Università
Ca'Foscari
Venezia

Corso di Laurea magistrale (*ordinamento ex D.M. 270/2004*)
in Economics

Tesi di Laurea

Ca' Foscari
Dorsoduro 3246
30123 Venezia

Option Pricing with Genetic Programming

Relatore
Ch. Prof. Marco Corazza

Laureando
Alice Preo
Matricola 829215

Anno Accademico
2014 / 2015

Contents

Introduction

I am glad to present here all the research I have done on Genetic Programming applied to Option pricing.

Chapter 1 wants to introduce the concept of Genetic Algorithms and Genetic Programming widely explaining what they are, their structures and all the features that characterized them.
Chapter 2 focuses on the presentation of the previous work and researches conducted on Genetic Algorithms and Genetic Programming, with special attention to the applications in the financial world and in particular to option pricing. In this chapter will be given a comprehensive and detailed presentation of this financial instrument and will be introduced the Black-Scholes formula.
Chapter 3 is developed in two experiments I run using the Matlab 2013a software in order to test the capabilities of Genetic Programming in approximating the Black-Scholes formula.
Chapter 4 closes this work presenting the conclusions that I have drawn carrying these experiments and some suggestions for future works.

Acknowledgments

It is a pleasure to thank the people that made my thesis possible.

I want to express my acknowledgment to my supervisor Professor M. Corazza, for his help and support and for his patience in correcting my mistakes.

I want to express my gratitude also to my family, who has always supported and trusted me, and to my friends, that has always stood by my side.

Chapter 1
Genetic Algorithms and Genetic Programming: a Wide Overview

1.1 Evolutionary Algorithms

In artificial intelligence, we refer to Evolutionary Algorithms (EAs) as a subset of Evolutionary Computation, a generic population-based meta-heuristic optimization algorithm. Often directly inspired by nature, meta-heuristics are general algorithmic frameworks, whose purpose is the identification of solutions for complex optimization problems and they represent a growing research area since a few decades. Meta-heuristics include categories based on different criteria: for instance, some meta-heuristics process a single solution (e.g. simulated annealing) while others process a set of solutions (and are called population based methods, e.g. evolutionary algorithms).

Evolutionary Algorithms are heuristics that mimic the processes of natural evolution in order to solve global search problems. These algorithms are based on the Darwinian principle of the "*survival of the fittest*", i.e. the most fitting individuals in a certain environment have greater possibilities to survive and pass on their "genes" to the next generations.

The "survival of the fittest" principle is common to others approaches that draw inspiration from natural and biological systems. Under the extensive classification of *Biologically Inspired Algorithm* it is possible to include not only the Evolutionary Algorithms, but also the Artificial Neural Networks, the Social Systems and the Immune Systems. Considering the Artificial Neural Networks, that mimic the capability of the neuron web of the human brain of process the inputs and converting them in meaningful output data, there exist three more subcategories: Multi-layer Perceptron, Radial Basis Function Networks and Self Organizing Maps. The Social Systems is categorized into Particle Swarm Optimization (which also

includes the Grammatical Swarm) and the Ant Colony Optimization. Finally the Immune Systems are divided into Negative Selection and Clonal Selection.[1]

Practically speaking, a population of *individuals* evolves from generation to generation through mechanisms similar to sexual reproduction and genes mutation. Each individual represents a possible solution to the investigated problem. This mechanism leads to a heuristic search which endorses regions in the search area where better solutions are more likely to be found, even though it does not completely neglect regions where are situated solutions with less probability of success. Thus, as described by Charles Darwin in *On the Origin of Species by Means of Natural Selection* (1859), over time, due to natural selection, the population as whole evolves embedding individuals whose genes are converted into structures and behaviors that enable those individuals to better perform in the environment and allow them to survive and reproduce.

Known as Evolutionary Algorithms (EA), this family of computational techniques, characterized by the underlying idea of natural selection, is commonly classified into Evolutionary Strategies (ES), Evolutionary Programs (EP), Differential Evolution (DE), Genetic Algorithms (GA) and Genetic Programming (GP). These techniques usually share the general outlines, diverging in term of specific technical details. Later on the attention will be focused mainly on Genetic Algorithms and Genetic Programming.

1.2 Genetic Algorithms: an Introduction

The development of the Genetic Algorithms, almost universally abbreviated to GAs, dates from the 1960s, but only in 1975 John Holland introduced them to the wide audience through its famous book *Adaptation in Natural and Artificial Systems*, where he rigorously drew up the basic principles, allowing the development of a new thriving branch of research.

[1] For a deep and comprehensive analysis of the various Biologically Inspired Algorithms refer to *Biologically*

Genetic Algorithms are mathematical adaptive heuristics which exploit a randomized search in order to solve mainly optimization problems. Based on an finite dimensional set (population) of individuals, i.e. solutions of the problem, GAs mimic the principles of natural selection and "survival of the fittest", adapting and evolving solutions to real world issues.

In nature, individuals in a population compete with each other to survive and to reproduce. Highly adapted individuals will spread their genes to an increasing number of individuals in each successive generation, often succeeding in producing, through genes recombination, even fitter offspring ("superfit individuals"). This process leads to evolution, explaining wherein new offspring suit better than the parents in their environment.

GAs simulate those processes, essential to evolution, working on a finite population of individuals. Each individual is evaluated as a candidate solution to the problem of interest. Given a quality function to be maximized as an abstract fitness measure, a "fitness score" is assigned to each individual considering how good as solution to the problem it is. Thus, the fitness function assigns a figure of merit to each solution. The initial population may be generated randomly, or using some heuristic method. Thus, for each individual in the population, genes are randomly assigned and there is a wide spread of individual fitness.

Although many variants of GAs exist, each potential candidate solution is traditionally encoded as a set of parameters (known as *genes*), which define the proposed solution to the problem the Genetic Algorithm is solving. These genes form a fixed-length binary string (0101 . . . 1), often referred to as a *chromosome*. The set of parameters represented by a particular chromosome is referred to as a *genotype*. The genotype is an abstract representation of an individual part of the population and it contains the information needed to compute an organism, the *phenotype*. In other words, the genotype is decoded through the fitness value in the phenotype, which represent a potential solution. The fitness of an individual depends on the performance of the phenotype, which can be computed from the chromosome, using the fitness function. The problem-specific fitness function links each string to a number representing its quality or fitness value, which basically provides a measure of performance and a measure of reproductive opportunities.

The execution of the Genetic Algorithm is a two-stage process starting with the current population. Selection is applied to the current population to create an intermediate one (*reproduction*). At the first generation, the current population corresponds to the initial population. Each string is evaluated according to the fitness function and, according to this value, the strings that present a higher performance are more likely to be copied and placed into the intermediate population. Then *recombination* and *mutation* are applied to the intermediate population to create the next generation. Through the crossover operation, two strings randomly chosen are recombined. In each of the two string, a crossover point is randomly picked, dividing the strings in two parts. Then, the parts are swapped, generating two new strings. The mutation operation, instead, applies only to a single point (bit), which is randomly selected only for some randomly chosen strings. The probability of the mutation operator is extremely low and it is generally applied with the aim to guarantee the population diversity.

Each string is called *schema* (plural "*schemata*") and each schema H, composed by an extended alphabet (the 0 and 1 of the binary alphabet, for example), describes a set of points from the search space of a problem that have particular features. In a population of strings of length L over an alphabet of size K, then a schema is identified by a string of length L over the extended alphabet of size K + 1, where the additional element of the alphabet is the asterisk (the symbol which identifies the indifference for the value in that position, i.e. the "*don't care*" symbol). The asterisk is a metasymbol. It is never explicitly processed by the Genetic Algorithm. There are (K + 1)^L schemata of length L. When an individual survives to the reproduction and recombination processes performing a high fitness value, it is not easy to identify which sequence or combination of symbols that characterize that string is responsible for the successful performance. Averages are the most reliable index that may disentangle this issue. If a particular combination of attributes is repeatedly associated with high performance (because individuals containing this combination have high fitness), it is possible to state that this particular combination of attributes is the reason for the observed performance. The same is true for combinations associated with low average performance. If a particular combination of attributes exhibits both high and low performance, then it may

have no explanatory power for the problem. The Genetic Algorithm implements this highly intuitive approach to identify the combination of attributes that is responsible for the observed performance of a complex nonlinear system.

Intuitively, it may appear that Genetic Algorithms operate only on the specific individual character strings that are actually present in the current population. In his book *Adaptation in Natural and Artificial Systems*, Holland focused the attention on the fact that Genetic Algorithms implicitly processes, in parallel, an extended volume of useful information regarding unseen Boolean hyperplanes (schemata). Hence, the Genetic Algorithm has the notable property of *implicit parallelism* (or *intrinsic parallelism*), which is involved as a number of solutions are worked on simultaneously improving efficiency and reducing the chance of premature convergence to local maxima[2].

The highly fit individuals have chances to get selected for reproduction and recombination by cross-breeding with other members in the population and originate new individuals identified as "offspring", which share some features received from the "parents". The least successful units of the population are less likely to reproduce and eventually they will most likely die out. Generation over generation, good genes are spread throughout the population, mixed and exchanged with other better performing genes as the process runs. Supporting this progression by mating the more fit individuals, more promising areas are explored. A well designed GA will lead to the convergence of the population to an optimal solution to the problem.

The parents are recombined through a "crossover" operator, which splits the two genetic structures apart at randomly chosen locations. This genetic operation allows new individuals to be created and new points in the search space to be tested. The recombination of the different part of the genetic structures of the parents creates two offspring. The two offspring are usually different from their two parents and different from each other and both contain genetic material from each of its parents.

[2] This topic will be further discussed and extended in Chapter 1.5.

The new offspring fitness is evaluated by the algorithm and replaces one of the relatively unfit members of the population. New genetic structures are created until the new generation is completed. The same criterion is iteratively followed for the establishment of successive generation until some previously defined termination criterion is met. The obtained final population presents a selection of possible solutions, which can be applied in order to solve the initial problem.

Improvements in the population are typical of the fitness-proportionate reproduction and crossover operations, because low-fitness individuals tend to be eliminated from the population and high-fitness individuals tend to be duplicated. Note that both of these improvements in the population come at the expense of the genetic diversity of the population. Random "mutation" operations on fixed-length strings may be occasionally introduced in order to guarantee genetic heterogeneity and avoid a rapid convergence to local maxima. The frequency of applying the mutation operation is controlled by a parameter called the mutation probability. Mutation is an asexual operation applied on only single individuals. It begins by randomly selecting a string from the mating pool and then randomly selecting a number between 1 and L as the mutation point, where L denotes the string length. The character (gene) selected as mutation point is changed. In the case of binary alphabet, the gene is simply replaced by the opposite value. Mutation is used with moderation in Genetic Algorithm works and it is considered as a secondary operation useful in reinstating lost diversity in a population because of previous exploitation. Thus, it can be pointed out that the Genetic Algorithm principally relies on the creative outcomes of crossover and the exploitative effects of the Darwinian principle of survival and reproduction of the fittest.

1.3 Genetic Algorithm's Basic Principles

The conventional fixed-length string Genetic Algorithm involves the determination of some features. First of all it must be determined the representation scheme, which is a mapping that explicates each possible point in the search space as a fixed-length string and it requires the

specification of the length L and the alphabet size K. Strictly speaking, before a GA can be run, a suitable *coding* (or representation) for the problem must be devised. Secondly, it is necessary to define a fitness measure able to evaluate every fixed-length string. Thirdly, it is crucial to fix the parameters and variables for controlling the algorithm, which are primarily the population size (M) and the maximum number of generations to be run or another termination criterion and, secondarily, the probabilities of reproduction, recombination and mutation (p_r, p_c, p_m). Further additional quantitative and qualitative control parameters and variables must be expressly nailed down in order to thoroughly specify how to run the Genetic Algorithm. One method of result identification is to designate, as result of the Genetic Algorithm, the best individual in the last generation of the population at the time of termination.

The Genetic Algorithm performs in a *domain-independent* way on the fixed-length strings. It operates making few assumptions about the considered problem and, for this reason, it belongs to the class of methods known as "*weak methods*". Even without knowing anything about the problem domain or the fitness function, the Genetic Algorithm shows a surprising rapidity and effectiveness in searching complex, highly nonlinear, multidimensional search spaces. Broadly speaking, the Genetic Algorithm searches for an unknown space for high-fitness point. Nevertheless, the choices made by the user about the representation scheme, the fitness measure and the above listed features (population size, number of generation, parameters determination) may influence how well the Genetic Algorithm will perform in a specific problem domain.

The basic Genetic Algorithm model relies on the paramount assumption that an individual's high fitness is due to the fact that it contains good schemata. Hence, with this reproductive system, good schemata receive an exponentially increasing number of trials in successive generations. This concept is clearly explained in the Holland's Schema Theorem, which will be later discussed.

Analyzing a method (e.g. Genetic Algorithms) that approach to a solution of a problem by investigating the search space, it can be easily understood the issue regarding the trade-off

between what are called *cost of exploration* and *cost of exploitation* of the already-evaluated points in the search space. This trade-off concerns in finding a compromise between the computer program's sources spent in exploring new points from a portion of the search space, which we believe may have above-average payoffs, and the sources spent in exploiting area in which several points have already been evaluated, in particular starting the new tests from schemata whose fitness have already been evaluated as relatively high, in order to find new solution with higher fitness value.

The Genetic Algorithm allows the user to progress the analysis of the search space by testing new and different points that are similar to the ones that have already provided above-average fitness, directing the search into more promising parts of the search space. The best and the average individual increase their fitness from generation to generation towards a global optimum. When at least 95% of the population share the same value it is possible to state that a gene has converged. Only when all the genes have converged also the population is said to be convergent. Once the population has converged to the global optimum, the fitness of the average individual will meet the fitness of the best individual.

Genetic Algorithms are predisposed to stochastic errors. Even in the absence of any selection pressure, these stochastic errors may lead to genetic drift[3], which cause gene variants to disappear completely and prematurely, reducing genetic variation. Increasing the mutation parameter can reduce the occurrence of genetic drift. Anyway, if the mutation rate is too high, the search may become random.

Mutation also helps to overcome another problem which occurs when genes from a few relatively fittest (but not globally optimal) individuals suddenly and rapidly come to prevail over the population, provoking the convergence to a local optimum. As the run progresses, particular values for each gene begin to predominate, so the range of fitness in the population reduces entailing premature convergence or slow finishing. In the first case, once the

[3] The term *genetic drift* refers to the change in the frequency of a specific gene in the population. This change in frequency means a reduction of the fraction of copies of that specified genes. For this reason *genetic drift* may cause the disappearing of that gene.

convergence is complete, crossover has limited possibilities to induce the exploration of new search space. Only mutation remains to explore entirely new ground. The schema theorem demonstrates that Genetic Algorithms naturally assigns an exponentially increasing number of trials to the best observed schemata, leading to a trade-off between exploitation of promising directions of the search space and exploration of less-frequented regions of the space (see also Vose, 1991). Broadly speaking, Holland's schema theorem states that reproduction opportunities must be allocated according to the relative fitness of each individual, but premature convergence may occur since the population is finite. In order to make Genetic Algorithms effective, it is necessary to modify the way individuals are selected for reproduction by controlling the number of opportunities that each individual gets for reproduction so that it is neither too large nor too small. The consequence is to compress the range of fitness preventing the occurrence of "super-fit" individuals from prematurely taking over. The same techniques used to combat premature convergence also combat slow finishing.

Nevertheless, the convergence to the global optimum, as before stated, cannot be guaranteed, but Genetic Algorithms are generally good at finding "acceptably good" solutions to problems taking an "acceptably short" time. It is important to notice that in general, if exact techniques already exist for finding the solution of a particular problem, they are likely to perform better than Genetic Algorithms in both speed and precision. On the other hand, GAs find their ground of application in complex area of search, where no such techniques exist or where the existing techniques require huge amounts of time for computational mathematical analysis. And even when the specific techniques exhibit good performances, clear improvements can be done by hybridizing them with GA.

1.4 Genetic Programming

1.4.1 Genetic Programming: an overview

The Genetic Programming (GP) is an evolutionary algorithm-based process inspired by biological evolution to find (computer) programs that implement a previously determined

task. It is an extension of Genetic Algorithm, where each element of the population is a program instead of a string of bit. The Genetic Programming consists on a set of instructions and a fitness function that measure how well a computer program has performed a task. Substantially, the Genetic Programming paradigm (where each individual is a computer program) continues the trend of dealing with the problem of interest in Genetic Algorithms by increasing the elaboration of more complex structures undergoing adaptation, which are general, hierarchical computers programs of constantly changing size and shape. The problem solving process can be restated as a search for a highly fit individual computer program in the search space. Genetic Programming is basically a solving problem procedure which provides a way to search for this fittest individual computer program.

John R. Koza introduced Genetic Programming in his book *Genetic Programming - On the Programming of Computers by Means of Natural Selection* in 1992. Through the Darwinian principle of survival and reproduction of the fittest and genetic recombination (crossover operation), the populations of computer programs are mated in terms of genes. The initial population is randomly generated and each individual computer program is composed of functions and terminals pertinent to the problem domain. As Koza specified in his work "*The functions may be standard arithmetic operations, standard programming operations, standard mathematical functions, logical functions, or domain-specific functions. Depending on the particular problem, the computer program may be Boolean-valued, integer-valued, real-valued, complex-valued, vector-valued, symbolic-valued, or multiple-valued. The creation of this initial random population is, in effect, a blind random search of the search space of the problem."*

Individuals are then evaluated according to their performance in the specified problem environment through a fitness measure, whose nature varies with the faced problem. Each computer program is run over a number of distinct *fitness cases* and its performance is evaluated as the sum or average over the selection of depictive various situations.

In generation zero the computer programs generally show poor fitness performance. However, some individuals will anyway perform better than others, showing higher fitness measures.

Genetic Programming will then exploit these differences in performance and, as for the Genetic Algorithm methods, the Darwinian principle of reproduction and survival of the fittest and the genetic operation of sexual recombination (crossover) are applied to generate a new offspring population of individual computer programs from the current one. In proportion to their fitness, the most performing computer programs will be selected for the current population through the reproduction operation. The procedure allows them to survive by coping them into the new population. New offspring are then created through sexual reproduction between two parental computer programs, which are always selected in proportion to their fitness. Normally, the parental programs exhibit divergences in size and shape; the offspring individuals are formed by the recombination of subexpressions selected from their parents and are typically of different sizes and shapes than their parents. The procedure ends with the replacement of old parents with the offspring. Each individual in the new generation is then evaluated for its fitness measure and the processes of reproduction and sexual recombination are run over and over again, creating the future generations. If two computer programs are somewhat effective in solving a problem, then some of their parts probably have some responsibility for their good performance. Recombining randomly chosen parts of the most performing programs, new computer programs may result event fitter and perform better in solving the problem.

Over many generations, the Genetic Programming algorithm will produce a population of individual computer programs with increasing average fitness in dealing with their environment, able to exhibit a rapid and effective capability to adapt in changes in the environment. In any generation the best-so-far individual is labeled as the outcome generated by the Genetic Programming (Koza 1992).

While in Genetic Algorithms is required a suitable *coding* (or representation scheme) that explicates each possible point in the search space as a fixed-length string and the specification of the length L and the alphabet size K, in Genetic Programming the role of preprocessing inputs and post-processing outputs is absent or minor. This avoids expensive loss of time.

Inputs, intermediate results and outputs are all expressed as functions, the natural terminology of the problem domain.

1.4.2 Detailed Description of Genetic Programming

Genetic Algorithms and Genetic Programming are both characterized by a structure, undergoing adaptation, composed by population of individual points from the search space instead of a single point. One of the most peculiar feature that differentiate genetic methods from other search techniques, is their ability to simultaneously manage a parallel search involving hundreds or thousands of points in the search space. In Genetic Programming, as before specified, the structures undergoing adaptation are hierarchically structured computer programs, whose sizes, shapes and contents may dynamically vary during the process, according to the changing environment. Rather, the structures that undergo adaptation in the conventional Genetic Algorithm are one-dimensional fixed-length linear strings, as before specified. The set of possible structures in Genetic Programming is the set of all possible compositions of functions that can be composed recursively from the set of N_{func} functions from $F = \{f_1, f_2, ..., fN_{func}\}$ and the set of N_{term} terminals from $T = \{a_1, a_2, ..., aN_{term}\}$. Each particular function f_i in the function set F takes a specified number $z(f_i)$ of arguments $z(f_1)$, $z(f_2)$ ..., $z(fN_{func})$. That is, the function f_i has a number of argument equals to $z(f_i)$.

The functions in the function set may include arithmetic operations (such as +, -, *, /), mathematical functions (such as sin, cos, exp, and log), Boolean operations (such as AND, OR, NOT), conditional operators (such as If-Then-Else), functions causing iteration (such as Do-Until), functions causing recursion, and any other domain-specific functions that may be defined.

For what concerns the terminals, instead, they are either variables, symbolizing inputs, or constant, for example a numerical or Boolean constant. The hierarchical structure undergoing

adaptation in Genetic Programming, formed by set of functions and terminals, must be selected in order to satisfy the conditions of *closure* and *sufficiency*.

As for the *closure* property, it will be satisfied only in the case in which the function set admits only functions well defined and closed for any combination of argument they may encounter. Practically speaking, the closure property requires that any value and data type that may be returned by any function in the function set and any value and data type that may be assumed by any terminal in the terminal set must be accepted by each of the functions in the function set as its argument. This condition is easily satisfied when the problem faced is simple, but ordinary computer programs usually contain complex variables and operators. Closure can be achieved in a direct and unambiguous way for the vast majority of problems simply introducing the *protected division function*, the *protected square root function* SRT, and the *protected natural logarithm function* RLOG, which allow the user to successfully overcome problematic situations when the division function encounters an attempt of division by 0, when the square root function encounters a negative argument or when the logarithm function encounters a nonpositive argument. According to the definition provided by Koza (1992), the *protected functions* are arrangements made in order to deal with problematic situations that some mathematical functions can encounter. For example, if the arithmetic operation of division encounter as its second operator a zero the function would not be defined. In this situation, the protected division function will instead return the value 1. For what concerns the protected square root function, when it encounters a negative argument it will return the square root of the absolute value of that argument. Instead, the protected natural logarithmic function returns 0 when the argument is 0 and operates in the absolute value of the argument when it encounters a nonpositive argument.

On the other hand, the *sufficiency* property admits only a set of terminals and a set of functions capable of expressing a solution to the problem. In other words, before starting to run the Genetic Programming, the user needs to be sure or at least to believe that there exist composition of the functions and terminals that can provide a solution of the problem. Identifying the variables and the set of functions that have sufficient explanatory power to

solve a peculiar problem may be obvious or may require a considerable insight, depending on the complexity of the problem.

1.4.3 Genetic Programming: the initial structure

Referring to the initial structure of Genetic Programming means referring to individuals constituting the initial population, which is randomly generated as a rooted, point-labeled[4] tree with ordered branches (Koza, 1992). Each individual is generated randomly selecting one of the functions from the set F (using a uniform random probability distribution). The root of the tree will be labeled with the selected function. Imposing the restriction of choosing the root exclusively from the function set F, ruling out the possibility of selecting the root of the tree from the terminal set, is a choice made according to the fact that the structure is required to be hierarchical, not a degenerate structure consisting of a single terminal. The root of the tree is usually called *point 1*.

Figure 1.1
Beginning of the creation of a random program tree, with the function + as initial node and two arguments chosen as roots of the tree.

Whenever a point of the tree is labeled with a function *f* from *F*, the number of lines radiating out from that point is the same number of arguments, z(*f*), taken by the function *f*. Then, the endpoint of each radiating line is labeled selecting randomly an element from the combined

[4] Each point is labeled with a function or one of the terminal set components(i.e. a variable, a constant).

set $C = F \cup T$ of functions and terminals (where F indicates the function set and T the terminal set). If a function has been chosen to be the label for any such endpoint, the generating process continues recursively. On the other hand, if a terminal is chosen to be the label for any point, that point becomes an endpoint of the tree and the generating process is terminated for that point.

1.4.4 Genetic Programming: the generative processes

The process for generating random trees of various sizes and shapes can be implemented in different ways. Among these different ways, two basic processes are the "*full*" method and the "*grow*" method. The depth of the tree is measured as the length of the longest non-backtracking[5] path from the root to an endpoint.

The "*full*" method is characterized by the specification of a predefined maximum depth of the trees that the length of every non-backtracking path between an endpoint and the root cannot exceed. This procedure can be implemented forcing the choice of the label of the points whose depth is less than the maximum to the function set *F*, and constraining the selection for the points at the maximum depth to the terminal set *T*. The "*grow*" method does not fix the depth of the trees with a previously specified length measure. This method of generating initial random population involves growing trees that are variably shaped, but still restrict the breadth of the path between an endpoint and the root to a maximum depth. The changing shapes and sizes of the trees are guaranteed by a random selection of the label for points at depths less than the maximum from the combined set C = F ∪ T consisting of the union set of

[5] The term *backtracking* refers to the backtracking algorithm. This technique is used to solve problems under some constraints and it is applied in the analysis of tree structures. The back tracking algorithm explores the tree recursively, starting from the root and systematically searching for a solution to a problem among all available options. Each node is evaluated by the algorithm as potential solution of the problem. If the node does not satisfy the constraints and cannot be considered as a solution, the algorithm excludes it from further searches and moves back to the previous node following the same path it has done to reach the node evaluated. From this position the algorithm moves along another branch in order to reach a new node. In Genetic Programming backtracking is not considered. For a more comprehensive explanation refer to *A theoretical evaluation of selected backtracking algorithms*, Kondrak and Van Beek (1997).

the function set *F* and the terminal set *T*, and, as in the "*full*" method, constraining the selection for the points at the maximum depth to the terminal set *T*.

A more uneven method called "*ramped half-and-half*" can be applied over a wide range of problems. According to Koza (1992), this generative method does best over a broad range of problems. The significant point that distinguish this method is the fact that no assumption is previously made or generically specified in advance on the size and shape of the solution. The "ramped half-and-half" generative method produces a wide range of trees characterized by changeful sizes and shapes, incorporating both the "full" and the "grow"method. The ramped half-and-half generative method consists of the creation of an equal number of trees using a depth parameter that ranges between 2 and the maximum specified depth. This is the generative method most preferred by Koza, as he explicitly specified in his book *Genetic Programming - On the Programming of Computers by Means of Natural Selection* (1992). He provides a clear explanation in order to clarify the concept behind the "ramped half-and-half" generative method. In his example, if the maximum specified depth is 6 (the default value in Koza's book), 20% of the trees will have depth 2, 20% will have depth 3, and so forth up to depth 6. Then, for each value of depth, 50% of the trees are created via the "full" method and 50% of the trees are produced via the "grow"method. Given a fixed value of depth, there is a sensible variation in shape and size from each tree to any other. Thus, the ramped half-and-half method creates trees having a wide variety of sizes and shapes. Empirically, this method has shown better performance than both the "full" and the "grow" methods in a large number of researches.

Another interesting issue considering the generative processes is the presence of duplicate individuals in the initial random generation. These individuals are unproductive deadwood and represent a waste in computational resources. Furthermore, their presence reduce the genetic diversity of the population. Avoiding duplicates in the initial population is desirable but not necessary; the creation of duplicates is more likely to happen when trees have small dimensions (as it is for a certain percentage of population in the "ramped half-and-half" and "grow" methods). In order to control and reduce the presence of duplicates, each newly

generated individual should allowed to become part of the initial population only after been checked for uniqueness. If a new individual is a duplicate, then the generating process is repeated until a unique individual is created. This procedure should be applied only during the creation of the first generation; any duplicate in the following generations must be considered as a product of the genetic operation of reproduction, thus a natural result of the genetic process. Occasionally, it might happen that it is necessary to substitute a small tree with one of larger size, if during the process the set of all the feasible tree of that given size has been exhausted. If duplicate checking is done, then the user will end up with 100% variety of the random population.

1.4.5 Fitness selection

The selection of the elements on which reproduction, crossover and the other secondary genetic operations (i.e. mutation, permutation, editing, encapsulation, decimation[6]) are applied influences the convergences in both Genetic Programming and Genetic Algorithms. The leading force drawn from the Darwinian natural selection is the *fitness*. A major selective pressure leads to a more rapid convergence of the algorithm, which may also causes a loss in the genetic diversity of the population and a failure in the search for the optimal solution. The most common methods used are: the *fitness-proportionate selection*, in which the probability of selection of an individual as parent is proportionate or equal to its normalized fitness; the *rank selection*, in which all the elements are ranked and chosen according to their relative fitness, rather than basing the choice on the absolute value of fitness; the *tournament selection*, in which a fixed number of elements is randomly drawn from the population and the one with the best fitness value is selected. The *fitness-proportionate selection* method is the one which presents a lower selective pressure. This method, described in Holland's *Adaptation in Natural and Artificial Systems* (1975), supports many of Holland's theoretical results.

[6] These operations will be extensively introduced and analyzed in Chapter 1.4.7.

In nature, the fitness is a measure of the degree of likelihood that an individual survives to the stage of reproduction and reproduces. In the artificial world of mathematical algorithms, as before said, the fitness is measured in order to govern and allow the control of the operations that modify the structures in the population. Creating an explicit fitness measure for each individual let the user approach the vast majority of applications of the conventional Genetic Algorithm and Genetic Programming, as clearly illustrated in Koza (1992). Through a well-defined evaluative process, a scalar fitness value is assigned to each individual. The most common fitness measure are well-illustrated in Koza (1992), where the author describes the four measures of fitness that are used:

- raw fitness,
- standardized fitness,
- adjusted fitness, and
- normalized fitness.

The *raw fitness* $r(i, t)$ for individual i at time t is the measurement of fitness that is stated in the natural terminology of the problem itself. The potential benefit of this approach is undermined by the non-comparability of performance of a particular individual across generations. The better value resulting after the fitness evaluation may be either smaller[7] or larger (when raw fitness is gain).

The *standardized fitness* $s(i, t)$ restates the raw fitness in a way that allows the user to rank all the values in a scale where the lowest numerical value is always the best. In certain problem domains (for example, in an optimal control problem, where the aim is to minimize costs), a lower value of raw fitness is better. In this situation standardized fitness equals the raw fitness for that problem

$$s(i,t) = r(i,t).$$

[7] For example when raw fitness corresponds to an error measure, e.g. when it coincides, incorporates or is derived by the Mean Squared Error.

It may be convenient and desirable to make the best value of standardized fitness equal to zero. If this is not already the case, it can be made so by subtracting (or adding) a constant. When for particular problem a greater value of raw fitness is considered a better value, then standardized fitness must be computed from raw fitness: standardized fitness equals the maximum possible value of raw fitness (denoted by r_{max}) minus the observed raw fitness. Basically, the reversal of the raw fitness is required

$$s(i,t) = r_{max} - r(i,t).$$

The *adjusted fitness* measure $a(i, t)$ is computed directly from the standardized fitness $s(i, t)$:

$$a(i,t) = \frac{1}{1 + s(i,t)}$$

where $s(i,t)$ is the standardized fitness for individual i at time t. The adjusted fitness is included between 0 and 1, where higher values correspond to better individuals in the population. Koza (1992) applies consistently the adjusted fitness, in particular since it has a significant feature: it can amplify the importance of small differences in the value of the standardized fitness as it approaches 0 (as often occurs on later generations of a run). Over generations, more importance is given to small differences that make the difference between a fit individual and a fitter one. This procedure is particularly significant if the standardized fitness actually reaches 0, when the perfect solution to the problem is finally achieved.

It is important to point out that adjusted fitness may be neither relevant nor used when specific methods of selection, different from fitness proportionate selection, are applied (for instance, tournament selection and rank selection).

If the method of selection employed is fitness proportionate, the *normalized fitness* $n(i, t)$ is obtained from the adjusted fitness value $a(i, t)$ as follows

$$n(i,t) = \frac{a(i,t)}{\sum_{k-i}^{M} a(k,t)}$$

where $k = \{1, \dots, i, \dots, M\}$. The normalized fitness is characterized by three significant and desirable features. First of all, it ranges between 0 and 1. Secondly, it is larger for the fitter individuals in the population. Thirdly, the sum of the normalized fitness values is 1. In Genetic Programming problems, the phrases "proportional to fitness" or "fitness proportionate" usually refer to the normalized fitness.

Facing some problems, the user may deal with situations in which the population essential to find the solution to the problem is required to be larger and larger. This situation entails an extremely time-consuming calculations because both the population size and the amount of time required to evaluate the fitness are large, especially when computer resources are limited. As Koza (1992) remarks, in many cases, the performance of the Genetic Programming can be considerably enhanced through the *greed over-selection* of the fitter individuals in the population. This procedure is applied when starts the selection of the individuals from the population for the various genetic operation (e.g., reproduction and crossover): the fitter individuals are given an even better chance of selection than is already the case with normalized fitness. This *greedy over-selection* amounts to a further adjustment to the fitness measure. This adjustment should be used to improve performance only when the population size is 1,000 or larger.

1.4.6 Primary Operations for Modifying Structures

Reproduction

The structures undergoing the Genetic Programming are modified through the application of two primary operations and five operations considered secondary, since the user can discretionary choose whether to apply or not these genetic procedures.

The primary operations are the Darwinian reproduction and the crossover (or sexual recombination).

The reproduction operation is the base of the Darwinian natural selection and survival of the fittest. This operation is asexual since its performance involves only one parental individual and it produces only one offspring. A single individual is selected from the population following some previously determined selection method based on fitness. Then, the selected individual is copied, with no modifications, from the current population into the new population (i.e., the new generation). There are many different selection methods based on fitness. The most popular, as before said, is fitness-proportionate selection. When this method is applied, the reproduction operation is said *fitness-proportionate reproduction.* If $f(s_{i(t)})$ is the fitness value of individual s_i in the population at generation *t*, then, under fitness-proportionate selection, the probability that individual s_i will be copied into the next generation of the population is

$$\frac{f(s_{i(t)})}{\sum_{j=1}^{M} f(s_{i(t)})}$$

Usually the fitness value of any individual is evaluated by the normalized fitness function $n(s_{i(t)})$. Thus, the probability that an individual will be copied into the next generation equals exactly its normalized fitness $n(s_{i(t)})$.

As an alternative to fitness-proportionate selection, the user may choose to apply the *rank selection*; in this case, as before explained, individuals are selected according to the rank

position obtained after the determination of the fitness value associated with them (Baker 1985). High-fitness individuals in the population are subjected to a concrete reduction in terms of potentially dominating effects since rank selection implies a limited amount of selection pressure in favor of such individuals. Furthermore, this kind of selection overstates the differences among individuals belonging to clusters characterized by very closed fitness values (Whitley 1989). In *tournament selection*, a fixed-number group of individuals (generally two) is randomly selected from the population and the individual which displays the better fitness value is then chosen.

Generally, the selection operation can be performed with replacement: this implies that parents can be selected several times for reproduction during the current generation. Thus, the rate of survival and reproduction for individuals with high fitness value is essential for Genetic Algorithms and Genetic Programming. Furthermore, reproduction is particularly important in terms of time that can be saved in calculation. The reason is simply the fact that individuals that are replicated in next generations do not need to be again measured in terms of fitness, since their fitness value will remain unchanged[8]. For example, if the reproduction operation is being applied to, say, 10% of the population on each generation, this technique alone results in 10% fewer calculations of fitness on every generation.

Crossover

The *crossover* (sexual recombination) operation has a relevant influence in Genetic Programming. This operation guarantees the variety among individuals in the population by creating new offspring formed of parts taken from each parent. Thus, crossover operation involves two parents from which are created two new offspring. On the contrary of reproduction, crossover is a sexual operation. Parents are selected according to the previously determined fitness-based selection method also used for selection in reproduction operations. Using a uniform probability function, one random point in each parent will be independently

[8] Except for specific cases in which the fitness function is for example normalized or standardized.

selected as the crossover point. The crossover fragment for a particular parent is the rooted subtree which has as its root the crossover point and which consists of the entire subtree lying below the crossover point. This subtree sometimes consists of one terminal. It is important to note that, normally, parents show unequal sizes. The crossover operation has always to take into consideration and implement the fixed parameters which define the maximum depth of the trees. Practically speaking, the first offspring is formed by getting rid of the crossover fragment of the first parent from the first parent and then inserting the crossover fragment of the second parent in what is called the *remainder* or, in other words, at the crossover point of the first parent. The same procedure is symmetrically followed for the creation of the second offspring.

The following figures are displaying a graphical example to clarify the crossover operation.

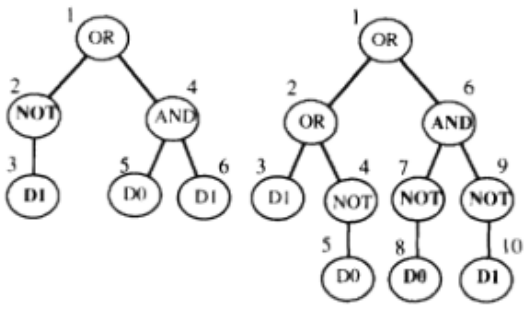

Figure 1.2

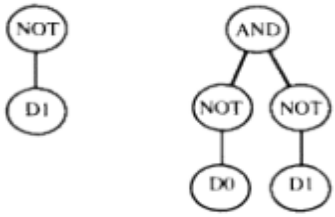

Figure 1.3

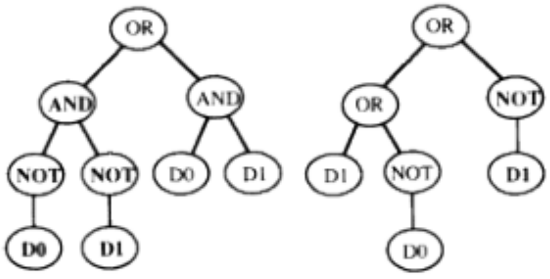

Figure 1.4

Figure 1.2 shows an example of two parental programs. Both trees above are numbered in a depth-first, left-to-right way. The crossover points of Parent 1 and Parent 2 are respectively located at second point (which corresponds to the NOT function) and at the sixth point (which corresponds to the AND function). In Figure 1.3 the crossover fragments are clearly highlighted and set apart. As shown in Figure 1.4, the crossover fragment obtained from Parent 1 is implemented in the remainder of Parent 2, creating a new offspring. The same procedure involves the crossover fragment created from Parent 2 and the remainder of Parent 1, entailing the creation of the second new offspring.

Note that in applying the crossover operation, the closure property if the functions has always to be respected in order to obtain feasible offspring.

Considering the crossover operation results, offspring can be generated in a variety of different combinations. If the crossover point of the first parent corresponds to a terminal point, then the subtree extracted from the second parent will be inserted in the first parent in the place of the terminal point, creating an extension of the first individual, while the terminal will be inserted at the location of the subtree in the second parent, shearing off the second individual. This procedure will often have the effect of producing an offspring with considerable depth.

In the case in which the both selected crossover points correspond to terminal points, the crossover operation will simply consist of a swap between the two terminals from one parent to the other.

There is also the possibility that the root of one parental happens to be selected as the crossover point. In this situation the entire parent will be inserted at the crossover point of the second part, becoming a subtree within the second parent with the result of producing a new offspring of considerable depth. Furthermore, the subtree extracted from the second parent will, instead, become the other complete offspring. In rare situations, it may happen that the crossover point selected from the first parent is the root of the individual (as in the previous

case) and the crossover point of the second parent consists of a terminal point. In this case the second offspring will consist only of one terminal point.

When the roots of the both parents are selected as crossover points, the crossover operation simply implies the reproduction of both individuals.

It may happen that an individual can be selected to embody both parents and incestuously mates with itself or two identical individuals mate. In these situations the resulting offspring are generally different since the crossover points, which are randomly selected, are situated in different positions in the parents. These cases are peculiar since they are completely in contrast with the case of the conventional Genetic Algorithm. When the crossover operation is applied to Genetic Algorithms, it operates on fixed-length character strings where the one selected crossover point will be situated in the same position in both parents. Thus, the incestuous mating of an individual produces two identical offspring that duplicate the parent. These results affect the genetic diversity of the population of the next generation. For both genetic methods, when an individual in the population shows extraordinarily good fitness relative to the other individuals currently in the population, the Darwinian reproduction operation will cause many copies of that one individual to be produced, even if its performance is mediocre in the search space as a whole. In fact, the reproduction operation entails the selection of a fixed percentage of the population, chosen probabilistically proportionate to fitness, that will may be copied into the next generation. This tendency towards convergence will be increased, since the extraordinary individual and its copies will be frequently selected to participate in crossover: incestuous mating among individuals will be recurrent. As before said, in Genetic Algorithms, when an individual incestuously mates with itself, the two resulting offspring will be identical: the result may be a strong tendency toward *convergence* which perilously leads to what is called *premature convergence*. Premature convergence involves the convergence of the population to a globally suboptimal result and generally happens when a mediocre individual in the search as whole shows extraordinary high performance in terms of relative fitness when compared to the other individual of the current generation. As Koza (1992) clearly illustrates, in this situation

(sometimes called "survival of the mediocre"), the conventional Genetic Algorithm fails to find the global optimum. Naturally, when the global optimum is found, the conventional Genetic Algorithm converges with high probability to that globally optimal individual. Once the convergence process is started, only the mutation operation may divert the trend, since in principle mutation may lead in any direction. Anyway, in practice, it is usually to observe a quick reconvergence of the population. Instead, Genetic Programming reacts differently to the issue: if an individual incestuously mates with itself, generally the crossover point will be in different points of the two parents (except in rare cases), producing two different offspring. In conclusion, it can be highlighted that, in Genetic Programming, crossover operation generates a counteracting force away from convergence.

Recalling what before anticipated, a maximum depth of the trees should be fixed at the beginning of any computation in Genetic Programming, in order to avoid extreme losses of time in complex calculation over few extraordinary large individuals. Once this maximum size is established, also the offspring created by the crossover operation must respect this parameter. What happens if, after crossover operation, a new offspring, which exhibits a not admissible size, is created? In this situation the operation must be aborted: the offspring will be eliminated and the first of its parents will be arbitrarily chosen to be copied into the next generation. When both offspring exceed the maximum depth admitted, then both parents will be reproduced into the new population. If it was possible to execute all the possible combination resulting from crossover with no boundaries in terms of depth size of the trees, the process would behave as the nature does. Nonetheless, as Koza (1992) illustrates, it is possible to establish a default value for the maximum permissible depth (for example 17^{9}) which guarantees limited negative influence for what concern the exploration and the constraining of the solutions.

[9] This example for the choice of the maximum depth of the tree just reports the number selected by Koza (1992).

1.4.7 Secondary Operations for Modifying the Structure

For what concerns the secondary operations in Genetic Programming, there are five more optional operations that can occasionally be used, whose worth require to be carefully examined, as their influence might be particularly important for the exploration of the search space. These five operations are:

- mutation;
- permutation;
- editing;
- encapsulation;
- decimation.

Mutation

The mutation operation is particularly useful since it introduces random changes in structures in the population. When applied to conventional Genetic Algorithms, it operates on strings and contributes to reintroduce or increment genetic diversity in a population that may be experiencing premature convergence. In fact, often a particular symbol (i.e. allele) occupying a distinct position on a chromosome string happens to premature extinguish because associated to strings with lower performances. Complications arise when that particular allele corresponds to the needed character that will allow the Genetic Algorithm to achieve optimal solution at a later stage of the run. Occasionally, the mutation operation may effectively produce beneficial outcomes reintroducing alleles necessary to reach the optimal solution but extinct during the run. As Holland (1975) and Goldberg (1989) underline, in conventional Genetic Algorithms the effects of mutation are relatively subordinated to the primary genetic operations and, for this reason, it is considered an almost unimportant operation.

The considerations about the occasional usefulness of mutation applied to strings in the conventional Genetic Algorithm are largely inapplicable when this operation is applied to

Genetic Programming. Mutation is an asexual operation operating only in one parent. The probability, according to which the selection is effectuated, is proportional to the normalized fitness and the result of this operation is one offspring. The initial step of mutation is the randomly selection of a point, called *mutation point*, that can be either an internal point (thus, corresponding to a function) or an external point (i.e. a terminal) of the tree. The second step involves the removal of both the point selected as mutation point and whatever is below that point. After this operation, a randomly generated subtree is inserted at that precise point, creating a completely new individual. Naturally also mutation must comply with the parameter that specifies the maximum size (depth) of the trees. Normally, this control parameter takes the same value of the parameter initially set for the maximum size of individuals in the original random population. It may happen that sometimes, at a randomly point of the tree, a single terminal is inserted after the mutation operation. Occasionally this point mutations occurs also in the crossover operation, when both selected crossover points correspond to terminal points.

In Genetic Programming functions and terminals are not bounded to fixed positions in a fixed structure and it is rare that a single function or terminal completely disappear from the population, at least in the early stage of the run, because of the low number of functions and terminals used in the process. For this reason, while in the conventional Genetic Algorithm mutation restores the diversity, especially in cases of premature convergences, this function is not essential in Genetic Programming. Furthermore, in Genetic Programming the crossover operation may itself produce the same effects of mutation whenever the two crossover points in the two parents are both endpoints of the trees. Thus, even though point mutation may be useful, the crossover operation already provides it.

Permutation

The *permutation* generalizes the inversion operation which applies in the conventional Genetic Algorithm. In conventional Genetic Algorithms, after selecting two different points of a single individual, all the characters included between these two points are reordered by reversing their positions. The effect is to put close together some alleles and move farther apart others with the purpose of establish a linkage between combinations of alleles that perform well when combined together, especially when the inversion operation is applied to relatively high-fitness individual.

As for reproduction and crossover, the individuals subjected to permutation are selected with a probability proportional to their fitness value. This asexual operation, which, thus, operates on only one parent, produces one offspring. It begins with the randomly selection of an internal point (i.e. a function). When the selected function has *k* arguments, a permutation is selected at random from the set of *k!* possible permutations and the arguments of the function are permuted according to the random permutation chosen. Sometimes immediate effect on the returned value may not be visible as long as the selected function has commutative properties. Permutation applied to Genetic Programming, as described above, is different from the inversion operation for the Genetic Algorithms. While in Genetic Programming permutation allows any one of the *k!* possible permutations to occur, which is randomly chosen, the inversion operation allows only one of the *k!* possible permutations, namely the simple reversal.

Editing

The editing operation aims at simplifying the structures of the individuals as Genetic Programming is running. The editing operation is asexual, i.e. it operates on one parent and produces one offspring. In order to make the editing operation applicable, a pre-established set of domain-independent and domain-specific editing rules must be defined for each

individual in the population[10]. The simplification process basically follows one simple rule: any function that presents no side effect, that is context independent and has only constant atoms as arguments, can be evaluated through the editing operation and can be replaced with the value obtained from the evaluation (domain-independent editing rule). A classical example is the numeric expression (+ 2 4) which encodes the summation of 2 and 4: this function will be substituted by the value 6. Another representative case could be the Boolean expression (AND T T), where T stands for True: in this situation the expression will be substitute simply by T. As before said, all these cases follow a pre-specified set of domain-specific editing rules, which covers all the situations that potentially can be simplified. In the Genetic Algorithms there are no equivalents to the editing operation since individuals are already encoded in fixed-length character strings with a uniform structural complexity.

In Genetic Programming, the editing operation can be applied in two different ways:

- the editing operation can be used *cosmetically*, or in other words, external to the run, in order to return a more readable output of displayed individuals.
- the editing operation may also operate during the run with the aim either for returning simplified output or for improving the overall performance of Genetic Programming.

Whatever is the motivation for which the editing operation is applied, it will be implemented to each individual in the population. A frequency parameter controls the recurrence of the editing operation across generations. There is a very unclear opinion over the actual result of the editing operation in Genetic Programming. Doubts are generally related to the difficult question whether simplifying in order to speed up the process is potentially helpful or prejudicial (as it decrease the number of structures available for the crossover, the mutation and the permutation operations) in finding the solution to problems with Genetic Programming.

[10] The terms domain-independent and domain-specific editing rules refer to specialized rules whose application is not affected or influenced by the run of the Genetic Programming.

Encapsulation

The encapsulation operator automatically identifies useful subtrees and gives them an encoded name in order to reference and use them later. In this way it is possible to decompose a larger problem into a hierarchy of smaller subproblems, easier to be solved. The automatic identification of the subproblems and the definition of a hierarchy are the fundamental steps for dealing with large problems. Encapsulation is an asexual operation: as in reproduction operation, the parent is selected with a probability proportionate to its fitness value and it produces one offspring. The first step of this operation is the randomly selection of a function (internal point) of the individual. The encapsulation operation cancels the subtree originated at the selected point and establishes a new function which automatically refers to the removed subtree. These new encapsulated functions are respectively named E_0, E_1, E_2, E_3, ..., according to their creation and they have no arguments (i.e. the functions are placed at a terminal point of the tree). The reference to the new function is then integrated at the selected point of the individual subjected to the encapsulation operation. The result of this operation is one offspring and one new subtree definition. Furthermore, the initial function set of the problem is completed through the integration of the newly created function allowing the mutation operation, if used during the run, to incorporate the encapsulated functions in the subtrees grown at the selected mutation point. The main positive development of the encapsulation operation is the creation of an indivisible single point which encapsulate the selected subtree and which is no longer subject to the potentially disruptive effects of crossover, becoming a potential building block for future generations.

Decimation

In some complex problems[11], the initial population may present high skewness in the distribution of the fitness values. This condition reveals that a very large share of the individuals has very poor fitness. In such situation, the main issue is the incredible amount of

[11] Complex problems usually display some penalty in fitness value in order to reduce the otherwise huge amount of time consumed for each single run. This happens for example in time optimal control problems or problems involving iterative loops.

time that can be spent and wasted on very poor individuals in the early stage of the process, especially in the first generations. Furthermore, there is a high probability that individuals with high fitness values easily start to dominate the population and reduce the genetic diversity. Even if in Genetic Programming the crossover operation guarantees high capabilities in reintroducing genetic diversity in the population, the selection of the parents participating in crossover is always based on fitness: in other words, crossover focuses on the few individuals that relatively perform better in terms of fitness value and for this reason the reintroduction of variety is not always obvious. The *decimation* operation offers a faster way to deal with this situation. As the term in itself suggests, the *decimation* operation reduces the number of individuals in a population letting survive a percentage of the population and eliminating the remaining individuals. Two parameters need to be established in order to correctly apply the decimation operator: a percentage and a condition specifying when the operation is to be invoked. The percentage parameter specifies the share of population that must be preserved while the other parameter defines on which generations the decimation parameter will be applied (for example, the percentage may be 10% and the operation may be invoked on generation 0). Immediately after the fitness calculation for generation 0, all but 10% of the population is deleted. Obviously, if decimation is applied on 10% of the population of generation 0, the user will provide an initial population composed by 10 times the individuals of the population desired for the remaining of the run. Individuals are selected probabilistically according to their fitness value and reselection is not allowed so as to guarantee the maximum variety among the individuals in the remaining population.

1.4.8 Termination Criteria and Result Designation

As the Genetic Algorithm, the Genetic Programming process is virtually never-ending. For this simple reason, the user must establish at the beginning of the run a *termination criterion* that has to be met and satisfied. This termination criterion can be either the achievement of a fixed maximum number G of generations that have to be run or a problem-specific *success predicate* that has been satisfied, which may for example involve finding a 100%-correct

solution to the problem[12]. In more complex problems (for example optimization problems) the exact solution may not be immediately recognizable or not even expected to be found. In these situations the user should apply as *success predicate* a lower criterion for success (instead of the 100%-correct solution). Another termination criterion applied in Genetic Programming is the method of *result designation*, which identifies the best individual ever appeared in any generation of the population, or in other words the *best-so-far* individual resulting from the run of the Genetic Programming. The best-so-far individual is reported as the result of the entire run as soon as the run meets the termination criterion. In alternative, the result designation method can select the best-of-generation individual in the population at the time the termination criterion is met. This method usually produce the same result as the best-so-far method; the explanation for this is the fact that the best-so-far individual is usually in the population at the time of termination. This correspondence happens in two cases. In the first case, due to its high fitness, the best-so-far individual is more likely to be copied into the next generations by the reproduction operation until the termination of the run. In the second case, the run is terminated at the generation in which the best-so-far individual is created and it satisfies the termination criterion.

1.4.9 Control Parameters

Genetic Programming is generally controlled by 19 control parameters, including two major numerical parameters, 11 minor numerical parameters, and six qualitative variables that select among various alternative ways of executing a run. The two major numerical parameter are the population size *M* and the maximum number of generations to be run *G*. The eleven minor parameters used to control the process are:

- The probability of crossover, p_c;
- The probability of reproduction, p_r;

[12] For example some individuals of the population display a standardized fitness of 0 (Koza,1992).

- The selected crossover points are equally allocated through a probability distribution p_{ip} (for example p_{ip} =90%) among the internal (function) points of each tree, while the remaining share of crossover points (10%) is equally distributed among the external (terminal) points of each tree;
- A maximum size (measured by depth), $D_{created}$, is established for individuals created by the crossover operation;
- A maximum size (measured by depth), $D_{initial}$, is established for the random individuals generated for the initial population;
- The probability of mutation, p_m;
- The probability of permutation, p_p;
- The parameter specifying the frequency, f_{ed}, of applying the editing operation;
- The probability of encapsulation, p_{en};
- The percentage and condition for invoking the decimation operation[13];
- The decimation percentage, p_d;

Moreover, the execution of the runs is influenced by six more qualitative variables:

- The generative method for the initial random population;
- The method of selection for reproduction and for the first parent in crossover;
- The method of selection for the second parent in crossover;
- The type of fitness measure applied;
- The application or not of the greed over-selection method;
- The application of the elitist strategy[14].

[13] The decimation operation requires to establish the percentage of individual that must survive and the condition for evoking the operation, which consist in setting the number of the generation at which the operation must operate. For example, setting this condition at 0 means that the decimation operation will be applied on the generation 0 (i.e. the initial generation).

[14] The elitist strategy allows the best-so-far individuals to pass on unchanged to the next generation in order to avoid decrease in the solution quality from generation to generation.

1.5 The Schemata

In the Genetic Algorithm the number of individuals actually present in the current population is just an infinitesimal share of the complete search space of the problem. In his book *Adaptation in Natural and Artificial Systems* (1975), Holland clearly shows how the Genetic Algorithm, processing fixed-length character strings, implicitly operates information about a massive number of unseen schemata. The Genetic Algorithm, in parallel, computes for each generation an appraisal of the average fitness of each unseen schemata. The paramount concept underlying the Schemata Theorem is, indeed, the implicit calculation that operates on the schemata. In other word, the Genetic Algorithm computes the reproduction and crossover operations on the M individuals actually part of the population, but still processes all the possible existing schemata.

Individuals are encoded in strings of length L and each gene can assume a value over an alphabet of size K. Instead, a schema is defined as a string of length L over an extended alphabet of size K plus the metasymbol *. The high consideration received by the concept of schema is due to the fact that a particular schema might be a relevant component of the final solution of the problem. Given a particular string expressed in a binary alphabet, for instance H=(0 0 0 1 0 1 0 1), the schemata connected to this string will consist of any possible schema whose structure match with that string; in other words each symbol matches the symbol of the identifier for all specific positions, where the *-symbol is matching anything, for example H=(0 * * 1 0 * 1). Thus, what the implicit parallelism implies is that one string's fitness tells us something about the relative fitness of more than one schema. The assumption underlying the Schema Theorem is the fact that individuals' high fitness values are due to the existence of a good schemata. According to Holland, for Genetic Algorithms using fitness-proportionate reproduction and crossover, those individuals who present good schemata, and therefore high fitness values, should receive higher chances to pass on their genes in the next generation. In case of binary alphabet, there are $(\mathrm{K}+1)^L$ schemata and each individual appears in 2^L cases; therefore, in a population of M individual strings there will occur $\mathrm{M}2^L$

schemata. Following Holland's thought, the expectation for each schema H to occur in the next generation is

$$m(H, t+1) \geq \frac{f(H,t)}{\overline{f(t)}} m(H,t)(1-\varepsilon)$$

where $m(H, t+1)$ is the number of each schema H expected to occur in the next generation, $m(H,t)$ is the number of each schema H in the current generation, $f(H,t)$ is the fitness value of the considered schema H, $\overline{f(t)}$ is the average fitness and ε is small. When the fraction $\frac{f(H,t)}{\overline{f(t)}}$ remains above the unit through several generations, the schema presenting above-average fitness has exponentially increasing possibility to occur in the next generation, too. Concerning about ε, it is determined by dividing the length $\delta(H)$ of the schema (i.e. the distance between first and last position of non * symbols) by $L - 1$, which is the number of points where crossover could operate). Thus, ε is small when the length of the schema $\delta(H)$ is short too, with the consequence preference of Genetic Algorithms to compute on short and compact schema.

In Genetic Programming a *schema* is the set composed by all the trees in the population that are formed by one or more particular subtrees. Assume that the feature that must be shared among the individuals belonging to the same schema is a subtree formed by *s* predefined points. All points are specified: "don't care" symbols do not exist in the Genetic Programming schema. The number of individuals that could potentially contemplate this feature is unlimited since the possible combinations are infinite, but the Genetic Programming procedure, as before said, consider only trees within a pre-specified maximum depth, which is provided both in the first generation, concerning the size of the initial random trees, and the successive generations, when the depth of individuals varies because of the crossover operation. Once the maximum size W is defined, the subset of interest will be finite. All the individuals belonging to this subset contribute to the computation of the average fitness of the schema, $\overline{f(H)}$, which is the average of all the fitness values of the considered individuals. Following the same approach used to study the Genetic Algorithm, Holland came to the

conclusion that, also in Genetic Programming, the occurrence of schemata in the following generations depends on the ratio of the fitness of the schema in interest to the average fitness of the population. In other words, what it is really interesting is not the absolute fitness value of one individual, but its relative value compared to all the other individuals existing in the population. When this ratio is high, it is possible to notice an increase in the number of the expected occurrence $m(H, t)$ of that schema in the next generation in an exponential way.

In contrast with what happens in Genetic Algorithms, the disruptive effect of the crossover operation in Genetic Programming is more likely to cause deviations from the near-optimal rate of growth (or decay) of a schema. For strings this effect is relatively small because of the limited distance between the points in the string contributing to the definition of schema ($\delta(H)$). In Genetic programming the disruptive effect is limited only when the schema corresponds to a small compact subtree; to overcome this problem, when the schema contains only a single well-defined subtree, these subprograms from relatively high-fitness programs become building blocks for constructing new individuals in an approximately near-optimal way. Over time, the consequences are the reduction of the search space and the increase of the fitness of the individuals. This process applies also to schemata with multiple specified subtrees.

Chapter 2
The Literature In Genetic Algorithms and Genetic Programming

2.1 Literature Overview

2.1.1 Genetic Algorithm and Genetic Programming in Financial Applications

The application of Genetic Programming has collected important results in the solution of problems in which the domain is poorly understood and the relevant variables are not specifically defined or are unknown. Generally speaking, when the domain is completely defined, there might be more specific tools able to solve the problem with good qualitative results which allow the user to avoid the intrinsic uncertainty typical of stochastic processes like Genetic Programming. On the other hand, Genetic Programming reveals all its potential capabilities, especially when it comes about new or not fully understood applications. Genetic Programming may help in understanding the true importance of variables and operations, revealing new problem solutions and unpredictable connections among variables. In other words, Genetic Programming can bring to light new approaches whose application could be extended to a wide variety of circumstances. Its contribution is important especially when size and shape of the solution are unknown. Instead, in the case in which the characteristics of the solution are known, it is possible to approach to the solution with more specific method (for instance the Genetic Algorithm, whose strings have a pre-specified length). One of the most interesting feature of Genetic Programming is its capability to cope with large amount of data and handle the presence of noise in the data. Furthermore, Genetic Programming shows more effectiveness in exploiting smaller dataset with respect to others nonparametric approaches, such as Neural Networks or Genetic Algorithms.

The application of Genetic Algorithms and Genetic Programming has been widely used in an enormous number of fields, reporting successful results as automatic programming tool, machine learning tool and automatic problem-solving engine. The first term (automatic programming tool) defines a computer program whose language is generated by commands that follow a precise automatic code; these tools are opposed to the ones which are manually processed and computed by the users. The machine learning tools are approaches based on algorithms that learn from data. Instead of being merely based on precisely programmed instructions, these tools are able to use the inputs provided by the user to build a model and process the data in order to make decisions and predictions.

During the last decades the financial application of GAs and GP have seen numerous advancements, starting from familiar applications, such as forecasting, trading, and portfolio management, and concluding with enhancements in more recent fields of study, such as cash flow management, option pricing, volatility forecasting, and arbitrage. This research area has been widely studied and a lot of introductory material is available. Bauer (1994) is one of the first and more complete textbooks on the introduction of GAs to finance. No comparable textbook can be consulted with regard to GP in computational finance. However, an exhaustive analysis of financial applications of GP is exposed in Smith and Chen (1998).

Because of the extremely vast area of application of Genetic Algorithms and Genetic Programming, the description of all these possible practical implementations is impossible. For the sake of clarity, an example of the huge work conducted on Genetic Programming is the research developed by Shu-Heng Chen (Chen, 2002), who has applied Genetic Programming over more than 60 papers in finance and economics.

Genetic Algorithms and Genetic Programming have been mainly applied to financial forecasting and trading, which are the most dynamic financial applications. Just for giving the idea of the variety of the subject, even if not updated to the latest years, an extensive review has been provided in *Evolutionary Computation in Economics and Finance: A Bibliography*, Chen and Kuo (2002). From the list of publications contained and classified according to their domain application, it results that about 40 former publications are focused on financial forecasting and 35 on trading. Obviously, the connection between forecasting and trading is

extremely close: the main aim of financial forecasting is to boost the productivity of trading. Excluding financial forecasting and trading, the remaining published application of GAs and GP are mainly focused on portfolio optimization, cash flow management, option pricing, volatility modeling and arbitrage.

Recalling Chen works, Chen has recently worked on the modeling of agents in stock markets (Chen and Liao, 2005), game theory (Chen, Duffy, and Yeh, 2002), evolving trading rules for the S&P 500 (Yu and Chen, 2004) and forecasting the Heng-Sheng index (Chen, Wang, and Zhang, 1999). In 2008 Chen examined the extent to which the return of financial trading rules, obtained through Genetic Programming, is correlated with the entropy rates of the price time series (Navet, N. and Chen, S.-H. 2008), deepening the preceding works on Genetic Programming in financial trading. Among these works dedicated to financial trading and Genetic Programming, it is necessary to cite also the work of Dempster and Jones (2001), where it has been developed a real-time adaptive trading system based on combinations of different indicators at different frequencies and lags.

Kaboudan shows that GP can forecast international currency exchange rates (Kaboudan, 2005), stocks (Kaboudan, 2000) and stock returns (Kaboudan, 1999). Tsang and his co-workers continue to apply GP to a variety of financial areas, including: betting (Tsang, Li, and Butler, 1998), forecasting stock prices (Li and Tsang, 1999; Tsang and Li, 2002; Tsang, Yung, and Li, 2004), studying markets (Martinez- Jaramillo and Tsang, 2007), approximating Nash equilibrium in game theory (Jin, 2005; Jin and Tsang, 2006; Tsang and Jin, 2006) and arbitrage (Tsang, Markose, and Er, 2005). Dempster also uses GP in foreign exchange trading (Austin, Bates, Dempster, Leemans and Williams, 2004; Dempster and Jones, 2000; Dempster, Payne, Romahi and Thompson, 2001). Pillay has used GP in social studies and teaching aids in education, (e.g. Pillay, 2003). Since 1995, the International Conference on Computing in Economics and Finance (CEF) has been held every year. It regularly attracts papers focused on Genetic Programming, many of which are on-line. In 2007 Brabazon and O'Neill established the European Workshop on Evolutionary Computation in Finance and Economics (EvoFIN).

2.2.2 Derivative Securities: A Focus on Options

Concerning the prediction of derivative securities behavior, Genetic Programming has been widely investigated and applied in the study of derivative securities' behavior.

The term "derivative security" defines a financial contract whose value is established by the market price of the underlying cash instrument at the time considered. In other terms, the price of the underlying asset determines the price of the derivative security.

The underlying asset can be

- Stocks;
- Currencies;
- Interest rates;
- Indexes;
- Commodities, like crude oil, gold and many more.

Furthermore, derivative securities can be grouped under three general headings:

- Futures and forwards;
- Options;
- Swaps.

Since the main aim of this work is the analysis of Genetic Programming applied to the option pricing, a brief introduction to options and the previous literature dedicated to the this topic is provided.

In finance, the term *option* defines the peculiar contract that confers on the *buyer* (owner) of the option the opportunity, but not the obligation, to buy or sell the underlying asset on which the option is written at the strike price on or before the specified expiration date. Options mainly diverge from the other derivative securities thanks to the rights that the possessor is entitled: the user is not obliged to buy or sell the underlying. The operation happens only if it is the most profitable choice, otherwise the possessor will not exercise his right. On the other hand, the corresponding *seller* has to fulfill the transaction: he is obliged to sell or buy if the buyer (owner) exercises the option. According to the purpose of the operation, option trading

gives various benefits; in particular, it limits the risk and provides a leverage protecting or enhancing a portfolio in increasing, decreasing or neutral markets.

Options can be classified in *call* or *put options*, which respectively give the right to buy or sell the underlying asset at the strike price on or before the expiration date. The other main classification is between *European* and *American* options: the first gives the opportunity to exercise the option only at the expiration date while the second group gives to the owner the right to exercise the option in any moment from the subscription of the contract until the date of expiration.

Option pricing is a current growing research topic, which is attracting the attention in both academic and practical financial fields. The price, or cost, of an option is known as the premium, the amount of money that grants the right of exercise the option. The premium is non-refundable, whether or not the option is exercised.

The value of the option premium is usually formed by the *intrinsic value* (the difference between the strike price of the option and the value of the underlying asset) and the *time value*, which refers to the difference between the premium and the intrinsic value. Generally, the time value of the option increases as the expiration date is further in the future. The underlying price is definitely the most influential component of the option premium as it influences directly the option price. Strike price instead defines the intrinsic value, if there is any. In particular, the premium augments its value if the option is in-the-money (and the option is more likely to be exercised) and, on the other side, it drops as the option becomes out-of-the-money (and the option will probably be not exercised). An option is considered *in-the-money* when its exercise is considered convenient; in the case of call options this happens when the underlying price is higher than the strike price, while put options are considered *in-the-money* when the underlying asset price is below the strike price. Vice versa, call options are considered *out-of-the-money* when the strike price is above the stock price; when considering put options, they are *out-of-the-money* when the strike price is lower than the underlying asset price. Finally, an option is defined *at-the-money* when the stock and the strike price are identical.

Option traders also consider the volatility of the option, which is a measure of the degree of fluctuation of the underlying asset's price: it displays the speed and magnitude of price

changes. Financial traders usually compare the *historical* and the *implied volatility* in order to understand if an option is over- or undervalued. The *historical volatility* measures the observed price changes in a specified time lapse and is normally calculated using the standard deviation from the average price calculated over that precise time period. The historical volatility is also known as *statistic volatility*. Once the statistic volatility has been calculated, it can be used in a standard option pricing model (as the Black-Scholes model) in order to derive the market value of an option. Normally the market value obtained through this calculation (known as theoretical value) is different from the current price of an option and this difference is defined as option mispricing. In other words, the theoretical value is an estimation evaluated through the application of a model. Its value should picture the worth of the option and it is calculated using known parameters and real data. Generally, these inputs vary during the lifetime of the option and fluctuations should be considered while applying the model. For this reason theoretical value is conceptually different from the current market price of an option. Traders are naturally interested in evaluating the effectiveness of the option pricing model applied and a good measure that allows to evaluate the efficiency of the model is given by the implied volatility. The *implied volatility* is the expected volatility or, in other words, the projection in the future of the volatility rate of the underlying asset price at the expiration date of an option. The implied volatility can also indicate the current market trend. If the implied volatility is higher than the historical volatility it could indicate that the market is expecting that some non-specified factors will significantly influence the trend of the underlying asset. In this case, generally, the option is overvalued. Implied volatility is generally not easily quantifiable as, in general, there is any closed-form formula. Normally, when the underlying asset shows high volatility peaks, it entails higher expected price fluctuations and consequently a higher option premium.

The *expiration date* has a consistent influence in option valuation. The probability that an option will be in-the-money is higher as the expiration date is far in the future. Of course, as the expiration date comes closer, the value of the option will experience a decrease.

An interesting effect on option value is ascribable to interest rates and dividends. The interest rate's influence is linked to the cost of owning the underlying asset and, as the rates increase, call premiums will rise and put premiums will decrease. The interest rate assumes the role of

an opportunity cost: when the interest rates are higher, also the opportunity cost of buying stocks becomes higher and buying call options rather than stocks becomes more attractive. In other words, buying call options instead of the stocks allows the investor to gain the same profit, controlling the same quantitative of the underlying asset but freezing a smaller amount of money compared to the money the investor would have used directly buying the underlying. This condition pushes up the call option demand and consequently the call option price (ceteris paribus). For what concerns put options, they can be considered as substitutes for shorting shares. Shorting shares implies a positive cash flow into investor's account, entailing earnings from interests. For this reason, buying put options can be profitable when speculating to downside trending. When, instead, interest rates increase, put option buyers can not earn on interests. In a scenario of rising interest rates, put options are less attractive than shorting shares causing a drop in put option demand and consequently a drop in put option premium. Following the same reasoning, a decrease in interest rates implies a rise in put options demand along with the premium increase.

Option prices are also conditioned by dividends: any time dividends are cashed, the underlying asset's price experiences a drop on the ex-dividend date. The considerations that must be taken into account are two: when the dividend's value increases, call prices will decrease and put prices will increase. On the contrary, as the dividend's value decreases, the effect on call and put options will be exactly the opposite.

The development of both an academic interest and a flourishing trading market in options started from 1973, when options became actively traded through a guaranteed clearing house at the Chicago Board Options Exchange. Nowadays options are traded through clearing houses on regulated markets or over-the-counter (when buyer and seller agree on a bilateral customized contract). From the 1970s till the present day a constant attention has been directed to the definition of an effective option pricing model. Generally, models exploit fixed certain parameters and data (such as the underlying price, the strike price and the expiration date) and calculated factors, such as the implied volatility, in order to derive the theoretical value of the option at a specified time. During the option lifetime, variable data and parameters will fluctuate and these changes will influence the position of the theoretical value.

2.2.3 Black-Scholes Formula

The first and most common approach to option pricing is the Black-Scholes model, published in 1973 by Fischer Black and Myron Scholes in a paper entitled "The Pricing of Options and Corporate Liabilities" published in the *Journal of Political Economy* and derived by the previous research of Robert Merton and Paul Samuelson. In that paper, a closed-form option pricing formulas were obtained through a dynamic hedging argument and a no-arbitrage condition. The Black-Scholes formula has been widely recognized as a milestone in the option pricing theory and it is commonly applied in finance, but the academic world agrees on the restrictiveness of the assumptions under this model, which shows systematic biases from actual option prices. Above all, it has been assumed that the underlying asset returns follow a normal distribution and discontinuous jumps are not considered possible. This assumption is a limit in real-world applications. In fact, a misspecification of the stochastic process will lead to systematic pricing and hedging errors for derivative securities linked to the underlying asset (Brabazon). A strong assumption is also made on the efficiency of the market: market movements cannot be predicted. Due to these lacks in accuracy and affinity to the real world option pricing behavior, some researches have been directed to new non-parametric approaches, as Genetic Programming (Chen 1997).

The Black-Scholes approach is applied to European put and call options, that can only exercised at the expiration date. The original shape of the model does not take into consideration any dividend during the analyzed lapse. Anyway, further adaption has made possible to account for dividends estimating the ex-dividend date value of the underlying asset.

The Black-Scholes pricing formula for call options:

$$c = SN(d_1) - N(d_2)Ke^{-rt} \qquad \text{where} \qquad d_1 = \frac{\ln(S/K) + \left(r + s^2/2\right)t}{s\sqrt{t}}$$

$$d_2 = d_1 - s\sqrt{t}$$

C = Call premium
S = Current stock price
t = Time until option exercise
K = Option striking price
r = Risk-free interest rate
N = Cumulative standard normal distribution
e = Exponential term
s = Standard Deviation
ln = Natural Logarithm

The formula composition can be analyzed in two parts: the first one, *SN(d1)*, consists in multiplying the current stock price, *S*, by the change in the call premium in relation to a change in the underlying price, *N(d1)*. From this multiplication it can be derived the advantage of buying the underlying. The second part, *N(d2)Ke^(-rt)*, shows, instead, the present value of paying the exercise price upon expiration. The difference between these two parts is the option value. The five variable needed in Black-Scholes formula, as shown in the formula above reported, are five: strike price, stock price, time to maturity, volatility and risk free interest rate.

2.2 Literature In GAs and GP Option Pricing and Related Works

Pricing financial products is certainly one of the most complex issues in finance and, when the underlying asset returns do not follow a precise stochastic process, an exact solution is generally not available (Chen 1997). Various nonparametric approaches, among the others of course also Genetic Algorithms and Genetic Programming, have been widely studied to test their capability to properly overcome the lack of the Black-Scholes model in flexibility.

A wide literature has investigated the features of these applications.

In 1994 James M. Hutchinson, Andrew W. Lo, and Tomaso Poggio published an innovative paper that inspired many further works: *A Nonparametric Approach to Pricing and Hedging Derivative Securities Via Learning Networks*. The authors propose a nonparametric approach for the estimation of the pricing formula of a derivative asset using learning networks. These nonparametric modelling tools exploit data in order to learn from them altering the connections between the input elements and analyzing the evolution of the problem results obtained. In this paper, Black-Scholes option prices are simulated and it is proved that the learning networks have the ability to recover the Black-Scholes rule. For this purpose, a two-year option prices set has been trained. A comparison is conducted analyzing four other approaches: ordinary least squares, radial basis function networks, multilayer perceptron

networks, and projection pursuit[15]. Furthermore, these methods have been tested to real market data from S&P 500 futures options, from 1987 to 1991.

The method presented by Hutchinson, Lo and Poggio is a nonparametric data-driven approach, a technique characterized by the fact that the data itself attempts to generate a model that determines both the behavior and the dynamics of the underlying asset and the connection between the considered asset and the prices of derivative securities. Meanwhile, assumptions on the underlying and on the pricing model are reduced to the minimum: lognormality and sample-path continuity are not taken into consideration and the parameters adapt to the changing environment during the run as data keep evolving through generations. The main inputs of the model are the underlying asset price, the strike price, the date of expiration, the volatility and the free-interest rate, while the output is the derivative's price obtained as problem's solution of the nonparametric data-driven problem.

The aim of this work is to prove the capability of learning networks to approximate the Black-Scholes formula. For this purpose, the learning networks are trained on option prices randomly generated through Monte Carlo simulation[16] in a world where the Black-Scholes rule is applied. In other words, the data generated via Monte Carlo simulation set up a training set, which consists of an input vector that includes the variables of interest, and an output vector. The input and output vectors are used together with the learning method, whose task is to process the input vector in order to obtain an output vector as closer as possible to the originally given one. In this test the output vector used for the training test is formed by the option prices calculated with the Black-Scholes formula. The resulting solutions obtained with the learning networks are compared with the Black-Scholes formula solutions both analytically and in out-of-sample experiments. The results confirm the considerations proposed by the authors: the Black-Scholes formula is recovered with extreme accuracy by the learning network.

---

[15] For more information see *Biologically Inspired Algorithms for Financial Modelling*, Anthony Brabazon and Michael O'Neill (2006) – Springer edition.

[16] Monte Carlo methods are a wide group of computational algorithms that, generally, randomly generate sample data according to a probabilistic function defined on a pre-specified domain. Refer to *Monte Carlo*, by George Fishman (2006) for a comprehensive introduction to Monte Carlo methods.

In 1997 Shu-Heng Chen and Who-Chiang Lee publish their paper *Option Pricing with Genetic Algorithm: The Case of European-Style Options*, an interesting study of the application of Genetic Algorithm in option pricing, focusing in particular on European call options, whose solution is compared to the one obtained from the Black-Scholes option pricing theorem. The obtained results are extremely promising, in particular when the authors consider *in-the-money* options, but some issues with the construction of the test structure limiting the performance of the model. The fitness functions, in fact, are calculated through a relative measure, the *absolute percentage error*, which evaluates, in absolute percentage terms, the residuals between the call option prices derived by the Genetic Algorithm and the call option prices derived by the Black-Scholes formula. This relative measure displays an asymmetric distribution, reaching values close to zero when the options are in-the-money and values up to 70% or 80% when the options considered are out-of-the-money. For this reason, the authors claim the necessity of new researches and in-depth analysis.

Another important research dedicated to option pricing was published in 1998 by Jay White, who applied genetic adaptive neural networks (GANNs) for pricing interest rate futures call and put options. In this work, Genetic Algorithms were used to implement the option pricing formulae evolving and determining the weights of the neural networks. In order to use the Genetic Algorithm in option pricing and therefore derive a formula that can be represented by a bit string, Chen and Lee (1997) approach the issue using a series expansion[17], truncating the infinite series to a finite one. Then, the authors let the Genetic Algorithm process the bit strings that encode the coefficients obtained by the finite series expansion.

As option pricing formula size and shape are not easily derivable, in more recent studies, it is common the application of the tree representation, typical of the Genetic Programming [Chen, Lee and Yeh (1999); Chindambaram et al. (2000); Keber (2000); Keber (2001)].

[17] Series expansions are mathematical methods used to calculate a function whose expression cannot be stated using the basic mathematical operators addition, subtraction, multiplication and division. Thus, this type of functions are expressed as the sum of powers in one of its variables, or by a sum of powers of another (usually elementary) function $f(x)$. Refer to *Finite series-expansion reconstruction methods*, Censor Y. (1983).

The paper of Hutchinson, Lo and Poggio has been considered as a benchmark work: their conclusions has been a comparison term and an inspiring starting point for many following papers. An example is the paper published in 1997 by Thomas H. Noe and Jun Wang, *The Self-Evolving Logic of Financial Claims*. This paramount study is reported in the book "Genetic Algorithms and Genetic Programming in Computational Finance", edited by Shu-Heng Chen, Springer Edition 2002. In this paper, Genetic Programming has been used as an optimization technique to price financial instruments; the purpose is to show how easily Genetic Programming can approximate the Black-Scholes formula even when trained on small data sample. Thus, applying the Genetic Programming to S&P 500 futures options, they found that GP performances in option pricing were at least comparable to the performance of artificial neural networks in Hutchinson et al. (1994).

Noe and Wang wonder about the existence of a pricing technique that does not require a specified pricing structure and does not need to predefine a clear relation between the underlying asset's price and the derivative's price. To overcome these restrictions encountered both in the Black-Scholes model and in non-parametric data-driven model (as the radial-bias neural network method used by Hutchinson, Lo and Poggio), Noe and Wang investigate the Genetic Programming approach. The data sample used for the search is relatively small with respect to the database generally needed, as Genetic Programming can perform well also with a restrict collection of data. In fact, the results show that this method can find a pricing formula that displays small pricing errors. Furthermore, in the following tests, the Black-Scholes formula has been incorporated in the initial population, with the meaning that the authors have employed as output vector the normalized ratio of the option price obtained with the Black-Scholes formula and the strike price. Thanks to these arrangements, the representation of the Black-Scholes formula depicted in a tree-shape has been possible.

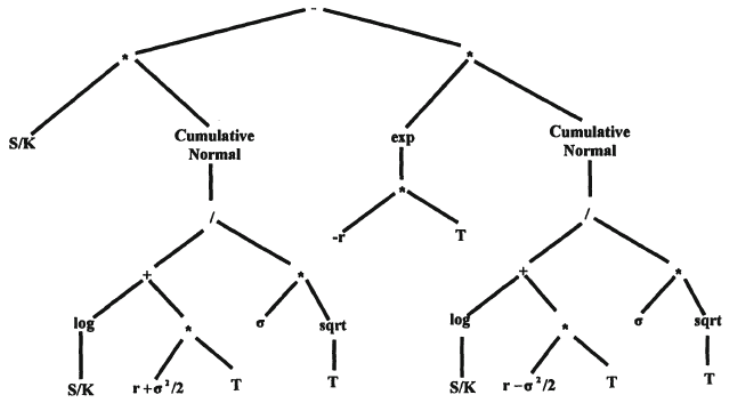

Figure 1 - Black-Scholes option pricing formula in tree representation.

The result is surprisingly interesting: the introduction of the Black-Scholes formula shortens the evolutionary process, but the accuracy in option pricing reached in this second test is substantially the same as in the previous test. Keeping Hutchinson, Lo and Poggio research as a sort of benchmark, Noe and Wang first apply Genetic Programming in a simulated market governed by the Black-Scholes pricing formula. In this first case, three terminal functions are defined: the ratio of the spot stock price and strike price, the time to maturity and a real number between -1 and 1. For what concerns the non-terminal functions, they are randomly chosen between a set of operators such as plus, minus, multiplication, division, logarithm, exponential, minimum, maximum, square root, and cumulative normal. The resulting option price formula is normalized by the strike price. The composition of the terminal and non-terminal sets easily allows Noe and Wang to introduce the Black-Scholes formula, depicted in a tree form representation (see Figure 1).

N randomly generated programs form the initial population and their fitnesses are evaluated. The tournament selection is adopted and termination criteria are set. In particular the process will conclude as the best so far program correctly assess a true pricing value to a previously fixed percentage of the options, or when there is no improvement for a selected number of consecutive generation, or when the maximum number of generations that can be run is reached.

Then the raw fitness of a trading program is defined as:

$$FIT_{raw} = \sum_{i=1}^{M} \left\{ \left| p_i - p_i^0 \right| + f\left(\frac{\left| p_i - p_i^0 \right|}{p_i^0} \right) \right\} + g(S)$$

where p_i is the option price resulting from the Genetic Programming process, p_i^0 is the option price obtained from the market, or in simulation, from the Black-Sholes option pricing formula. *S* is the number of nodes in a tree and *f* and *g* are monotone increasing functions. In other words, the first term represents the raw fitness, the second one an increasing function which measure the relative pricing errors and the third one controls the size of the genetic programs.

In the second test, the Black-Scholes formula is incorporated in the Genetic Programming approach. Assumptions are made on volatilities in the population and the number of total generation is reduced. The test proves the ability of Genetic Programming to recover the pricing formula in a Black-Scholes world.

Finally, in the third test, Genetic Programming applies to the S&P 500 futures options; data are collected from the Chicago Mercantile Exchange. The application of Genetic Programming to S&P 500 futures option shows that the results obtain are better or at least comparable to the one obtained by Hutchinson, Lo and Poggio (1994).

In 1998 N. K. Chidambaran, Chi-Wen Jevons Lee, and Joaquin R. Trigueros published their research *An Adaptive Evolutionary Approach to Option Pricing via Genetic Programming*, investigating the relationship between the option price, its contract terms and the behavior of the underlying asset price. Exploiting the capability of Genetic Programming of incorporating preexisting and commonly used formulas, the model searches for the best approximation to the true pricing formula. With the aid of Monte Carlo simulations, Chidambaran, Lee and Trigueros prove that, when stock prices follow a jump-diffusion process, Genetic Programming is able to reach a high potential option pricing formula, comparable if not preferable to the one obtained with the Black-Scholes model. The approach used by the

authors is non-parametric but, as largely said before, Genetic Programming displays advantages over other learning networks since it can cope with smaller database, on the contrary of neural networks employed by Hutchinson, Lo and Poggio (1994). One interesting feature of this work is a comparison between six different alternative parent-selection methods: Best, Fitness-proportionate, Fitness-overselection, Random, Tournament with 4 individuals and Tournament with 7 individuals. Findings show that the Fitness-overselection method is the one with the most promising results for option pricing. The model incorporates the Black-Scholes formula in the initial gene pool. This shortcut simplifies the process to search for the best option pricing model. In fact, in this way the searching process starts from an already locally optimum solution. Furthermore, the Black-Scholes model easily adapts to a jump-diffusion process[18], making the assumption of normal distributed returns no more necessary and, thus, approaching to the real world structure.

The first test run by Chidambaran, Lee and Trigueros (1998) aims to verify the ability of Genetic Programming to implement the Black-Scholes formula. The dataset has been generated through Monte Carlo simulation. Stock returns are assumed to follow a diffusion process $dS(t)/S(t) = \mu dt + \sigma dW(t)$, while compound expected returns, standard deviation and risk-free interest rate are arbitrarily chosen by the authors. The formula for the calculation of the stock price is $S(t) = e^{\sum_{i=1}^{t} Z_t}$ and for each stock price realization has been generated a sample of call options. The Black-Scholes formula has been used in order to obtain option prices for each simulated option and, following Hutchinson, Lo and Poggio (1994), annual volatility and risk-free interest rate are constant throughout the options' lifetime. Results obtained in the first simulation are compared with a simulation in a jump-diffusion world, as described by Merton (1976). As a closed form solution is available, it is possible to compare the pricing errors from the Genetic Programming model and for the Black-Scholes one. In this scenario, the Genetic Programming formula reaches better solutions in 10 out of 10 runs in comparison with the Black-Scholes approach. When the solutions are applied to the S&P

[18] The jump-diffusion process is a combination of the jump process and the diffusion process, introduced by Merton (1976). The jump process is a stochastic process characterized by discrete movements (jumps) instead of a smoothed continuous movement. The diffusion process, in probability theory, is defined as a solution to a stochastic differential equation.

Index, Genetic Programming performs almost as well as in the previous case and it is preferred to Black-Scholes formula in 9 out of 10 cases. When the comparison involves the study of five equities, Genetic Programming shows better results for 4 of the 5 stocks considered. The same result is reached if Genetic Programming is compared to the results obtained by Hutchinson, Lo and Poggio (1994), while the resolution time is definitely smaller with respect to what is necessary for learning networks.
Summing up, results show that Genetic Programming formulas beat the Black-Scholes equation in 9 out of 10 tests when the jump-diffusion process is selected for generating the stock-prices and in 10 out of 10 runs when the analysis is executed on S&P Index options. Furthermore, also the third test, which has been run over five stocks of the sample, shows how this approach outperforms the Black-Scholes model in 4 out of 5 stocks.
Since the Genetic Programming approach can incorporate the Black-Scholes formula, these solutions can be considered as an adaptation of the Black-Scholes model extended in order to remove the restrictions on the underlying assumptions.

A more recent study was published in 2007 by Anthony Brabazon, Conall O'Sullivan and Zheng Yin, *Adaptive Genetic Programming for Option Pricing*. In this paper an adaptive Genetic Programming method is applied to option pricing. Crossover and mutation probability dynamically vary during the runs. In this case, the experiments are conducted analyzing market option price data. The tests have been designed such that a total of twenty Genetic Programming runs were launched, ten of which use in the process only fixed parameters, while the other ten use dynamic adaptive parameters. The aim of the paper is to show the outperforming capability of the adaptive Genetic Programming with respect to the more classic approach with fixed parameters. As expected by the authors, the results prove that better results are reached when parameters are free to vary and adapt to the surrounding environment.

Beyond the pure option price analysis, various studies have focused on the capability of Genetic Programming to generate performing hedging strategies. Chen has studied this topic and in 1999, after a cooperation with Wo-Chiang Lee and Chia-Hsuan Yeh, he has released

the paper *Hedging Derivative Securities with Genetic Programming*, a study case based on the previous works of, among the others, Hutchinson, Lo, and Poggio (1994), Chen and Lee (1997), Noe and Wang (1997) and Trigueros (1997). The previous studies have shown how Genetic Programming can recover the Black-Scholes using both randomly generated data and real historical data, e.g. S&P Index. This work, instead, focuses on the potential capability of Genetic Programming, compared with the Black-Scholes model, in developing hedging strategy. In this study data used to train and test the Genetic Programming model are daily closing prices of S&P 500 index options obtained from the Chicago Board Options Exchange and performance are evaluated according to a notion of tracking error. Results show that only 20% of the 97 tests run by the Genetic Programming outperform the Black-Scholes model, displaying a lower tracking error. This unsuccessful conclusion may find its explanations in the extremely short temporal period took into consideration. Indeed, the authors claim that a test based on a single year seems to be too limited. Nonetheless, considering the previous work, this paper display results that outperform the ones found in the previous literature, showing an interesting room for improvement.

In 2012, edited by Sebastian Ventura, the book Genetic Programming – New Approaches and Successful Applications is released. Among all the interesting researches published in this book, a new focus on the dynamic hedging is presented by Fathi Abid, Wafa Abdelmalek and Sana Ben Hamida in their paper *Dynamic Hedging Using Generated Genetic Programming Implied Volatility Models*. The aim of this paper is the analysis of a correct approach to forecast the volatility of financial derivatives. Forecasting volatility is one of the crucial issues in trading and risk management of derivatives as the estimation of the volatility has a huge influence in dynamic hedging. Using Genetic Programming as an approach for volatility estimation should allow the users to free the search from strong assumptions concerning the underlying asset price trends. The core focus of the paper is the influence on option contracts prices by new information and the variations in expectations and by the changes in the value of the underlying asset; obviously, the dynamic hedging would be risk-free only in a world volatility is perfectly predictable. For this reason, the more precise and accurate the prediction of the volatility is, the more performing the hedging model will be. The dataset used is the

daily prices for the European S&P 500 index calls and puts options traded on the Chicago Board of Options Exchange from 02 January to 29 August 2003. The paper is structured following two parts. The first part studies the generation of implied volatility from option markets using static and dynamic training of Genetic Programming. The static training implies the independent application of Genetic Programming on single sub-samples of the entire dataset, while in the dynamic training the Genetic Programming trains on all the sub-samples are trained at the same time just changing the training sub-sample during the process. The second part analyzes the precision of implied volatility models generated through Genetic Programming related to dynamic hedging. The results prove the relevance of the implied volatility forecasting in hedging strategies. Hedge performances resulting from Genetic Programming runs are higher than those achieved in a Black-Scholes world. In summary, the conclusions show that the best Genetic Programming hedging performance is obtained for in-the-money call options and at-the-money put options in all the tested hedging strategies.

Chapter 3
The Experiment

3.1 Design of the Experiment

Once the Genetic Programming has been defined and its features widely described, we finally want to test practically the potentiality of this tool. Numerous applications have been reported in the previous chapter in order to present the most interesting works in the empirical research. A particular focus has been placed on the studies lead on option pricing.

The following experiment has been designed using only simulated data. Tests have been devised in order to study the option pricing capability of Genetic Programming in a scenario of poor information and simple mathematical operators.

Through the software *Matlab R2013a* I generated a random population of 500 European call option prices written on a stock which pays no dividends. This population has been generated following a uniform distribution applied to the variables strike price, distributed in the interval [70;130], time to maturity (expressed in years) and volatility, randomly generated and taking values respectively in a interval of [0.5;3] and[0.005;0.2]. Instead, I kept fixed the Stock price at 100 and the free-risk interest rate at 3%. Each option displays a different price/strike ratio, time to maturity and volatility. The sample is equally divided in in-the-money and out-of-the money options in order to guarantee an equal proportion.

Crossover and Mutation probability are set at pc=0.7 and pm=0.3. Each formula obtained after each Genetic Programming run in Matlab was assessed by a *Fitness Value*, which, as said before, summarize the capability of the formula to resemble the Black-Scholes solution. Furthermore, each formula is also coupled with another descriptive value, the *Mean Squared Error (MSE)*, or in other words an estimator that calculate the average of the squared

differences between the estimator (the Call option price obtained with the Genetic Programming solution) and the true value (the Call option price obtained with the Black-Scholes formula). The fitness selection method applied is the *tournament selection*, in which a fixed number of elements is randomly drawn from the population and the one with the best fitness value is selected.

The test has been designed in two different step. In the first step I considered only 250 European call options, 125 in-the-money and 125 out-of-the-money and I considered only seven variables. The second part of the experiment has been designed using the original 500 European call options and adding to the seven variables other 14 more variables. The mathematical operators used in both tests are the sum, the subtraction, the multiplication, the division and the exponential operator.

3.2 First test

After calibrating the Matlab code, the first step consisted in generating 48 runs over the sample data. Each run differs from the other because of the combinations of the population size (which can take values among 50, 75, 100, 250 and 500), the *maxtreedepth* operator (which defines the maximum size of the tree and can take values 8, 10 or 12) and the number of generations run (whose values vary among 50, 75, 100 and 250). Unfortunately, because of the limits of the personal computer used during this experiment, I could not always test the Genetic Programming over populations of 500 individuals.

In the first test I considered as inputs seven variables, namely the Stock Price, the Strike Price, the Risk-Free Interest Rate, the Maturity (expressed in years) and the Volatility. The two further variables are two constant used in order to implement the exponential function and the square root operator. The first constant is the natural number 2.718, which has been introduced with the purpose of teaching to the code how to reproduce the exponential function. The other constant is 0.5 and it has been included in the variables with the purpose to make the code learn the square root operator. The output consists obviously in the Black-Scholes formula results for each call option. These results are obtained using the Matlab

formula “blsprice”, which directly refers to the Black-Scholes formula for European Call options, reported in Figure 1.

$$C = SN(d_1) - N(d_2)Ke^{-rt} \qquad \text{where} \qquad d_1 = \frac{\ln(S/K) + \left(r + s^2/2\right)t}{s\sqrt{t}}$$

$$d_2 = d_1 - s\sqrt{t}$$

C = Call premium
S = Current stock price
t = Time until option exercise
K = Option striking price
r = Risk-free interest rate
N = Cumulative standard normal distribution
e = Exponential term
s = Standard Deviation
ln = Natural logarithm

Figure 1 – Black-Scholes formula for European call option pricing.

Because of the reduced number of inputs, I was not expecting to obtain a precise replication of the Black-Scholes formula results. In particular, in this first part of the experiment, the code have no chance to learn how to implement neither the Cumulative standard normal distribution nor the logarithmic ratio between Stock and Strike price, both used in the Black-Scholes formula. As in the real world nothing seems to prove that the assumptions above the Black-Scholes model should hold (in particular the assumption that claims that the underlying asset returns follow a normal distribution), I chose to not implement these two functions and the experiment has been designed in order to be also directly tested to real financial data.

On the contrary, the purpose of the test was the study of the Genetic Programming in order to understand its capability to well perform even in the presence of a “poor” information environment. In other words, this first analysis focused on the study of the performance of the Genetic Programming approach in terms of Fitness Values, MSE values and the effective distribution of the prices obtained with Genetic Programming with respect to the values obtained with the Black-Scholes formula.

After running the 48 Genetic Programming tests over the different combinations of population size, tree maximum depth and maximum number of generations, I obtained 48 formulas, each

one with its related fitness and MSE value. Each formula has been applied to the 250 European call options forming the database. Thus, a complete Excel spreadsheet has been generated in order to report for each option its true value (the one obtained with the Black-Scholes formula) and all the values obtained with the 48 Genetic Programming formulas. Due to the variety of combinations in the parameter set (population size, maximum depth of the tree, maximum number of generations) that changes for each run, the Genetic Programming formulas are characterized by extremely various performances in terms of fitness and MSE values.

In order to clearly understand the effective capability of Genetic Programming in pricing, I used as discerning terms between performing and less performing formulas the fitness and the MSE values and I compared the two sets. As the sample was wide (I run 48 tests), I did not consider for this comparison the obtained formulas that provided a constant result[19], the formulas able to perform only in a in-the-money scenario[20] and the formulas that encountered different mathematical issues in the process of calculation of the option prices and whose results presented some calculation errors[21]. In this first analysis, after removing the above listed problematic formulas, I obtained a sample of 34 formulas. Then, I made a discretionary selection of the formulas with respect to their performance in terms of fitness and MSE values. I set, as bounding limit for the selection, fitness value higher than 0.70 and MSE lower than 600. Setting this discretionary limits helped me in reducing to 19 the number of formulas to be considered more performing.

---

[19] For example the formula obtained setting popusize=50, maxtreedepth=8 and c=50, that displayed a particularly low fitness value (0,307857) and an extremely high MSE (1482,868763) and the following structure

$$f(x) = -10828{,}464000 * (\mathrm{EXP}) - 29435{,}119054$$

where EXP is the number 2.718, kept constant for all the options analyzed, used with the aim to help the code in learning and reproducing the exponential function.

[20] This is the case of the formula obtained setting popusize=100, maxtreedepth=8 and c=250,

$$f(x) = -23743{,}819503 * (\mathrm{EXP}) + 0{,}004983 * ((\mathrm{ST})/(((((\mathrm{VOL}) - (\mathrm{EXP}))/(\mathrm{ST})) + ((\mathrm{PR}) - ((\mathrm{VOL}) * (\mathrm{RATE})))) - (\mathrm{PR}))) + 0{,}013640 * 0{,}195304 * (\mathrm{ST}) + ((\mathrm{VOL}) * ((((\mathrm{ST})/(\mathrm{MAT}))/(\mathrm{MAT}))\hat{}(\mathrm{MAT}))) - 21{,}249741 * ((((\mathrm{ST}) - (\mathrm{MAT})) - ((\mathrm{PR}) + (\mathrm{VOL})))\hat{}(\mathrm{RATE})) + 64560{,}504422$$

even though it was displaying a high fitness value (0,835407) and relatively low MSE (304,943618).

[21] As I did not set any mathematical restriction, in some formulas happened to appear calculations impossible to be mathematical solved (as the square root of a negative number)

The choice of discerning the Genetic Programming formulas with respect to their fitness and MSE values is connected to the fact that the mere comparison of the prices obtained with the Genetic Programming tests with the values calculated with the Black-Scholes formula would not have helped in understanding the effective capability of Genetic Programming in pricing. I am not expecting the Genetic Programming to perfectly recover the Black-Scholes formula results. For this reason I keep considering the Genetic Programming formulas displaying high fitness values and low MSE values even though the prices obtained with these formulas may show discrepancies with the Black-Scholes formula prices.

3.2.1.The Graphical and Analytical Comparison

At this point, I chose to plot graphically the formulas in order to compare their behavior with the trend of Black-Scholes formula results. I focused on the graphical representation of the most performing formula, but I have also included three graphs representing formulas characterized by low fitness and high MSE (Table 2). The comparison between more performing and less performing formulas will be graphically presented with scatter plot graphs and time series graphs, while the Genetic Programming formulas will be presented analytically in a chart (Table 3 and

Concerning the Genetic Programming formulas with high performances, I graphically compared the Black-Scholes results with groups of 4 formulas and, secondly, singularly with the formulas that graphically displayed a trend closer to the one of Black-Scholes formula one. In this process I used firstly a scatter plot where the variable on the Y axis is the Black-Scholes formula. At this point, I selected the five formulas that display the higher correlation with the Black-Scholes formula (Table 1).

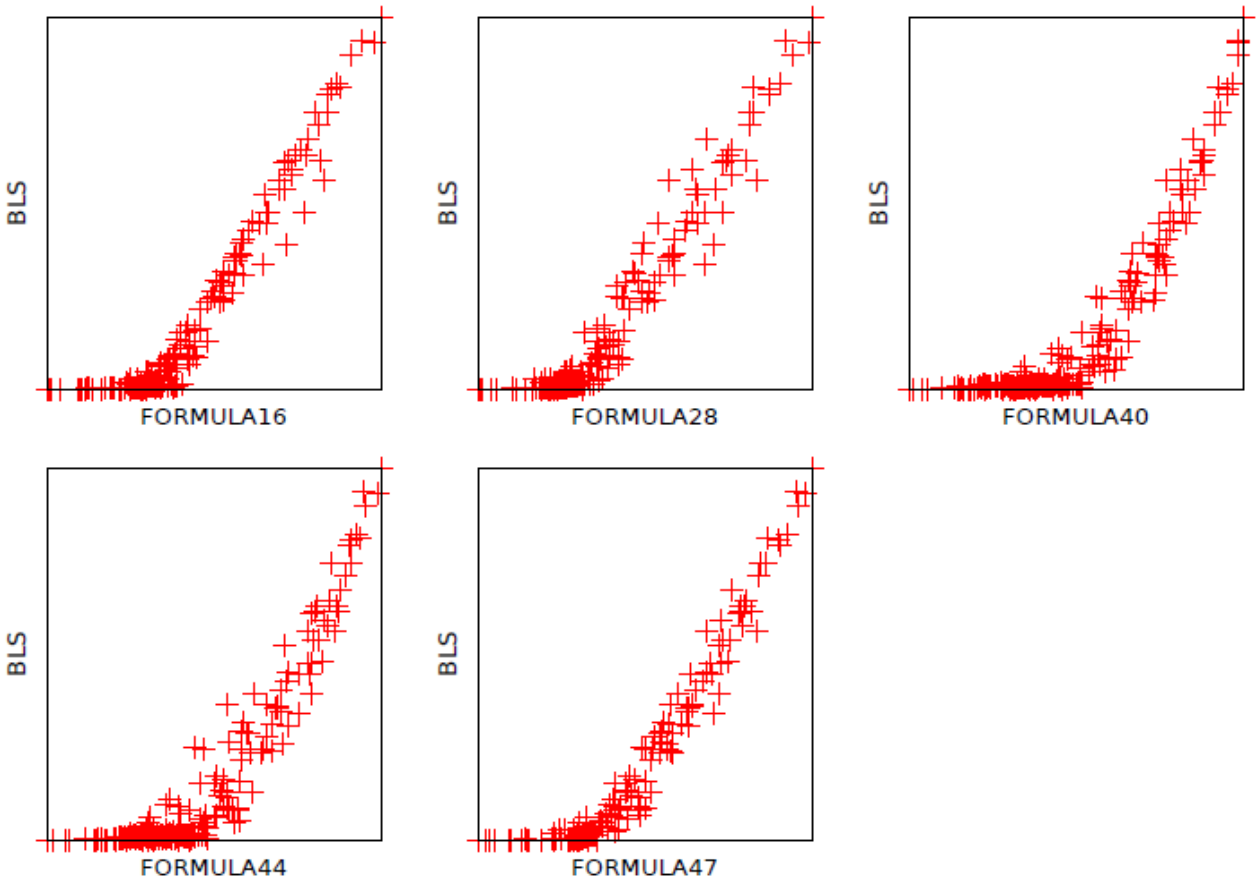

Table 1 – Scatter plot analysis of the relation between Black-Scholes formula and the more performing Genetic Programming formulas.

For the sake of clarity, a graphical representation of poor performing formulas could help in understanding why the formulas previously shown in Table 1 can be considered good in approximating the results of the Black-Scholes formula one. For this reason I present them in Table 2.

From the sample of 34 option formulas obtained after removing the problematic formulas, I selected the 15 Genetic Programming formulas that displayed the worst scatter plot graphs. From this selection I chose to excerpt from that sample the five formulas with the lowest fitness values. The discerning term has been fixed as fitness values below 0.60.

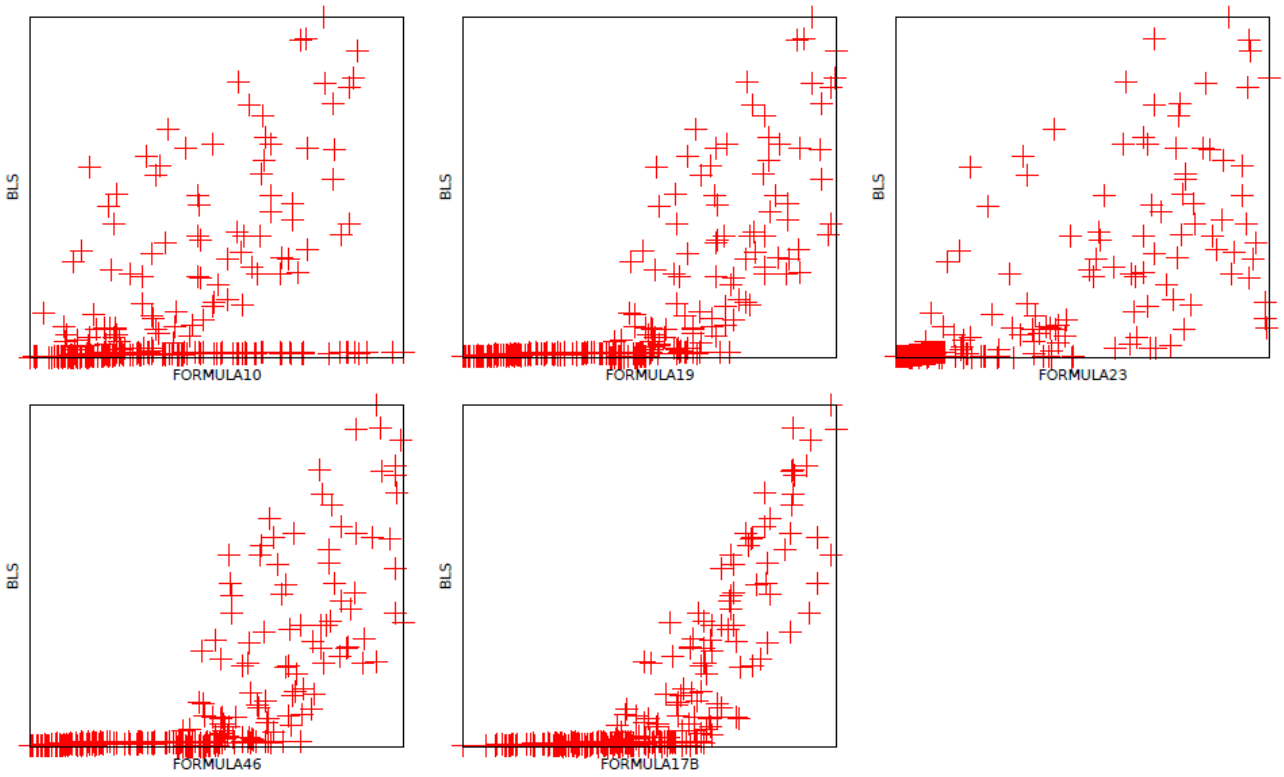

Table 2 – Scatter plot analysis of the relation between Black-Scholes formula and the less performing Genetic Programming formulas.

Table 2 clearly shows the low correlation between the formulas considered and the Black-Scholes one supporting the choice of focusing on the investigation of Genetic Programming formulas according to the fitness and MSE values.

In order to enhance the understanding of the scatter plot analysis, I generated a correlation matrix including both the more and less performing formulas with the purpose of obtaining more precise statistical data.

	FORMULA16	FORMULA28	FORMULA40	FORMULA44	
BLS					
1,0000	0,9462	0,9525	0,8810	0,8962	BLS
	1,0000	0,9825	0,9238	0,9383	FORMULA16
		1,0000	0,9262	0,9317	FORMULA28
			1,0000	0,9860	FORMULA40
				1,0000	FORMULA44

FORMULA47	FORMULA10	FORMULA19	FORMULA23	FORMULA46	
0,9471	0,4757	0,7129	0,7408	0,7110	BLS
0,9903	0,5021	0,6676	0,7109	0,6627	FORMULA16
0,9925	0,5076	0,7065	0,7415	0,7012	FORMULA28
0,9440	0,5675	0,7898	0,7601	0,7878	FORMULA40
0,9518	0,5234	0,7406	0,7377	0,7371	FORMULA44
1,0000	0,4976	0,7007	0,7428	0,6953	FORMULA47
	1,0000	0,6779	0,4489	0,6751	FORMULA10
		1,0000	0,8449	0,9977	FORMULA19
			1,0000	0,8246	FORMULA23
				1,0000	FORMULA46

FORMULA17B	
0,7684	BLS
0,7898	FORMULA16
0,8018	FORMULA28
0,8663	FORMULA40
0,8419	FORMULA44
0,8005	FORMULA47
0,4184	FORMULA10
0,7687	FORMULA19
0,7780	FORMULA23
0,7494	FORMULA46
1,0000	FORMULA17B

Table 3 – Correlation matrix between Black-Scholes formula and the selected Genetic Programming formulas.

The correlation matrix displayed in Table 3 confirms the results drawn from the scatter plot graphs. Indeed the Genetic Programming formulas considered less performing show lower correlation values with respect the formulas considered more performing, with a minimum value reached by the Formula 10, whose associated value of correlation is 0.4757.

The five more performing formulas selected through a simple graph analysis are also the formulas that displays higher fitness values and, in particular, the lower MSE values. From the first graphical analysis of the scatter plots, I analyzed the trends of these five formulas with respect to the Black-Scholes one, removing the formulas that displayed a lower correlation with the Black-Scholes formula. I made this choice after a further comparison between these five formulas in terms of fitness and MSE values, that is reported in Table 4. The remaining three formulas are compared and shown in different graphs in Table 5, Table 6. Table 7 and Table 8 in order to guarantee to the reader a better comprehension of the graphs.

The following chart displays the features of the five more performing formulas, allowing a comparison between their characteristics. The two formulas eliminated after the scatter plot analysis show a small fitness value and a higher MSE value when compared with the other three formulas. This comparison validate the choice of not taking them into consideration for the final graphical analysis.

	popusize	**Max tree depth**	**Max nr of generations**	**Fitness value**	**MSE Value**	**Formula**
FORMULA 16	250	8	250	0,918	174,01	-170159,586470 * (EXP) -4625,176260 * (ST) + 0,000077 * ((ST)^(EXP)) + 30662749,642916 * (((((RATE)*(ST))-((PR)-(PR)))/((ST)/(VOL)))/(EXP)) -57341481,430289 * ((MAT)/((PR)-(EXP))) -338402,467542 * (VOL) + 589437,513732 * (MAT) + 1701,327835 * ((ST)*(EXP)) +462569,774091

FORMULA 28	250	10	100	0,917765	176,183	-67602,723483 * (EXP) -81116320927,718185 * ((PR)-((VOL)+((ST)+(PR))))-81116323010,613144 * (ST) + 2433489625,881561 * (MAT) + -766,277878 * ((SQRT)-((EXP)*(ST))) + -0,753958 * ((VOL)*(ST)) -81115753577,374222 * (((((RATE)+((EXP)^(SQRT)))^(RATE))^((RATE)*((SQRT)*(RATE))))*((VOL)+((MAT)*(RATE)))) +184140,726949
FORMULA 40	250	12	50	0,779557	472,284	-0,210191 * (((((VOL)+(MAT))+((ST)*((ST)/((((ST)+((EXP)^(MAT)))-(RATE))+(PR)))))*((ST)/(PR)))/((SQRT)+(VOL))) + 0,310988 * (ST) -22909,085822 * (EXP) -15,619677 * ((VOL)/(MAT)) +62260,687125
FORMULA 44	250	12	75	0,810046	406,962	-1662,501269 * (ST) + 611,550917 * ((ST)*(EXP)) -293152,699453 * ((VOL)/((((ST)/(RATE))+(PR))) + 2,889952 * (((VOL)/(RATE))-(((RATE)*(EXP))^(((MAT)*(VOL))*(MAT)))) -0,134150 * (((PR)^(EXP))*(SQRT)) +18342,42034
FORMULA 47	100	12	100	0,906710	199,868313	-400510247,048350 * (EXP) -2121,501538 * (ST) -16848,036845 * ((EXP)-(MAT)) + 4233377,299564 * ((EXP)*((RATE)-(((EXP)-(PR))+((RATE)*(VOL))))) + 345228,685402 * (VOL) -6198,004946 * ((EXP)*(MAT)) + 780,451583 * ((ST)*(EXP)) -31070295,881522

Table 4 – Analytical description of the five more performing Genetic Programming formulas.

The choice of removing Formula 40 and Formula 44 has been made because of the extremely high values of the MSE (both above 400) in comparison with the other Genetic Programming formulas taken into consideration.

Before presenting through time series graphs the three selected most performing formulas, I will present a chart, equivalent to the previous one shown in Table 3, which analytically presents the five less performing Genetic Programming formulas displayed in Table 2.

	popusize	**Max tree depth**	**Max nr of generations**	**Fitness value**	**MSE Value**	**Formula**
FORMULA 10	75	8	100	0,5735	913,6438	-31,685016 * (((((MAT)/(PR))-(VOL))*((MAT)-((RATE)^(((MAT)/(EXP))^((VOL)^(VOL))))))/(EXP)) -84536,647655
FORMULA 19	100	10	50	0,5087	1052,547	18,832794 * ((VOL)*(((MAT)*(SQRT))+((VOL)-(RATE)))) + 762,275178 * ((PR)/(EXP)) -280440,66946
FORMULA 23	100	10	75	0,5501	963,7693	-106642,681788 * (((PR)+(EXP))*(VOL)) + 10954157,767224 * (VOL) -0,240591
FORMULA 46	75	12	100	0,505	1058,569	-10255,408317 * (EXP) + 10,489846 * ((VOL)*(MAT)) +27875,582188
FORMULA 17B	50	10	50	0.593	870.788634	-254.439211 * ((EXP)*((PR)+(((SQRT)+(VOL))-(EXP)))) - 0.143044 * (ST) + 711.071810 * (VOL) +67640,529523

Table 5 – Analytical description of the five less performing Genetic Programming formulas.

Analyzing Table 5, it appears clear that the fitness and the MSE values play an paramount role in evaluating the capability of Genetic Programming in option pricing. These five formulas have been chosen according to their low fitness values (below 60) but they are also

the five formulas that displays the highest MSE values. Obviously these two indicators are linked and with a cross-check of these values we can derive important information about the performance of the Genetic Programming formulas.

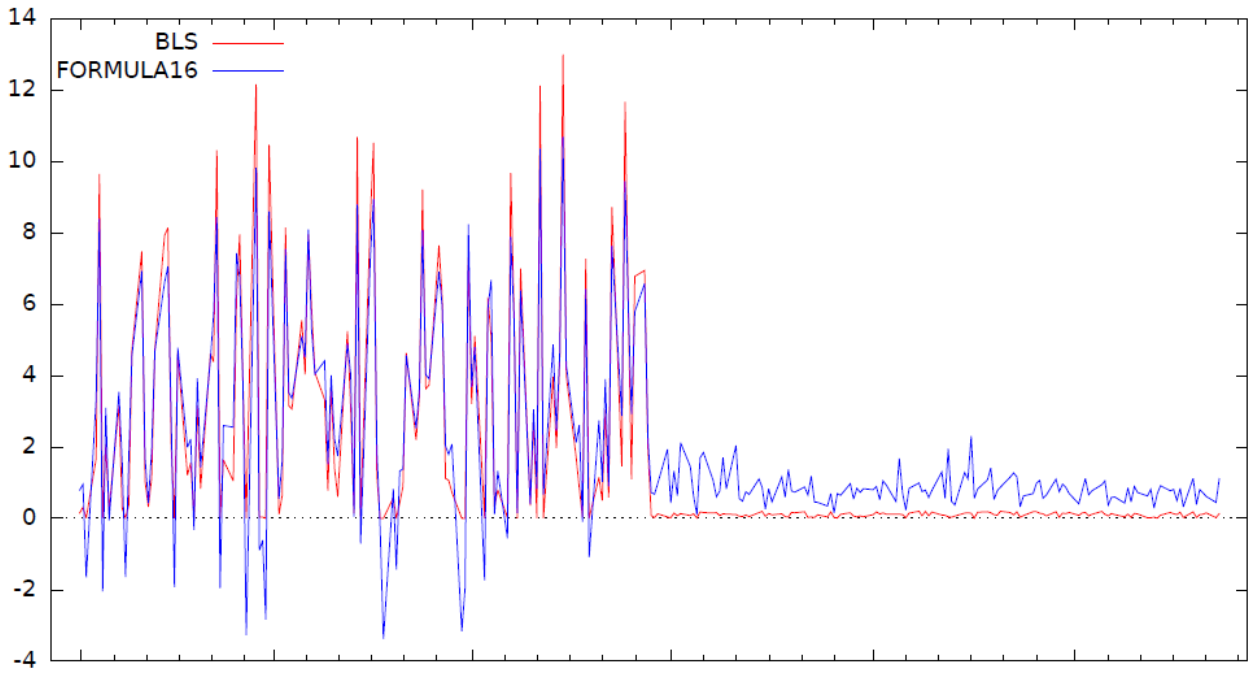

Table 6 – Graph comparing the trend of the Black-Scholes formula and the Genetic Programming Formula 16.

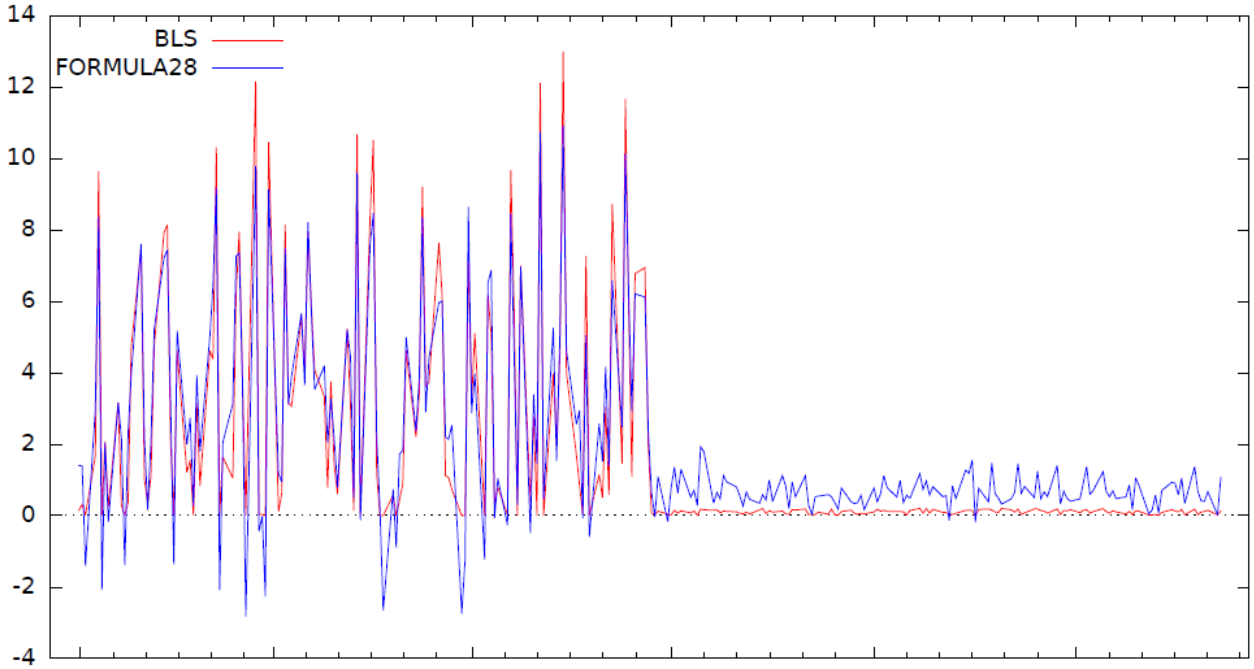

Table 7 – Graph comparing the trend of the Black-Scholes formula and the Genetic Programming Formula 28.

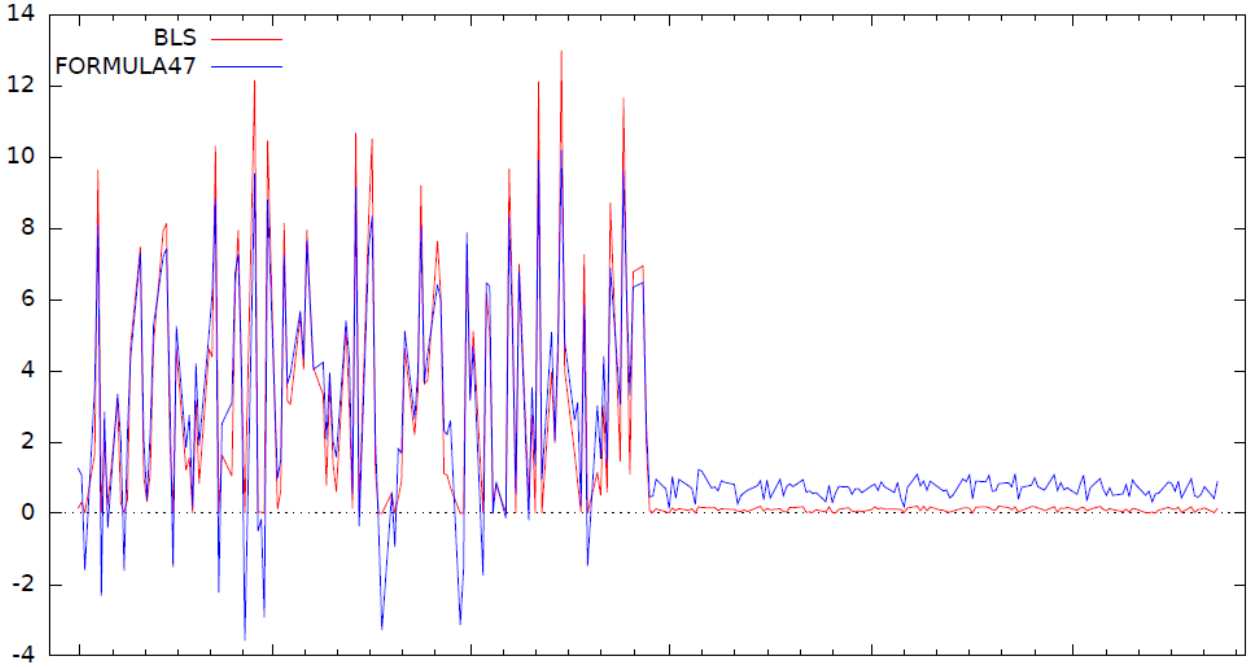

Table 8 – Graph comparing the trend of the Black-Scholes formula and the Genetic Programming Formula 47.

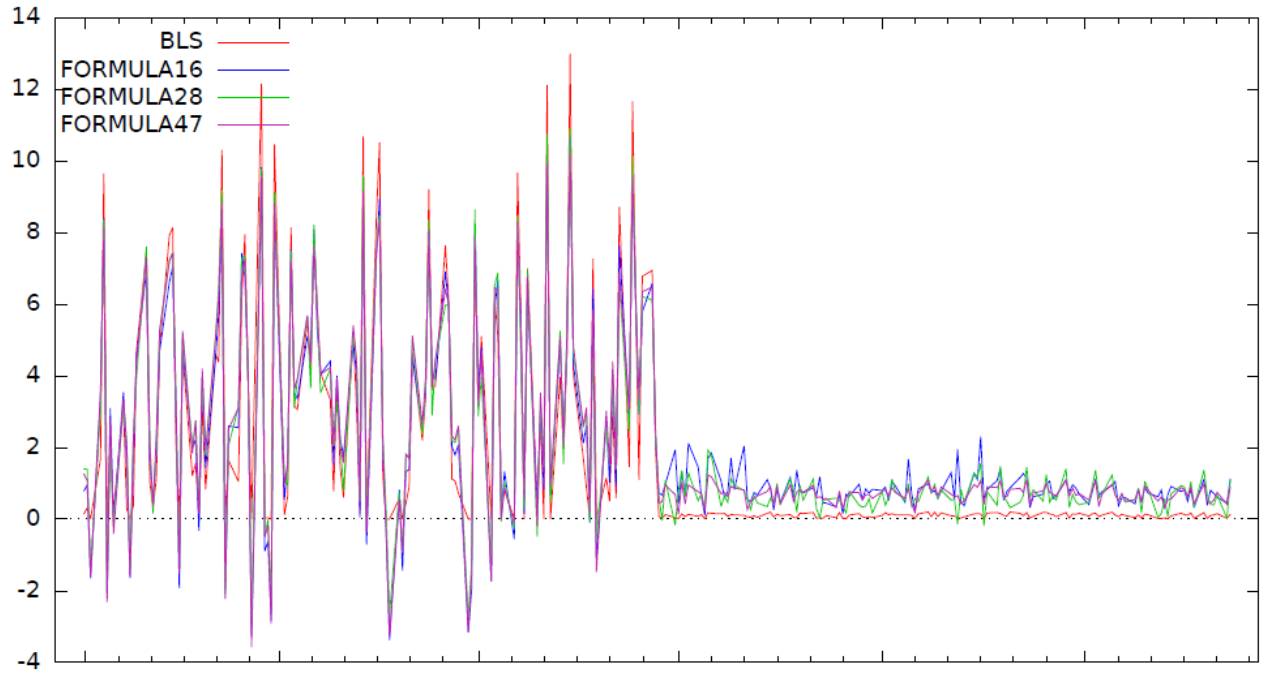

Table 9 – Graph comparing the trend of the Black-Scholes formula and the Genetic Programming Formula 47.

From the graph analysis and the comparison obtained in Table 2, it is possible to draw some conclusion from this first part of the test. The Genetic Programming formula that displayed better performance where the ones presenting a fitness value above the 0.9 and the MSE below 200. The last three formulas taken into consideration where precisely the only three formulas displaying an MSE below 200.

Following the results obtained in the graphs and comparing them with the starting dataset, I can state that the Genetic Programming formulas performance are closer to the Black-Scholes formula when considering in-the-money European call options[22].

[22] As we can see in the graphs, GP formulas trends are more close to the Black-Scholes one in the left half of the graphs, where are represented the in-the-money options. Anyway we have to limit the comparison to the positive plane of the graphs, thus not considering the drops of GP formulas when displaying negative prices.

After focusing on the more and less performing Genetic Programming formula, I took into consideration the whole dataset, considering all the 48 obtained Genetic Programming formula, also the ones that displayed mathematical calculation problems. Considering the fitness values and the MSE values as indicators of good performance, and according to the results so far obtained, I chose to analyze the whole database dividing the resulting formula according to the fitness and MSE values to them associated. I chose to consider, as previously done, performing the formula with Fitness value higher than 0.70, MSE lower than 630 and free from any kind of mathematical calculation issues (as before done). After this selection, the number of Genetic Programming formula that can be considered at least sufficiently good in approximating the Black-Scholes formula are 27 out of 48.

3.3 Second Test

The second part of the experiment is run over a wider number of input variables, designed in order to provide to the code more possibilities to approximate the Black-Scholes formula. The input variables integrated are 21. In addition to the starting 7 variables, I added the logarithm of the ratio between the Stock price and the Strike price (ln(P/S)), the square root of the Maturity time (SQRT(T)) expressed in years, the normal distribution of the logarithm of the ratio between the Stock price and the Strike price (N[ln(P/S])), the normal distribution of the square root of the Maturity time expressed in years (N[SQRT(T)]), the d_1 factor, the d_2 factor, the normal distribution of the d_1 factor ($N(d_1)$), the normal distribution of the d_2 factor ($N(d_2)$), the numerator of d_1, the numerator of d_2, the denominator of d_1 (which corresponds to the denominator of d_2), and more three variables that corresponds to the normal distribution of respectively the numerator of d_1, of the numerator of d_2 and of the denominator of d_1.

The run of this second part of the experiment basically follows the same design and method applied in the first part. I used the complete simulated dataset, randomly generated at the beginning of the experiment. I kept fixed the crossover and the mutation probabilities respectively at 0.7 and 0.3. The fitness selection method applied is again the tournament

selection and obviously the Genetic Programming formulas obtained have been evaluated with a Fitness value and a MSE value.

Anyway, because of the variation in the number of inputs, some adjustment were necessary. As I have been using an extremely larger dataset with respect to the first test due to the addition of 250 more European call options and the introduction of 14 more variables, I reduced the number of runs. I generated 39 runs and each of them differs from the others because of the different combination of population size (which can take values among 50, 75, 100, 25023), the maxtreedepth operator (which defines the maximum size of the tree and can take values 8, 10 or 12) and the number of generations run (whose values vary among 50, 75, 100 and 250). Most of the tests run were extremely time consuming, taking more than one hour to complete the process and unfortunately, because of the limits of the personal computer, I could run only few tests with population size equals to 250.

While in the first tests I did not allowed the Genetic Programming code to implement neither the normal cumulative function nor the natural logarithmic function, in this second test I introduced both these function manipulating my dataset and introducing them directly through the calculation of the new input variables that I add. The purpose of this test is to verify if the Genetic Programming code I used can approximate the Black-Scholes formula learning from the new variables better than in the first part of the experiment.

As in the previous test, I constructed a new complete Excel spreadsheet in which all the Genetic Programming formulas has been used to calculate an estimator of the 500 European call options.

Unlike in the first test, in this second one it is immediately clear, looking at the values obtained, that the Generated Programming formulas displays a lower heterogeneity in terms of fitness values calculated. All the formulas obtained show a fitness values larger than 0.90.

[23] I exclude any test with population size equals to 500 individual, which was instead run in the first experiment.

On the other hand, unless few cases, all the formulas are related to high MSE values with respect with the ones obtained in the first part of the experiment24.

Another important difference with the first test is the fact that no formula presents constant results or encounters mathematical problems with the calculation of both in-the-money and out-of-the-money options. For these reasons, for my first analysis, I did not have to remove formulas from my sample.

Following the same procedure I previously applied, I firstly compared in terms of a graphical representation the 39 formula and compared their behavior with the one displayed by the Black-Scholes formula. After the graphical analysis I further studied the more and less performing formulas in term of an analytical comparison.

3.3.1.GRAPHICAL ANALYSIS

The first comparison has been made with the aid of scatter plot graphs, which display the correlation of results obtained with the various Genetic Programming formulas with the Black-Scholes ones.

Graphical results, as I expected after seeing the results obtained in terms of fitness values, are satisfactory. I report here below the graphical results for the formulas that could be considered more and less performing. As almost all the Genetic Programming formulas obtained in this second test display high fitness values, I chose as discretionary bounding limits fitness values above 0.97 and MSE below 1229.17, which the average Mean Squared Error value of the sample set constituted by 39 formulas. Then, as in the first part of the experiment, I made a graphical comparison. The following graphs presented in Table 1 are the graphs that show the higher correlation between the Genetic Programming formulas and the Black-Scholes one, among all the more performing formulas I analyzed.

[24] While in the first part of the experiment the highest MSE displayed was 1482.868763, in this second test the average MSE is 1229.17.

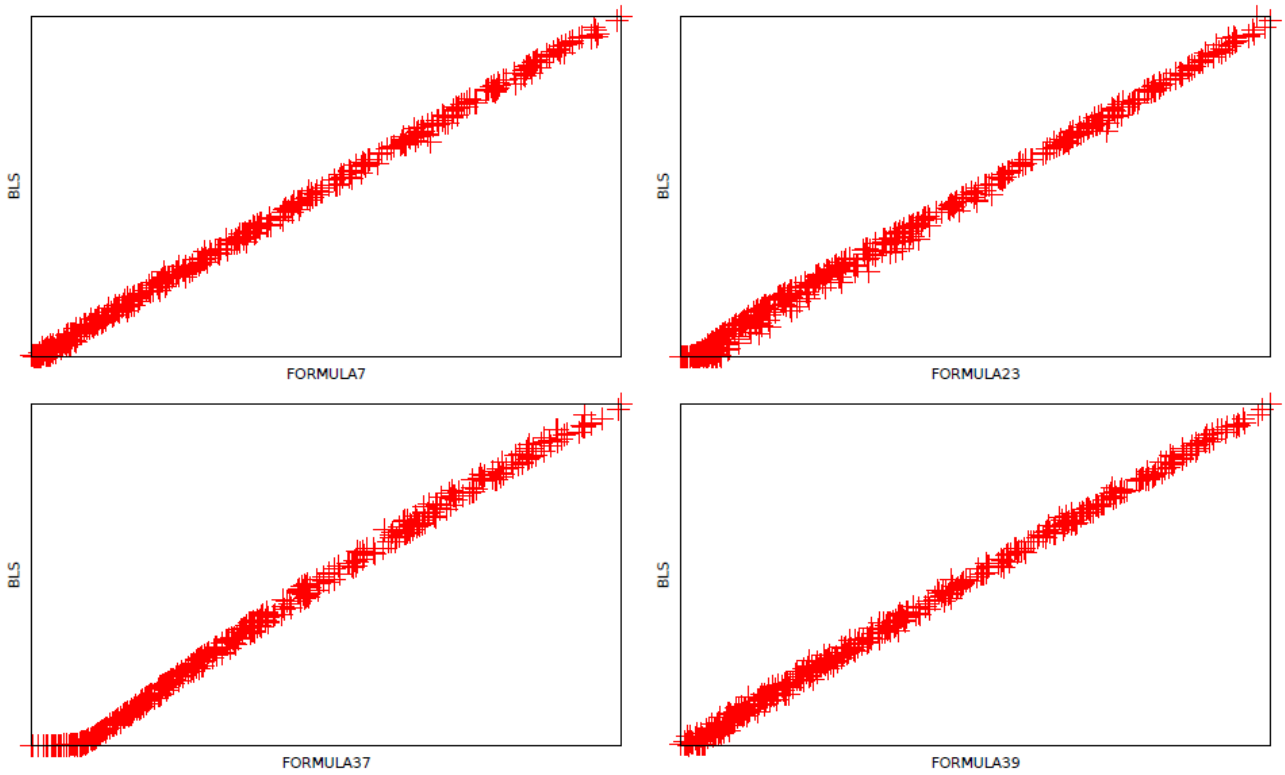

Table 1 – Scatter plot analysis of the relation between Black-Scholes formula and the more performing Genetic Programming formulas.

In Table 2, instead, are reported the graphs that display the lower correlation between Genetic Programming formulas and the Black-Scholes one.

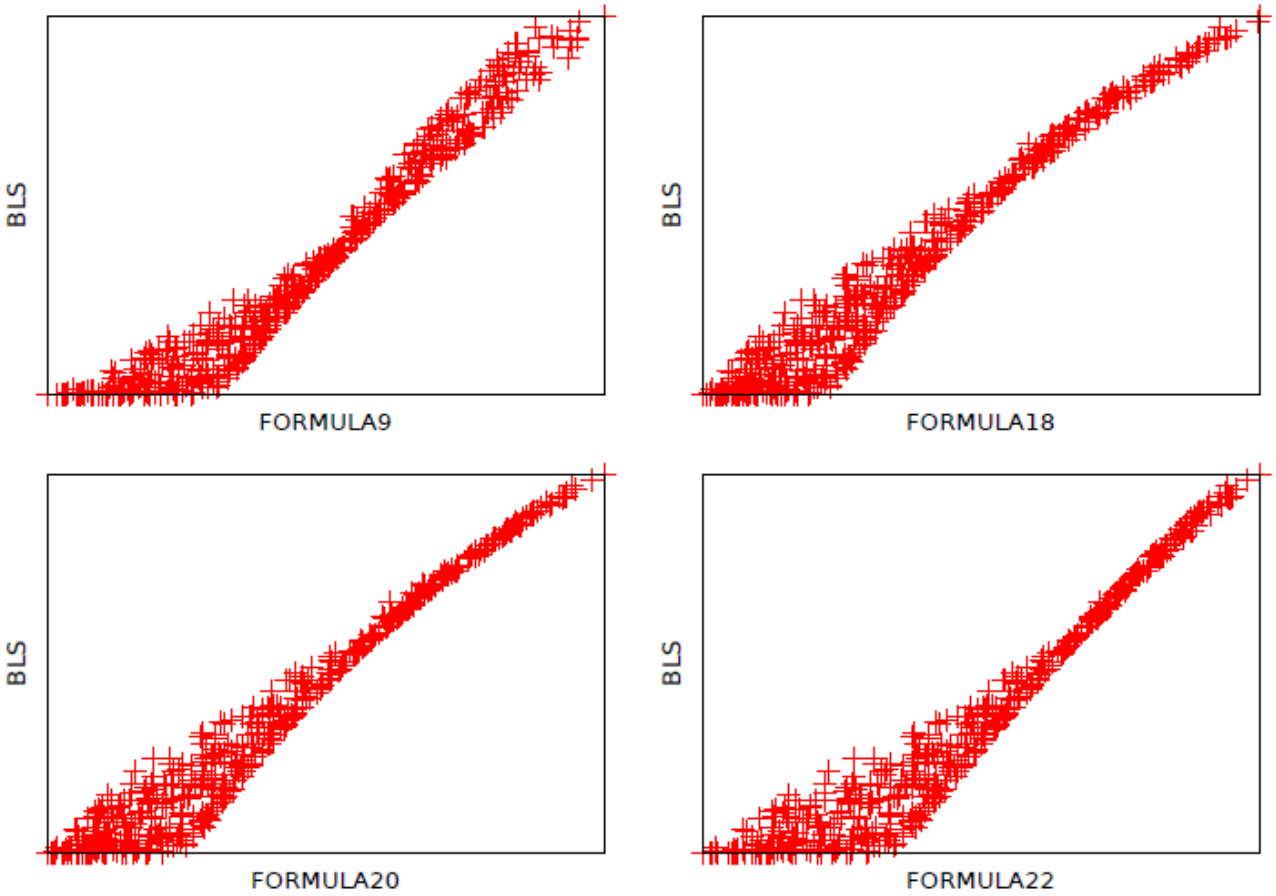

Table 2 – Scatter plot analysis of the relation between Black-Scholes formula and the less performing Genetic Programming formulas.

Table 1 shows the extremely high correlation between the Genetic Programming formulas selected as the more performing and the Black-Scholes formula. Nonetheless, also the formulas considered in Table 2 display a good correlation with the Black-Scholes one. Comparing these graphs with the ones obtained in the first experiment tells us how the new variables included as new inputs in the dataset have extremely influenced the capability of the Genetic Programming code to recover the Black-Scholes formula.

In order to understand how strongly all these Genetic Programming formulas graphically presented in Table 1 and 2 are correlated with the Black-Scholes formula, I created a correlation matrix, presented in Table 3.

BLS	FORMULA7	FORMULA9	FORMULA18	FORMULA20	
1,0000	0,9995	0,9723	0,9819	0,9811	BLS
	1,0000	0,9731	0,9825	0,9817	FORMULA7
		1,0000	0,9812	0,9865	FORMULA9
			1,0000	0,9990	FORMULA18
				1,0000	FORMULA20

	FORMULA22	FORMULA23	FORMULA37	FORMULA39	
	0,9665	0,9986	0,9981	0,9992	BLS
	0,9673	0,9981	0,9980	0,9997	FORMULA7
	0,9940	0,9729	0,9741	0,9714	FORMULA9
	0,9886	0,9831	0,9854	0,9830	FORMULA18
	0,9938	0,9822	0,9838	0,9822	FORMULA20
	1,0000	0,9678	0,9683	0,9670	FORMULA22
		1,0000	0,9975	0,9979	FORMULA23
			1,0000	0,9976	FORMULA37
				1,0000	FORMULA39

Table 3 – Correlation matrix between Black-Scholes formula and the selected Genetic Programming formulas.

In Table 3 the correlation coefficients between the Black-Scholes formula and the Genetic Programming formula are displayed. The Genetic Programming formulas considered less performing are highlighted in red. It appears that obviously the formulas that graphically display a higher correlation also show values higher than 0.99 in the correlation matrix.

Nonetheless, also the formulas that graphically looks less performing show correlation values above 0.966. In order to draw more consistent conclusions I also present a chart that summarize all the features of the Genetic Programming formulas taken into consideration.

	popusize	maxtreedepth	max generation nr	Fitness value	MSE	FORMULA
FORMULA 18	50	10	100	0,959856	1699,687551	4935,391412 * ((num d1)/(Strike)) +6,112178
FORMULA 7	50	8	100	0,998896	52,386932	-26,153737 * (N(d2)) + 120,205762 * ((N(num d2))*(N(d1))) +1,542835 * (N(Sqrt(T))) + 8,924008 * (d1) + 1,584866 * (ln(P/S)) - 8,961602 * (d2) + 0,131279 * ((N(d2))/((den d1)/(ln(P/S)))) -24,708360 * ((N(d2))/(N(num d1))) + 16,427998 * (N(d1)) - 1,61839
FORMULA 9	100	8	100	0,945344	2594,295688	13,369819 * ((den d1)-(r)) + 55,860916 * (num d1) +5,705871
FORMULA 20	100	10	100	0,958251	1776,197638	101,321054 * ((N(num d2))*(num d1)) +6,240229
FORMULA 23	100	10	250	0,997132	136,133857	-3570,280431 * (N(num d2)) -1284,624564 * ((den d1)*(den d1)) + 63,808801 * ((num d2)*(N(d1))) + 3605,814423 * (N(num d1)) -

						15,615279
FORMULA 22	75	10	250	0,931234	3130,180110	56,460913 * (num d1) +7,050409
FORMULA 37	50	12	250	0,996303	175,496388	34,594291 * (num d1) + 76,427356 * (N(den d1)) + 29,094004 * ((num d2)*(((N(d2))/(N(den d1)))-(SQRT))) -37,226416
FORMULA 39	100	12	250	0,998370	77,373792	-0,938838 * ((Strike)*(N(d2))) + 2,523461 * ((T)*(N(d2))) + 94,714028 * (N(d1)) +0,147342

Table 4 – Analytical description of the selected Genetic Programming formulas.

Table 4 helps us in understanding which are the factors that influence the performances of the Genetic Programming formulas. As in Table 3, the formulas considered less performing are highlighted in red. It is clear, again, that high fitness values are linked to the more performing formulas: they all display values above 0.995. Focusing on the formulas considered less performing, we can see that they also show high fitness values, always above 0.93. Nevertheless, they are all related to MSE values much higher than the more performing formulas, which instead are the ones displaying the lower Mean Squared Error values, even if compared with the entire data sample of the 39 Genetic Programming formulas.

In order to conclude the graphical comparison I also plot the formulas in the following time series graph.

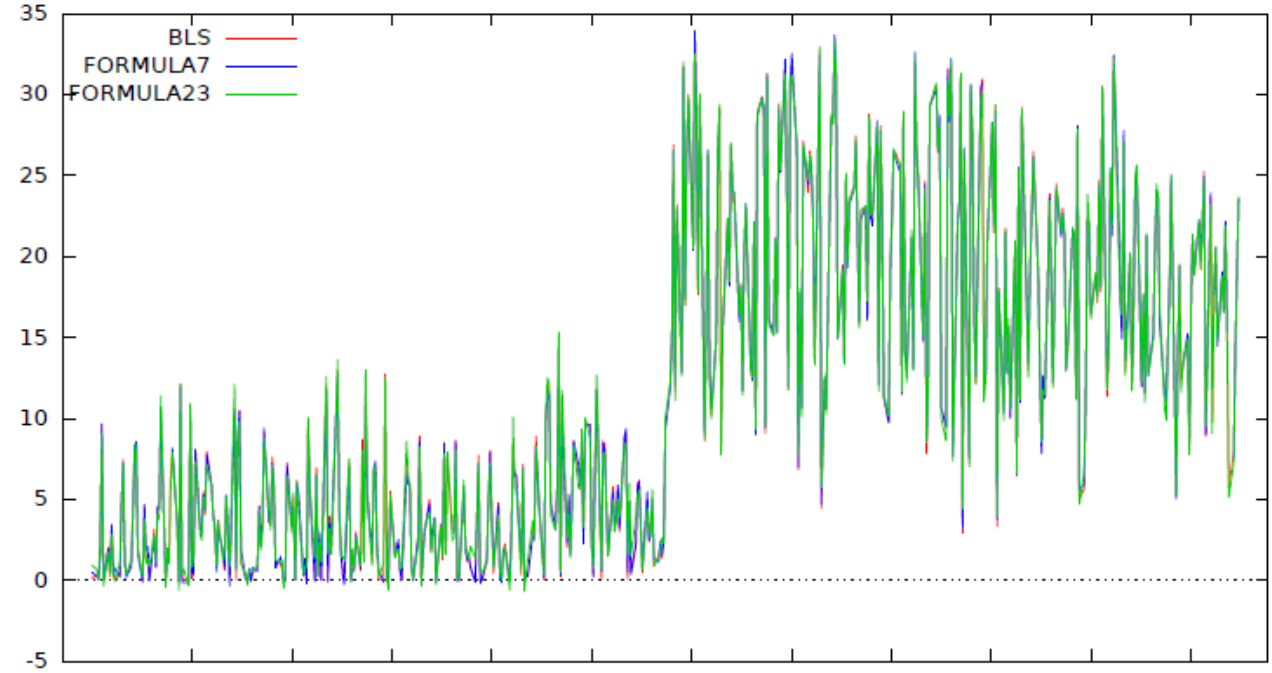

Table 5 – Graph comparing the trend of the Black-Scholes formula and the Genetic Programming Formulas 7 and 23

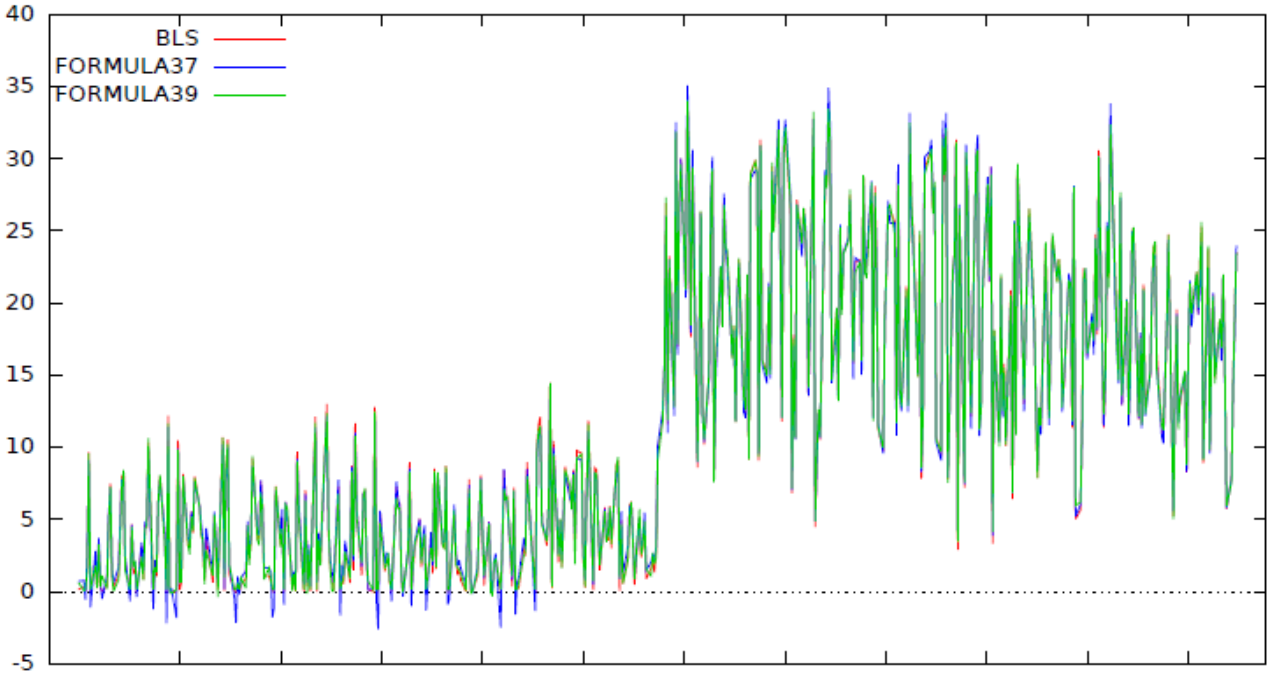

Table 6 – Graph comparing the trend of the Black-Scholes formula and the Genetic Programming Formulas 37 and 39

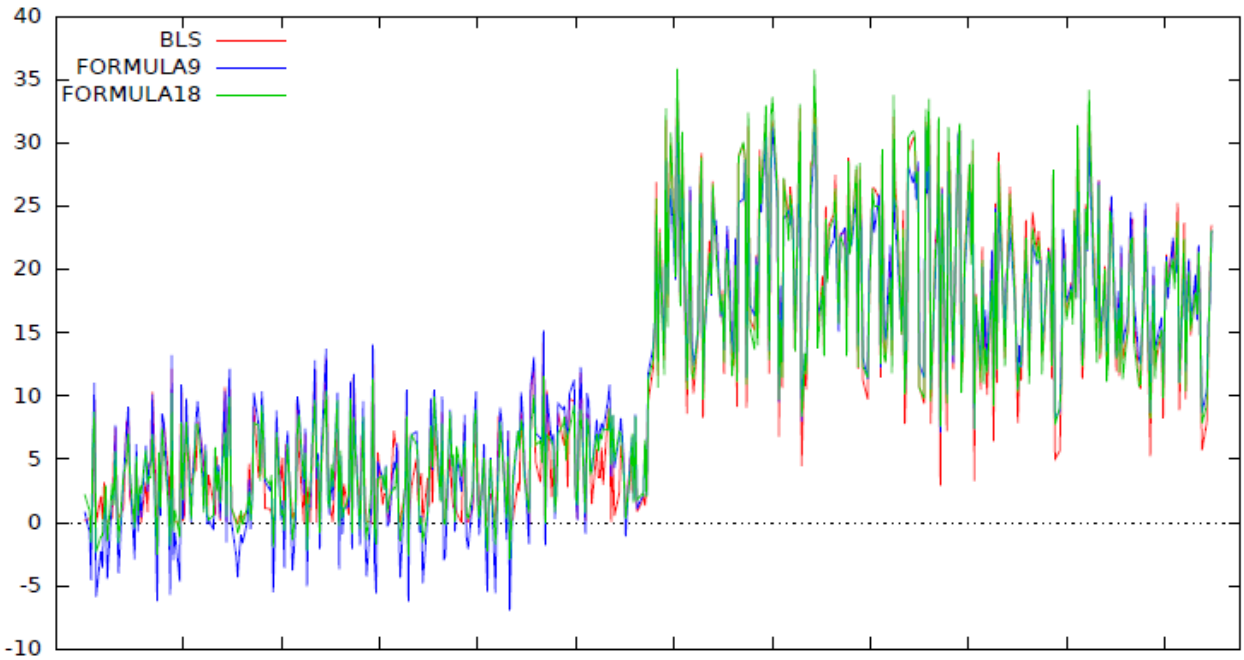

Table 7 – Graph comparing the trend of the Black-Scholes formula and the Genetic Programming Formulas 9 and 18

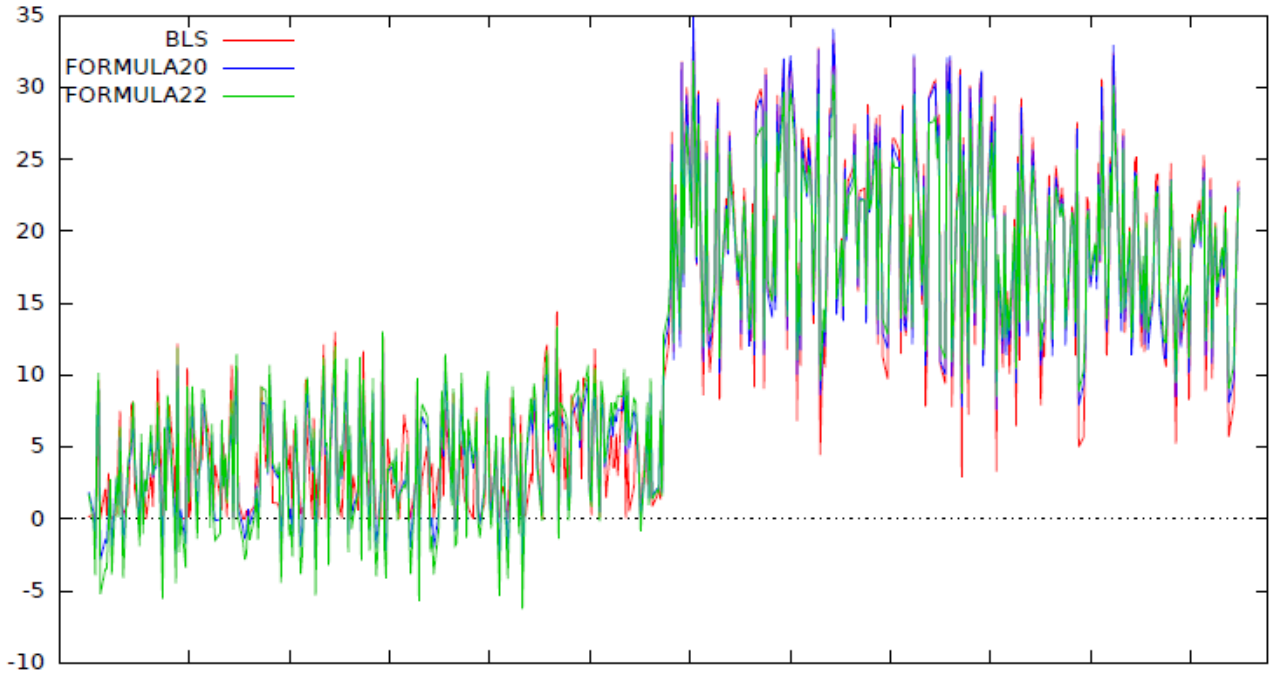

Table 6 – Graph comparing the trend of the Black-Scholes formula and the Genetic Programming Formulas 20 and 22

Table 5 and 6 graphically show the behaviors of the more performing Genetic Programming formulas, while Table 7 and 8 displays the graph of the less performing Genetic Programming formulas. The differences in behavior are clear. Prices obtained by the less performing formulas show more discrepancies with the prices obtained with the Black-Scholes formula, especially when considering the in-the-money options, which in the graphs are located in the left half of the graph. Curiously, the only formula among the more performing that shows some discrepancy is the Formula identified by the number 37, especially considering the in-the-money options, where it displays some negative values.

For the sake of completeness, I am interested in understanding also which is the number of Genetic Programming formulas that can be considered as more performing taking into

account the whole dataset of 39 formulas. According with the conclusions drawn from this second experiment, I chose to set as bounding constraints in order to define which formula can be considered performing the Fitness Value above 0.97 and the MSE below 750.

The formulas satisfying these constraints that can be considered more performing are 15 out of 39. The reason of this low ratio may be find in the fact that for 17 different setting combinations25 the Genetic Programming code repeatedly obtained the same 3 formulas, all of them displaying low fitness values and high MSE.

[25] Combinations of population size, maximum depth of the tree and maximum number of generations run.

Chapter 4
Conclusions

This work has been designed with the purpose of analyzing the approach and application of Genetic Programming in the financial world, with a special focus on the option pricing. Because of the strong connections between Genetic Programming and Genetic Algorithms, I started this work (Chapter 1) by presenting a wide overview of both of them, first contextualizing them in the world of Biologically Inspired Algorithms and Evolutionary Algorithms, and then introducing a comprehensive and complete description of their basic principle, their theoretical foundations and their structures.

I explained the fundamental concept of *survival of the fittest*, which has inspired this whole class of algorithms, and I introduced the concepts presented in the Holland's Schema Theorem, linked to both Genetic Algorithms and Genetic Programming. Following the paramount and pioneering work of John Koza in the field of Genetic Programming, I thoroughly presented a description of the elements that undergo the structure of Genetic Programming, analyzing the initial structure, its generative process, the fitness selection issue and all the primary and secondary operations.

Following, in Chapter 2, I introduced an overview of the existing literature of Genetic Algorithms and Genetic Programming in Economics and Financial fields. This chapter opens with a descriptive explanation of the potential capabilities of the Genetic Programming as problem-solving tool and the presentation proceeds with the introduction to the major papers and books published in the field of Genetic Algorithm and Genetic Programming application to the financial fields. The released literature that deals with this topic is extremely vast and various, covering all the possible financial applications, in particular the one related with financial derivative securities. I focused the attention on a peculiar argument, the option pricing. After covering a wide introduction to options, including all the features of this financial tool, I presented and explained the Black-Scholes formula, giving special attention to its mathematical implication and to the assumption underlying the application of this formula in the real world.

Starting from this point, I introduced seven papers I found particularly interesting. The first one, in particular, *A Nonparametric Approach to Pricing and Hedging Derivative Securities Via Learning Networks* published in 1997 by James M. Hutchinson, Andrew W. Lo, and Tomaso Poggio, became an inspiring work for further researches. In this first paper neither Genetic Algorithms not Genetic Programming are applied, but the other papers I took into consideration were all inspires by this work.

My third Chapter has been inspired by the papers I presented in Chapter 2, in particular by two researches. The first is the work realized by Shu-Heng Chen and Who-Chiang Lee, that published in 1997 their paper *Option Pricing with Genetic Algorithm: The Case of European-Style Options*, focused on the application of Genetic Algorithm in option pricing, in particular on European call options, and in which solutions are compared to the ones obtained from the Black-Scholes option pricing theorem. The second study is the work of Thomas H. Noe and Jun Wang, *The Self-Evolving Logic of Financial Claim*, published in 1997. In this paper, Genetic Programming has been used as an optimization technique to price financial instruments; the purpose is to show how easily Genetic Programming can approximate the Black-Scholes formula even when trained on small data sample.

Starting from these analysis, I decided to conduct an experiment on a set of simulated data with the aim of studying the capability of Genetic Programming in approximating the Black-Scholes formula for European call options in two different scenarios. The experiment has been devised in two different tests, the first with a reduced set of 7 inputs and the second with a larger set of 21 inputs. The main implication of this different set of variables was the fact that in the first experiment the Matlab code I used was not in the condition to replicate the Cumulative normal distribution function and the Logarithmic function, both part of the Black-Scholes formula, while in the second experiment I introduced both these function manipulating my dataset and introducing them directly through the calculation of the new input variables that I add.

As the assumptions underlying the Black-Scholes formula do not hold in the real world (especially the assumption according to which the underlying asset returns follow a normal distribution) I chose in the first experiment to not force the Genetic Programming in replicating the Black-Scholes formula calculations. Designed in this way, the first experiment

could easily be tested with real data, in the real world, where the Black-Scholes assumption are not proven to hold.

Considering the results that I obtained, the first test displayed a good capability in approximating the behavior of the Black-Scholes formula prices. Taking into consideration the best performing Genetic Programming formulas have shown performance closer to the Black-Scholes formula when considering in-the-money European call options. Considering the whole set 27 out of 48 genetic Programming formulas can be defined at least sufficiently good in approximating the Black-Scholes formula.

For the second test, instead, I immediately noticed how every Genetic Programming formula displayed high fitness values (almost all the formulas presented fitness values above 0.95) but also a much higher average MSE with respect to the values observed in the first test. Considering the GP formulas that displayed fitness values above 0.998 and MSE values under 200, they can approximate the Black-Scholes solution with a satisfying level of precision. Considering the whole database of 39 GP formulas obtained in the second test, instead, it is possible to state that 15 Genetic Programming functions display good performances.

According to the results that I found, further researches should be carried on, firstly exploring the settings combinations that I was not able to run and wider database, with more data, and, secondly running the Matlab code and testing the capabilities of Genetic Programming in option pricing in a real data set. In particular it would be really interesting see how the first test behave when applied to data from the real world, where Black-Scholes formula assumption do not hold.

References

Abid, F., Abdelmalek, W. and Hamida, S. B. 2012 "Dynamic Hedging Using Generated Genetic Programming Implied Volatility Models." INTECH.

Allen, F. and Karjalainen, R. 1999. "Using genetic algorithms to find technical trading rules." *Journal of Financial Economics* 51: 245-271.

Bauer, R. J. 1994. *Genetic algorithms and investment strategies*. John Wiley & Sons.

Beasley, D., Bull, D. R. and Martin R. R. 1993. "An overview of genetic algorithms: part 1, fundamentals." *University Computing* 15(2): 58-69.

Beasley, J. E., Meade, N. and Chang, T.-J. 2003. "An evolutionary heuristic for the index tracking problem." *European Journal of Operational Research* 148: 621-643.

Bianchi, L., Dorigo, M., Gambardella, L.M., Gutjahr, Walter J. 2008. "A survey on metaheuristics for stochastic combinatorial optimization

Black, F. and Scholes, M. 1973. "The pricing of options and corporate liabilities." *Journal of Political Economy* 81(3): 637-654.

Brabazon, A. and O'Neill M. 2006. "Biologically Inspired Algorithms for Financial Modelling." Springer Edition.

Brabazon, A. and O'Neill M. 2008. "Natural Computing in Computational Finance Vol. 1, 2, 3 and 4." Springer Edition.

Chen, S-H. and Lee, W-C. 1997. "*Option Pricing with Genetic Algorithm: The Case of European-Style Options."* Seventh International Conference on Genetic Algorithms. Michigan, State University.

Chen, S-H. and Lee, W-C. 1997. “Option pricing with genetic algorithms: a second report.” *International Conference on Neural Networks* 1: 21-25.

Chen, S-H. 2009 “Genetic Algorithms And Genetic Programming In Computational Finance.” Springer Edition.

Chen, S-H., Lee, W-C. and Yeh, C-H. 2009. “Hedging Derivative Securities with Genetic Programming.” International Journal of Intelligent Systems in Accounting, Finance & Management, 237–251.

Chidambaran, N. K., Lee, C. J. and Trigueros, J., 1998. “An Adaptive Evolutionary Approach to Option Pricing via Genetic Programming.” *Conference on Computational Intelligence for Financial Engineering*.

Chidambaran, N. K. 2003. “Genetic Programming With Monte Carlo Simulation For Option Pricing.” P*roceedings of the 2003 Winter Simulation Conference.*

Chiong, R. “Nature-Inspired Algorithms for Optimisation.” Springer.

Davis, L. 1991. *Handbook of genetic algorithms*. Von Nostrand Reinhold.

Deboeck, G. J. 1994. *Trading on the edge: neural, genetic and fuzzy systems for chaotic financial markets*. John Wiley & Sons.

Dempster, M. A. H. and Jones, C. M. 2001. “A real-time adaptive trading system using genetic programming.” *Quantitative Finance* 1: 397-413.

Folino, G. 2003. “Algoritmi evolutivi e programmazione genetica: strategie di progettazione e parallelizzazione” Consiglio Nazionale delle Ricerche, Istituto di Calcolo e Reti ad Alte Prestazioni (ICAR).

Glover, F. and Kochenberger, G. A. 2003 “Handbook of Metaheuristic.” Kluwer Academic Publishers.

Goldberg, D. E. 1989. *Genetic algorithms in search, optimization and machine learning*. Addison-Wesley.

Holland, J. H. 1975. *Adaptation in natural and artificial systems*. University of Michigan.

Hui, A. "Using Genetic Programming to Perform Time-Series Forecasting of Stock Prices"

Hull, J. 1997. "Options, Futures, and Other Derivatives." Pearson Education Inc.

Hutchinson, J.M., Lo, A. W. and Poggio, T. "A Nonparametric Approach to Pricing and Hedging Derivative Securities Via Learning Networks." *The Journal of Finance*, Vol. 49, No. 3, 851-889.

Iba, H. and Sasaki, T. 2002. "Using genetic Programming to Predict Financial Data"

Koza, J. R. 1992. *Genetic programming: on the programming of computers by means of natural selection*. The MIT press.

Navet, N. and Chen, S.-H. 2007. "Financial data mining with genetic programming: a survey and look forward." Available at *www.loria.fr/~nnavet/*.

Navet, N. and Chen, S.-H. 2008. "On predictability and profitability: would GP induced trading rules be sensitive to the observed entropy of time series?" In *Natural Computing in Computational Finance*, Springer.

Neumann, F. and Witt, C. 2010 "Bioinspired Computation in Combinatorial Optimization – Algorithms and Their Computational Complexity." Springer Edition.

Noe, T. H. 1997 "*The Self-Evolving Logic of Financial Claims*", Genetic Algorithms and Genetic Programming in Computational Finance, edited by Shu-Heng Chen, Springer Edition.

Poli, R., Langdon, W. B. and McPhee, N. F., 2008. "A Field Guide to Genetic Programming." ISBN 978-1-4092-0073-4 .

Potvin, J.-Y., Soriano, P. and Vallee, M. 2004. “Generating trading rules on the stock markets with genetic programming.” *Computers and Operations Research* 31: 1033-1047.

Riolo, R., Vladislavleva, E., 2013 Ritchie, M. D. and Moore, J. H. “Genetic Programming Theory and Practice.” Springer Edition

Rodgers, A. and Prugel-Bennett, a. 1999. “Genetic drift in GA selection schemes.” *IEEE Transactions on Evolutionary Computation* 3(4): 298-303.

Yin, Z., Brabazon, A. and O’Sullivan, C. 2007. “Adaptive Genetic Programming for Option Pricing”