulette Wheel	Single Point	Elitist

Table 1: Operators for each GA model (source: self made)

The different combinations of operators work well with many sets of parameters but thoroughly studying every possibility is out of the scope of this dissertation due to the number of tests required. However, the GA models are studied with several different sets of parameters to conclude in general terms what models work best.

The parameters for the GA are: initial population, crossover rate, mutation rate and number of generations. In this section, a certain set of parameters for the GA is represented with the values for the parameters in parenthesis: (initial population, crossover rate, mutation rate, number of generations).

The first set of parameters is (100,0.8,0.2,1000). The results are presented graphically as it is the best way of analyzing them.

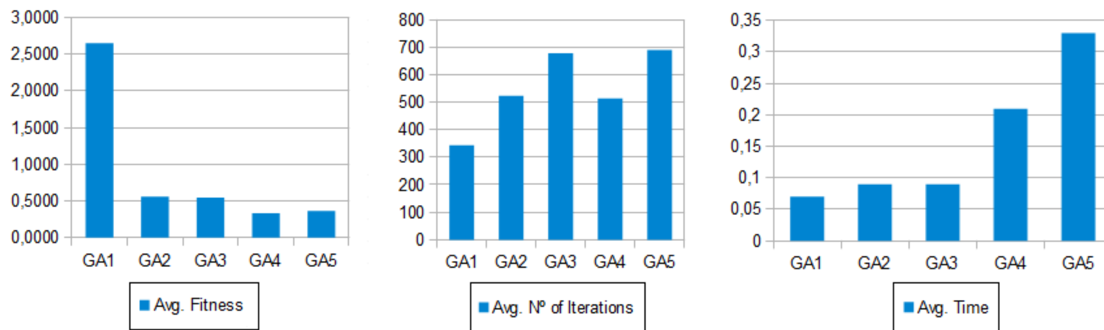

Figure 34: Graphical representation of the performance of the GA models with the 1st set of parameters (source: self made)

The testing done with the first set of parameters shows that the GA1 model, the more stochastic approach of the algorithm, obtains the worst solutions and has the shortest simulation time. The GA2 and GA3 models obtain significantly better solutions than the GA1 at a cost of a slightly longer simulation time. The last two models, the GA4 and GA5, do improve the solution obtain but it is at

a cost of a double or triple simulation time. This is due to the fact that these two models have the most complex operators, such as the greedy and elitist variations.

The second set of parameters is (1000,0.8,0.2,1000). This time the size of the initial population has been increased to 1000. With this set of parameters the solutions obtained for the GA1 are the worst again while the remaining models give similar solutions. In terms of the computational cost or simulation time, the same pattern as for the first set of parameters is presented. The GA1 model is the fastest. The GA2 and GA3 models take approximately twice the time the GA1 does. The GA4 model simulation time is as well twice the time taken by GA2 and GA3. Finally, the GA5 model has the longest simulation time being about twice the GA4 model simulation time.

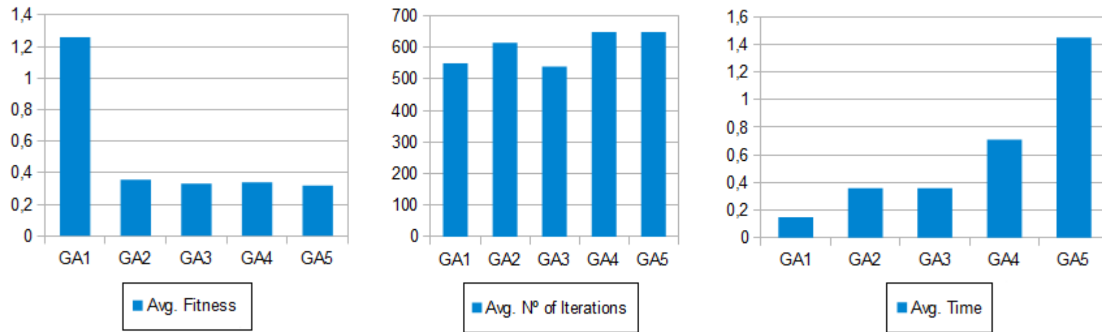

Figure 35: Graphical representation of the performance of the GA models with the 2nd set of parameters (source: self made)

FIGURE.

The third set of parameters is (100,0.6,0.2,1000). The results obtained for this set of parameters is almost identical to the results obtained for the second set of parameters. The best solutions are still found by the GA5 at the expense of a very high simulation time, compared to the other models.

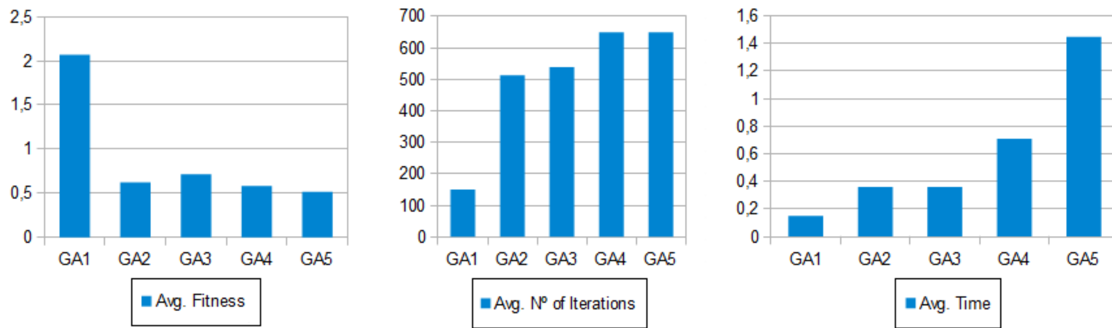

Figure 36: Graphical representation of the performance of the GA models with the 3rd set of parameters (source: self made)

For the fourth set of parameters, (100,0.8,0.4,1000), the test results show that the GA4 outperforms the GA5 model even when the simulation time is shorter. In fact, the GA2 and GA3 also obtain better solutions than the GA5 model. This is due to the fact that the Mutation Rate has been increased and thus the elitist replacement does not work as well as it does for lower rates of mutation.

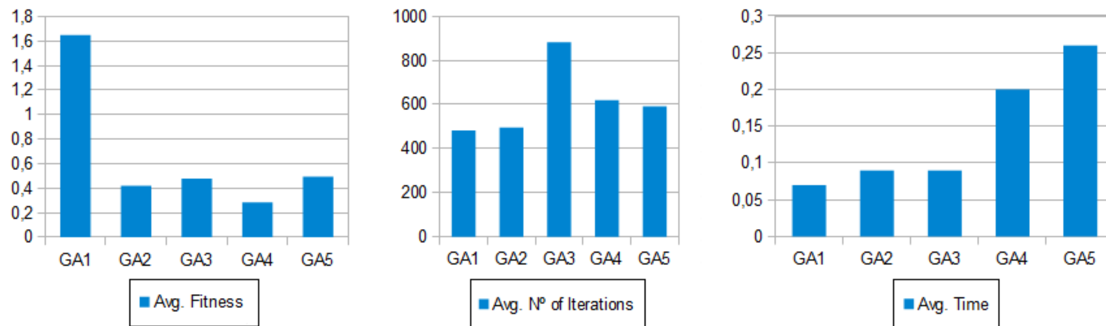

Figure 37: Graphical representation of the performance of the GA models with the 4th set of parameters (source: self made)

The fifth and last set of parameters corresponds to (100,0.8,0.2,10000). The number of generations has been increased tenfold. Here again the GA4 outperforms the other models even though the GA5 has a slightly longer simulation time. The results obtained through the testing of the models vary very little from one set of parameters to another, as can be concluded from the figures in this section.

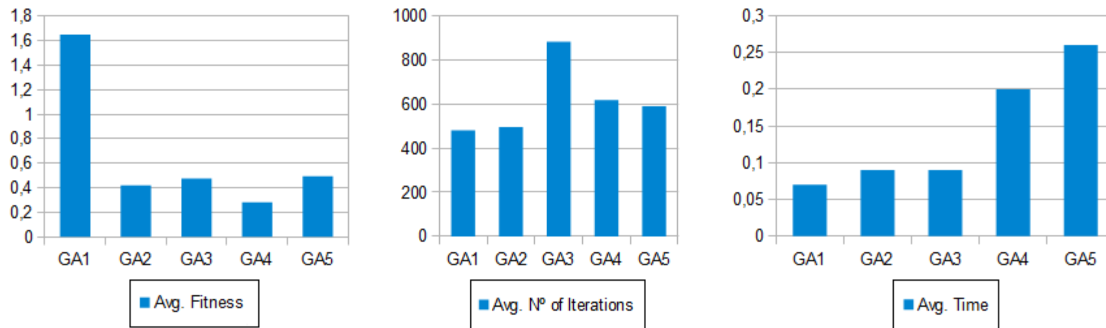

Figure 38: Graphical representation of the performance of the GA models with the 5th set of parameters (source: self made)

In conclusion, the GA4 model obtains the best results with the same set of parameters. This is accompanied by a moderate increase in the simulation time and therefore this model may not be the most efficient. An additional simulation has been performed to check if the results obtained with the GA2 model, which are already acceptable, are better than the ones obtained with the GA4 model for the same simulation time. This can be done by increasing the number of generations run with the GA2 model.

The simulations reveal that the GA2 model is more efficient than the GA4 model since it does find a better solutions in the same time, therefore the GA2 model will be henceforth used in the future testing and simulations. Greedy and Elitist seem to be a big handicap to the exploration facet although they increase the exploitation. Together with the complexity and computational cost of the added code these strategies need a finer tuning to be able to outperform a more stochastic approach. There is also the possibility that such strategies work better when facing more complex problems, in other words, problems with a high number of variables or highly constrained.

Once the best GA model has been found, the GA parameters are tested more thoroughly using such model (GA2).

In the first place, the Initial Population size is tested. This is done by representing the average solution f(x) obtained for different Initial Population values. The results show that the improvement on the solution obtained is not significant at all above a size of 2000 or, at least, the improvement is very small with big differences in the Initial Population. Therefore, although the best results are obtained for the highest Initial Population, this may be inefficient due to a very high computational cost of generating the initial population. It can be noted that from a population of around 100 onwards, the solutions obtained are quite similar. If the computational cost is an issue perhaps a size of 100 for the Initial Population would work better. On the other hand, if computational cost is not an issue using a very large size could be beneficial.

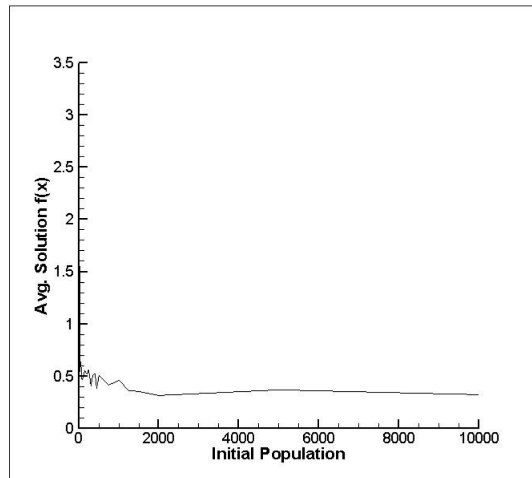

Figure 39: Graphical representation of avg. solution f(x) with initial population (source: self made)

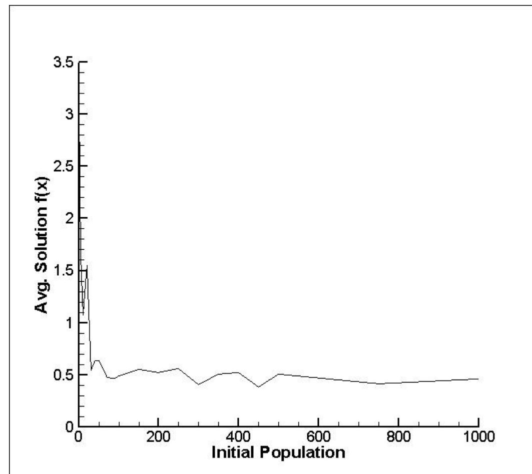

Figure 40: Graphical representation of avg. solution x with initial population (source: self made)

Together with the average solution f(x) obtained, the average number of iterations until the solution is found is also studied. This is indeed another indicator that allows to understand how the method behaves. The figures below show that for Initial Population sizes under 2000 the average number of iterations until the solution is found is around 600. For sizes above 2000, the number of iterations seems to decrease the larger the Initial Population is. It must be taken into account that these simulations have been made for a number of generations equal to 1000, which certainly affects the solutions obtained since this number is in fact the maximum number of iterations run.

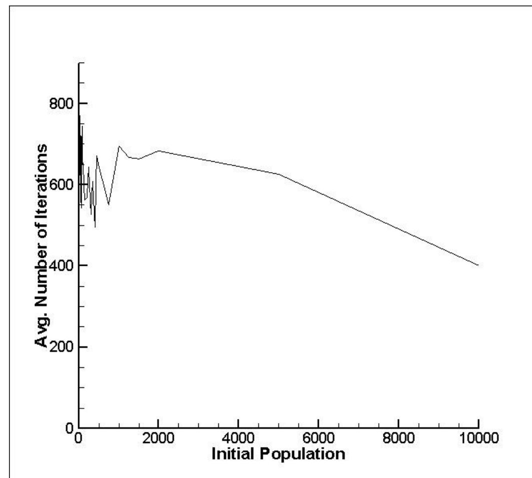

Figure 41: Graphical representation of avg. number of iterations until best solution is found with initial population (source: self made)

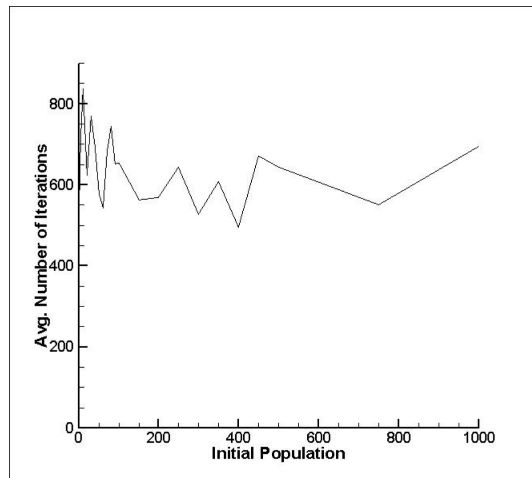

Figure 42: Zoomed graphical representation of avg. number of iterations until best solution is found with initial population (source: self made)

The behaviour of the GA with different values of the Crossover Rate is studied next. The testing shows that there are not meaningful differences in the solutions obtained although these do exist. The average solution f(x) improves with higher values of the Crossover Rate up to around the value 0.8, for higher rates the solutions obtained seem to be less precise. Along the lines of the state of the art of

GA, here as well the best value for the Crossocer Rate seems to be 0.8. This way the best chromosomes are sometimes passed down to the next generation unmodified but most of the time the crossover does take place as expected.

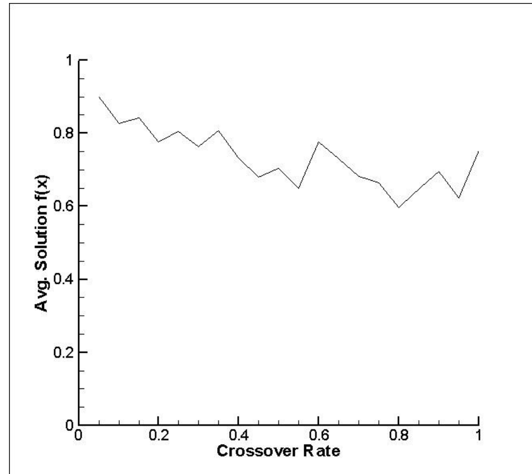

Figure 43: Graphical representation of avg. solution f(x) with crossover rate (source: self made)

The testing performed on the average number of iterations until the best solution is found does not seem to behave in a certain way according to the Crossover Rate used. As can be seen from the figures, the number of iterations do not seem to follow a particular pattern or logic. One thing that may be pointed out is that for a Crossover Rate of 1 the number of iterations until the best solution is found is one of the highest obtained probably due to the fact that, although the exploration is good, the exploitation is not as good as for lower rates.

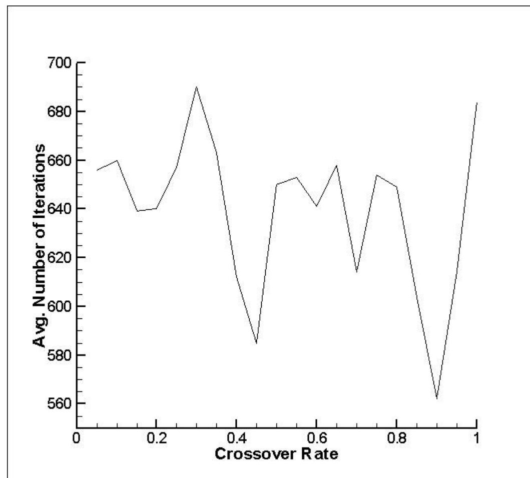

Figure 44: Graphical representation of avg. number of iterations until best solution is found with crossover rate (source: self made)

The Mutation Rate is tested next. The results show that the average solution f(x) has an optimum for a value of the Mutation Rate equal to around 0.4. It appears that this value ensures a good exploration of all the search space but does not turn the GA into an excessively stochastic method. Bear in mind that the results obtained for every parameter depend on the operators used for the method.

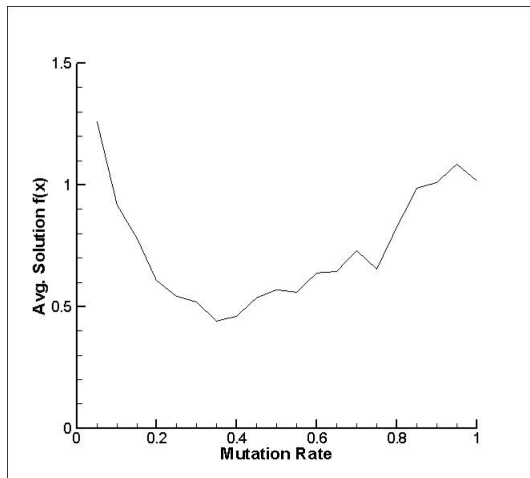

Figure 45: Graphical representation of avg. solution f(x) with mutation rate (source: self made)

The average number of iterations until the best solution is found does seem to be influenced by the Mutation Rate, unlike with Crossover Rate. Here, the number of iterations is the highest for the value for which the best results are obtained, a Mutation Rate of 0.4. This behaviour could be due to the fact that in this scenario the exploration is the highest without resulting in an excessive randomness, therefore the algorithm keeps improving the solution obtained for more iterations than with other values of the parameter studied.

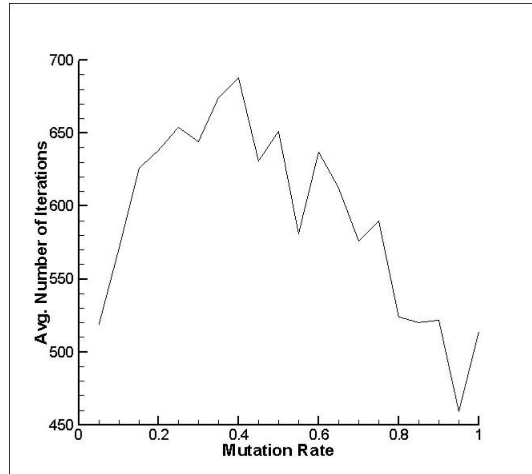

Figure 46: Graphical representation of avg. number of iterations until best solution is found with mutation rate (source: self made)

Last but not least, the behaviour of the GA with the number of generations evolved is tested. The simulations show, as expected, that the longer the method is run the better the solutions obtained. Since the actual algorithm does succeed in evolving the population to better adapt the environment, or the problem statement in the case under study, the solutions improve generally with every new generation. This is true until the population comes to a point where all the members are almost identical and therefore the crossover no longer serves its purpose of exploring the search space. This is the main reason for which a relatively high Mutation Rate has been chosen, since it prevents the population from being copies of the same chromosome by introducing random genes.

It should also be noted that the average solution f(x) draws a logarithmically decreasing function with the number of generations. Thus, if the number of generations is small a light variation of this number results in a great improvement in the solution obtained. On the other hand, if the number of generations is high a huge variation in the number of generations is needed to improve significantly the solution obtained.

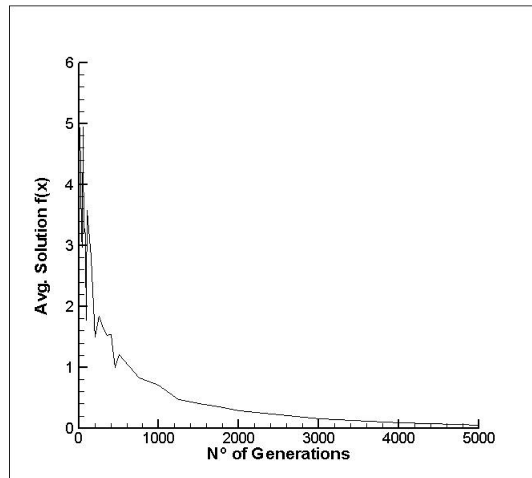

Figure 47: Graphical representation of avg. solution f(x) with number of generations (source: self made)

Finally, as is shown in the figure below these lines, the average number of iterations until the best solution is found increases linearly with the number of generations. Again, this is due to the fact that the algorithm continually improves the population throughout the simulation and therefore the best solution is usually found in the last generations. This implies that the algorithm can be always be run for more generations if a better solution is required. This is undoubtedly the GA's forte, the method appears to improve the present solution to a given problem indefinitely. The robustness of this method is unmatched, at least by the methods studied in this dissertation.

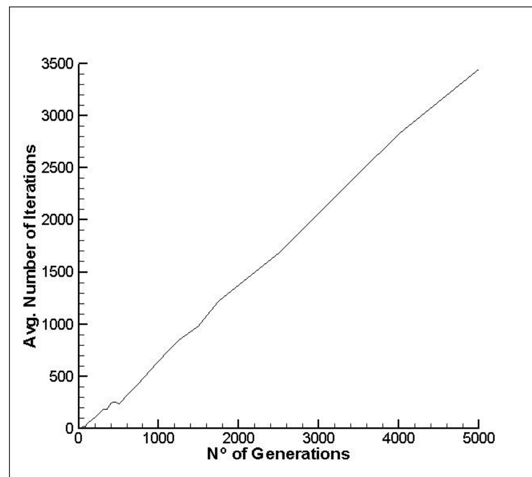

Figure 48: Graphical representation of avg. number of iterations until best solution is found with number of generations (source: self made)

7.5 Simulated Annealing

In the SA implementation the number of operators and parameters is considerably lower than in the GA. The design pattern used is the same as in the mentioned method. Here as well, the implementation of some of the operators is different for continuous and combinatorial optimization problems.

Like in the GA again, this method can be used in constrained optimization by means of Penalty Methods or by implementing specific operators for this type of problems.

Following the dissertation's structure, the testing done regarding the optimal parameters for the SA is also presented in this section. Although the SA uses different variations of the same operators according to the type of problem, the algorithm of this method is quite straightforward and does is not as prone as the GA to be implemented in multiple ways. Therefore, a fixed set of operators has been chosen for implementing the method and the parameters have been tested to determine the best values of these.

7.5.1 Implementation

The implementation of the SA is identical to that of the GA in terms of the programming language used and the software design pattern. C++ is the object oriented programming language chosen and the design pattern is again a combination of the Strategy Pattern and the Template Method. In the same way as for all metaheuristics, given their stochastic nature, the simulations must be run a number of times and the results averaged.

The SA starts off by generating a random initial approximation and then, every iteration, it picks a neighbouring point to the current approximation and decides probabilistically if the current approximation is changed to the neighbouring point. Every iteration the temperature decreases. Thus, as the algorithm progresses the probability with which the current approximation will jump to a neighbouring point with less energy decrease with the temperature.

The algorithm is presented in terms of the operators, or functions, involved. The implementation is presented introducing separately each operator in the same way as for the GA.

The encoding used is again strings of numbers of the variable type double. In the Travelling Salesman Problem these numbers correspond to the cities coordenates, for example. The temperature is of utmost importance in this method and must be stored in such way that all the functions can access it. No lists are necessary in the implementation of this method since there is only one approximation at any given time.

Initial Approximation

This operator generates a random initial approximation in the search space. The implementation of this operator is the exact same as the Initial Population operator in the GA with the sole exception that here only one population member is required, which will be the initial approximation. Since the implementation is analogous the pseudocode is not rewritten here, check the Initial Population implementation where the code is included.

Pick Neighbour State

The Pick Neighbour State operator selects a neighbour point to the current approximation. This point is picked randomly from all the neighbouring points. The pseudocode for this operator is uncomplicated and can be described as:

```
for every variable in the current state

  generate random variable between zero and the Search Diameter

  set neighbour variable to the addition of the current state variable plus the random number generated minus half the Search
```

In the implementation of the Pick Neighbour State operator for this dissertation, the neighbourhood of a point has been described as the hypercube centered on the given point and with a side equal to the Search Diameter parameter. In other words, the difference between a point and a neighbour is at most equal to half the Search Diameter for each variable. This is due to the way the algorithm has been implemented.

Acceptance Probability

The Acceptance Probability operator decides whether the current state should be changed to the neighbour state or not. The decision is made probabilistically according the the temperature in the current iteration. As explained earlier in this dissertation, the acceptance probability function used in this study is the one proposed by Kirkpatrick et al..

The pseudocode for this operator is very straightforward. In the case of a minimization problem, the code is structured in the following manner given the current state's energy (e), the neighbour state's energy (e') and the temperature (T):

`if` $e < e'$

  `change to neighbour`

`if` $e > e'$

  `generate random number between zero and one`

  `if random number generated` $< exp((e - e')/T)$

    `change to neighbour`

Cooling Schedule

The Cooling Schedule used in the implementation of the SA for this study consists in decreasing the temperature by one every iteration. There is no need to include pseudocode for this operator since it is trivial. At any point in the algorithm's main loop, the variable storing the current temperature must be decreased one degree. It is usually advised to implement this operator together with the Termination Criteria since both modify global variables once per iteration, namely temperature and iteration counter.

Termination Criteria

The Termination Criteria operator's implementation in the SA is the exact same as the one used for the GA. When the desired number of iterations has been reached, this operator exits the algorithm's main loop.

7.5.2 Testing

Since the SA is usually implemented in a specific manner, not like GA that can be implemented in many ways, there is only one model of the method. The parameters of the method are the Search Diameter, Initial Temperature and Number of Iterations. In the same way as the GA, a number of simulations need to be run in order to average the results obtaining representative values. The number of simulation runs is again 1000.

The interaction between the different parameters is neglected, therefore when studying the Search Diameter, for example, the Initial Temperature and the Number of Iterations are fixed to certain values. The same is done with the rest of the parameters. The default, or fixed, values for the parameters are $Search\,Diameter = 15\%$, $Initial\,Temperature = 500$ and $Number\,of\,Iterations = 1000$.

The problem used in the testing of the SA parameters is Rastrigin's Function in one dimension.

The results obtained from testing the SA performance for different Search Diameters reveal that, for the problem under study, while the solution is better for small diameters there is a point at which the diameter is too small to allow a complete exploration of the search space and therefore the global optimum is not found. In the case tested here, it is for diameters smaller than the 10% of the search space that the global optimum is not found. The simulations show that the best value for the Search Diameter is the smallest that still allows the algorithm to explore thoroughly the search space.

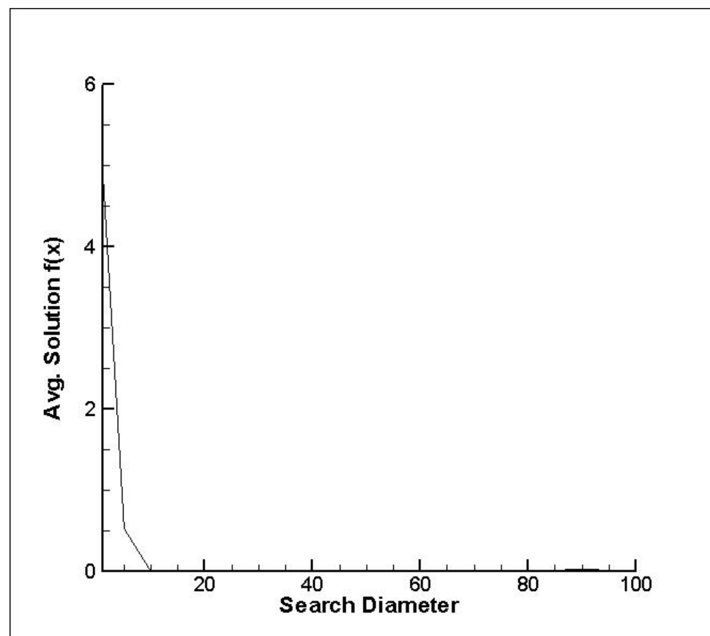

Figure 49: Graphical representation of avg. solution f(x) with search diameter (source: self made)

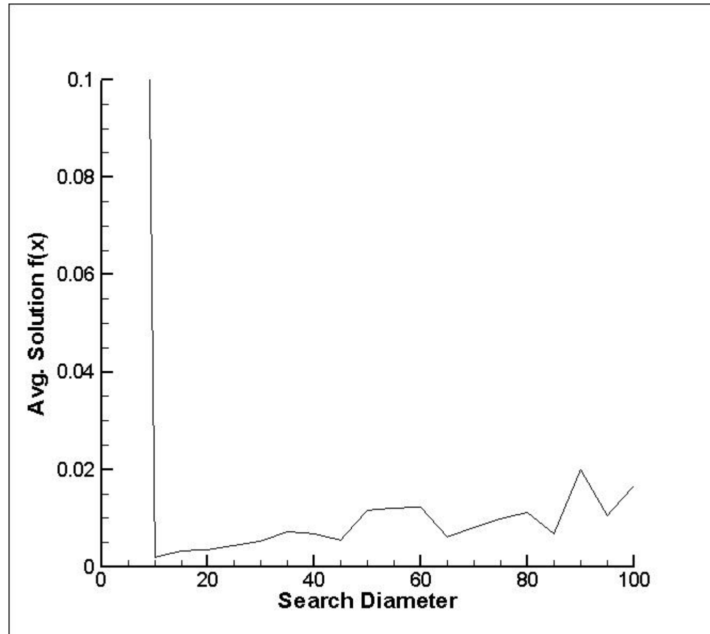

Figure 50: Zoomed graphical representation of avg. solution f(x) with search diameter (source: self made)

The average solution x obtained in the tests carried out behaves in the same way as the average solution f(x) above. Best value for the Search Diameter appears to be around 15% again.

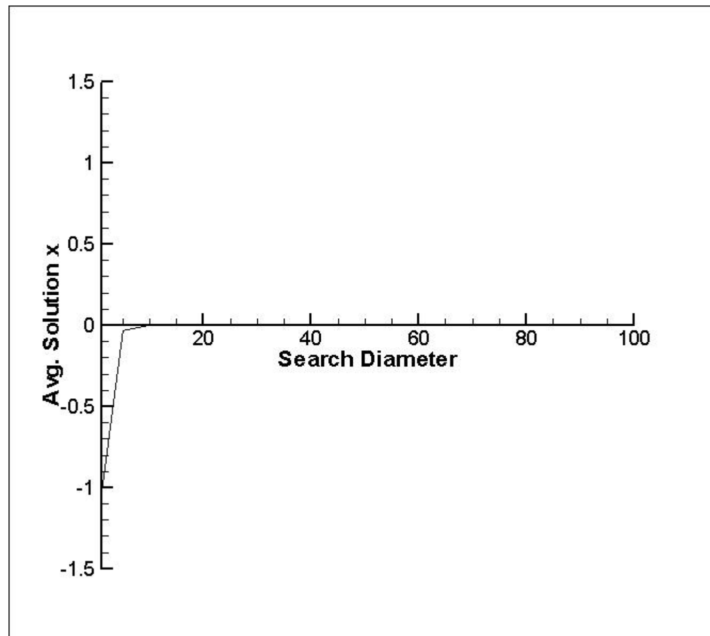

Figure 51: Graphical representation of avg. solution x with search diameter (source: self made)

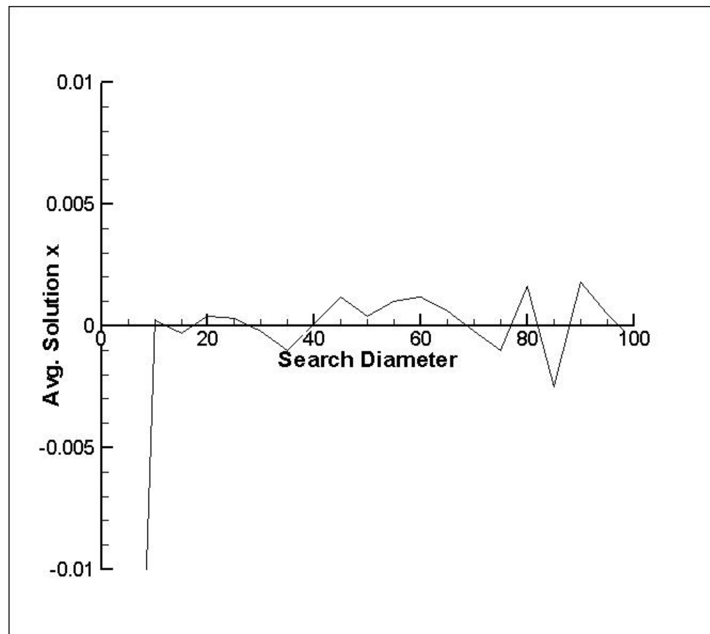

Figure 52: Zoomed graphical representation of avg. solution x with search diameter (source: self made)

Although the algorithm is run for a certain number of iterations, the best solution found does not coincide with the solution of the last iteration. While the SA has been run for 1000 iterations, the best solution is found most of the time around the iteration 500. It shall be noted that for a Search Diameter smaller than 10% the best solution is found earlier but this is due to the fact that the exploration is very poor for those values of the studied parameter.

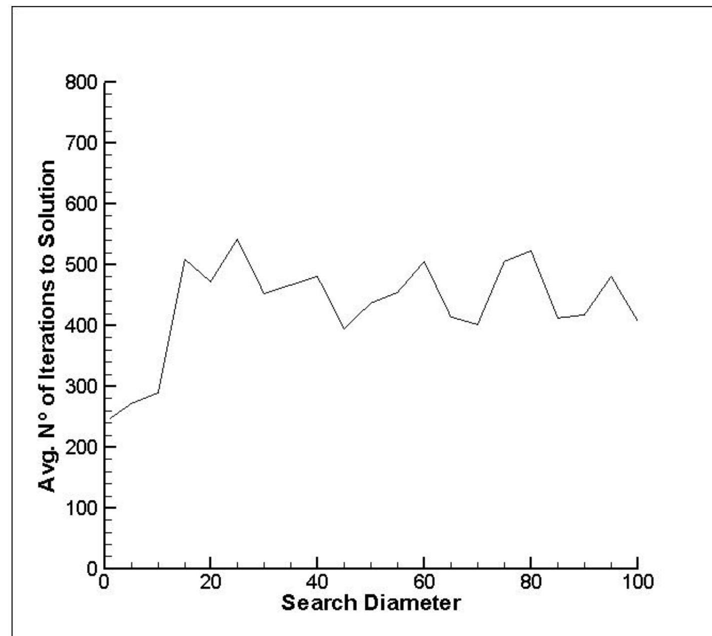

Figure 53: Graphical representation of number of iterations until best solution is found with search diameter (source: self made)

Below, the results of testing the Initial Temperature parameter are presented graphically. The best value for the Initial Temperature depends greatly on the number of iterations run. The figures show that if a high Initial Temperature is used then the solutions obtained are not the best. This is due to the fact that when the temperature reaches zero the algorithm concentrates solely on exploitation improving greatly the solution obtained. In the case presented, if the Initial Temperature is higher than 500 then, since 1000 iterations have been performed for the simulation, the algorithm's exploitation is not great. On the other hand, if the Initial Temperature is set to a value lower than 500 then the algorithm has at least another 500 iterations with temperature equal to zero where it can exploit the solution obtained.

In general terms, the best value for the Initial Temperature parameter corresponds usually to around half the number of iterations run. Thus the method is balanced between exploration and exploitation.

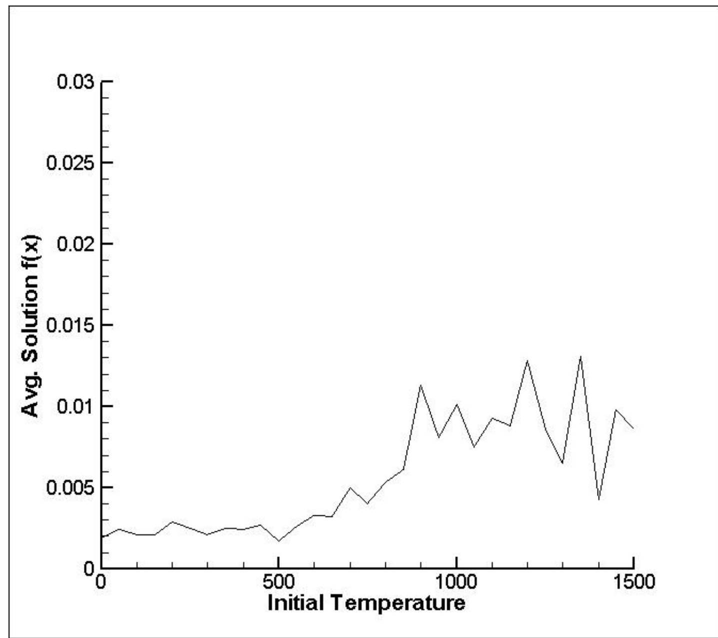

Figure 54: Graphical representation of avg. solution f(x) with initial temperature (source: self made)

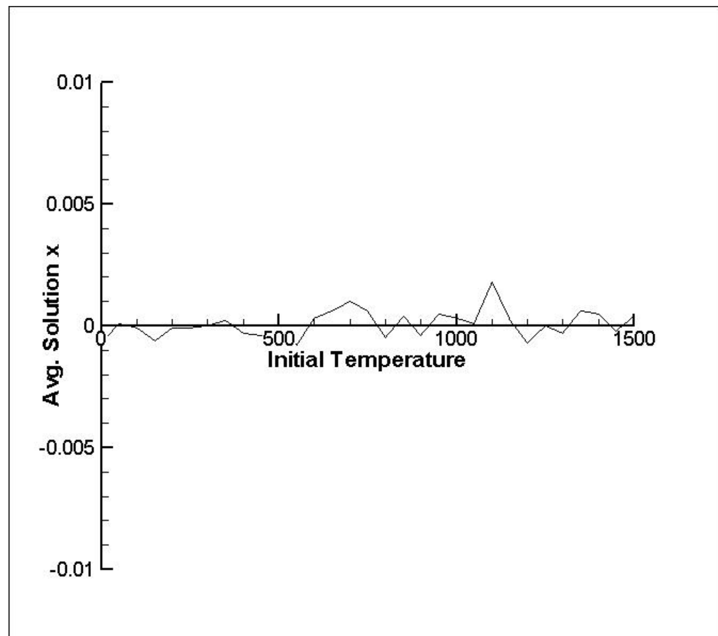

Figure 55: Graphical representation of avg. solution x with initial temperature (source: self made)

Here again, even though the simulation is run for 1000 iterations, the best solution is usually found around the 500 iteration. It can also be concluded that in almost every scenario the best solution is found after a relatively low number of iterations at a temperature equal to zero.

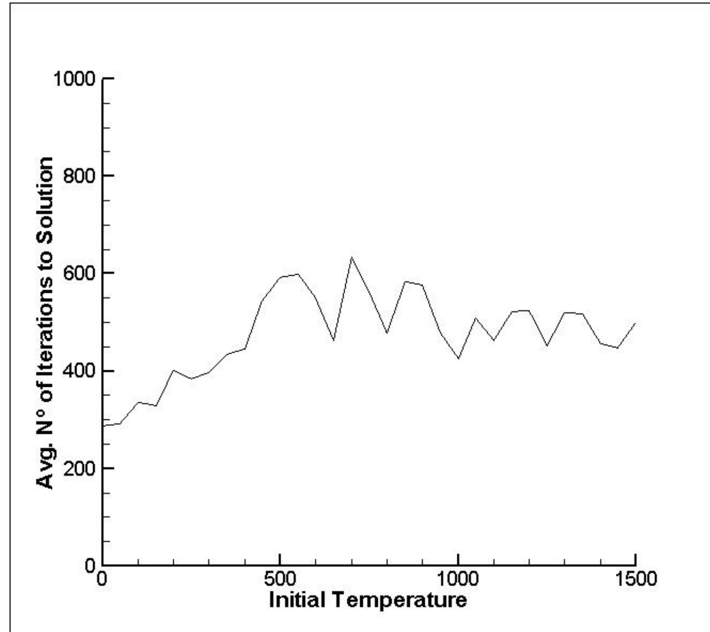

Figure 56: Graphical representation of avg. number of iterations until best solution is found with initial temperature (source: self made)

As expected, the higher the number of iterations simulated the better the solution obtained. However this is only true up to about 1000 iterations, for a higher number of iterations no significant improvement in the solution is noted. This is due to the Initial Temperature, which has been set to 500. As mentioned before, the SA seems to work best for a Initial Temperature of half the number of iterations run.

Although the above partially explains the testing results, there is another limitation intrinsic of the implementation used that also must be taken into account. The figure below shows how the solution stabilizes around 0.002 and does not improve anymore with the number of iterations. The first explanation for this, the one above, states that the Initial Temperature should increase with the number of iterations run in order to improve the solution obtained. The second explanation is related to the implementation of the method. The SA has been programmed to generate random numbers with a precision of 0.001, therefore when searching for a neighbour point to jump to, the randomly generated neighbour only has a precision of 0.001. This behaviour can be corrected by modifying the implementation.

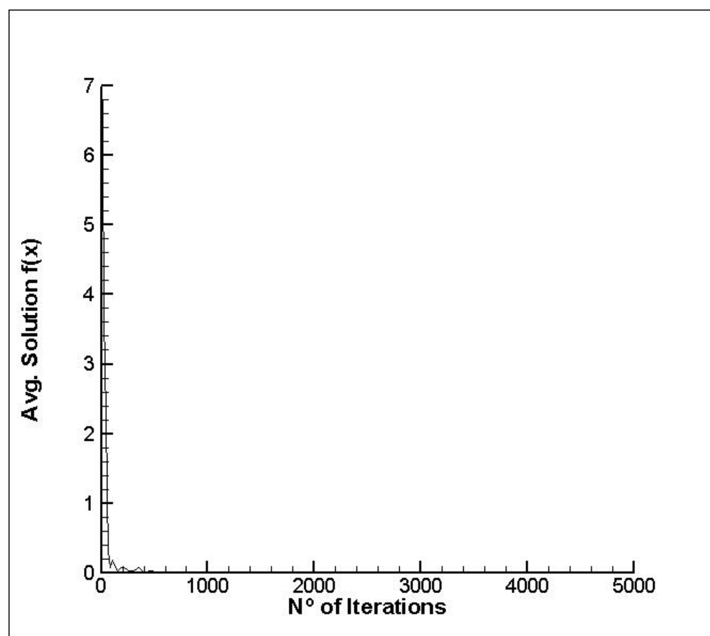

Figure 57: Graphical representation of avg. solution f(x) with number of iterations (source: self made)

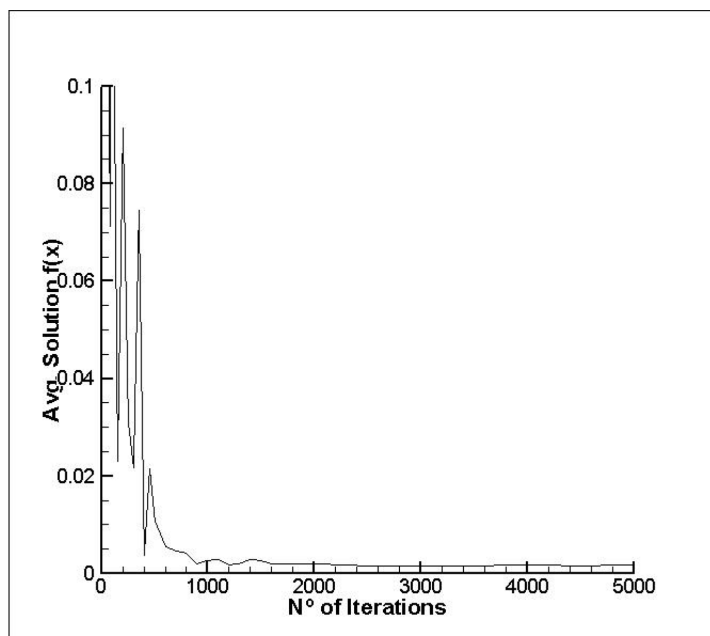

Figure 58: Zoomd graphical representation of avg. solution f(x) with number of iterations (source: self made)

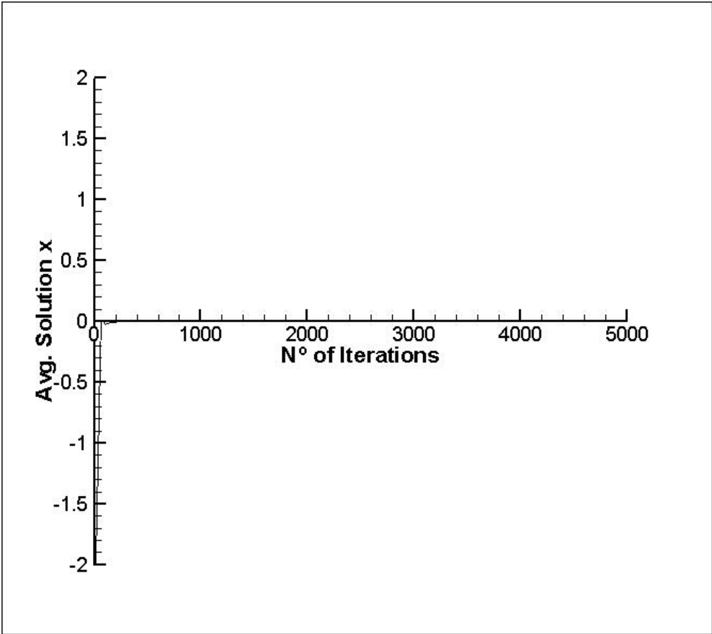

Figure 59: Graphical representation of avg. solution x with number of iterations (source: self made)

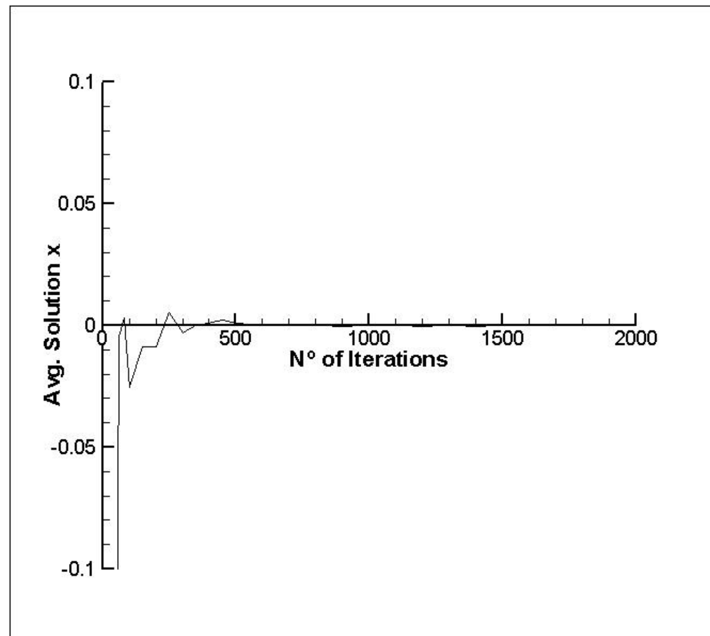

Figure 60: Zoomed graphical representation of avg. solution x with number of iterations (source: self made)

The figure below these lines presents the number of iterations until the best solution is found. The test shows that the number of iterations until best solution increases with the number of iterations run. This implies that running more iterations does improve the solution obtained but, according to the results of the precedent tests, the improvement on the solution is very small after the point in which the temperature reaches zero.

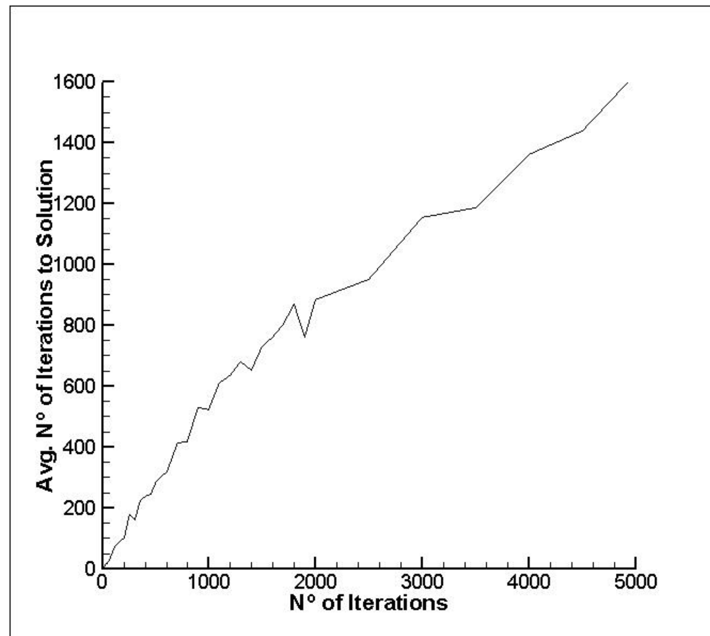

Figure 61: Graphical representation of avg. number of iterations until best solution is found with number of iterations (source: self made)

8 Simulation Results and Discussion

The optimization methods presented in this dissertation are used to solve the different optimization problems studied. The aim of the simulations is to be able to see how metaheuristics fare against classical optimization methods and determine whether metaheuristics can be successfully applied to engineering optimization.

The most favourable values for each method's parameters have been chosen, according to the testing done previously in this study. Although the optimal value for the number of iterations is included, in order to carry out the simulations a range of values for the number of iterations is used to study the performance of each method according to the computational cost.

Parameters		
Newton's Method	Initial Approximation	*variable*
	Tolerance	0.000001
Exhaustive Search	Grid Size	*variable*
Pure Random Search	Nº of Iterations	*variable*
Genetic Algorithm	Initial Population	100
	Crossover Rate	0.8
	Mutation Rate	0.4
	Nº of Generations	*variable*
Simulated Annealing	Search Diameter	15
	Initial Temperature	$\frac{1}{2}$·Nº of iterations
	Nº of Iterations	*variable*

Table 2: Parameters used in the simulations for each method

The parameters that appear as variable are so because they are used to exert a control over the method's computational cost. The comparison of the different methods can only be done in terms of the error in the solution obtained and the computational cost. The latter is expressed through the simulation time since all the simulations have been run un the same computer and therefore the time is proportional to the computational cost. Besides, time is the limiting resource in almost every real life attempt of solving an optimization problem. In addition to the error in the solution and the computational cost, the aim of this section is also to elucidate the performance of the methods in terms of robustness.

Since the computational resources for this dissertation are limited and due to the fact that the solutions obtained using some of the methods studied must be averaged from a number of simulations, the problems proposed are relatively simple and can be usually be solved in a few seconds. Bear in mind that if a single simulation takes one second then the number of simulations run in order to average the results obtained can take a significant time lapse.

The simulations results are arranged according to the type of optimization problem solved. Since all the methods studied have been already tested, the way in which every algorithm behaves is already known and their performance can be predicted. Special attention has been put in this dissertation into explaining how the metaheuristics work and how these can be implemented and applied to engineering optimization problems. Due to this fact, the aim of the simulations carried out is to clearly state if the metaheuristics perform successfully in the problems studied. Therefore, the results are presented as briefly as possible in order to make clear the conclusions reached. For a more thorough understanding of how each method works, the section in this dissertation presenting the testing of each method should be read.

The results of the simulations performed have been represented graphically by means of B-spline functions.

8.1 Unconstrained Continuous Optimization

Unconstrained Continuous Optimization is the most general type of optimization problem and the least demanding in terms of the requisites of the solving method used. Thus all the methods studied in this dissertation can be used to solve this sort of problems.

As mentioned before, the different methods are compared in terms of the solution obtained for a certain computational cost. The computational cost in Pure Random Search and both metaheuristics, GA and SA, is directly related to the number of iterations run. Furthermore, in continuous optimization, the computational cost in Exhaustive Search can be set through the Search Diameter parameter. On the other hand, there is no way to exert a control over the computational cost in Newton's Method since this method always obtains the same solution for a given problem with a certain computational cost.

Notice that the solutions obtained with Newton's Method are not presented together with the solutions obtained by means of the rest of the algorithms studied. This is due to the fact that a direct comparison is not possible since Newton's Method gives a much more precise solution when the global minimum is found but does only find such minimum for certain initial approximations. Therefore, while the other algorithms can be run for a longer duration in order to improve the solution found with no limitation, Newton's Method does not work the same way and will only find the global minimum in certain situations.

8.1.1 De Jong's Function

The 2D version of De Jong's function is used in the simulations. As mentioned in the corresponding section of this dissertation, the usual search space for this optimization problem is usually restricted to the hypercube $-5.12 \leq x_i \leq 5.12$, $i = 1, ..., n$. De Jong's function is a very simple optimization problem which consists only of one global and local minimum located at the origin.

In order to compare the performance of the studied optimization methods in finding the global minimum of De Jong's function, a number of simulations have been run to be able to analyze the solutions obtained according to the computational cost for each method. The results of the simulations have been used to represent graphically the behaviour of each method in a way that a direct comparison can be made.

The results are presented in the figure below these lines.

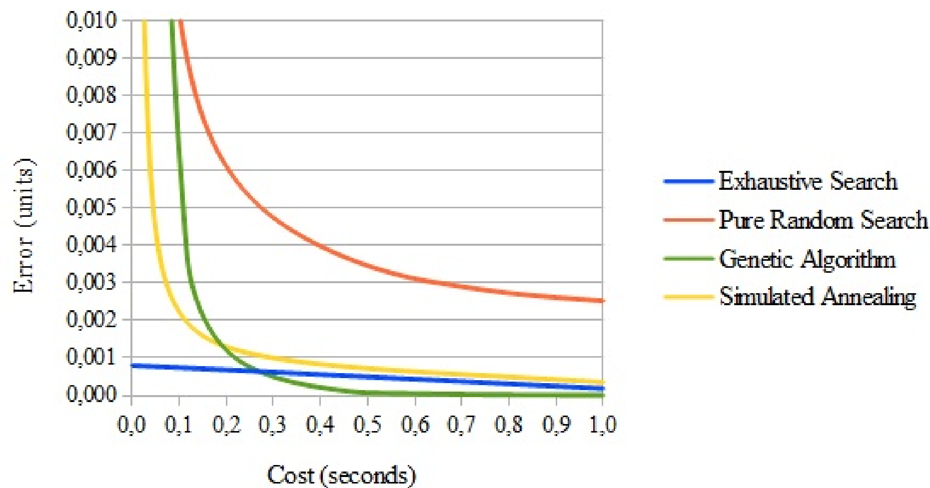

Figure 62: Graphical representation of De Jong's function simulations results (source: self made)

The first observation to be made from the simulations performed is that Exhaustive Search improves the solution obtained linearly with the computational cost while Pure Random Search, Genetic Algorithm and Simulated Annealing improve the solution drawing a logarithmically decreasing function with the computational cost. This is expected given the way in which the Grid Size parameter works in Exhaustive Search applied to continuous optimization. The smaller the grid size, the larger the number of points in the search space thus the error decreases and the computational cost increases linearly with the number of points.

For very low simulation times the Exhaustive Search gives the best solutions, the reasoning behind this is that not enough iterations have been run of the other methods in order to effectively explore the search space and exploit the solutions found. When enough time is given to the metaheuristic algorithms studied, these can successfully find better solutions than the Exhaustive Search. In a problem as simple as the one being solved here, enough time seems to be around 0.2 to 0.3 seconds for the GA and slightly over 1 second for the SA. For more complex optimization problems the simulation time required is obviously higher but the behaviour of the methods is qualitatively the same.

Amongst the optimization methods studied, Pure Random Search has the worst performance. It behaves in a similar way to GA and SA, although these metaheuristics have a guided search that allows them to perform way better than their more stochastic counterpart. The results of the simulations show that even an exhaustive search of the whole space is better than the pure random approach.

The Genetic Algorithm obtains the best solutions for the first optimization problem studied. The performance of the Simulated Annealing is better for low computational costs but falls off for longer simulation times with respect to GA. Since De Jong's function has only one minimum and the curvature is of the same sign in any direction, the exploitation capabilities of a method are more valuable in this

problem than the exploration facet. The simulations indicate, until further comprobations are made, that GA probably has better exploitation capabilities than SA.

Given De Jong's function curvature and the fact that only one minimum exists, Newton's Method always finds the best solution with a smaller error and less computational cost than the methods above.

8.1.2 Rastrigin's Function

Here again the 2D version of the function is used since it allows a visual interpretation of the results of the simulations. The search space is the same as in De Jong's function, namely the hypercube $-5.12 \leq x_i \leq 5.12$, $i = 1, ..., n$. Unlike De Jong's function, this optimization problem has an infinite number of local minima and therefore serves the purpose of testing the robustness of each method together with their exploration capabilities. Rastrigin's function is symmetrical and therefore makes the exploration that much simpler. The global minima is located at the origin.

In the same way as for De Jong's function simulations, the results obtained have been represented graphically in the figure below.

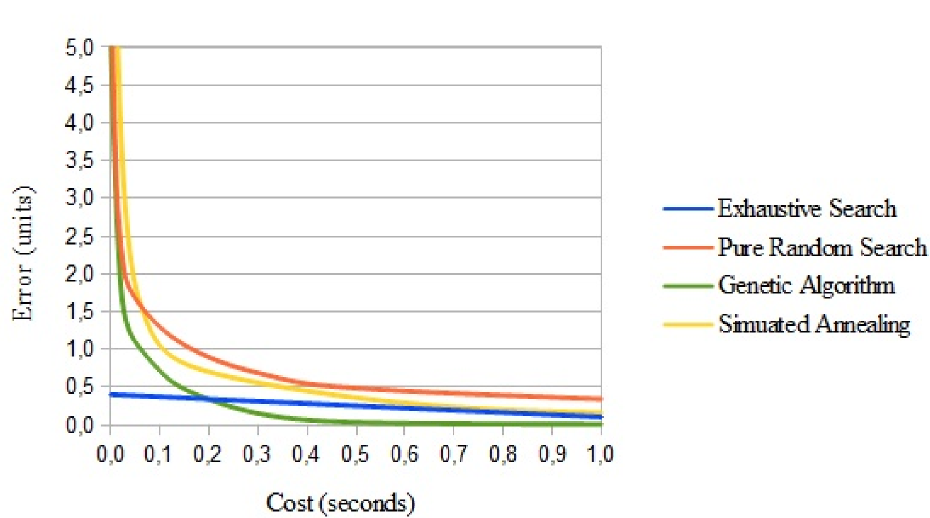

Figure 63: Graphical representation of Rastrigin's function simulations results (source: self made)

In general terms, the results obtained are pretty much the same as for De Jong's function optimization problem. Rastrigin's function consists in an infinite number of local minima as explained before, thus an adequate exploration of the search space is essential to be able to find the global minimum. Due to this fact, the Pure Random Search algorithm performs far better than before compared to

the metaheuristics since the stochastic component ensures a good exploration of the search space. Nonetheless, the metaheuristics still outperform the pure random approach as expected.

The Exhaustive Search, in the same way as for De Jong's function, obtains better solutions than the other methods for very low simulation times. Its improvement of the solution obtained is also presented here as linear with the increase in the computational cost. Despite this fact, it must be mentioned that the Exhaustive Search does not strictly improve the solution with the computational cost given that the vertices of the search grid may be closer to the solution even if the grid size is bigger. These irregularities have been neglected and the improvement has been considered linear.

The Genetic Algorithm gives better results than the Simulated Annealing for Rastrigin's function too. Therefore, apparently the GA's exploration capabilities in unconstrained continuous optimization are also better than the SA's.

When facing an optimization problem with several local minima, such as the Rastrigin's function, Newton's Method does no longer find the global minimum in any circumstances. Only for certain initial approximations does Newton's Method find the global minimum. Since these initial approximations are not known beforehand, said method is not reliable anymore and fails in finding the solution to the optimization problem in most cases. This proves that the lack of robustness is Newton's Method biggest drawback.

8.1.3 Six-Hump Camel Back Function

The Six-Hump Camel Back function only exists in its 2D version. It is, together with the above, one of the usual benchmarks used for testing optimization algorithms. This function is asymmetric and has a finite number of local minima, six to be more precise, which makes it harder to search for the global minima. The function has two global minima located at $(x_1, x_2) = (-0.0898, 0.7126)$ and $(0.0898, -0.7126)$.

The results obtained through the simulations are presented in the following figure.

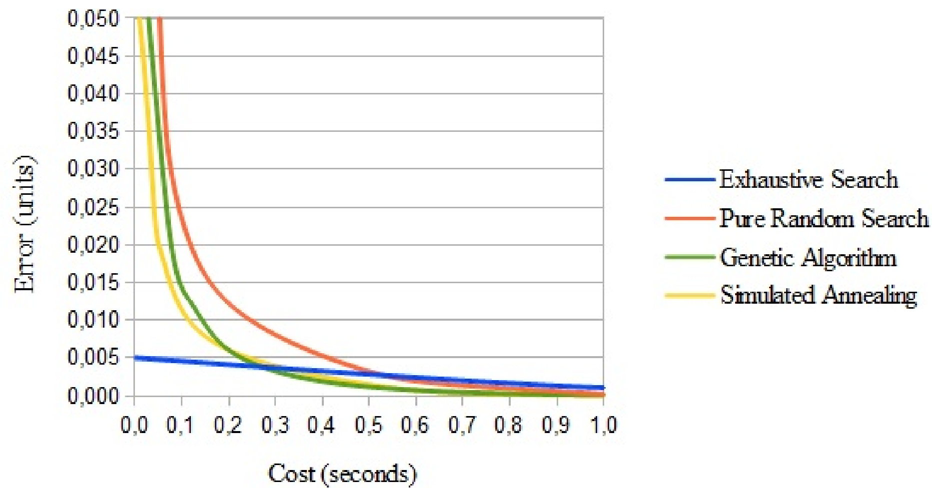

Figure 64: Graphical representation of Six-Hump Camel Back function simulations results (source: self made)

The simulations carried out for the last unconstrained continuous optimization problem studied confirm the conclusions drawn from the previous simulations. The performance of the different methods is almost the exact same as the observed for De Jong's and Rastrigin's functions. Again, the Exhaustive Search obtains the best solutions for very low computational costs. Although, in solving this problem, Exhaustive Search is outperformed even by Pure Random Search for simulation times over 1 second.

The metaheuristic algorithms, GA and SA, give the best results once more. Nonetheless, this time around both methods obtain very similar solutions with no noticeable difference in the precision of the solution for simulation times over 0.6 seconds. The SA gives slightly better results than GA for lower computational costs and slightly worse as the simulation time increases. The similarity in the solutions obtained with both metaheuristics could be due to the fact that two global minima exist and thus it is easier to find the solution to the problem.

Given the fact that there are only six local minima in this function belonging to a common pit and that two global minima exist, the results obtained are almost as good as the ones for De Jong's function. This is also the reason why Pure Random Search gives such good results, together with the fact that the search space is smaller than for the other two functions studied and therefore a random search of the space is more likely to find the global optimum.

8.2 Constrained Continuous Optimization

In Constrained Continuous Optimization only the points in the continuous search space that verify the constraints are feasible solutions to the problem. Therefore Newton's Method can not be applied

naturally to these type or problems. With the sole exception of the aforementioned method, the rest of the algorithms studied are used to solve the selected benchmark problem.

Exhaustive Search and Pure Random Search check if the constraints are verified for every solution evaluated with the problem's objective function and if the solution does not verify the constraints then it is not stored as the best solution. Remember that a discretization of the search space is carried out in the Exhaustive Search.

The metaheuristics, GA and SA, use the Death Penalty approach to check if a solution is feasible. Death Penalty consists in checking if the constraints are verified every time a new approximation or population member is generated, if these are not verified then the solution is generated again. There are many other penalty methods but this is the simplest to implement. For the problem studied this approach works fine but for highly constrained problems Death Penalty usually entails very high computational costs and therefore must be avoided in favour of other penalty methods.

The optimization of Schmit's Structure is proposed here as a benchmark for constrained continuous optimization because it can be used to understand how the studied methods behave before this type of optimization problems and is related to engineering optimization. This problem belongs to the branch of structural optimization.

8.2.1 Schmit Structure

The structural problem posed herein, as explained before, has at least two known local minima corresponding to the isostatic and hyperstatic solutions of the structure. Both solutions are presented again in the table below these lines. Since this is a constrained optimization problem, the points of the actual search space are not known beforehand until the constraints are checked for any given point. This significantly complicates the exploration and exploitation of the search space.

Solutions to Schmit Structure		
Variable	x_1	x_2
Isostatic	1.066771	0
Hyperstatic	0.788675	0.408248

Table 3: Isostatic and hyperstatic solutions for schimt structure (source: self made)

The solutions in the table above are dimensionless to ease the analysis of the results obtained during the simulations.

The results of applying the methods studied to this optimization problem are the following.

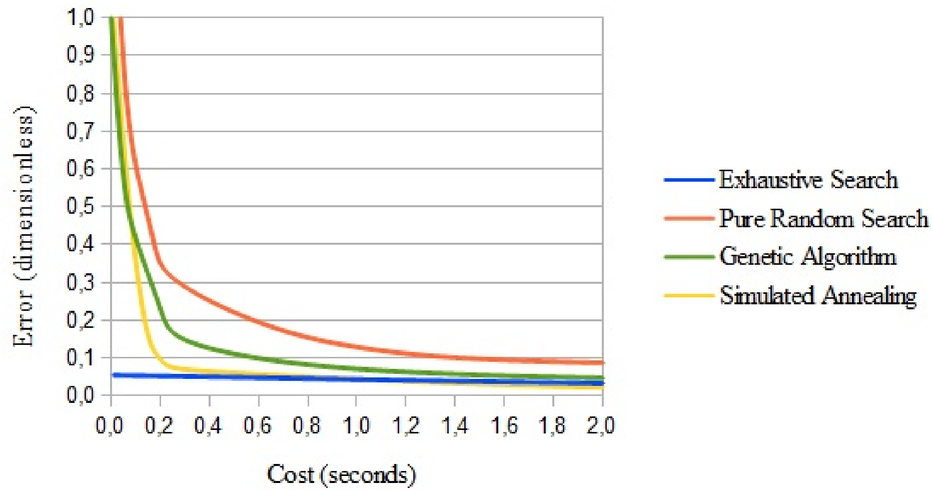

Figure 65: Graphical representation of Schmit structure simulations results (source: self made)

The first observation to be made from the simulations carried out for the structural optimization problem under study is that the Exhaustive Search gives the best results up to a relatively high simulation time. This is due to a peculiarity of Schmit Structure's problem and does not imply that the Exhaustive Search performs better than the other methods in constrained continuous optimization problems. With a Grid Size of 0.1 for the Exhaustive Search the best solution found in the search space is $(0.8, 0.4)$. Despite being a coarse grid, which implies a low computational cost, the solution is very good given the fact that the global optimum happens to be casually very close to $(0.8, 0.4)$. If, for instance, the solution had been close to $(0.75, 0.35)$ then the Exhaustive Search would no longer have the best performance.

The main difference between the structural optimization problem studied here and the unconstrained optimization problems solved above is obviously the fact that the constraints must be verified by any candidate solution. This is also the reason why, unlike in unconstrained continuous optimization, the best solutions here are obtained with SA instead of GA. Since Simulated Annealing works only with one approximation at a time, instead of a whole population like in Genetic Algorithm, the constraints must only be checked once per iteration and thus the computational cost is much lower. Therefore, for the same computational cost or simulation time the results obtained with SA are better than those obtained with GA for constrained continuous optimizaton.

As mentioned before, other ways of checking if the constraints are verified exist for the metaheuristics studied. Many of these belong to the penalty methods but others rely on algorithms that allow the metaheuristics to generate only new approximations or solutions that already verify the constraints, so these need not be checked. Such methods obtain better results than Death Penalty, due to the fact that the computational costs are lower, but are harder to implement.

8.3 Combinatorial Optimization

In the same way as in constrained optimization, all the methods studied but Newton's Method can be used to search for the optimum of a Combinatorial Optimization problem. The reason behind this is that a combinatorial optimization problem can be understood as a constrained optimization problem where the search space is discrete and the constraint is that the same value can not appear in two different variables. In other words, in combinatorial optimization solutions (a series of variables) are generated by picking values from several available (discrete search space) and no value can be picked twice in the same solution (constrained problem). Since the search space is not continuous nor differentiable, Newton's Method can not be used.

Unlike in continuous optimization, when using Exhaustive Search in combinatorial optimization there is no continuous search space to discretize. Therefore, there is no Grid Size parameter and thus there is no control over the computational cost of applying this method. This fact prevents Exhaustive Search from being compared to the other methods like done before. When facing a combinatorial optimization problem Exhaustive Search always finds the exact solution but usually does so at a very high computational cost.

The Travelling Salesman Problem has been selected as the representative of combinatorial optimization. TSP is a widely studied combinatorial optimization problem which has a variety of applications in engineering, for example planning or logistics. It has also been used in the simulations of this dissertation due to the fact that it is commonly utilized as a benchmark for computational optimization methods such as the metaheuristics studied herein.

8.3.1 Travelling Salesman Problem

In the first place, the distribution of cities must be specified. This is done by defining an euclidean space and introducing a set of points or cities between which the euclidean distance is defined. Since an emphasis has been put in this dissertation into studying real applications for the algorithms analyzed, the distribution of cities chosen does also correspond to real life scenarios.

A map of Spain has been drawn and the ten largest cities, according to the population, have been included in the study. The goal is to find the shortest closed tour through all of the cities. The distance from one city to another is considered to be described by a straight line. Therefore, the case studied corresponds to the problem of finding the shortest tour to travel through all the cities by airplane. If the tour was to be done by car, for example, then the euclidean distanced between any two cities should be substituted by the driving or road distance.

The distribution of cities studied is drawn in the figure below these lines.

Figure 66: TSP distribution of cities (source: self made)

The best tour found with the Exhaustive Search, and therefore the best tour possible, is presented in the following table together with the computational cost of finding such tour.

Shortest Tour	
Tour Length	$2287\,km$
Simulation Time	$49\,seconds$

Table 4: TSP Exhaustive Search simulation results (source: self made)

The solution obtained is represented visually in a more appropiate way in the figure below. Although the resulting tour can be easily predicted by human logic due to the way in which the cities are distributed, this does not make it easier for a computational method to find the correct solution. Thus the distribution chosen is as good as any other in testing the method's behaviour.

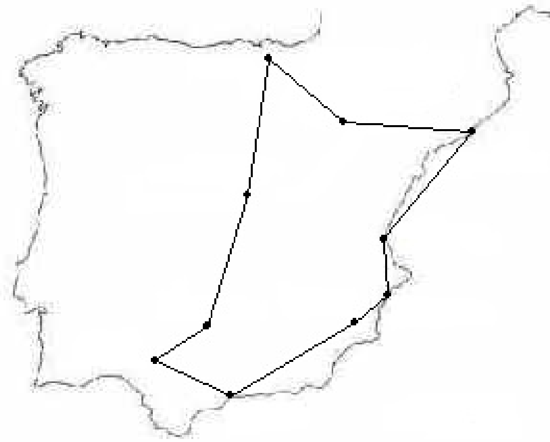

Figure 67: Graphical representation of TSP Exhaustive Search simulation results (source: self made)

As explained before, the Exhaustive Search gives the best results and thus has the best performance as long as it can be run in a practical time lapse. When the number of cities is too large, this method can not be utilised and metaheuristics become the best approach to solving combinatorial optimization problems. Exhaustive Search becomes impractical even for twenty cities.

The results of searching for the shortest tour with Pure Random Search, Genetic Algorithm and Simulated Annealing are presented graphically in order to facilitate the comparison of their performance.

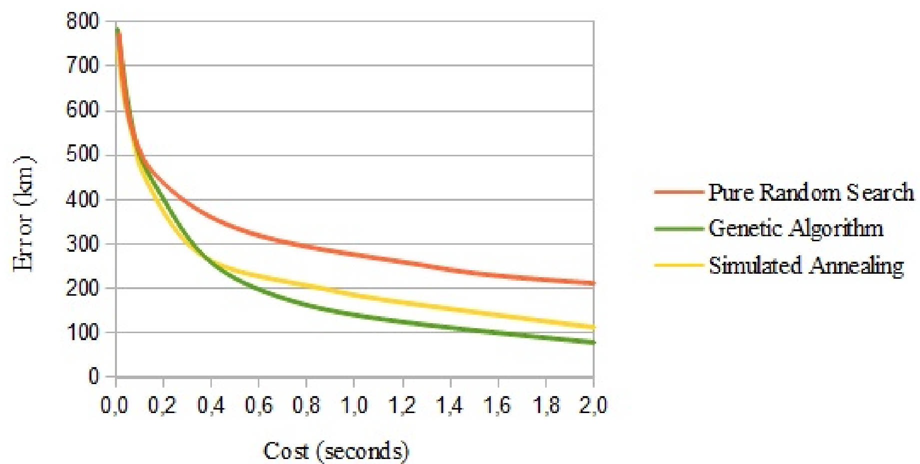

Figure 68: Graphical representation of TSP simulations results (source: self made)

The simulations show, in the same way as for the other types of optimization problems studied, that the metaheuristics GA and SA find considerably better results than Pure Random Search. Again, the reason behind this is that the search is being directed through heuristics instead of being purely stochastic.

The computational cost required to obtain successful solutions for the TSP is high due to the fact that even with ten cities the search space is huge. To be more precise, the number of possible tours is over three and a half million in the case studied. Despite the number of possible tours, both metaheuristics have an error of around 4% with a simulation time of only 2 seconds. Thus the results are quite promising and show that an acceptable solution could be found for a TSP with twenty cities in a reasonable simulation time.

Moreover, the current implementations of the GA and SA can be further improved which would also results in a better performance of these methods in front of this type of optimization problems. For instance, the way in which two parent tours are crossed to generate an offspring or a tour is mutated can be modified.

In general terms, the way in which the methods studied behave in combinatorial optimization problems is pretty much the same in which they behave before the rest of problems. In other words, here as well, the improvement in the solution found draws a logarithmically decreasing function with the computational cost.

9 Conclusions

In the final section of the dissertation, the conclusions extracted from the study are summarised. The aim set at the start of this work was to evaluate the feasibility of applying the recently devised metaheuristic algorithms to engineering optimization. Therefore, the conclusions shall clearly state if this is possible while explaining the benefits, if any, of using these methods over more classical approaches.

The field of engineering optimization is broad and has applications in many areas such as structural design optimization or logistics, both studied herein. Most of these optimization problems can be put as one of the three types of problems included in this study: unconstrained continuous optimization, constrained continuous optimization and combinatorial optimization.

Several classical algorithms have been compared to the metaheuristics, Genetic Algorithm and Simulated Annealing, when applied to the benchmark optimization problems. Many other optimization methods exist that have not been studied in this dissertation, nonetheless this does not prevent from making a judgement about whether the metaheuristics can be successfully applied to the problems under study.

The overall results of the simulations show that not only metaheuristics can be used to solve engineering optimization problems but, in fact, these usually outperform classical methods when doing so. Bear in mind that the simulations have been carried out at a relatively low computational cost for practical reasons, for higher computational costs the performance of the metaheuristics studied is expected

to be significantly better. At any rate, the conclusions reached are always qualitative rather than quantitative.

Calculus based methods such as Newton's Method usually fail in finding the global optimum of a optimization problem because they have no robustness and tend to fall into local optima and can not escape. Newton's Method has a very effective exploitation since it does find the optimum very fast and with virtually no error when in the vecinity of such optimum. However, the exploration of the search space in this method is almost non-existant. It can also be used only in continuous optimization with a differentiable objective function which is a strong limitation.

Despite the above, it is possible to take advantage of Newton's Method exploitation capabilities when used together with another optimization algorithm. For instance, in unconstrained continuous optimization, a metaheuristic method can be used to search for the pit where the global optimum lies and then Newton's Method can be applied using as initial approximation the solution obtained from the metaheuristic. This approach benefits from the exploration of the metaheuristic and the exploitation of Newton's Method.

Brute force methods, such as Exhaustive Search, and pure stochastic algorithms, such as Pure Random Search, have the disadvantage that the search follows no logical steps or, in other words, is not directed. Any directed search method, whether calculus based or based on heuristics, is expected to perform better than the aforementioned algorithms. If there is no limitation at all in the computational cost then Exhaustive Search can be deemed appropiate since it finds the exact solution. Nonetheless, most of the time for complex optimization problems the computational cost of running Exhaustive Search is prohibitive. Pure Random Search is always outperformed by other stochastic algorithms with a directed search, such as the studied GA and SA.

Genetic Algorithm and Simulated Annealing, the two metaheuristics studied, have been able to find the global optimum for every optimization problem proposed. Furthermore, these have obtained the best results amongst the various methods implemented even at low computational costs. The performance of these methods compared to that of classic methods can be quantified in terms of the simulation time elapsed to obtain a certain improvement of the solution.

Several tables have been used to summarise the results obtained from the simulations for elapsed times of 1 second for unconstrained continuous optimization and 2 seconds for constrained continuous optimization and combinatorial optimization. The solutions obtained are presented in terms of the actual error and the percentage error. The methods included are Pure Random Search (PRS), Exhaustive Search (ES), Genetic Algorithm (GA) and Simulated Annealing (SA).

		PRS	ES	GA	SA
De Jong's Function	Error(units)	$0,00252$	$0,002$	$0,0000056$	$0,00035$
	Error(%)	-	-	-	-
Rastrigin's Function	Error(units)	$0,34$	$0,11$	$0,0071$	$0,12$
	Error(%)	-	-	-	-
Six-Hump Camel Back Function	Error(units)	$0,0002$	$0,0005$	0	0
	Error(%)	$0,0194$	$0,0484$	0	0

Table 5: Summary of simulations results for unconstrained continuous optimization (source: self made)

For unconstrained continuous optimization, Genetic Algorithm is able to reduce the error of the solution obtained by Exhaustive Search in over a 90% for the three problems studied for a simulation time of 1 second. This improvement increases with the simulation time. In the same problems, Simulated Annealing reduces the error obtained by Exhaustive Search in a range from 0 to 50% for the same simulation time of 1 second. The improvement increases again with the simulation time. Therefore, while Simulated Annealing might obtain the same results as Exhaustive Search for a simulation time of 1 second in some of the problems studied, for longer simulation times the performance of SA is guaranteed to be better than that of Exhaustive Search. The three mentioned methods outperform Pure Random Search.

		PRS	ES	GA	SA
Schmit Structure	Error(units)	$0,08$	$0,0334$	$0,045$	$0,025$
	Error(%)	$3,0337$	$1,2666$	$1,7065$	$0,9840$

Table 6: Summary of simulations results for constrained continuous optimization (source: self made)

For constrained continuous optimization, Genetic Algorithm does no longer obtain the best results. In this type of optimization problem Simulated Annealing has an edge on Genetic Algorithm due to the fact that constraints must be checked for every candidate solution and, therefore, since GA has a larger number of candidate solutions the computational cost drastically increases for highly constrained problems.

		PRS	ES	GA	SA
Travelling Salesman Problem	Error(units)	212	-	79	113
	Error(%)	9, 2698	-	3, 4543	4, 9410

Table 7: Summary of simulations results for combinatorial optimization (source: self made)

For combinatorial optimization, since no constraints must be checked, Genetic Algorithm gives again the best results. Both metaheuristics outperform Pure Random Search as expected. In this problem, Exhaustive Search can not be directly compared to the other methods studied since it can not be run for the chosen simulation time of 2 seconds. If Exhaustive Search is run it returns the shortest tour in 49 seconds while both metaheuristics find a tour with an error below 5% in under 2 seconds.

As computer science advances computational cost is less of an issue and thus metaheuristics are everyday more powerful. If the simulations carried out in this dissertation were to be repeated in one year time, using the exact same implementations, the results obtained would be significantly better than the ones today due to the increase in computational power. For these reason metaheuristics will probably be widely used in the near future to solve a broad variety of optimization problems, including engineering optimization. Moreover, given the current interest in ecological processes, nature based algorithms such as GA and SA are being thoroughly studied.

References

[1] Antoniou, A. *Practical Optimization: Algorithms and Engineering Applications.* Springer. March 2007. p. 1-79

[2] Bäck, T.; Fogel, D. B.; Michalewicz, Z. *Evolutionary Computation 1: Basic Algorithms and Operators.* Institute of Physics Publishing. May 2000. p. 1-78 127-137 166-190 235-325

[3] Bentley, Peter J. *Evolutionary Design by Computers.* Morgan Keufmann Publishers. June 1999. p. 1-219

[4] Chambers, Lance D. *The Practical Handbook of Genetic Algorithms: Applications.* Chapman & Hall/CRC. December 2000. p. 1-3 41-46 99-133

[5] Douglas, Scott. H. *Variations on a Simple Genetic Algorithm.* Rochester Institute of technology.

[6] Goldberg, David E. Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley. January 1989.

[7] Hartmann, Alexander K. *Optimization Algorithms in Physics.* Wiley-VCH. September 2001. p. 159-183

[8] Holland, J. H. *Adaptation in Natural and Artificial Systems.* MIT Press. April 1992. p. 1-32 89-164

[9] Kirkpatrick, S.; Gelatt, C. D.; Vecchi, M. P. *Optimization by Simulated Annealing*. SCIENCE, Volume 220, Number 4598, 13 May 1983.

[10] Koh, Chan Ghee; Perry, Michael J. *Structural Identification and Damage Detection using Genetic Algorithms*. CRC Press. January 2010. p. 1-45

[11] Man, Kim-Fung; Tang, Kit-Sang; Kwong, Sam. *Genetic Algorithms: Concepts and Desings*. Springer. March 1999. p. 1-62

[12] Mitchell, M. *An Introduction to Genetic Algorithms*. MIT Press. April 1998. p. 1-113 155-187

[13] Molga, M.; Smutnicki, C. *Test functions for optimization needs*. April 2005. p. 1-43

[14] Navarrina Martínez, Fermín. Una metodología general para optimización estructural en diseño asistido por ordenador. UPC. May 2008. II 15-16b

[15] Papadimitriou, Christos H.; Steiglitz, Kenneth. Combinatorial Optimization: Algorithms and Complexity. Dover Publications. January 1998.

[16] Pham, D. T.; Karaboga, D. *Intelligent Optimisation Techniques: Genetic Algorithms, Tabu Search, Simulated Annealing and Neural Networks*. Springer. February 2000. p. 1-47 51-141 187-217

[17] Rao, Singiresu S. *Engineering Optimization: Theory and Practice*. John Wiley & Sons, Inc. July 2009. p. 1-114 316-324 389 489-519 715-761

[18] Schneider, J. J.; Kirkparick, S. *Stochastic Optimization*. Springer. November 2006. p. 79-88 157-166 211-229 299-310

[19] van Laarhoven, P. J.; Aarts, E. H. *Simulated Annealing: Theory and Applications*. Kluwer Academic Publishers. October 1988. p. 7-12 55-88

[20] Yang, Xin-She. *Engineering Optimization: An Introduction with Metaheuristic Applications*. Wiley-Blackwell. July 2010. p. 3-24 61-68

[21] http://chaos4.phy.ohiou.edu/~thomas/complex/ga.html

[22] http://geneticalgorithms.ai-depot.com/Tutorial/Overview.html

[23] http://kal-el.ugr.es/~jmerelo/ie/ags.htm

[24] http://sourcemaking.com/design_patterns/

[25] http://subsimple.com/genealgo.asp

[26] http://www.cumps.be/nl/blog/read/design-patterns-strategy-pattern

[27] http://www.coolsoft-sd.com/ArticleText.aspx?id=4

[28] http://www.econ.iastate.edu/tesfatsi/holland.gaintro.htm

[29] http://www.economicsnetwork.ac.uk/cheer/ch13_1/ch13_1p16.htm

[30] http://www.heatonresearch.com/articles/9/page1.html

[31] http://www.hindawi.com/journals/ijfr/2009/527392/

[32] http://www.nd.com/products/genetic/termination.htm

[33] http://www.obitko.com/tutorials/genetic-algorithms/

[34] http://www-optima.amp.i.kyoto-u.ac.jp/member/student/hedar/Hedar_files/go.htm

[35] http://www.talkorigins.org/faqs/genalg/genalg.html