Poznan University of Technology
Faculty of Computer Science
Institute of Computer Science

Master's Thesis

RESEARCH INTO IMPLEMENTATION OF EXPLANATORY MODELLING BASED ON GENETIC PROGRAMMING IN A DISTRIBUTED ENVIRONMENT OF APACHE SPARK

Jakub Guner, 106529

Supervisor
dr hab. inż. Krzystof Krawiec

This thesis is a part of research project:
"Explanatory modelling system for big data (LUCID)"
TANGO2/340026/NCBR/2017

Praca powstała w ramach projektu:
"Środowisko wyjaśniania i modelowania dużych zbiorów danych (LUCID)"
TANGO2/340026/NCBR/2017

Poznań, 2017

Contents

Chapter 1

Introduction

Computers are at the verge of taking over the role of scientists. In 2009, Cornell University researchers, Michael Schmidt and Hod Lipson, built a software system that synthesized natural laws from data [SL09]. The two authors explain the essence of their experiment:

> "the discovery of physical laws, from scratch, directly from experimentally captured data [...] without prior knowledge about physics, kinematics, or geometry."

What it means is that it is now sufficient to feed the computer with raw measurement data from the real life experiments and the software is capable of coming up with any function or equation that, firstly, binds the variables captured during the experiment, and secondly, embodies the natural law that governed the experiment in the first place.

Should scientists worry about their job security? Many other professionals certainly do. The technological unemployment has been tied to the development of computers from the very beginning. In the book *The Innovators* [Isa14], Walter Isaacson describes the origins of ENIAC – the first electronic, general purpose computer. At the time, the major concern was calculation of firing tables for American artillery during World War II. The task was initially performed by over 170 people who operated mechanical adding machines. Isaacson recounts:

> "women math majors were recruited from around the nation. But even with all of this effort, it took more than a month to complete just one firing table".

John Mauchly and J. Presper Ecker of University of Pennsylvania convinced the army to fund the construction of ENIAC, promising it would greatly reduce the amount of time required to prepare the firing tables. Eventually, the machine delivered:

> "it was able to perform five thousand additions and subtractions in one second, which was more than a hundred times faster than any previous machine".

As a result, dozens of mathematicians, at the time called *computers*, had to move on to other, less routine occupation. This trend continues today. IRRC Institute estimates that the automation of brick and mortar stores will lead to disappearance of at least 6 million retail jobs in the United States alone [SVC17]. The extreme version of the future is Amazon Go - a completely automated, cashier-less grocery store [ama17]. Autonomous cars are another example. It is predicted that about 4 million truck and cab drivers are going to loose their jobs [Sim17] to the likes of auto-pilot Tesla truck and driver-less Uber.

However, despite the marvels of machine learning and artificial intelligence in the examples above, is it possible for computers to replace scientists as well? Is it viable to tackle probably the most difficult, creative and demanding job out there? To answer this questions we need to understand the realm of scientific method and see which parts of it, if any, can be automated.

1.1 Scientific method

The goal of science is to provide testable and universally true explanations for the phenomena of the natural world. Scientists worldwide agree that *scientific method* is the best way to obtain unbiased and accurate description of reality. Scientific method is a series of steps that lead from the initial inspection to the final conclusions. It begins with the observation of an event or process and asking a question why it is the way it is. The next phase is to form a hypothesis, a possible explanation that may or may not be true. Then a hypothesis-based forecast is made and compared with reality through experiment. If the prediction and the result of the experiment match then the hypothesis can be accepted and becomes a law or a scientific theory. Otherwise it has to be rejected and a different one proposed and tested.

The crucial aspect of scientific method is that it offers two complementary ways of understanding reality: prediction and explanation. Science not only allows us to anticipate *what* is going to happen but also gives us tools to explain *why* and *how* it is going to happen. The explanations are embodied in *models* - simplified versions of reality that are comprehensible by humans. The process of constructing such models is called *explanatory modelling*.

In the context of this thesis, explanatory modelling is going to be restricted to synthesis of mathematical equations or functions that bind the physical quantities measured during the experiments. This is the part of the scientific method that can be automated and performed by the computer. Scientists are still required to choose an area of interest and perform an experiment. However, the task to :

> "extract some information about how nature is associating the response variables to the input variables" [Bre01]

as Leo Breiman, a renowned statistician and one of the pioneers of machine learning, put it, can be ceded to the computers. For example, let's say we have measurements of four physical quantities of objects moving in the gravitational field. The computer would be able to synthesize the equation 1.1 that holds true for all data points.

$$x_1 = \frac{6.674 * 10^{-11} * x_2 * x_3}{x_4^2} \tag{1.1}$$

In this form, the equation is vague and difficult to interpret. It needs to be cleaned and transformed before being presented to the scientific community. The x_2 and x_3 are masses of two interacting objects and x_4 is the distance between them. They should be given more convenient names: m_1, m_2, and r respectively. Then, x_1 is the resulting force of attraction between the two objects; the better symbol would be F. Finally, the value of $6.674 * 10^{-11}$ is a universal constant, so it should be named as the *gravitational constant*, given proper units ($m^3 kg^{-1} s^{-2}$) and replaced with the symbol G. Thus, the Newton's law of universal gravitation was synthesized (1.2).

$$F = \frac{G * m_1 * m_2}{r^2} \tag{1.2}$$

The natural processes can be extremely complicated and diverse. They can be described in a number of ways with varying levels of accuracy and complexity. A lot of data is also required to capture and understand their intricacies. For example, the particle detectors of Large Hadron Collider at CERN can deliver about 25 GBs worth of data per second [CER17]. Hence, any explanatory modelling software that hopes to be used in large-scale, practical applications needs to answer the following three fundamental questions:

- How to come up with *any* model?
- How to produce the *good* models?
- How to have *a lot of* good models to choose from?

In 2016, a group of computer scientists from Poznan University of Technology, Poland realized that they have answers for the aforementioned questions that can benefit not only the scientific community, but also businesses and the general public. The story of turning their ideas into practice unfolds in the next section.

1.2 The two pillars of Data Science

Data Science is a process of applying scientific method to any field of interest that one can gather data about. In contrast to natural sciences, which focus on universal laws and repeatability, Data Science is interested mainly in rules and regularities that are exclusive to the environment the data are collected in. Thus, it is widely used in business to optimize unique operations and to discover opportunities in particular segments of market.

The field of Data Science is built upon two paradigms. The first of them is Machine Learning (ML) - "a field of study that gives computers the ability to learn without being explicitly programmed" [Sam59]. It stems from the research on Artificial Intelligence and was defined in 1950s by Arthur Samuel, an IBM computer scientist who experimented with the software that could play checkers. Samuel wanted the computer to play the game better than he did, so instead of programming the reactions for possible combinations of pieces, he devised an algorithm for assessing the state of the game and consequences of available moves. The key to success was to allow the software to play many games that generated a lot data that made the computer better and better with each iteration. Today, Machine Learning is a wide field that deals with the problems of, among others, classification, regression and clustering. And still, over 60 years later, the principle remains the same: give the computer access to a large amount of information so that it figures what should happen next. However, in most cases these insights are unavailable for humans to analyze and directly benefit from, as they are encoded only in machine-readable form. We shall turn to a different paradigm for help.

The second pillar of Data Science is Data Mining (DM) - "the *process* of discovering interesting patterns and knowledge from *large* amounts of data" [Han05]. The aim of Data Mining is to offer humans clear, concise and useful intelligence from the data they already collected in their databases. It is a stark contrast to Machine Learning, where the discovered knowledge is assumed to be hidden. The Data Mining techniques include, among others, rule induction, frequent sequence mining and market-basket analysis. However, the discovered patterns are useless if they are not interpreted by humans and result in no action or change of operations. That often requires dedicated personnel of data analysts and generates extra costs. In the end, the majority of Data Science today is centered around Machine Learning, leaving database administrators, corporate managers and end-users ignorant of *why's* and *how's* of decisions made by modern smart software.

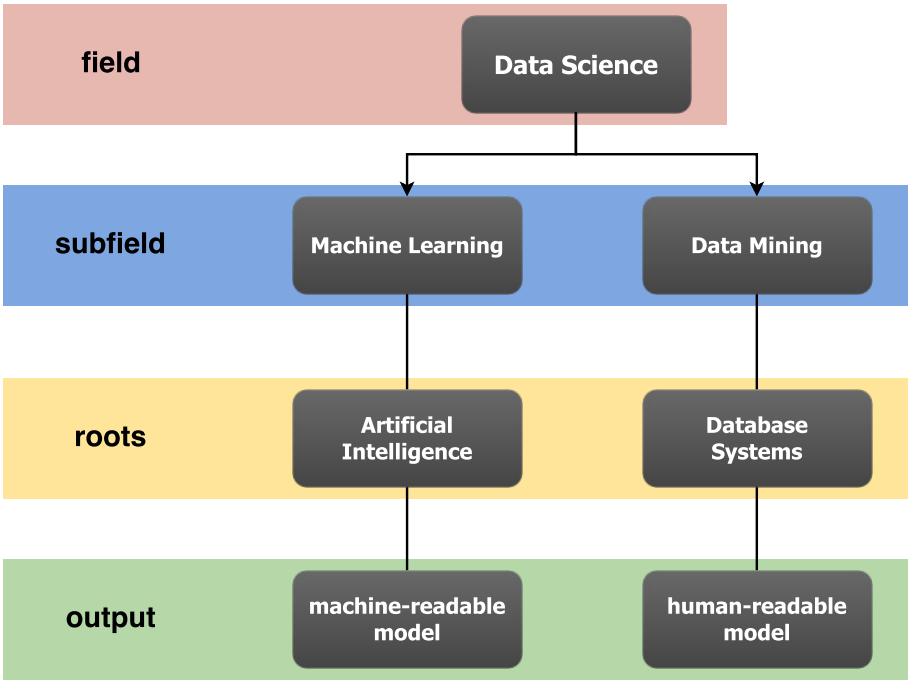


Figure 1.1: The overview of the field of Data Science

The contrast between the predictive power of popular Machine Learning algorithms and limited availability of reasoning behind automated decision-making processes, inspired the researchers of Poznan University of Technology, Poland to change the *status quo*. They decided to utilize the recent advances in Computer Science in an attempt to build a new system that could achieve both goals: prediction accuracy of ML models and clarity of DM ones. The project was named Lucid and uses explanatory modelling, as defined in Section 1.1, to tackle the problem of limited transparency. What is more, in order to handle large volumes of data, Lucid is to be implemented in a distributed environment of Apache Spark [Fou17b] - one of the most popular cluster-computing frameworks.

Lucid was approved for funding by the Polish National Centre for Research and Development [fRD16], with the goal of making the end product "available to entities from business and academia" [IoCS17]. However, before the proper development began in full speed, a prototype was required to validate the proposed approach to *explanatory modelling*. The prototype is going to answer the question of coming up with *any* model with *symbolic regression* - a method of generating random mathematical expressions that makes no *a priori* assumptions about the shape or complexity of the function. As a result, any mathematical equation or function is possible. The task of producing the *good* models is going to be fulfilled with *Genetic Programming* (GP) – a metaheuristic, that mimics the Darwinian evolution to create functions that best fit their environment, the collected data. Apache Spark is going to allow to have *a lot of* good models to choose from. The scope of Lucid's prototype is going to be presented in the next section.

1.3 Goal and scope of the thesis

The hypothesis put forth in this thesis is that **the efficient implementation of explanatory modelling system based on genetic programming is possible on top of Apache Spark**. Such framework is expected to improve the quality of the results with the increase of the amount of training data and the size of GP population, while preserving good scalability. However, fine tuning of the GP model and optimization of the distributed deployment are beyond the scope of this thesis.

This project serves as a bridge between the science of GP and the engineering of distributed systems. As a result, the process of pairing the abstract algorithmic concepts with specific tools and programming libraries forms the core task of the thesis. Each part of GP can be implemented in a number of ways, and not always the first attempted combination is the best one. Hence, the following chapters will include the evaluation of solutions proposed for each element of GP paradigm. The detailed goals of the thesis include:

- analysis of the internal components of Apache Spark,
- evaluation of available data structures suitable to store the GP's population,
- implementation of a prototype system that specializes in symbolic regression,
- experiments with hand-crafted and real-life data,

The thesis consists of six chapters, including this introduction. Genetic Programming is introduced in Chapter 2. Apache Spark and its components are presented in Chapter 3. Chapter 4. covers the implementation of the prototype and the results of the experiments are presented in Chapter 5. The thesis concludes with evaluation of the project in Chapter 6.

Chapter 2

Genetic Programming

Technological unemployment is one of the issues studied by the MIT Initiative on the Digital Economy [otDE17]. The two leaders of the Initiative, Erik Brynjolfsson and Andrew McAfee, explain how the Industrial and Digital Revolutions differ in taking advantage over human workers: "Computers and other digital advances are doing for mental power [...] what the steam engine and its descendants did for muscle power" [BM14]. The automation of muscle power during Industrial Revolution was easy to control and predict because it was *imperative* – the machines and engines did exactly and only what they were tasked with by their operators. On the other hand, the automation of mental power in the time of Digital Revolution is becoming more and more unpredictable as it is *declarative* – only the goal of computation is specified and the computer is given the freedom to find the best way to achieve it. In order to better understand this process, Brynjolfsson and McAfee name two aspects of mental power the computers are trying to replicate: ability to shape the environment and ability to understand it.

The first ability, influencing the environment, is accessible to computers by means of Artificial Intelligence (AI). Stuart Russell and Peter Norvig explain that the AI is based on *rational agents* that "operate autonomously, perceive their environment, persist over a prolonged time period, adapt to change, and create and pursue goals" [RN03, p. 4] to achieve the best possible result. Machine Learning is a branch of AI that tackles the challenge by allowing the agents to learn from data, as explained in Section 1.2. The insights derived from the observations of the environment allow ML systems to respond to new, previously unseen events. In the last decade, this approach has gained in importance and popularity both in the academic circles and in the business operations.

Nevertheless, the capacity to understand the environment, the second aspect of mental power, remains elusive. In order to deliver models of reality that are accessible both to humans and machines, we have to turn to a rather niche branch of ML – Genetic Programming (GP). It is a type of *evolutionary computation* – a paradigm "based on the mechanics of natural selection and natural genetics", that combines survival of the fittest with "structured yet randomized exchange to form a search algorithm with some of the innovative flair of human search" [Gol89, p. 1]. Just like the evolution in nature can create species that are well adapted to their environment, GP can devise executable models that fit the environment of observations "without requiring the user to know or specify the form or structure of the solution in advance" [PLM08].

There are three main features of GP:

- It maintains a working population of candidate solutions
- Genetic operations, like mutation and crossover, are used to alter candidate solutions
- Each candidate solution is a computer program (or other executable entity)

This chapter will cover all three of the above characteristics of GP. Each feature will be explained in detail and the corresponding place of GP in the hierarchy of algorithms will be presented along the way.

2.1 Population-based, biologically inspired metaheuristic

The synthesis of mathematical model that embody, or at least closely approximate the natural law, is a type of *optimization problem*. A finite set of variables can be used to construct a countably infinite set of mathematical formulas. The goal of the optimization is to find the formula that matches the data best and makes generalizations that allow it to overcome measurement errors. In general, such problems can be solved using exact, approximate and heuristic algorithms. This thesis focuses on *symbolic regression* a heuristic algorithm, that will be presented in detail in section 2.3. However, as heuristics are problem-specific, we need to cover metaheuristics first, since they provide higher-level abstractions and are problem-independent.

Sean Luke of George Mason University, explains that key feature of metaheuristics is that they use "some degree of randomness to find optimal (or as optimal as possible) solutions to [...] *I know it when I see it* problems" [Luk13, p. 9] – the ones where it is difficult to arrive with a good model, but being given a model, it is relatively easy to assess its quality. There are a few main families of metaheuristics: local search, simulated annealing, tabu search, and population-based approaches. Evolutionary computation is the most popular approach within the last group. In this section, two features of EC will be presented. First, single-solution algorithms will be contrasted with the population-based metaheuristics. Then, the biological inspirations of EC will be explained. The section will conclude with an attempt to devise a hierarchy of algorithms and metaheuristics.

2.1.1 Single solution vs population

The goal of exact algorithms is to use the input data and a set of conditions to produce some output. For example, sorting algorithms accept an array of numbers and a sorting order (ascending or descending) and return the same array but with the numbers rearranged in the desired permutation. Quicksort algorithm [Hoa61] uses the divide-and-conquer principle to break the full set of numbers into smaller and smaller sets that can be easily sorted through swapping. Another example is the knapsack problem [Mat96], where a subset of elements has to be chosen that maximizes the total value of selected elements but satisfies the limit imposed on their weight. The recognized algorithm for the knapsack problem is the one driven by dynamic programming [Pis97]. It starts with an empty solution and uses the bottom-up approach to construct the final one step-by-step, expanding the initial one along the way. However, like many other classical approaches, quicksort and knapsack algorithm iteratively improve only a single solution to the problem being posed. There are also single-solution metaheuristics, like local search, tabu search and simulated annealing, that randomly pick a complete, initial solution and explore its *neighborhood* – a set of solutions that slightly differ from the starting point. One solution from from the neighborhood is selected and process repeats.

On the other hand, population-based metaheuristics do not construct one, final result, but instead they maintain a set of complete, feasible solutions. The population is initially constructed at random, and the quality of solutions varies from very poor to very good. The job of the algorithm is to guide the population towards the final, adequate results. Particle Swarm Optimization (PSO) [KE95] is one of the best-known examples of this approach. PSO maintains a population of candidate solutions, represented as *particles*, that move in the virtual search space looking for an optimal position. Their movement is influenced by the best-so-far known coordinates and the location of neighboring particles. PSO does not guarantee to find the globally best solution, but ensures that the very good one is found quickly and efficiently.

2.1.2 Biological inspirations

Many population-based metaheuristics, if not all of them, are bio-inspired. In this thesis, a basic classification of them is introduced. The first group is evolutionary computation that is modeled after the Darwinian theory of evolution and the science of genetics that developed in the 20th century. Evolutionary computation begins with creation of the initial, random population of candidate solutions, known also as the first *generation*. The weakest individuals are eliminated in the selection stage, directly inspired by *natural selection*. The remaining individuals serve as parents for the next generation. The process continues until the evolving "species" adapts well to the imposed requirements.

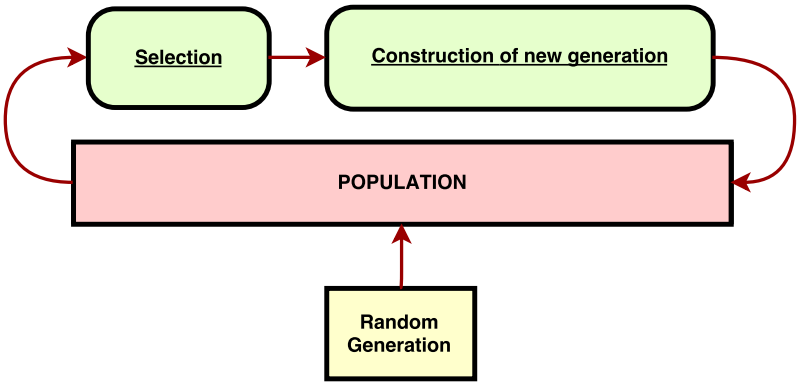


Figure 2.1: The basic steps of evolutionary computation

The other class of meteheuristics is inspired by the behaviour of groups of animals and the interactions between the individuals in those groups. Hence, the name *behavioral computation* is proposed. Peter Miller, the author of the book *Smart Swarm*, explains the principle of the method: "a group of individuals who respond to one another and to their environment in ways that give them the power, as a group, to cope with uncertainty, complexity and change" [Mil10]. PSO, introduced in the previous subsection, is a type of behavioral computation as "it's modeled not after evolution per se, but after swarming and flocking behaviors in animals" [Luk13, p. 55].

2.1.3 Hierarchy

Algorithms and meteheuristics can be classified based on their inspirations, purpose, paradigms or even time complexity. The classification attempt presented at Figure 2.2 is not meant to be complete or exhaustive. It rather aims to help understand the place of GP in the context of population-based meteheuristics and how it relates to other techniques.

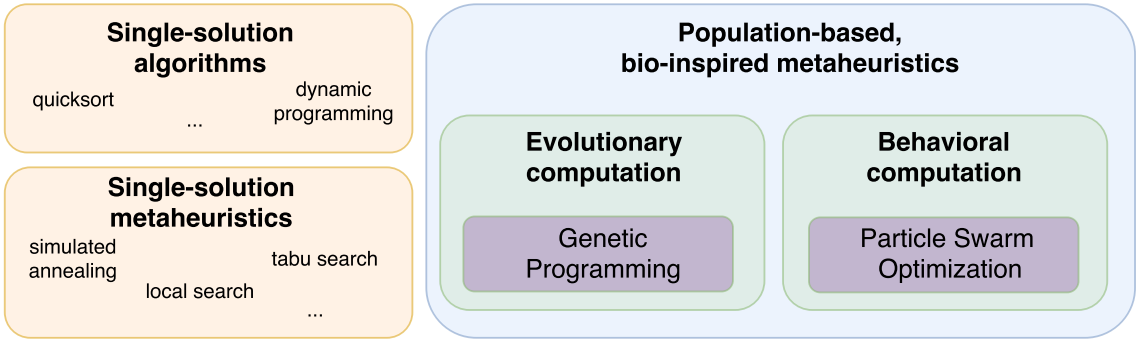


Figure 2.2: Hierarchy of algorithms: single solution vs population based

2.2 Evolutionary computation

There are three main variants of evolutionary computation: Genetic Algorithm, Evolutionary Strategies and Genetic Programming. They all differ in the way they represent candidate solutions, but they all share the concept of generations and the set of genetic operations that transform one generation to the other. This section introduces *selection* as a way to differentiate between poor and well performing candidate solutions. Then, *mutation*, *copying* and *crossover* are presented as means of producing offspring with parents' genetic material. The section ends with an updated diagram of evolutionary computation.

2.2.1 Selection

The goal of natural selection is to ensure that only individuals with good traits are able to pass on their genes to the next generation. The traits, like strength, resistance to diseases or defensive mechanisms, are known as *phenotype* – the visible manifestation of *genotype*, the genetic information encoded in DNA. The usefulness of each trait is determined only in the context of environment – brown fur may be beneficial in woods and mountainous areas, but it is undesirable in polar regions. In consequence, the genotype first has to be mapped to phenotype, then the quality of the traits is assessed in the ecosystem and only then, the strong and healthy individuals are able to live long enough to find a mate and pass on their genes. As a result, the entire population adapts to the environment.

Natural selection is replicated in evolutionary computation in two stages: evaluation and selection. In the evaluation stage, the information encoded in the genome, be it a binary string or an abstract syntax tree, is translated into a formal object (executable expression, graph, vector, trajectory) that is then evaluated and tested against training data or during simulations. The score achieved during the tests is called *fitness*, and it summarizes the degree of adaptation to the environment.

The most popular form of selection is *tournament selection* – "a number of individuals are chosen at random from the population. These are compared with each other and the best of them is chosen to be the parent" [PLM08]. Tournament selection not only ensures that "a single extraordinarily good program cannot immediately swamp the next generation with its children" [PLM08] but also surprisingly well recreates mating from nature. To become a parent, one does not have to be the best in the entire population, but rather the best in the random sample of individuals.

2.2.2 Mutation

Random mutations in genome happen regularly in nature. Usually they are undesirable, especially when it comes to humans, as they often lead to diseases like color blindness or haemophilia. However, from the perspective of evolution, they can be justified as a way to introduce new features and abilities to an already established population. One of the most vivid examples are polar bears, close relatives to brown bears. In the distant past, one random mutation allowed them to loose pigment in their fur making it translucent, but appearing white from the distance [LLF$^+$14]. The individuals with the mutation benefited from better camouflage and improved heat circulation, as the sun rays could penetrate their translucent fur and heat their skin, which, in fact, is black. That gave them better chances of survival in polar regions and finding a mate. Soon, their genes, and mutation they carried, dominated the whole species.

Mutation in the context of evolutionary computation is defined as a (possibly randomized) function $f : S \Rightarrow S$, where S is the solution space. In contrast to nature only one parent is required to generate mutated offspring. Sometimes, the mutation is not applied and the genes pass on unchanged to the next generation. It is known as *copying*, where an individual outlives its siblings and is still a viable parent during the next generation.

2.2.3 Crossover

Most species rely on sexual reproduction, rather than self-replication, to produce offspring. The necessity for two parents of different sexes has been studied extensively by biologists. Richard Dawkins, in his famous book *The Selfish Gene*, gives a viable explanation for origins of separate sexes, he names, for the sake of argument, A and B:

> "Time and effort devoted to fighting with rivals cannot be spent on rearing existing offspring, and vice versa. Any animal can be expected to balance its effort between these rival claims. The point I am about to come to is that As may settle at a different balance from the Bs and that, once they do, there is likely to be an escalating disparity between them" [Daw89, p. 300]

The specialization of sexes is an interesting field of study on its own. However, the power of sexual reproduction comes from the fact that it enables *genetic recombination* – random mix of genes from both parents. Individual genes do not work in isolation. More often than not, it is their specific combination that matters. Each parent may have have a good trait on its own, but their combination in the offspring may be much more than the sum the two.

The EC's equivalent of genetic recombination is *crossover* – "the creation of a child program by combining randomly chosen parts from two selected parent programs" [PLM08]. It simplifies the process as it does not require the parents to be of different sexes – any two individuals that pass the tournament selection can be involved in crossover.

When the execution of genetic operations: mutation, copying and crossover is finished, the old generation is removed and the new one takes its place. Figure 2.3 presents the "circle of life" within evolutionary computation.

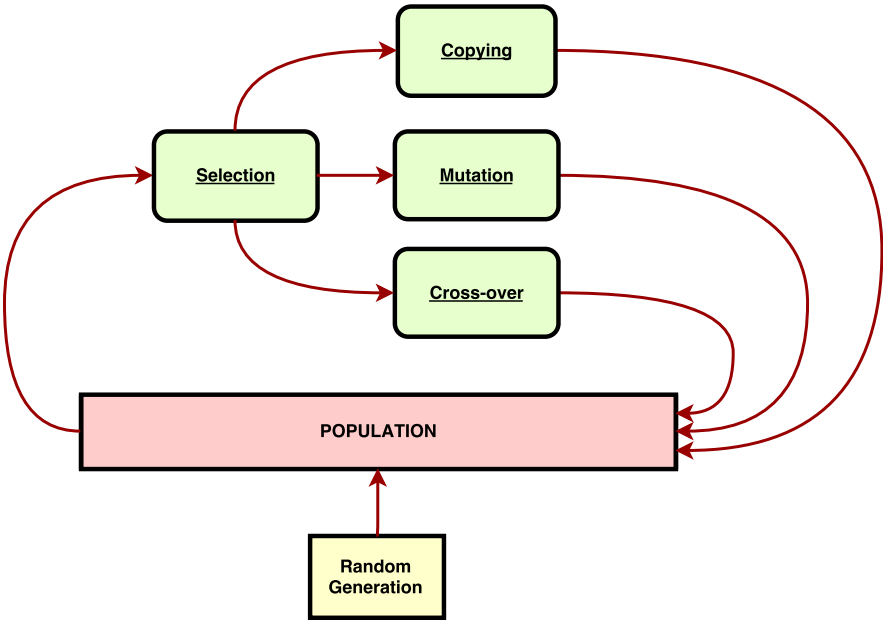
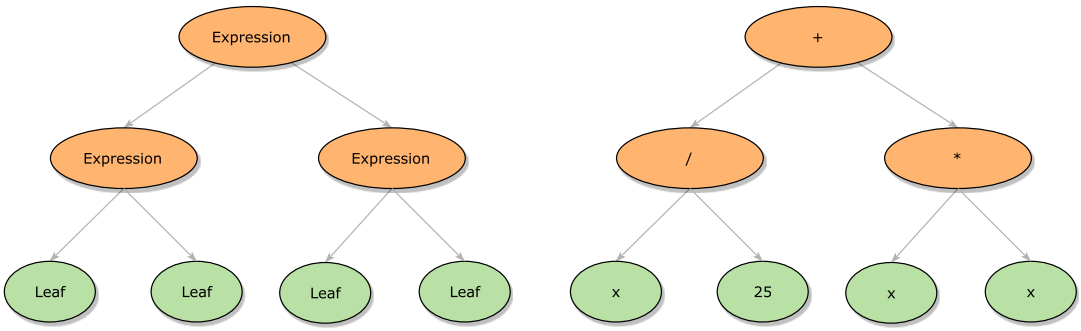


Figure 2.3: Genetic operations in the context of evolutionary computation

2.3 Genetic Programming and Symbolic Regression

Genetic Programming is a variant of evolutionary computation, where a population of *computer programs* is evolved. Each *program* is usually represented as an *abstract syntax tree* (AST) – a tree equivalent of source code that maintains the semantics. The nodes of the tree may implement fundamental building blocks of any programming language: variables, constants, operations, conditional expressions and loops. Common applications of GP include design of controllers, synthesis of boolean circuits and Symbolic Regression. GP is known for its versatility as it "has been successfully used as an automatic programming tool, a machine learning tool or an automatic problem-solving engine" [PLM08].

This thesis is limited to Symbolic Regression (SR), a type of problem where candidate solutions in GP populations are mathematical formulas that are required to map an input real-valued vector to an output one. SR, in contrast to e.g. linear or polynomial regression, makes no assumptions about the shape of the function and any model, no matter how complex, may be a viable solution. Mathematical expressions can easily be expressed with a subset of building blocks of a programming language. Hence the ASTs will contain only numerical variables and constants, and binary algebraic operations that are present in all programming languages like addition, subtraction, division and multiplication (see Figure 2.4b).



(a) A generic tree (b) AST equivalent of $x^2 + \frac{x}{25}$

Figure 2.4: Abstract syntax trees of depth 2

2.3.1 Genetic operations in GP

Genetic operations were introduced in section 2.2. The representation of programs as ASTs enforces GP-specific versions of these operations. The initialization stage may take two forms: *Full* method and *Grow* method. In the Full method all trees are balanced and have fixed depth "by restricting the selection of the label for points at depths less than the maximum to the function set F, and then restricting the selection of the label for points at the maximum depth to the terminal set T" [Koz92]. The Grow method allows any subtree to terminate with leaves at any depth within the set limit "by making the random selection of the label for points at depths less than the maximum from [...] the union of the function set F and the terminal set T, while restricting the random selection of the label for points at the maximum depth to the terminal set T" [Koz92].

The popular combination of these two methods is known as *ramped half-and-half* – "we usually do not know (or do not wish to specify) the size and shape of the solution in advance. The ramped half-and-half generative method produces a wide variety of trees of various sizes and shapes" [Koz92]. Mutation can be either *point*, where only a single randomly chosen node is mutated or *subtree*, where the whole randomly selected subtree is replaced with a new, randomly generated one. Finally, *subtree crossover* is implemented as swapping of two randomly selected subtrees between parents.

Chapter 3

Apache Spark

Writing a large-scale explanatory modelling (see section 1.1) system is a task that actually encompasses two distinct areas of Computer Science. The first of them is naturally the aspect of data analysis, either in the form of Genetic Programming or any other Data Science technique that may be of interest in the future. The second part of the task is the down-to-earth implementation of the system in a distributed fashion, having taken into account the networking, fault-tolerance and load-balancing between the machines in a computing cluster. The effort to implement both aspects of such system would not only be enormous but turns out not to be necessary. At a very early stage of Lucid's design, a decision was made to inherit the distributed processing functionality from an existing framework and focus development efforts on the data analysis part.

Apache Spark [Fou17b] was chosen as the base for Lucid. Spark is an open-source, cluster-computing framework, created by Matei Zaharia, of UC Berkeley, in the late 2000s and early 2010s [ZCF+10]. Spark addressed the criticism of earlier distributed systems by introducing new method of storing data across many processing units and a new model of transformation-based computations, that proved to be much faster than the previous solutions. As of 2017, Apache Spark is one of the most popular *big data* frameworks, used in production in many large companies and scientific institutions [Fou17a]. Three times a year, Spark Summits gather thousands of developers, both in the United States and Europe [Dat17a]. Even large corporations began to release their own Spark-native libraries. Intel published BigDL: Distributed Deep Learning Library for Apache Spark [Int17] and Microsoft released Microsoft Machine Learning for Apache Spark (MMLSpark) [Mic17]. The goal of Lucid is to continue this trend and reach the vast community gathered around Apache Spark, while extending its functionality with efficient implementation of selected methods of explanatory modeling, in particular GP and rule-based models.

This chapter introduces Apache Spark. First, Scala, the implementation language of Apache Spark, is presented. Then, the general architecture of Spark is shown, followed by the description of distributed data collections available in Spark. The chapter concludes with an overview of GraphX - a built-in graph processing library that is essential for the implementation of the part of Lucid's functionality that relies on Genetic Programming.

3.1 Scala

Scala [dL17] is a general-purpose programming language, originally developed in early 2000s by Martin Odersky of École Polytechnique Fédérale de Lausanne, Switzerland [Ode06]. Simplicity and programmer's productivity were the main goals for Odersky and his team. Hence, Scala combines the object-oriented and functional programming paradigms and keeps the resulting source code short and concise. The language was designed to run on top of the Java Virtual Machine (JVM) and is thus compatible with existing Java libraries. Even though Scala originated as a purely academic endeavor, it quickly became popular in the industry [OR14].

Matei Zaharia, the creator of Apache Spark, reveals why the cluster-computing framework was implemented in Scala. It turns out that the team at the Berkeley's AMPLab made the programmers' productivity their main priority, just like the scientists in Switzerland. They decided to deliver simple-to-use APIs, compatibility with existing Big Data tools, and an efficient runtime

Figure 3.1: Exhibition floor of Spark Summit Europe. Amsterdam, 2015

environment [ZCF+10] [Zah14]. Scala seemed like a natural choice. Its short syntax enabled concise API of Spark. The interoperability with JVM technologies made the integration with Java-based Hadoop ecosystem possible. And finally, the fact that Scala functions can be serialized, sent across the network and executed remotely, allowed for an efficient distributed system. The success of Apache Spark and its vision of making data-processing simple did not go unnoticed by the creators of Scala. In 2015, Martin Odersky named Apache Spark the "ultimate Scala collections" [Ode15].

3.2 Fundamentals of Apache Spark

Within eight years since its inception, Apache Spark became a unified engine for processing tabular and graph data both in batch and streaming modes. The overall architecture of Spark's components is presented in Figure 3.2. Spark Core API forms the basis of the framework as the remaining components are built on top of it. Spark SQL enables querying distributed datasets as though they were a relational database. The Streaming component provides tools for processing streams of data, and as of Spark 2.2 it is considered not only the easiest open-source streaming framework, but also the fastest [ZHA17]. MLlib package contains Machine Learning algorithms and exposes a unified Pipeline API. GraphX is responsible for distributed processing of graphs. It is essential for the implementation of GP in this thesis and will be presented in detail later in this chapter.

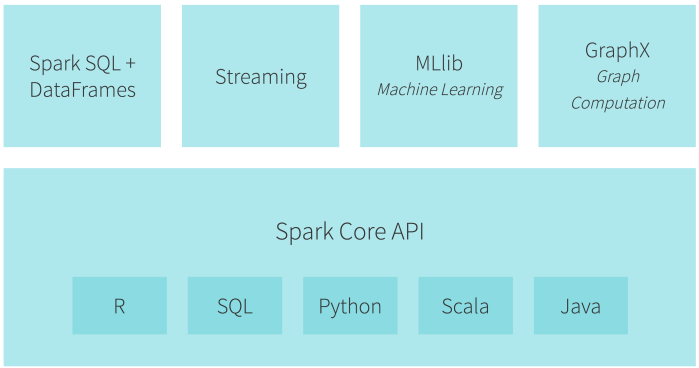


Figure 3.2: Architecture of Apache Spark as presented in [Dat17b]

The other aspect of Apache Spark that makes it appealing to developers is that it is available in as many as five programming languages. Spark SQL, Spark Streaming and MLlib are supported by all five: Scala and Java as native JVM languages, SQL as embedded code, Python through *pyspark* package and finally R by means of *SparkR* and *sparklyr* projects. However, the API of GraphX, the latest addition to Spark, is currently only available in Scala.

3.3 Distributed collections

The goal of Apache Spark is to provide fast and easy API for processing data in a cluster. The best way to do it is to distribute data across the machines and execute operations in parallel. It is possible thanks to *Resilient Distributed Datasets* (RDDs) – a fundamental collection within Apache Spark. RDDs are "based on coarse-grained transformations (e.g., map, filter and join) that apply the same operation to many data items" [ZCD+12]. It means that typically the programmer only needs to write a simple function that operates on a single record and it will be applied to all records in parallel. What is more, such functional approach allows RDDs "to efficiently provide fault tolerance by logging the transformations used to build a dataset (its lineage) rather than the actual data" [ZCD+12]. As a consequence, in case of a hardware failure, the lost data can be easily recomputed using original data source and the saved history of transformations.

However, RDDs are not free of drawbacks. They offer ambiguous access methods (numeric indexes) and lack type-safety. Consequently, they are now considered low-level API and the developers are discouraged from using them directly. The alternatives include *DataFrames* and *Datasets*, powerful expansions built on top of RDDs. DataFrames are inspired by *dataframes* known from R and Python's Pandas [pan17]. They introduce schema and basic type checking and can be queried with SQL just like a table from a traditional relational database. *Datasets*, on the other hand, are statically typed (e.g. `Dataset[Person]`) to provide compile–time type–safety and ease of use typical for local collections (e.g. `List[Person]`). For more information, please refer to *Spark Programming Guide* [Fou17c] and *Spark SQL, DataFrames and Datasets Guide* [Fou17d].

3.4 GraphX

Apache Spark was designed to cover a wide spectrum of workflows that involve processing of business or research data. However, many practical use cases are based not on independent data points but on graphs of multiple interconnected entities. The primary examples include social networks and chemical compounds. Such graphs do not easily fit into the tabular form of `RDDs` or `DataFrames` and their processing involves exchange of data between neighboring nodes. In order to accommodate this new approach, a graph-focused library was needed within Apache Spark. The good approach was not to build it from scratch, but instead use existing data structures, such as `RDDs`, as an efficient back-end. Hence, GraphX [XGFS13] – a built-in, full-feature graph processing library – was written using 12 times less lines of code than Spark Core itself [GXD+14].

Graph computations can take two forms. The first of them is a one-time transformation of the graph with an external function, e.g. selection of a subgraph. The other is iterative exchange of messages between the nodes, that, upon arrival, may change the properties of the destination node. Hence, to allow the execution of the first form of computation, GraphX enables the user to create a property graph – a "directed multigraph with user-defined objects attached to each vertex and edge" [Fou17e]. The processing is based on transformations, like in the case of `RDDs`, but here programs "describe transformations from one graph to the next either through operators which transform vertices, edges, or both in the context of their neighborhoods (adjacent vertices and edges)" [XGFS13]. This approach allows to store records in a way that reflects the relationships between them and to process data in a neighborhood-aware manner.

On the other hand, message passing algorithms tend to exhibit recursive behaviour "as properties of vertices depend on properties of their neighbors which in turn depend on properties of *their* neighbors" [Fou17e]. Direct implementation of such algorithms is challenging as the stop conditions may be difficult to observe. Pregel [MAB+10], a model of recursive graph computations, was proposed by Google engineers to alleviate those difficulties. With Pregel, the programmer defines what data shall be passed between the adjacent nodes and what each node should do with the incoming messages. The computation terminates when no messages are sent anymore. GraphX provides an optimized variant of Pregel and it proves to be very easy to use. The built-in implementation of PageRank [PBMW99], the most popular approach to scoring the importance of nodes in a graph (e.g. web pages on the Internet), requires only 20 lines of code [GXD+14].

Chapter 4

Genetic Programming on Apache Spark

The authors of *A field guide to genetic programming* concluded that:

> "one of the reasons behind the success of GP is that it is easy to implement own versions, and implementing a simple GP system from scratch remains an excellent way to make sure one really understands the mechanics of GP" [PLM08].

This chapter describes the core of the thesis – implementation of a GP system in a distributed environment of Apache Spark. SparkGP shall be the name of the system from now on. SparkGP is implemented in Scala, integrates tightly with the API of Apache Spark and follows the principles of functional programming. Immutability is achieved by means of *case classes*, a syntactic sugar of Scala. Case classes ensure thread-safety and make reasoning about the code simpler thanks to pattern matching. Data-processing workflows are easily built around the design pattern called *builder* – the transformations are "chained" as a series method calls on the underlying collections. Builder pattern treats functions as first-class citizens and accepts them as arguments to enable remote execution.

The source code of SparkGP is organized around the main loop of Genetic Programming, as shown in Figure 4.1. The main collection is the *Population* of candidate solutions. In each iteration the solutions are evaluated (see Section 4.2) and a new population (new generation) is constructed as a transformation of the old one (see Sections 4.4 and 4.5).

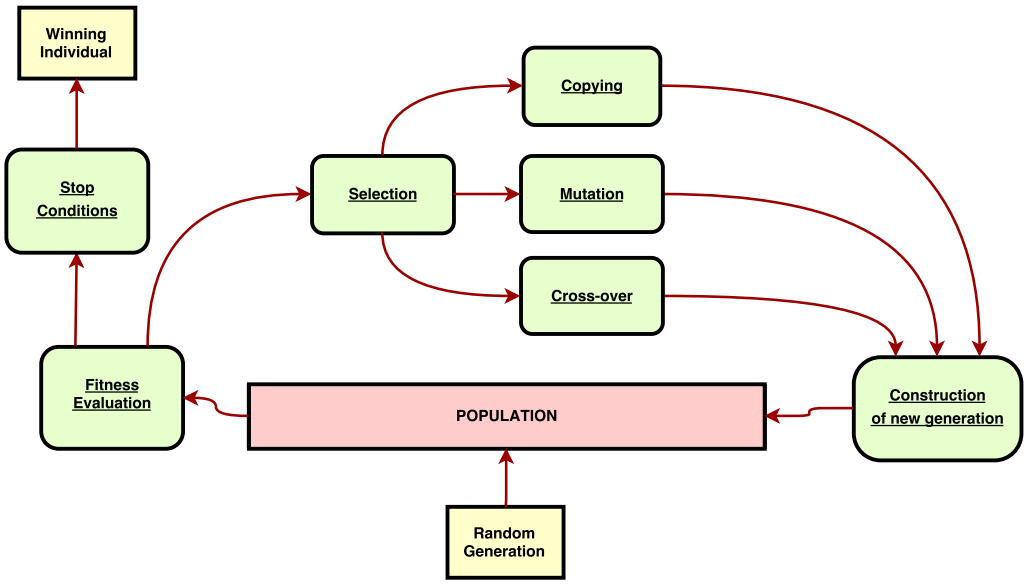


Figure 4.1: The complete Genetic Programming loop

Listing 4.1 presents the code responsible for the loop itself. First the fitness of the population is evaluated (line 3) and then the stop condition is checked. If satisfied, the winning solution is returned (line 5), otherwise a new generation is constructed and the loop continues (lines 6-7). The annotation `@tailrec` in the first line ensures that the function is tail recursive and can be optimized by the Scala compiler. However, before the loop can begin, an initial, random population has to be generated and stored in a proper fashion. The next section describes how it is done.

Listing 4.1: Genetic Programming loop

```
1  @tailrec
2  def loop(population :Population, stats:RuntimeStats):(Graph[TreeNode, String], Fitness)={
3    val (fitnessDS, bestFitness) = evaluateFitness(population, trainingData)

5    if(stopCondition(bestFitness, stats)) selectWinner(population, stats, bestFitness)
6    else loop(nextGeneration(population, fitnessDS),
7                          updateRuntimeStats(population, stats, bestFitness))
8  }
```

4.1 Population

Abstract syntax trees (ASTs) were introduced in Section 2.3 as a way to easily store and modify mathematical expressions. SparkGP uses ASTs to represent individual solutions in a population. The initial idea for their implementation was to use an existing, single-threaded GP framework and port parts of it for distributed computations. SWIM and FUEL libraries [Kra16] were chosen as the base of the work. These frameworks store each tree as a in-memory graph of JVM objects connected by references. The population is exposed as a collection, eq. `List[Op]`, where each individual `Op` object is the top-level operation in a given tree and its child nodes are accessible with `op.args`. This approach works well in a single-JVM environment.

However, when the population is supposed to be distributed between many machines, the `List` has to be replaced with Apache Spark's `Dataset`. This poses a problem, as each object within a `Dataset` has to be self-contained and serializable. An attempt to create a `Dataset[Op]` ended with a `java.io.NotSerializableException`, because each top-level node, stored directly within the `Dataset`, was backed by an entire in-memory graph of objects that could not be easily serialized and sent across the network to another machine. The idea to re-use the code from already available libraries had to be abandoned and a new approach was required.

4.1.1 Implementation in GraphX

Sachin Tendulkar, a famous Indian cricketer, once described his attitude towards obstacles and victories: "when people throw stones at you, you turn them into milestones" [Cri10]. The same mindset, of turning the very thing that caused an issue into a winning factor, was employed to tackle the challenge of distributed ASTs. `Dataset[Op]` failed due to an implicit in-memory graph of objects. Hence, the second approach, was based on an idea to explicitly handle graphs of user-defined objects. It succeeded thanks to GraphX – a library provided by Apache Spark and designed specifically for graph processing (see section 3.4).

Trees in general and Abstract Syntax Trees in particular, as a special type of graphs, can easily be modeled within GraphX. A single AST is constructed as a group of nodes, where each node has a specific, user-defined object attached to it. These objects determine whether the node is a root (`Root`), an inner node (`Expression`) or a leaf (`Constant` or `Variable`). Listing 4.2 presents the complete list of Scala's case classes used to construct companion objects for ASTs. The nodes belonging to a single AST are connected using edges to form one coherent structure. An example is presented on Figure 4.2. To form a population, multiple trees are stored in one `Graph` as disjoint subgraphs. The process of their construction is described in the next subsection.

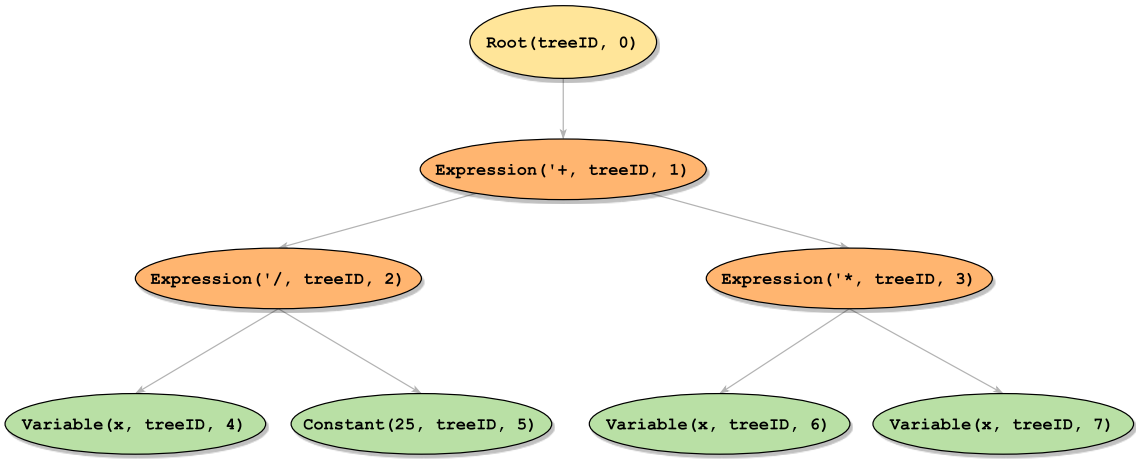


Figure 4.2: GraphX AST equivalent of $x^2 + \frac{x}{25}$

Listing 4.2: Case classes for node companion objects

```
1  trait TreeNode extends Serializable with Product {val treeID:Long; val localNodeID:Long}

3  case class Root(treeID:Long, localNodeID:Long) extends TreeNode
4  case class Expression(operation:Symbol, treeID:Long, localNodeID:Long) extends TreeNode

6  trait TreeLeaf extends TreeNode
7  case class Constant(value:Double, treeID:Long, localNodeID:Long) extends TreeLeaf
8  case class Variable(name:String, treeID:Long, localNodeID:Long) extends TreeLeaf

10 case class ComputedTree(computedValue:Double, treeID:Long, localNodeID:Long) extends TreeNode

12 case class SynthesizedNode(formula:String, treeID:Long, localNodeID:Long) extends TreeNode
13 case class SynthesizedTree(formula:String, treeID:Long, localNodeID:Long) extends TreeNode
```

4.1.2 Initial, random population

Ramped half-and-half (see Subsection 2.3.1) is the most popular method of generating the initial, random population in GP. However, for simplicity purposes, SparkGP uses only the Full method to achieve this goal. Still, implementation of Grow and Ramped half-and-half methods is possible within the setting and briefly outlined in Subsection 6.1.

The process of constructing initial population takes place in two stages. Each stage generates a different type of information so that the graph can be constructed at the end of the process. First, the technical and GraphX-specific data is generated (listing 4.3). Each tree is given a unique identifier, and each node is given a local ID (within its tree) and a global vertex ID. Parent nodes are also given the IDs of their direct descendants. All these information are gathered in `DetailedNode` objects and passed to `generateTree` function.

Listing 4.3: Initial random population

```
1  val treesRDD =
2    spark.range(populationSize).rdd //treeID
3      .cartesian(generateLocalIndexes) //localNodesIDs + childrenIDs
4      .zipWithUniqueId() //vertexID
5      .map{
6        case ((treeID, (localNodeID, localChildOneID, localChildTwoID)), globalVertexID) =>{
7          DetailedNode(globalVertexID, treeID, localNodeID, localChildOneID, localChildTwoID)
8        }
9      }
10     .keyBy(_.treeID)
11     .groupByKey()
12     .flatMapValues(generateTree)
13     .values
14     .cache()

16 new Population(treesRDD)
```

In the second stage, the function `generateTree` associates the nodes with randomly generated expressions, variables and constants (listing 4.4) – the instructions for building the first generation. All nodes that have children are assigned a binary algebraic operation randomly chosen from provided function set, thus realizing the Full method. Whenever a leaf is encountered, the function randomly chooses whether to attach to it a constant or a variable and proceeds accordingly.

Listing 4.4: Random nodes

```
1  def generateInnerNode(node:DetailedNode, childOneGlobalNodeID:Long, childTwoGlobalNodeID:Long) :
       ((Long, TreeNode), List[Edge[String]])={

3    def generateNode = {
4      (node.globalNodeID, Expression(pickRandomOperation, node.treeID, node.localNodeID))
5    }
6    def generateEdges= {
7      Edge(node.globalNodeID, childOneGlobalNodeID, ARGUMENT_ONE) ::
8      Edge(node.globalNodeID, childTwoGlobalNodeID, ARGUMENT_TWO) :: Nil
9    }
10   (generateNode, generateEdges)
11 }


14 def generateConstantLeaf(leafNode:DetailedNode) = {
15   Constant(nextInt(100).toDouble, leafNode.treeID, leafNode.localNodeID)
16 }


19 def generateVariableLeaf(leafNode:DetailedNode) = {
20   val variableIndex = nextInt(VARIABLES.size)
21   val variableName = VARIABLES(variableIndex)
22   Variable(variableName, leafNode.treeID, leafNode.localNodeID)
23 }
```

Once the initial population is constructed the loop of Genetic Programming can begin. The next section shows how Pregel is used to evaluate the fitness of ASTs stored within the graph.

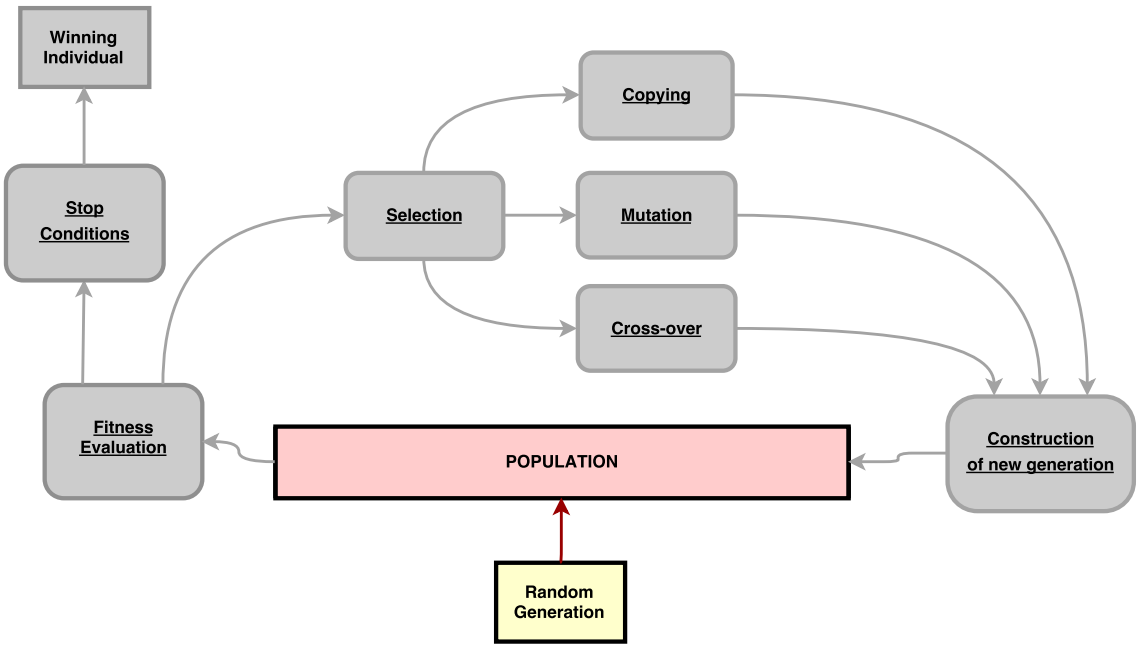


Figure 4.3: The parts of GP implemented so far (in color)

4.2 Fitness evaluation

The fitness of a candidate solution was introduced in Subsection 2.2.1. It is a measure of how well a solution is adapted to the environment. In the context of symbolic regression, fitness is defined as an average absolute error made by a solution when confronted with training data. The evaluation of fitness may be the single most expensive part of GP run, as each member of the population has to be tested against every record from the training set. To speed up the computations, SparkGP uses Pregel (see Section 3.4) to test all candidate solutions in parallel for a given record from the training set.

The evaluation follows the bottom-up tree contraction pattern, where all nodes gradually become `Constant` nodes (see lines 19 - 26 in Listing 4.5). It begins with the all `Variable` nodes being replaced with `Constant` nodes that contain appropriate value from the training set (see Figures 4.4 and 4.5 and lines 1 - 6). Then, all `Constant` nodes send their values to their parents – the `Expression` nodes. Each `Expression` uses the incoming arguments to calculate its value with accordance to the algebraic operation it holds, and becomes a `Constant` itself (see Figure 4.6 and lines 8 - 17). The processes continues until it reaches the `Root` node, which, upon arrival of the final value, turns into `ComputedTree` (Figure 4.7 and line 22). The process repeats with original trees for all testing examples and the fitness is calculated as the average of absolute differences between the expected values and the ones stored in `ComputedTree` nodes. For simplicity purposes, the version of the algorithm presented here uses only a single value from the training set ($x = 2$). In practice, a vector of numbers is used, so that the whole population can be tested against multiple records at a time. The values of fitness for all candidate solutions are returned in the form of a DataFrame.

Listing 4.5: Transformation functions passed to Pregel

```
1  def replaceVariablesWithTestData = {
2    treeNode match {
3      case v:Variable => Constant(findCorrespondingValue(v.name), v.treeID, v.localNodeID)
4      case _ => treeNode
5    }
6  }

8  def calculateValue(expr: Expression) = {
9    val result: Double = expr.operation match {
10     case + => receivedArguments.map(_._2).sum
11     case * => receivedArguments.map(_._2).product
12     case / => $(DIVIDENT) / (if (\$(DIVISOR) != 0) \$(DIVISOR) else 1)
13     case - => $(MINUEND) - $(SUBTRAHEND)
14     case pow => math.pow($(BASE), $(EXPONENT))
15   }
16   Constant(result, expr.treeID, expr.localNodeID)
17 }

19 def contractTree = {
20   treeNode match {
21     case expr: Expression => calculateValue(expr)
22     case root: Root => ComputedTrees(receivedArguments(0)._2, root.treeID, root.localNodeID)
23     case constant: Constant => constant
24     case constants: Constants => constants
25   }
26 }

28 if(firstIteration) replaceVariablesWithTestData
29 else contractTree
```

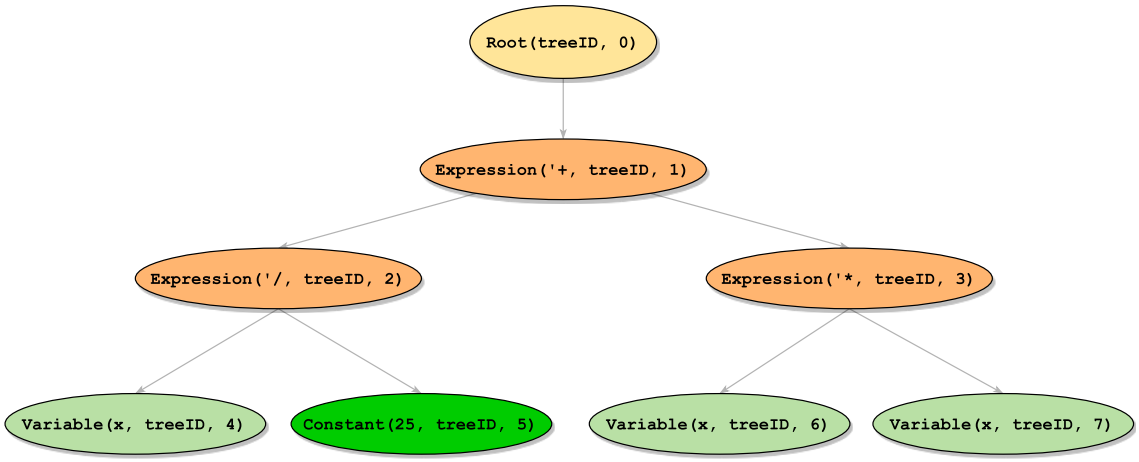


Figure 4.4: Initial state of the AST. Constant leaf highlighted

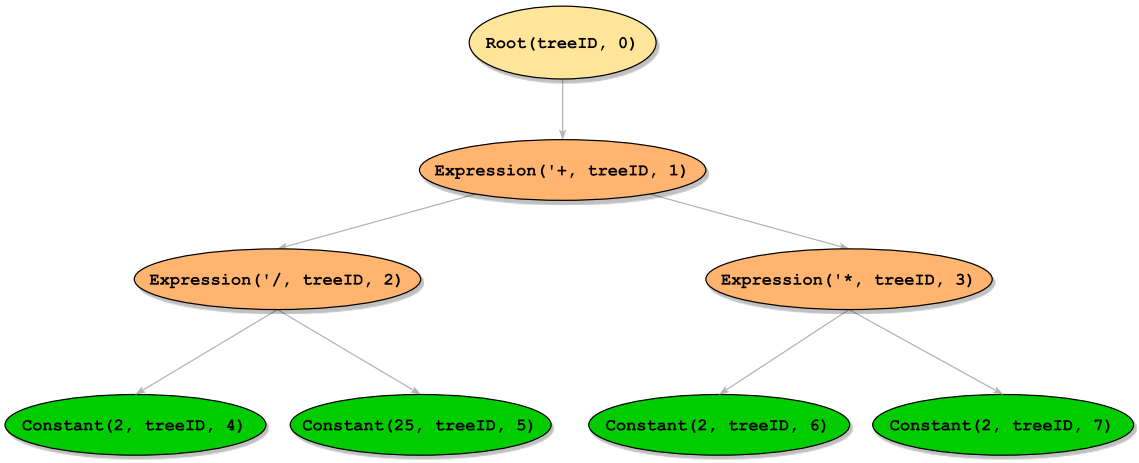


Figure 4.5: Variable leafs replaced with a value from training set

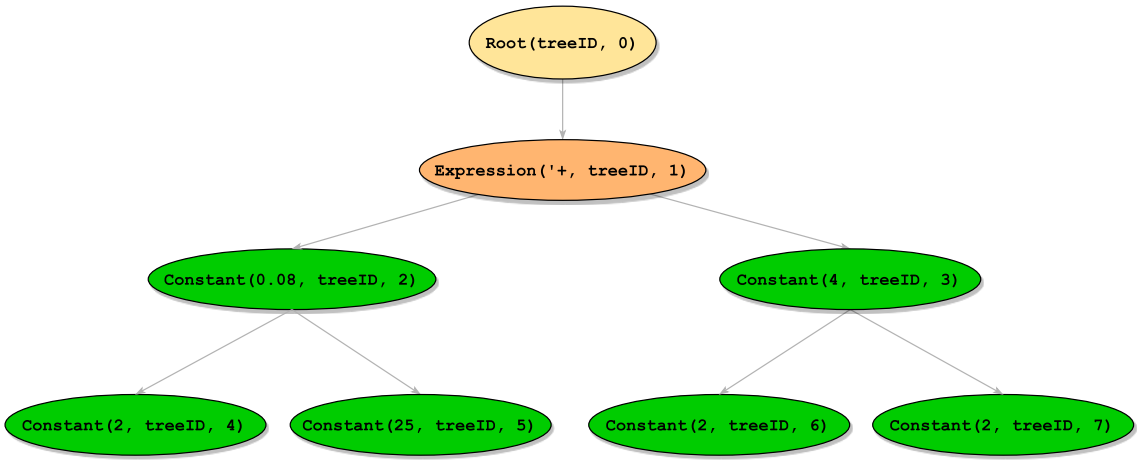
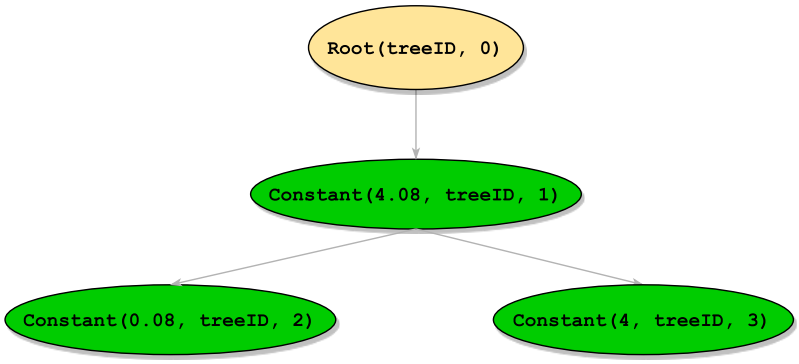
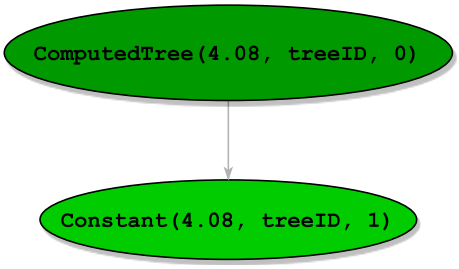


Figure 4.6: Constant values propagated to parent nodes which become Constants themselves



(a) The propagation of values towards the root



(b) Finally, the Root turns into ComputedTree

Figure 4.7: Final stages of tree contraction

4.3 Tournament selection

Tournament selection is used to determine which candidate solutions become parents for the next generation. Two tournaments are organized for each new candidate solution (see Listing 4.6). First, a random sample of parent IDs is taken and a join with the fitness DataFrame is performed. The winners of the tournaments are found with the the help of `reduceByKey` function which compares the fitnesses achieved by competing parents.

Listing 4.6: Tournament selection

```
spark
  .range(populationSize)
  .flatMap { newTreeID =>
    for{
      tournamentID <- 0 to 1
      candidateID  <-  Random.shuffle((0 until popSize).toList).take(tournamentSize)
    }yield {
      (newTreeID, tournamentID, candidateID)
    }
  }
  .toDF("newTreeID", "tournamentID", "candidateID")
  .join(fitness, col("candidateID")===col("treeID"))
  .select("newTreeID", "tournamentID", "treeID", "value")
  .rdd
  .keyBy(x=>(x.getLong(0), x.getInt(1)))
  .reduceByKey((x, y) => if (x.getDouble(3) < y.getDouble(3)) x else y)
  .values
```

Thus the fitness evaluation and tournament selection are implemented. The result of their work can now be passed on to search operators whose implementation will be presented in the next section.

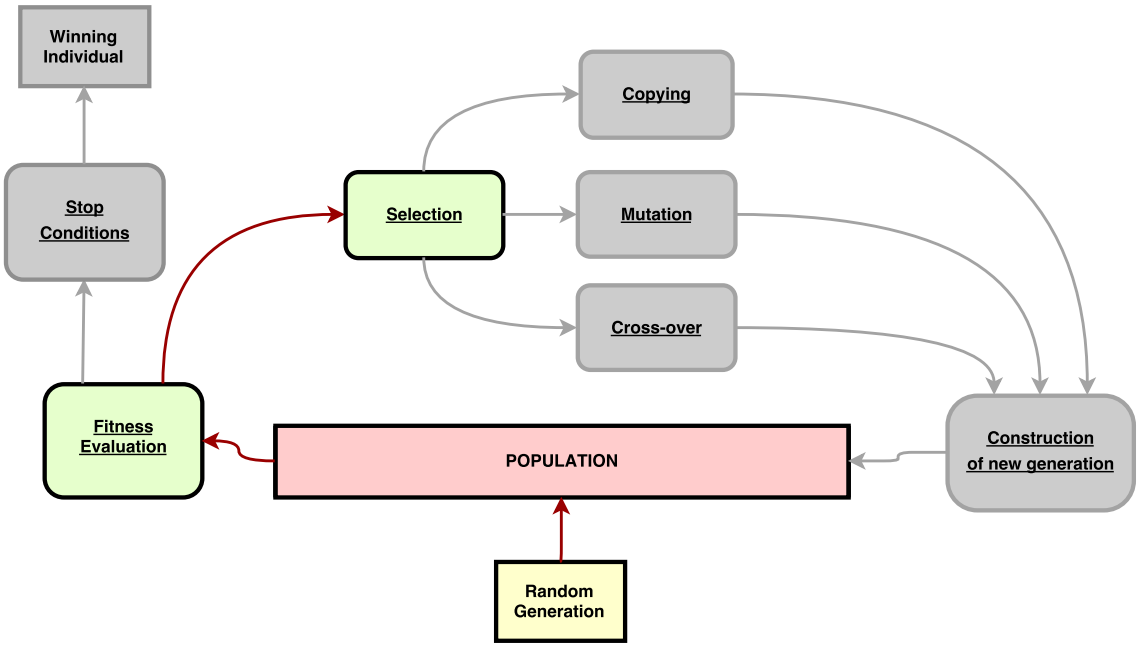


Figure 4.8: Fitness evaluation and Selection join Population and Random Initialization

4.4 Search operators

Search operators are used right after the two parents are determined. However, they do not create complete, new solutions, i.e. ASTs as graphs within Graphx. Rather, search operators manipulate only the companion objects (`Constants`, `Variables` and `Expressions`) to prepare instructions of how the next generation should look like. The reasoning behind this approach will be explained in section 4.5.

The algorithm chooses at random which search operator to use in order to create a new individual (Listing 4.7). Copying is implemented as simply returning the original `parentOne`. The following two sections present the details of mutation and crossover.

Listing 4.7: Random application of genetic operations

```
1  .reduceByKey({
2      case (parentOne, parentTwo) =>{
3        val THRESHOLD_OF_COPY=0.2
4        val THRESHOLD_OF_MUTATION=0.4+THRESHOLD_OF_COPY
5        val THRESHOLD_OF_CROSSOVER=1.0

7        val newTreeNodes=Random.nextDouble() match {
8          case x if x <= THRESHOLD_OF_COPY => parentOne.toArray
9          case x if x <= THRESHOLD_OF_MUTATION => mutateIndividual(parentOne.toArray)
10         case x if x <= THRESHOLD_OF_CROSSOVER => {
11           crossOver (parentOne.toArray, parentTwo.toArray)
12         }
13       }

15       newTreeNodes
16     }
17 })
```

4.4.1 Mutation

The prototype of Lucid implements *point mutation*, where only a single randomly selected node of AST is altered. Once the local ID of the node to be mutated is determined (Listing 4.8, line 3), a mapping on the RDD of nodes is performed. Should the node match the mutation point, it is replaced with an equivalent node (inner or leaf) of random nature. The remaining nodes are not changed and the modified collection of nodes is returned.

Listing 4.8: Point mutation

```
1  def mutateIndividual(originalVertices:RDD[TreeNode]):RDD[TreeNode] = {

3    val localMutationPoint = pickLocalMutationPoint

5    originalVertices
6      .map{
7        case treeNode:TreeNode if (treeNode.localNodeID == localMutationPoint) =>
             replaceWithEquivalentNode(treeNode)
8        case treeNode:TreeNode => treeNode
9      }
10 }
```

4.4.2 Crossover

The version of subtree crossover implemented in SparkGP is a little simplified with respect to the traditional tree-swapping crossover [PLM08]. In general, two crossover points are picked independently, one for each parent. Here, however, the crossover point is the same for both parents. Crossover, in contrast to mutation, requires not one but two parents. Consequently, two offsprings are possible – the first offspring that is based upon the first parent with a subtree coming from the second parent, and the second offspring that is based upon the second parent with a subtree coming from the first one. In order to create only a single offspring, once the crossover point is picked, the *primary parent* is randomly selected and the generation of instructions for the offspring is based upon the primary parent. It happens in two stages.

In the first stage, each node from the primary parent that lies below the crossover point is replaced with the corresponding node from secondary parent. In the second stage, all other nodes from primary parent, the ones not lying below the cross-over point, are selected. The instruction for offspring is formed as a concatenation of the two collections of nodes.

Listing 4.9: Subtree cross-over

```
1  def createOffspring(parentOne: Array[TreeNode], parentTwo: Array[TreeNode], crossOverPoint: Int):
       Array[TreeNode] = {
2    val newSubtree = for {
3      parentNode <- parentOne if PopulationUtil.isDescendantOf(parentNode.localNodeID,
         crossOverPoint)
4      instruction <- parentTwo if parentNode.localNodeID == instruction.localNodeID
5    } yield {
6      instruction
7    }
8    val theRest = parentOne.filterNot(x => PopulationUtil.isDescendantOf(x.localNodeID,
       crossOverPoint))
9    val finalCollection = newSubtree ++ theRest

11   finalCollection
12 }

14 val crossOverPoint: Int = pickRandomCrossOverPoint

16 if(Random.nextDouble() >= 0.5) createOffspring(parentOne, parentTwo, crossOverPoint) else
     createOffspring(parentTwo, parentOne, crossOverPoint)
```

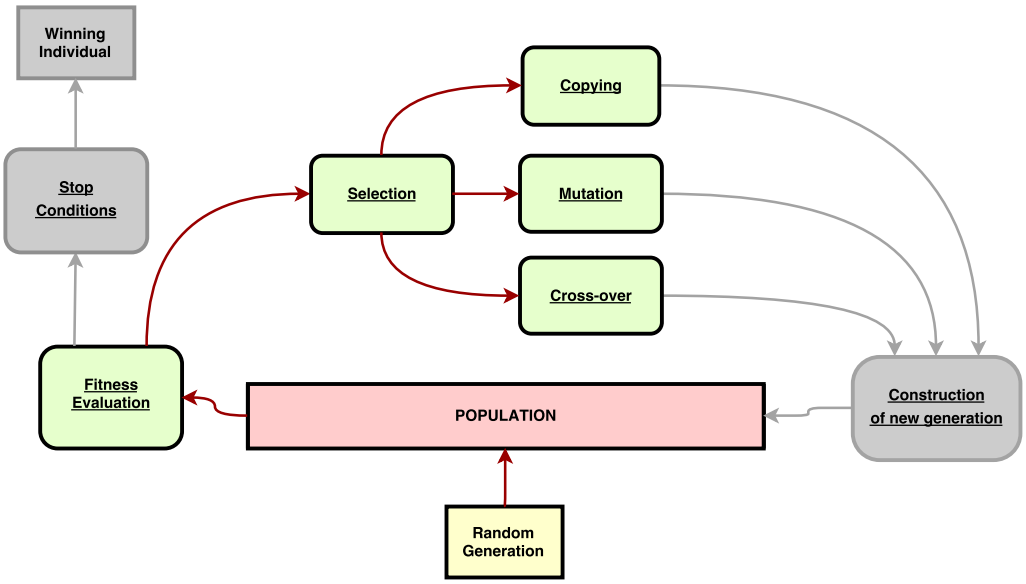


Figure 4.9: Search operators advance the implementation of the GP loop

4.5 Next generation

Genetic operations are usually expected to create final versions of candidate solutions for inclusion in the next generation. In the context of this thesis, it seemed logical to follow this pattern and create new solutions in the form of separate ASTs to be later united into one graph, one population, at the end of the loop. However, as GraphX is designed for analytical purposes and its API forbids to modify and extend the structure of an existing graph, each new generation required to deconstruct the small graphs of individual solutions and build a completely new, big graph that contained the whole population. The process was even more complex than the construction of initial population (see subsection 4.1.2) and resulted in poor performance.

The alternative approach was proposed that not only improved computation time and memory consumption, but also made the code shorter and easier to understand. Throughout the run of the algorithm, only a single graph is constructed and used as a template, or scaffolding, for all generations. As the size of population remains constant and all ASTs are balanced trees of fixed depth, the vertices and edges between them are constructed once and do not change. What do change is the companion objects attached to those vertices. Genetic operations provide only *instructions* for the next generation – a new set of companion objects – that overwrite the old ones with the help of `graph.joinVertices` function (Listing 4.10, line 8).

Listing 4.10: Original graph updated with instructions for next generation

```
1  def nextGeneration(nextGenInstructions: RDD[((Long, Long), (TreeNode, Long))]): Population = {
2    val instructionsForUpdate: RDD[(VertexId, TreeNode)] =
3                          translateLocalCoordinatesToGlobal(nextGenInstructions)
4    new Population(updateGraph(instructionsForUpdate))
5  }

7  private def updateGraph(instruction: RDD[(VertexId, TreeNode)]):Graph[TreeNode, String] = {
8    graph.joinVertices(instruction) {
9      (_, _, newOps) => newOps
10   }.cache()
11 }
```

4.6 Stop conditions

Only one stop condition, i.e. reaching the limit of generations, is implemented. It was done to maintain predictability during the performance testing that is described in the next chapter. However, two other stop conditions are feasible. The first of them is when perfect, or near-perfect, solution is found and the fitness falls below a certain threshold. The other popular stop condition is stagnation, where the population reached local optimum and is not able to leave it. The telltale sign of stagnation is a lack of improvement in fitness in a number of consecutive generations.

Listing 4.11: Stop conditions

```
1  def stopCondition(bestFitness:Fitness, stats:RuntimeStats):Boolean={
2    def generationsLimitReached = {
3      stats.generationsLeft == 0
4    }

6     generationsLimitReached
7  }
```

Throughout the generations, the algorithm keeps track of the best so far created candidate solution. Once the stop condition is met, the best so far solution is compared with the fittest from the last generation and the better of the two is returned. To facilitate the presentation of the best result, it is synthesized to Latex math source code. For example, the raw output may look like this:

```
\[((x) \times (45.0)+(34.0) \times (42.0)) \times ((x+x) \times (x+40.0))\]
```

and when pasted to Latex document it looks the following way:

$$((x) \times (45.0) + (34.0) \times (42.0)) \times ((x + x) \times (x + 40.0))$$

4.7 Summary

The implementation of Genetic Programming in a distributed environment proved to be a challenge in terms of finding new ways to realize well-established concepts. However, the source code of SparkGP is concise (about 1200 LOC) and easy to extend thanks to the features of Scala programming language. The main loop of GP is encoded in the `GPLoop` class, that can be extended with the Scala's traits (Listing 4.12). Should other versions of selection, mutation or crossover be implemented in the future, they can be easily mixed-in to the existing framework.

Listing 4.12: Construction of configurable loop of GP

```
1  val gp = new GPLoop($(populationSize), $(treeDepth), $(inputCols), $(labelCol))
2    with TournamentSelection
3    with PointMutation
4    with OnePointCrossOver
```

Chapter 5

Experiment

The completed GP framework has to be tested whether it works properly and exhibits the expected performance. The tests were conducted to see how the efficiency of SparkGP and quality of its results are affected by two factors: the size of the GP population and the size of the training set. Three benchmarks were used for testing: Quartic function, Pagie-1 and Dow Chemical dataset. The following section presents the configuration of the testing environment that was used throughout all the tests. The details and evaluation of local and distributed experiments are covered in Sections 5.2 and 5.3, respectively.

5.1 Common settings

The selection of benchmarks was based on the evaluation of testing approaches specific to symbolic regression published in [WMC+13]. Table 5.1 presents the details of the selected datasets. The first of them is the Quartic function, usually used to verify basic capabilities of symbolic regression systems. However, Quartic function is often believed to be non-representative of real-life problems [WMC+13]. Hence, the following two benchmarks were also used: Pagie-1, a more complex function of two variables, and Dow Chemical, a dataset of 57 variables measured during chemical experiments. These two are considered to be more difficult and to better assess the quality of a GP system. As presented in Listing 4.5 in Section 4.2, 5 binary arithmetic operations were chosen to drive the symbolic regression efforts in SparkGP: addition, subtraction, multiplication, division and exponentiation.

Name	**Variables**	**Formula**	**Generator**	**Size**
Quartic	1	$x^4 + x^3 + x^2 + x$	E[-312, 312, 1.0]	624
Pagie-1	2	$\frac{1}{1+x^{-4}} + \frac{1}{1+y^{-4}}$	E[-5, 5, 0.4]	625
Dow Chemical	57	properties of chemical reactions	–	747

Table 5.1: Benchmarks used in testing of SparkGP; E[*a, b, c*] stands for a series of points evenly spaced between *a* and *b* (inclusive) with an interval of *c*

Genetic Programming, being a metaheuristic, can not only be adapted to a range of specific applications, but also can be configured with a multitude of parameters. As these tests focus on general performance and producing the results of best quality possible is beyond the scope of the thesis, SparkGP was not fine-tuned and the values of parameters were chosen arbitrarily (Table 5.2).

Parameter	**Value**
Number of runs	10
Depth of AST	4
Size of tournament	7
Limit of generations	8
Probability of a variable as a terminal	0.50
Probability of a constant as a terminal	0.50
Probability of choosing a particular variable	1 / (number of variables)
Constants – uniformly distributed integers from range	-50 to 49
Probability of copying	0.2
Probability of mutation	0.4
Probability of crossover	0.4

Table 5.2: Values of parameters of GP used in testing of SparkGP

Two parameters were observed during the tests. The processing time was measured with 1 second precision. The quality of solutions was evaluated as the *Root Mean Square Error* (RMSE) of regression. The RMSE was calculated as a square root of the average of the squares of regression errors. The remaining two sections in this chapter contain box plots that visualize the values of processing time and RMSE. The bar inside each box stands for the median within the sample; hinges of the box denote the 1st and 3rd quartile. The whiskers extend to the most distant values in the sample that are within the limit of 1.5 times inter-quartile range above or below the respective hinges. The outliers are plotted separately as dots.

5.2 Local computations

The first batch of tests were conducted in a local, non-distributed system in the single-threaded mode on the Intel i7, 3.3 Ghz processor. Their goal was to test SparkGP in a simple environment and get an idea about its performance characteristics before deployment to a cluster.

In the first test, for each benchmark the computation time and RMSE were measured for five different sizes of population: 100, 200, 500, 750 and 1000 solutions. The results are presented on Figures 5.1 and 5.2. Only in the case of the easiest framework, the Quartic function, the bigger population means better RMSE. For Pagie-1 and Dow Chemical dataset, the changes in the size of the population render no observable trend. The probable explanation is the limit of 8 generations applied during the tests. In the case of Quartic function, 8 generations were enough for the population to efficiently converge. For more complex benchmarks, Pagie-1 and Dow Chemical, the limit of 8 generations was too low and processing was terminated before a reasonable candidate solution was created.

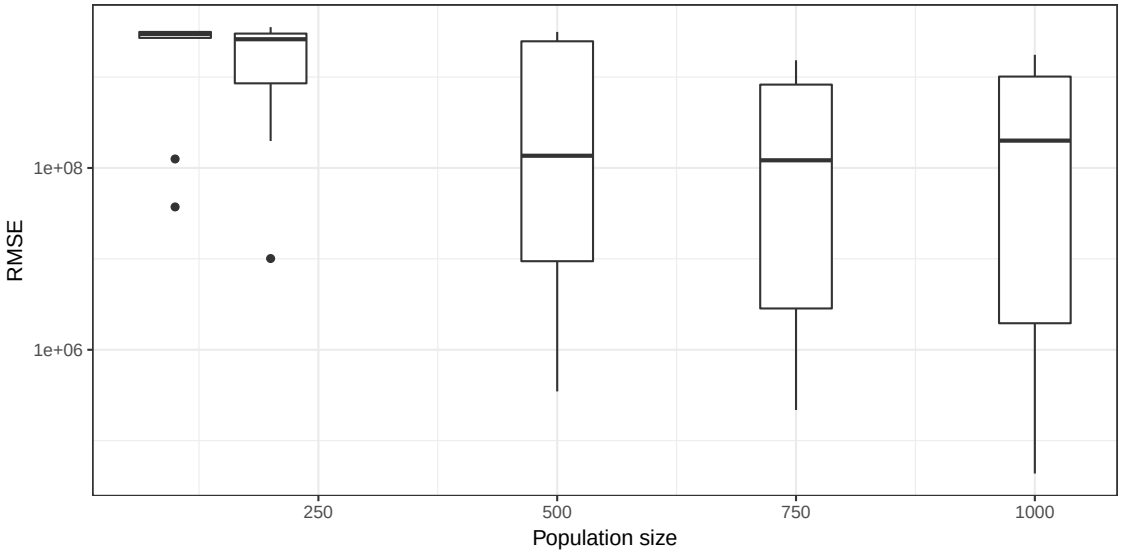
On the other hand, the framework seems to scale really well. The tenfold increase in the size of population lead to an increase in computation time that did not exceed the factor of 3 for all benchmarks. However, this may be due to a relatively small datasets being processed and a significant overhead imposed by Apache Spark and GraphX. As the size of the population grows, the overhead becomes less and less significant leading to a sublinear computation time.

The second test was performed to check how the variable size of the training set affects the performance – Figure 5.3. The population size was kept constant at 500. The original sizes of training sets for Quartic function, Pagie-1 and Dow Chemical were 624, 625 and 747 records respectively. The size of each set was increased by factors of 1.5, 2.0, 2.5 and 3.0 by multiple reuse of records from the original dataset. As the duplicated records brought no new information for the GP and the limit of 8 generations was still in place, the RMSE remained equally unpredictable as in the previous test. On the other hand, the time complexity seems to be sublinear – tree times bigger training set resulted in the computation time growing only by a factor of 2. Again, even though the linear relationship is expected, the sublinear one might be the result of overheads of diminishing importance.

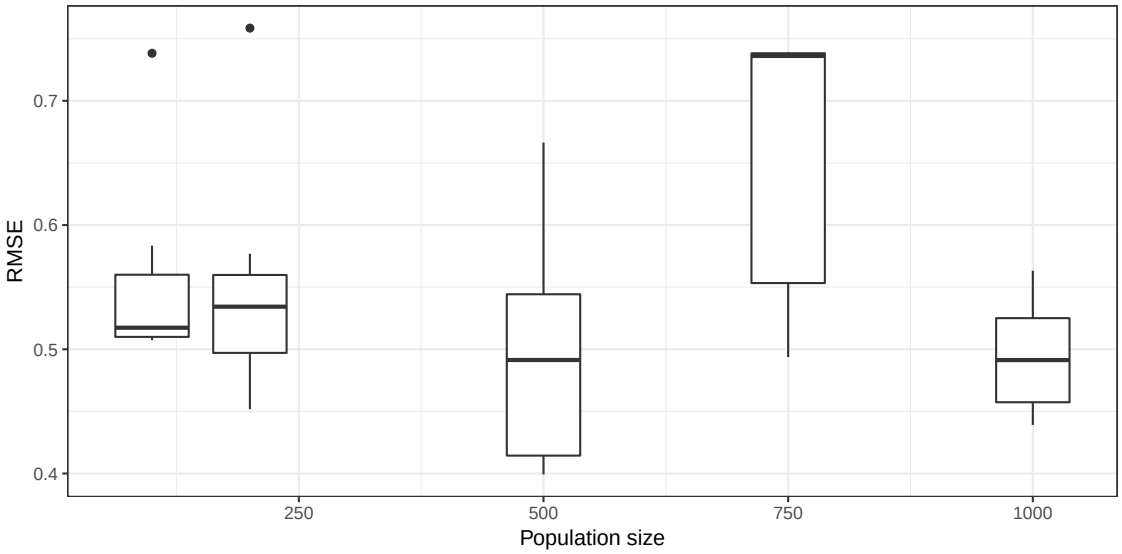
In order to get an idea of how the generated solutions look like, one of the fittest functions created for the Quartic benchmark is presented in equation 5.1. In such form, the solution is difficult to reason about, so it has to be manually simplified – equation 5.2. The denominator simplifies to 0, but as SparkGP protects against division by zero by replacing it with 1, the whole equation reduces to the nominator. The generated function captures the two most important elements of Quartic function: x^3 and x^4. As a result, the regression errors are relatively small and the function gets a very good value of fitness.

$$\frac{(((x) \times (x)) \times ((x) - (0.0))) + (((x) \times (x)) \times ((x) \times (x)))}{(((x) \times (0.0)) \times ((19.0) \times (x))) \times (((x) \times (x)) \times ((36.0) \times (-12.0)))} \tag{5.1}$$

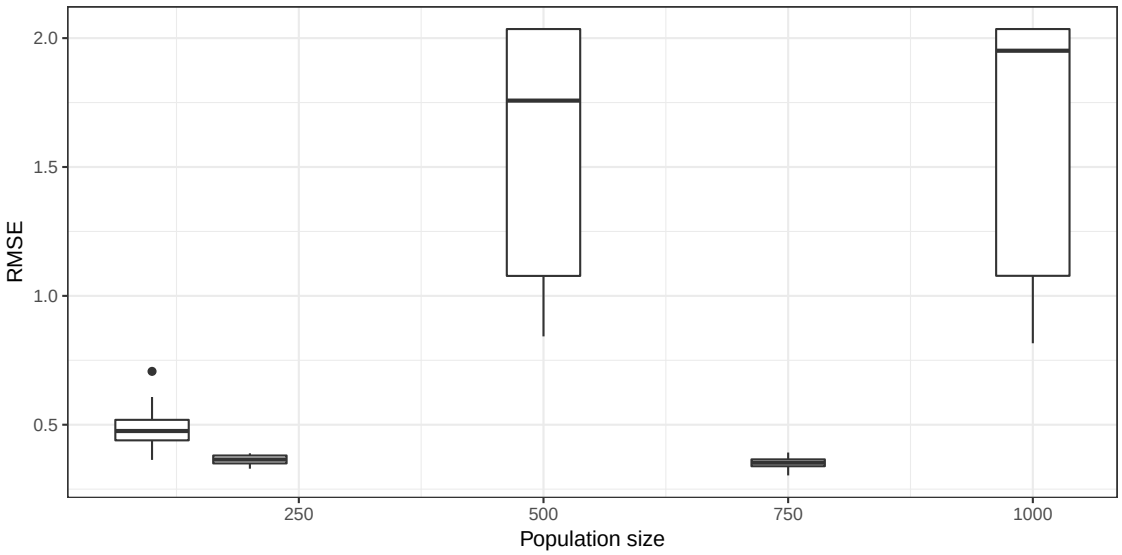
$$\frac{x^3 + x^4}{0} \tag{5.2}$$



(a) Quartic function

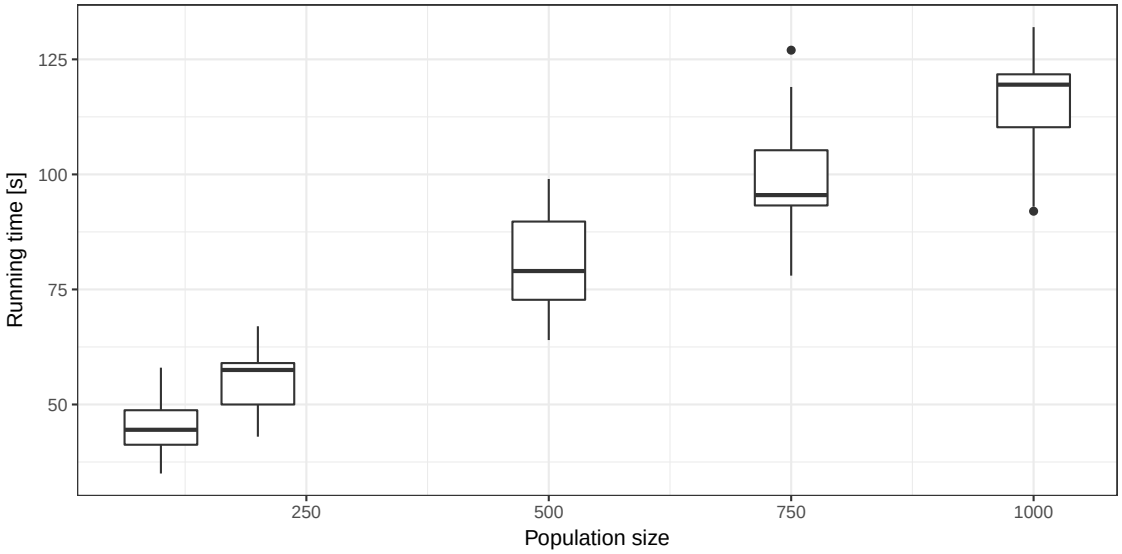


(b) Pagie-1

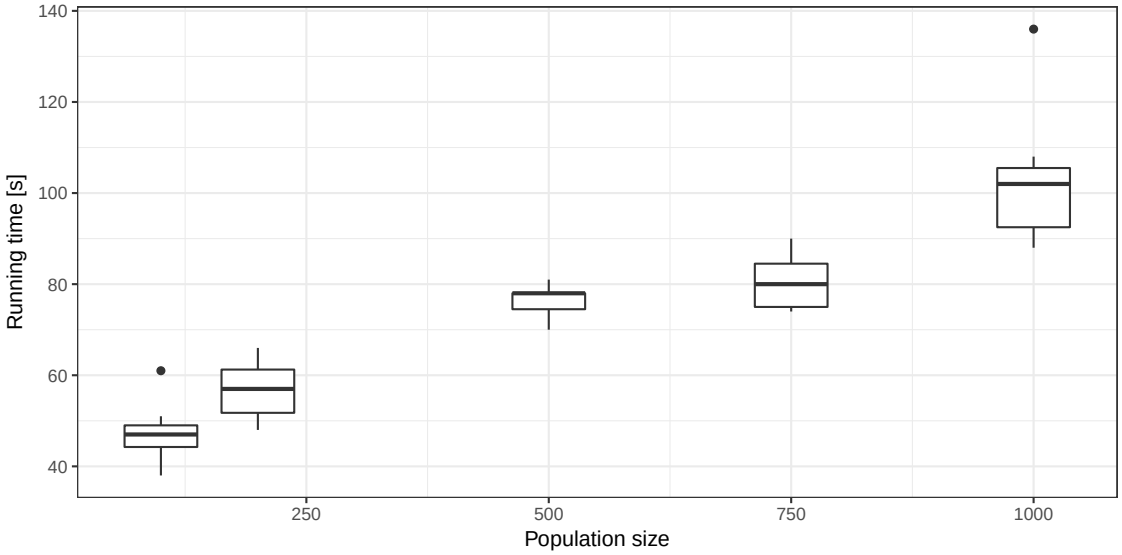


(c) Dow Chemical dataset

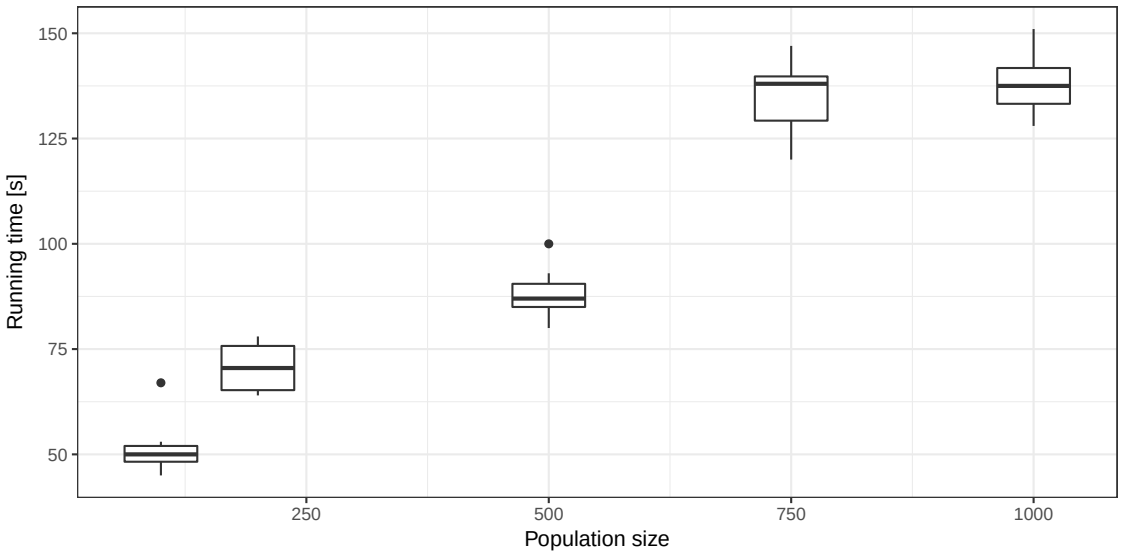
Figure 5.1: RMSE on test set versus the size of population



(a) Quartic function

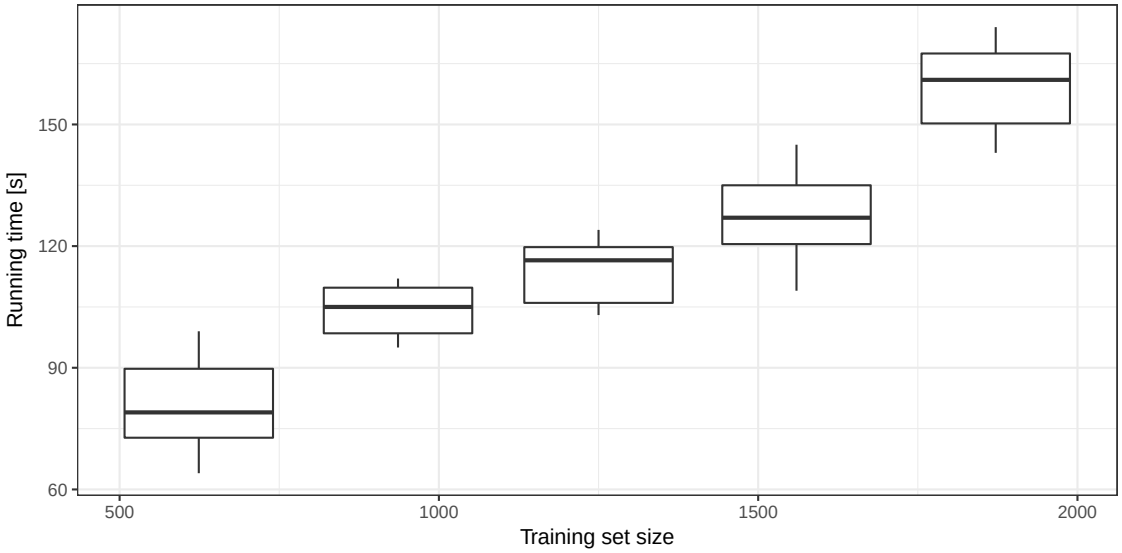


(b) Pagie-1

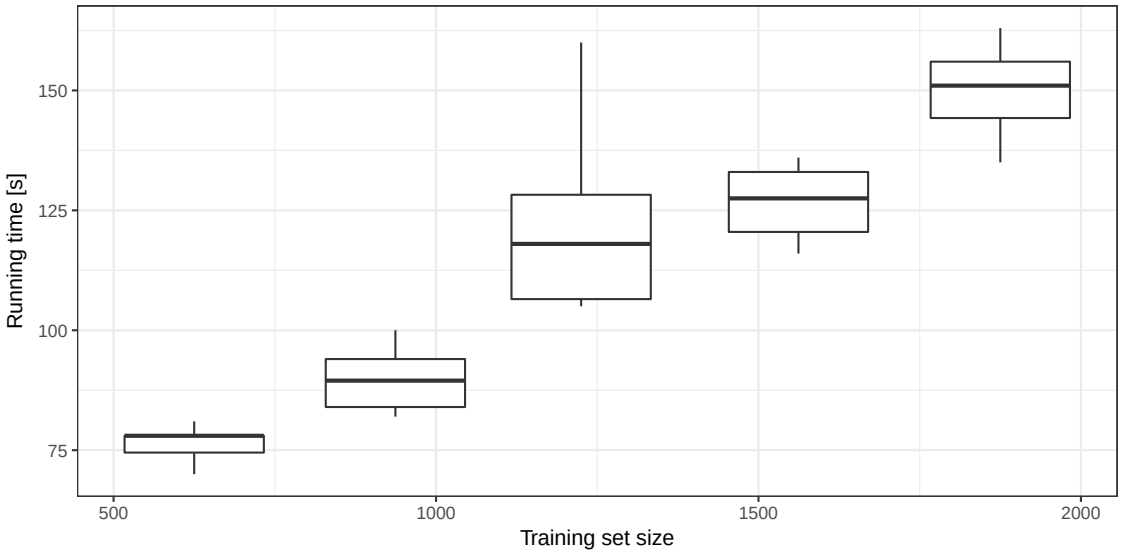


(c) Dow Chemical dataset

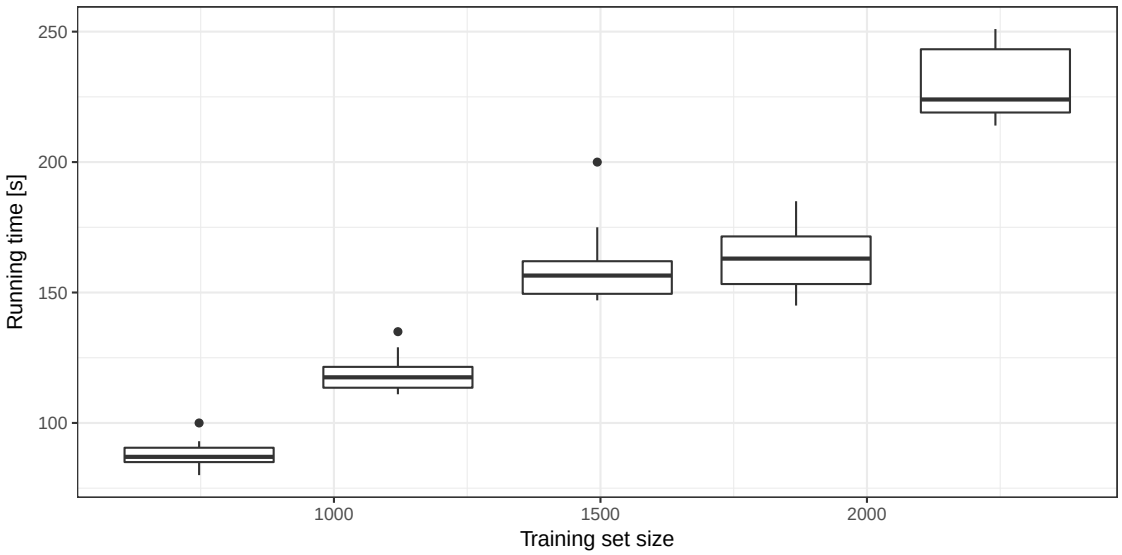
Figure 5.2: Computation time versus the size of population



(a) Quartic function



(b) Pagie-1



(c) Dow Chemical dataset

Figure 5.3: Computation time versus the size of training set

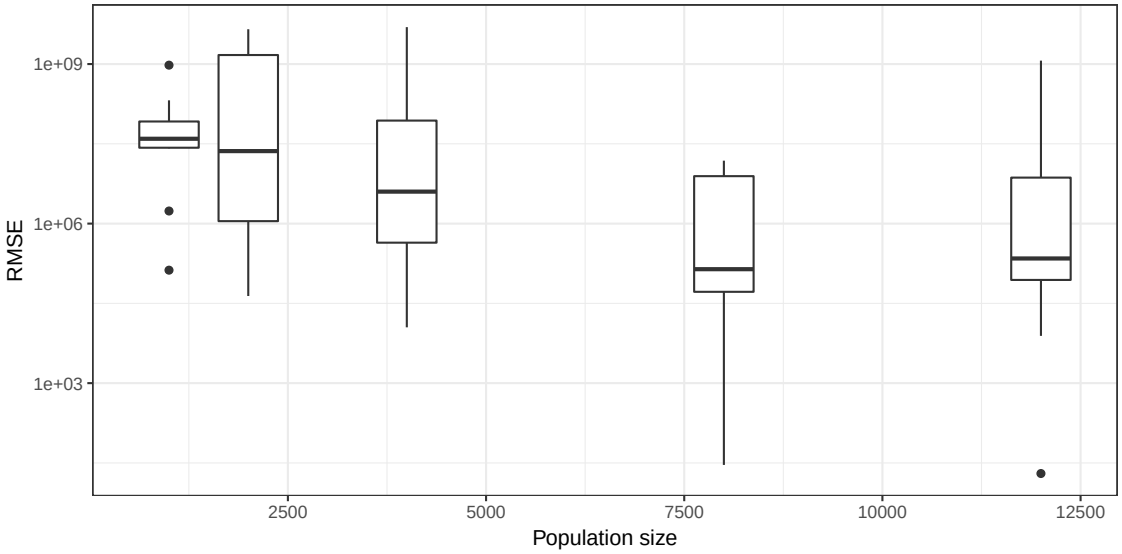
5.3 Distributed computations

SparkGP was built not as a standalone application, but on top of Apache Spark to enable computations both in local and cluster modes, without the need to alter the source code. The preliminary experiments in the local mode were conducted to verify the correctness of the program and to build a foundation for the design of the distributed experiments.

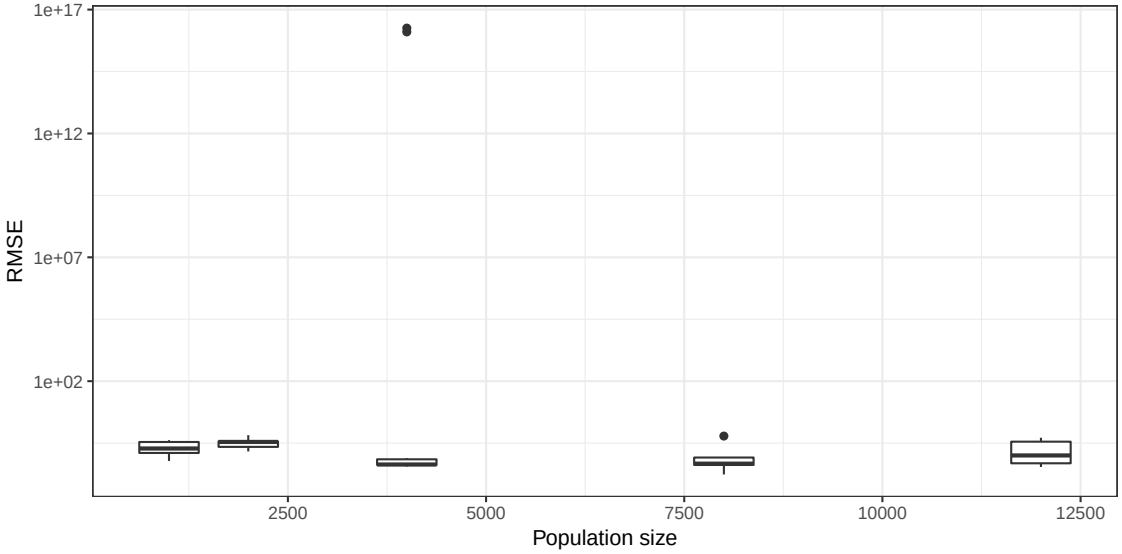
The goal of the tests in the distributed environment was to see how the migration from a single node to a cluster affects the performance and quality of GP runs. However, it turned out that SparkGP suffers from *shuffling* – an excessive exchange of data between the nodes. Early runs showed that the run for exactly the same parameters takes 2 minutes in local mode, 11 minutes in a cluster composed of 1 master and 2 worker nodes and more than 30 minutes in a cluster of 4 workers. The observed traffic was in the order of terabytes – an obvious anomaly that requires investigation and further optimizations. In such situation, the tests with variable cluster size were deemed impractical and too expensive on clusters provided by Amazon Web Services. Consequently, the distributed tests mirror the ones performed in previous section, but this time a cluster of two worker nodes and one master node was used.

We considered five values of the population size: 1000, 2000, 4000, 8000 and 12000. The results are presented on Figures 5.4 and 5.5. Similar to the local mode, RMSE tends to improve for Quartic function and remains more or less constant for Pagie-1 and Dow Chemical benchmarks. An increase in the limit of the generations to about 50, a standard value in GP experiments, was considered. However, it would extend the processing time over 6 times and that was not cost-effective in a rented cluster. The limit of 8 generations remained. On the other hand, the computation time seems to be logarithmic: as the population is increased by a factor of 12.5 the processing time increases not even by a factor of 2.0. The overhead imposed by Apache Spark for small populations is even more significant when compared to local mode. However, in the distributed mode the overhead diminishes faster than in local mode.

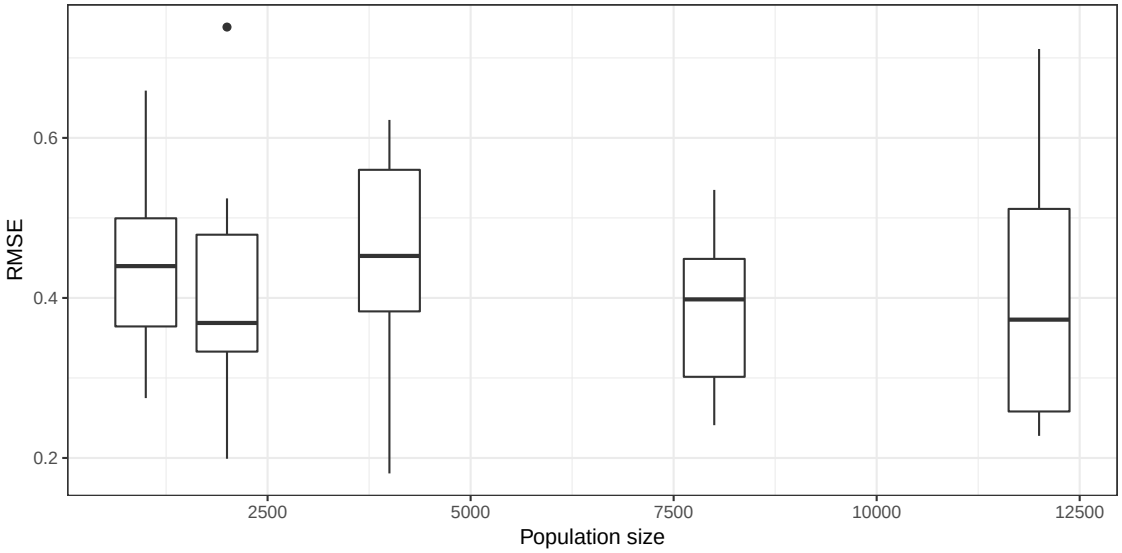
In the second test, the size of population was kept constant (4000 candidate solutions) and the size of training sets was increased by factors of 1.5, 2.0, 2.5 and 3.0 by means of duplicating the records from the original datasets. Here, the processing time looks to be linear – when the size of the training set grows by a factor of 3.0, the time increases by a factor of about 2.3 for Quartic function and Pagie-1 and by a factor of about 3.1 for Dow Chemical benchmark. The shift from sublinear to apparently linear relationship can be attributed to the fact that the analyzed datasets probably reached the values optimal for the cluster size, and the network and CPU cycles were properly utilized.



(a) Quartic function

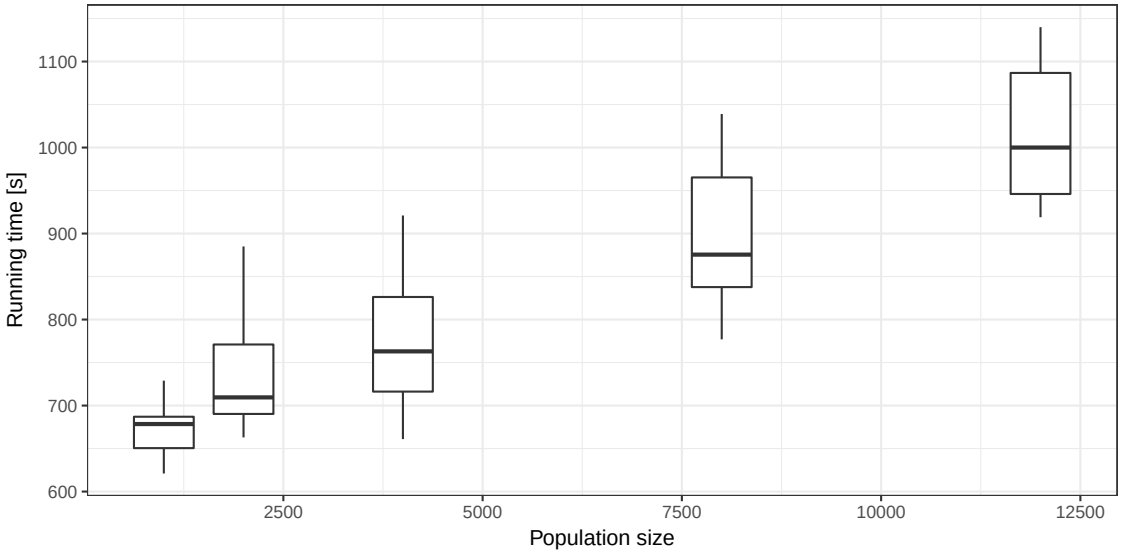


(b) Pagie-1

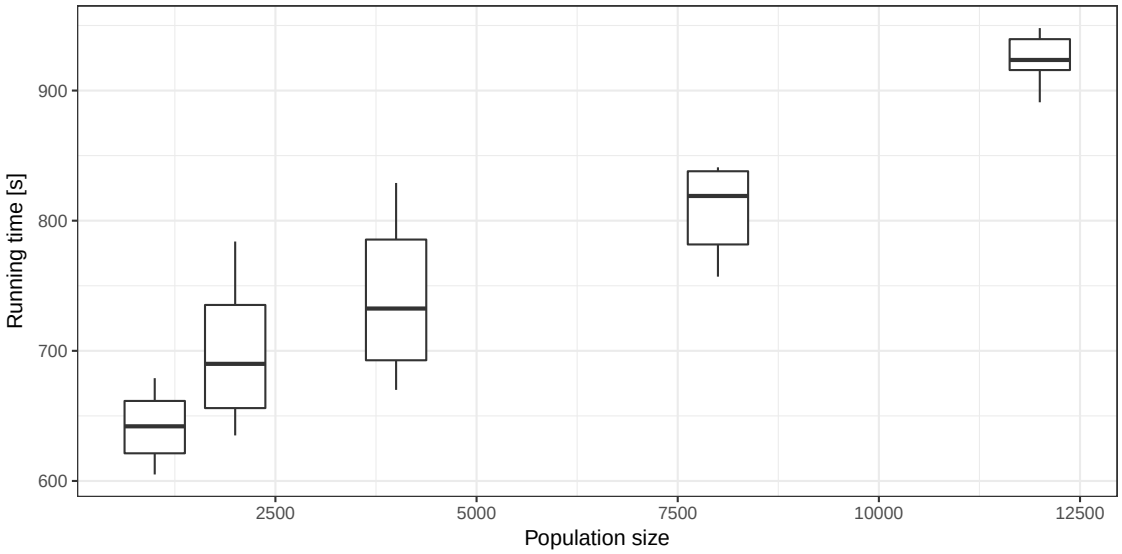


(c) Dow Chemical dataset

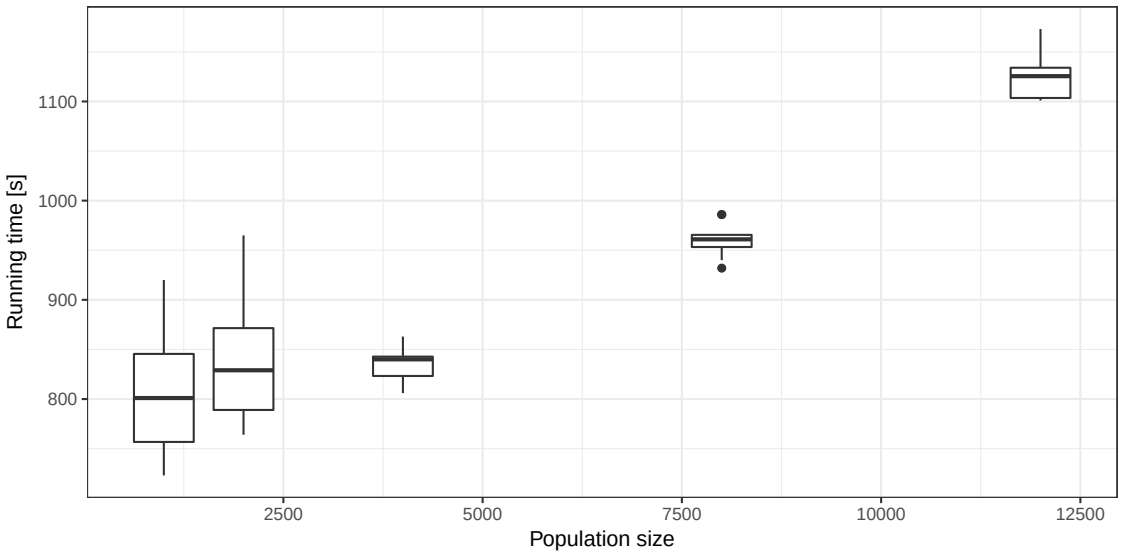
Figure 5.4: RMSE on test set versus the size of population



(a) Quartic function

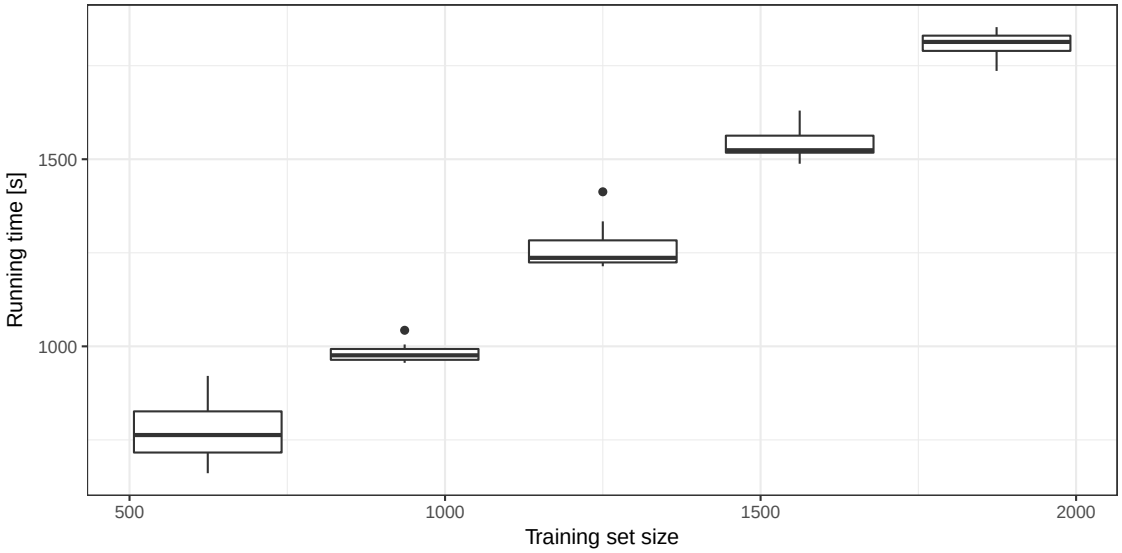


(b) Pagie-1

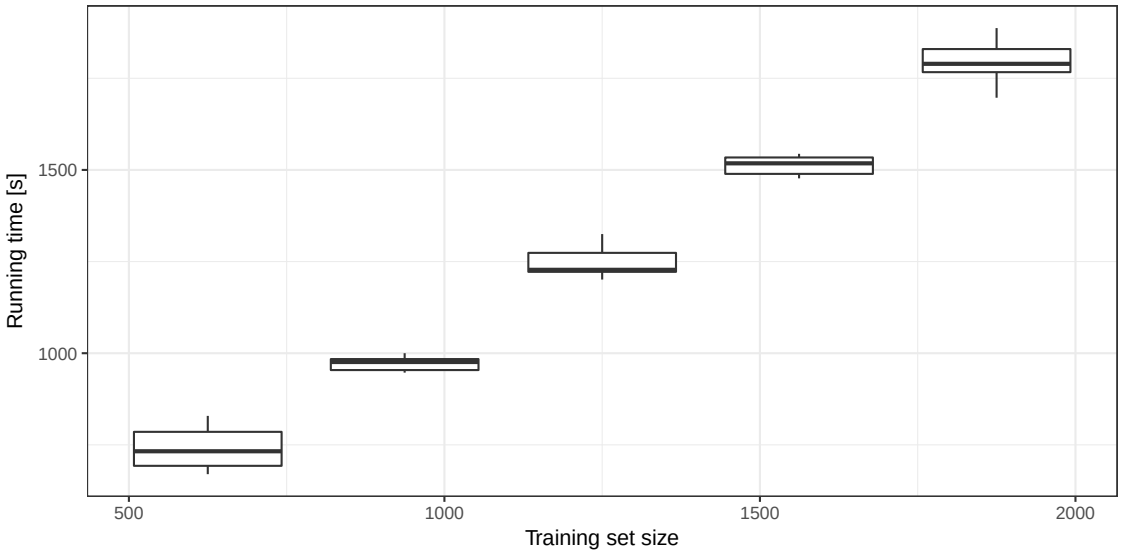


(c) Dow Chemical dataset

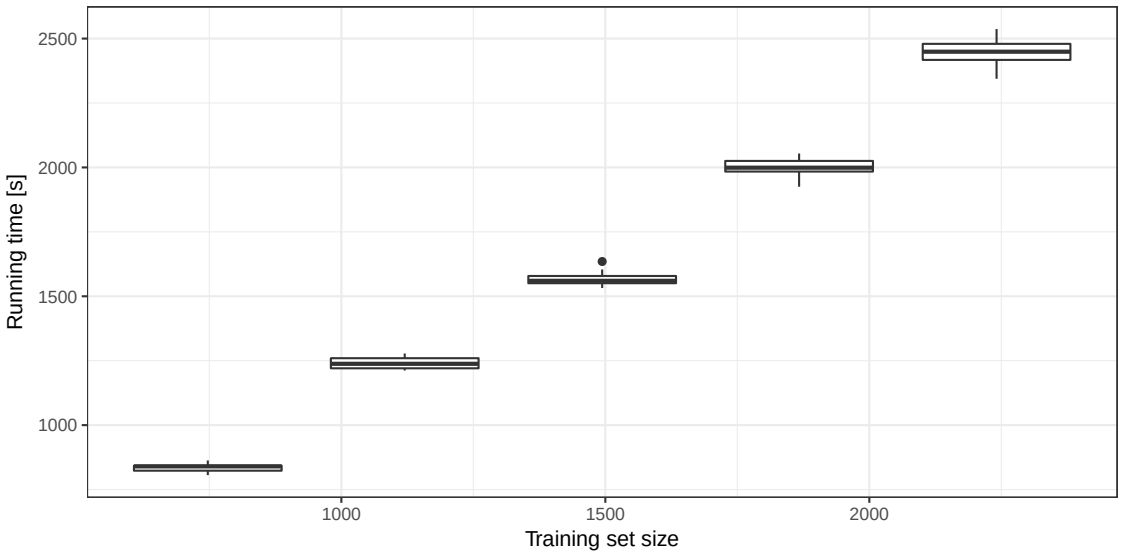
Figure 5.5: Computation time versus the size of population



(a) Quartic function



(b) Pagie-1



(c) Dow Chemical dataset

Figure 5.6: Computation time versus the size of training set

Chapter 6

Summary

SparkGP demonstrates that implementation of an explanatory modelling system based on Genetic Programming is possible on top of Apache Spark. The system scales well with the size of the working population (logarithmically) and the the size of the training set (linearly). On the down side, overall performance is not satisfactory, especially in comparison to other standalone GP libraries. Also, the challenge of *shuffling* shows that the development of such framework is not a trivial task and will require more investigation in the future.

Nevertheless, the detailed goals of this thesis, introduced in Section 1.3, have been achieved. The components of Apache Spark have been analyzed and the GraphX, instead of DataFrames, was chosen as the base for the GP population. The symbolic regression system has been implemented and tested against synthetic and real-life data. This chapter contains the last two sections of the thesis. In the first of them, the ideas for future development of SparkGP and Lucid are presented. The very last section contains final remarks and serves as a closure to the social issues established in the Introduction.

6.1 Future development

The source code of SparkGP has been written as a starting point for Lucid – the research and development project that will continue for the following two years. As a result, SparkGP is limited to basic functionality, as some ideas were either deemed too complex to begin with or abandoned due to time constraints. However, multiple solutions were thoroughly discussed throughout the prototyping stage and they can be implemented in the future. This section presents the most important technical and scientific extensions to the project for the Lucid's development team to consider.

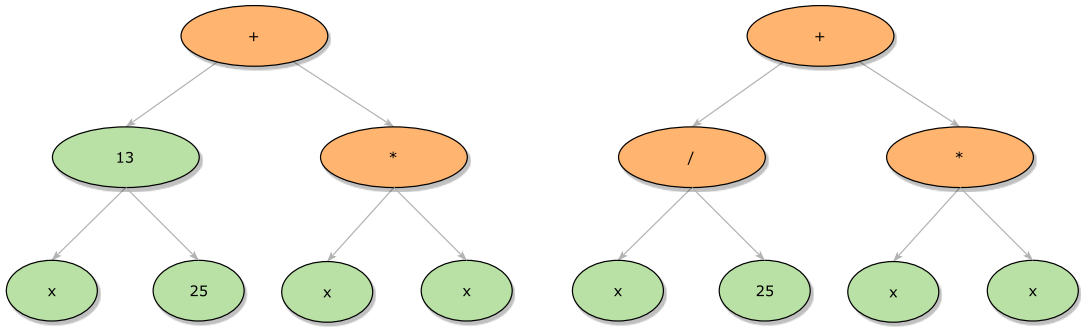
Extended symbolic regression

Currently, the symbolic regression module is limited to five binary arithmetic operations: addition, subtraction, multiplication, division and exponentiation. Such configuration is sufficient for early demonstrations and performance tests but seriously limits the ability to model complex relationships between variables. The possible extensions include other binary operations like n-th roots and logarithms. Trigonometric functions, as unary operations, can be implemented by ignoring the second argument of the AST's node.

More variants of GP

The implementation of Full method makes a strong assumption that all inner nodes of the AST are `Expressions` and only leaves can contain terminals, i.e. `Constants` and `Variables`. The alternative Grow method (see Subsection 2.3.1) may turn any node into a terminal with a probability equal to the depth of the node divided by the depth of the tree. Hence, there will never be a terminal in the root and all leaves are guaranteed to be terminals. When a terminal is chosen for one of the nodes in the middle of the tree, all nodes below it should also be generated but shall become "inactive", thus acting similar to recessive genes. This is because the depth is constant for all ASTs (see Section 4.5). Should the terminal in question be mutated to an `Expression` later on, the entire subtree becomes active. A combination of the Full and Grow methods will result in

Ramped Half-And-Half initialization. Subtree mutation and and other versions of crossover can easily be added as traits for easy composition with the existing loop of GP.



(a) Original AST, equivalent to $13 + x^2$ (b) Mutated AST, equivalent to $\frac{x}{25} + x^2$

Figure 6.1: Activation of subtree in the Grow method

Integration with MLlib Pipelines

Apache Spark is known not only for its efficiency but also for its MLlib package that provides consistent interface to a variety of Machine Learning algorithms. The third-party developers are encouraged to write new algorithms in a Spark-native way and integrate them with MLlib's `Pipelines` – an API inspired by scikit-learn that unifies all stages of data preprocessing, knowledge extraction and hyper-parameter tuning. A `Pipeline` consists of `Transformers`, modules that transform a single record a time, e.g a classification model that adds a prediction field based on other fields, and `Estimators`, e.g. implementation of a classifier, that use the entire dataset to produce `Transformers`. In this context a `SymbolicRegression` would be an `Estimator` that produces `SymbolicRegressionModel`, a `Transformer`, that is then applied to user's data.

However, even though the `Estimator` and `Transformer` classes of Apache Spark are public and it is possible to extend them, still, as of July 2017, the `Params` traits used to configure these two stages are marked *spark-private* and unavailable to external developers. What is more, there is no official documentation for developers of third-party libraries. Once these two obstacles are resolved, the team of Lucid should proceed to integrate the library with the `Pipeline` API. The `SymbolicRegressionModel` should contain a compiled version of the winning solution of the GP run. The compilation to bytecode will be possible thanks to *quasiquotes* - a feature built-in to Scala that enables treating `Strings` as source code for dynamic compilation at runtime.

Hybrid approach

Genetic Programming is a useful technique when one does not know the form or the complexity of the desired function. On the other hand, when the right shape of the function is found, GP struggles to fine-tune the constants present in the solution, as all changes made are random. A solution has been proposed and used in the past [RP$^+$06] to apply a combination of GP and gradient descent algorithm. GP is used to generate functions of different shapes and forms, and for each version the gradient descent is applied to fine-tune the values of constants. Should the need for very precise prediction arise, SparkGP can be extended with such a feature.

Island model

Island model of GP introduces the concept of *islands* - isolated subpopulations that evolve independently. Only once in a while a solution is allowed to *migrate* between the islands, potentially providing novel pieces of code to the subpopulation it has been accepted to. The migrations occur according to the *topology* of connections between islands. The goal of the island model of GP [CHMR87] is to avoid a situation where the entire population converges to a single local optimum and all solutions are very similar.

Island model is especially interesting for distributed implementations of GP, as each node in a cluster may host a single subpopulation and exchange solutions with other nodes only when the conditions of migration are satisfied. This not only increases chances of exploring dissimilar regions of search space but also greatly reduces the communication overhead in the cluster. Island model will be an interesting endeavor once Lucid reaches a stable version.

6.2 Final thoughts

The research conducted throughout this thesis demonstrated that efficient synthesis of human-readable models of reality is possible. Explanatory modelling, a part of scientific method, can be automated by means of Machine Learning and Genetic Programming. Does it mean that the profession of a scientist is going to vanish as the computers take over the field? No. Even though conducting experiments, taking measurements and discovery of natural laws are getting more and more computerized and automated, the scientists are here to stay, but their role will change. *Sapiens: A Brief History of Humankind*, a book by Yuval Noah Harari, an Israeli historian, explains that the field of science is not limited to experiments:

> "Science needs more than just research to make progress. It depends on the mutual reinforcement of science, politics and economics. Political and economic institutions provide the resources without which scientific research is almost impossible. In return, scientific research provides new powers that are used, among other things, to obtain new resources, some of which are reinvested in research" [Har15, Chapter 14: Discovery of Ignorance]

Science does not exists in isolation. It is dependent on funding from governments and is expected to benefit society in order to secure future funding. We do not have to look far for an example – Lucid resides within the described feedback loop of science. It would not exist without funding from The National Center for Research and Development, a government body, and its goal is to put theoretical research of Computer Science into practice so that other scientists and businesses may use it for their own gain. The prospering businesses mean more taxes and that means bigger government budget. The loop closes.

Consequently, it is my belief that, as the experiments become increasingly automated, the scientists will engage more in work that the computers probably will never be able to perform – lecturing, promotion of science, and dialogue with society and governments about which areas of science should we focus on, invest in and develop.

Bibliography

[ama17] Amazon Go. Frequently Asked Questions. https://www.amazon.com/b?node=16008589011, 2017.

[BM14] Erik Brynjolfsson and Andrew McAfee. *The second machine age: Work, progress, and prosperity in a time of brilliant technologies*. WW Norton & Company, 2014.

[Bre01] Leo Breiman. Statistical Modeling: The Two Cultures (with comments and a rejoinder by the author). *Statist. Sci.*, 16(3):199–231, 08 2001.

[CER17] CERN. Processing: What to record? https://home.cern/about/computing/processing-what-record, 2017.

[CHMR87] J. P. Cohoon, S. U. Hegde, W. N. Martin, and D. Richards. Punctuated Equilibria: A Parallel Genetic Algorithm. In *Proceedings of the Second International Conference on Genetic Algorithms on Genetic Algorithms and Their Application*, pages 148–154, Hillsdale, NJ, USA, 1987. L. Erlbaum Associates Inc.

[Cri10] Cricages. "When people throw stones at you, you turn them into milestones" : Sachin. http://www.cricages.com/sachin-zone/when-people-throw-stones-at-you-you-turn-them-into-milestones-sachin/, 2010.

[Dat17a] Databricks. 10th Spark Summit Sets Another Record of Attendance. https://databricks.com/blog/2017/06/09/10th-spark-summit-sets-another-record-attendance.html, 2017.

[Dat17b] Databricks. Apache Spark Ecosystem, 2017. [Online; accessed June 12, 2017; https://databricks.com/spark/about].

[Daw89] R. Dawkins. *The Selfish Gene*. Oxford paperbacks. Oxford University Press, 1989.

[dL17] École Polytechnique Fédérale de Lausanne. The Scala Programming Language. www.scala-lang.org, 2017.

[Fou17a] The Apache Software Foundation. Apache Spark™ - Companies and Organizations. http://spark.apache.org/powered-by.html, 2017.

[Fou17b] The Apache Software Foundation. Apache Spark™ - Lightning-Fast Cluster Computing. https://spark.apache.org/, 2017.

[Fou17c] The Apache Software Foundation. Apache Spark™ - Spark Programming Guide. https://spark.apache.org/docs/latest/programming-guide.html, 2017.

[Fou17d] The Apache Software Foundation. Apache Spark™ - Spark SQL, DataFrames and Datasets Guide. https://spark.apache.org/docs/latest/sql-programming-guide.html, 2017.

[Fou17e] The Apache Software Foundation. GraphX Programming Guide, 2017. [Online; accessed June 21, 2017; https://spark.apache.org/docs/latest/graphx-programming-guide.html].

[fRD16] The National Centre for Research and Development. Wspólne Przedsięwzięcie TANGO - wyniki oceny wniosków pełnych złożonych w II konkursie. http://www.ncbr.gov.pl/gfx/ncbir/userfiles/_public/programy_krajowe/tango/tango_2/lista_rankingowa_pozytywnie_ocenionych_wnioskow_tango2_na_www.pdf, 2016.

[Gol89] David E. Goldberg. *Genetic algorithms in search, optimization, and machine learning*. Addison-Wesley Professional, 1 edition, January 1989.

[GXD+14] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. GraphX: Graph Processing in a Distributed Dataflow Framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 599–613, Berkeley, CA, USA, 2014. USENIX Association.

[Han05] Jiawei Han. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[Har15] Y.N. Harari. *Sapiens: A Brief History of Humankind*. HarperCollins, 2015.

[Hoa61] C. A. R. Hoare. Algorithm 64: Quicksort. *Commun. ACM*, 4(7):321–, July 1961.

[Int17] Intel. BigDL: Distributed Deep Learning Library for Apache Spark. https://github.com/intel-analytics/BigDL, 2017.

[IoCS17] Poznan University of Technology Institute of Computing Science. Surveying explanatory modeling in business and R&D. https://drive.google.com/file/d/0BxzNL3e9Te1TRWNOTzZ0V1NTWlU/view, 2017.

[Isa14] Walter Isaacson. *The innovators. How a group of hackers, geniuses and geeks created the digital revolution*. Simon & Schuster, London, 2014.

[KE95] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Neural Networks, 1995. Proceedings., IEEE International Conference on*, volume 4, pages 1942–1948 vol.4, Nov 1995.

[Koz92] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[Kra16] Krzysztof Krawiec. *Behavioral Program Synthesis with Genetic Programming*, volume 618 of *Studies in Computational Intelligence*. Springer International Publishing, 2016. http://www.cs.put.poznan.pl/kkrawiec/bps.

[LLF+14] Shiping Liu, ElineD. Lorenzen, Matteo Fumagalli, Bo Li, Kelley Harris, Zijun Xiong, Long Zhou, ThorfinnSand Korneliussen, Mehmet Somel, Courtney Babbitt, Greg Wray, Jianwen Li, Weiming He, Zhuo Wang, Wenjing Fu, Xueyan Xiang, ClaireC. Morgan, Aoife Doherty, MaryJ. O'Connell, JamesO. McInerney, ErikW. Born, Love Dalén, Rune Dietz, Ludovic Orlando, Christian Sonne, Guojie Zhang, Rasmus Nielsen, Eske Willerslev, and Jun Wang. Population Genomics Reveal Recent Speciation and Rapid Evolutionary Adaptation in Polar Bears. *Cell*, 157(4):785 – 794, 2014.

[Luk13] Sean Luke. *Essentials of Metaheuristics* . Lulu, second edition, 2013. Available for free at http://cs.gmu.edu/∼sean/book/metaheuristics/.

[MAB+10] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A System for Large-scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[Mat96] G. B. Mathews. On the Partition of Numbers. *Proceedings of the London Mathematical Society*, s1-28(1):486–490, 1896.

[Mic17] Microsoft. Microsoft Machine Learning for Apache Spark. https://github.com/Azure/mmlspark, 2017.

[Mil10] P. Miller. *The Smart Swarm: How to Work Efficiently, Communicate Effectively, and Make Better Decisions Using the Secrets of Flocks, Schools, and Colonies*. Penguin Publishing Group, 2010.

[Ode06] Martin Odersky. A Brief History of Scala. http://www.artima.com/weblogs/viewpost.jsp?thread=163733, 2006.

[Ode15] Martin Odersky. BDSBTB 2015: Martin Odersky, Spark – the Ultimate Scala Collections, 2015. [Online; accessed January 19, 2016; https://www.youtube.com/watch?v=NW5h8d_ZyOs].

[OR14] Martin Odersky and Tiark Rompf. Unifying Functional and Object-oriented Programming with Scala. *Commun. ACM*, 57(4):76–86, April 2014.

[otDE17] MIT Initiative on the Digital Economy. ABOUT US. http://ide.mit.edu/about-us, 2017.

[pan17] Python Data Analysis Library. http://pandas.pydata.org/, 2017.

[PBMW99] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[Pis97] David Pisinger. A Minimal Algorithm for the 0–1 Knapsack Problem. *Operations Research*, 45(5):758–767, 1997.

[PLM08] Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008. (With contributions by J. R. Koza).

[RN03] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition, 2003.

[RP+06] Maria E Requena-Pérez et al. Combined use of genetic algorithms and gradient descent optmization methods for accurate inverse permittivity measurement. *IEEE Transactions on Microwave Theory and Techniques*, 54(2):615–624, 2006.

[Sam59] A. L. Samuel. Some Studies in Machine Learning Using the Game of Checkers. *IBM J. Res. Dev.*, 3(3):210–229, July 1959.

[Sim17] Andrew G. Simpson. 4 Million Driving Jobs at Risk from Autonomous Vehicles: Report. http://www.insurancejournal.com/news/national/2017/03/27/445638.htm, 2017.

[SL09] Michael Schmidt and Hod Lipson. Distilling Free-Form Natural Laws from Experimental Data. *Science*, 324(5923):81–85, 2009.

[SVC17] Michael Shavel, Sebastian Vanderzeil, and Emma Currier. Retail Automation: Stranded Workers? Opportunities and risks for labor and automation. https://irrcinstitute.org/news/6-to-7-5-million-u-s-retail-jobs-at-risk-due-to-automation/, 2017.

[WMC+13] David R. White, James McDermott, Mauro Castelli, Luca Manzoni, Brian W. Goldman, Gabriel Kronberger, Wojciech Jaśkowski, Una-May O'Reilly, and Sean Luke. Better GP Benchmarks: Community Survey Results and Proposals. *Genetic Programming and Evolvable Machines*, 14:3–29, 2013.

[XGFS13] Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. GraphX: A Resilient Distributed Graph System on Spark. In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, pages 2:1–2:6, New York, NY, USA, 2013. ACM.

[Zah14] Matei Zaharia. Apache Spark User List: Why Scala? http://apache-spark-user-list.1001560.n3.nabble.com/Why-Scala-td6536.html, 2014.

[ZCD+12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[ZCF+10] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. *HotCloud*, 10(10-10):95, 2010.

[ZHA17] Matei Zaharia, Tim Hunter, and Michael Armbrust. Expanding Apache Spark Use Cases in 2.2 and Beyond, 2017. [Online; accessed June 12, 2017; https://www.youtube.com/watch?v=qAZ5XUz32yM].

Poznan University of Technology
Faculty of Computer Science
Institute of Computer Science

Typeset using LaTeX in Computer Modern.

BibTeX:

```
@mastersthesis{ key,
   author = "Jakub Guner ",
   title = "{Research into
implementation of explanatory modelling
based on genetic programming
in a distributed environment of Apache Spark}",
   school = "Poznan University of Technology",
   address = "Pozna{\'n}, Poland",
   year = "2017",
}
```