CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY

ASSIGNMENT OF MASTER'S THESIS

Title:	Parallel Implementation of Symbolic Regression
Student:	Bc. Tomáš Malíček
Supervisor:	Ing. Tomáš Borovička
Study Programme:	Informatics
Study Branch:	Knowledge Engineering
Department:	Department of Theoretical Computer Science
Validity:	Until the end of summer semester 2017/18

Instructions

Symbolic regression fits a set of experimental observations with a symbolic formula that best approximates the function generating those observations. Genetic programming typically approaches this problem.

1) Review and theoretically describe symbolic regression and genetic programming.
2) Study possible ways of parallelization and propose a suitable approach for effective MapReduce implementation of symbolic regression by genetic programming.
3) Implement a chosen approach in the Scala language and evaluate the scalability of the proposed solution.

References

Will be provided by the supervisor.

doc. Ing. Jan Janoušek, Ph.D. Head of Department	prof. Ing. Pavel Tvrdík, CSc. Dean

Prague February 15, 2017

Czech Technical University in Prague

Faculty of Information Technology

Department of THEORETICAL COMPUTER SCIENCE

Master's thesis

Parallel Implementation of Symbolic Regression

Bc. Tomáš Malíček

Supervisor: Ing. Tomáš Borovička

9th May 2017

Acknowledgements

I would like to thank Ing. Tomáš Borovička and Ing. Ondřej Stuchlík for providing patient guidance during this master thesis project.

Declaration

In Prague on 9th May 2017 ....

Citation of this thesis

Malíček, Tomáš. *Parallel Implementation of Symbolic Regression.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2017.

Abstrakt

Svět kolem nás je plný neprozkoumaných dat. Tato diplomová práce se zaměřuje na jejich prozkoumání pomocí symbolické regrese, která je založena na hledání vzorečku nejlépe popisujícího hodnoty funkce použité pro vytvoření datasetu. Evoluční algoritmy, genetické programování a symbolická regrese a její užití jsou teoreticky popsány. Na základě teoretické části je navržena a implementována paralelní verze symbolické regrese pomocí genetického programování v jazyce Scala za užití clusterového enginu Apache Spark. Jsou provedeny experimenty stran škálovatelnosti navrženého řešení. Výsledky těchto experimentů ukazují, že symbolická regrese může být navrženou paralelní implementací významně urychlena.

Klíčová slova symbolická regrese, paralelizace, Scala, Apache Spark, cluster, škálovatelnost, genetické programování, evoluční algoritmus

Abstract

The world around us is full of unexplored data. This thesis focuses on their exploration using symbolic regression, which searches for a symbolic formula that best approximates values of the function used for the generation of the dataset. Evolutionary algorithms, genetic programming, and symbolic regression and its applications are theoretically described. Based on the description, a parallel implementation of symbolic regression using genetic programming is proposed. The program is implemented in the Scala language based on Apache Spark. Experiments about the scalability of the parallel implementation are performed. The results of these experiments show that the parallel implementation can significantly speed-up symbolic regression.

Keywords symbolic regression, parallelization, Scala, Apache Spark, cluster, scalability, genetic programming, evolutionary algorithm

Contents

List of Figures

List of Tables

Introduction

Motivation

Computer systems and other electronic devices are producing large amounts of data. The main purpose is to find relationships in datasets and derive useful wisdom to push forward expert knowledge in many fields. Various regression techniques have been proposed to explain the underlying essence of data.

Conventional regression techniques are based on an invariable structure with parameters, which are used to adjust a model with that structure for a particular dataset. However, in a particular field, there can be a lack of knowledge about an underlying system that is described by a dataset; therefore it can be difficult to choose the regression technique. An approach called symbolic regression was proposed for these cases. This type of regression is unique for its less limited structure of models, which are built in each execution from scratch using a subset of evolutionary algorithms called genetic programming. At the beginning, a set of random models is created by randomly combining functions, constants, and variables. Each model from this group is evaluated to gain knowledge about how well it fits a selected dataset. Better models are used for further improvements of the set of models by recombining them together. This process is repeated until a sufficient error rate of the best model is reached, and that model represents the final solution.

Goals

The goals of the thesis are following:

- Theoretically describe evolutionary algorithms, genetic programming, and symbolic regression.
- Propose a parallel implementation of symbolic regression using genetic programming and implement it.
- Evaluate the scalability of the proposed solution.

Structure

This thesis consists of five chapters.

1. The first chapter introduces and describes evolutionary algorithms.
2. The second chapter describes genetic programming (GP), methods used in GP, and its parameters.
3. The third chapter introduces symbolic regression, its applications, details, and an example run.
4. The fourth chapter describes the proposed parallel implementation of symbolic regression, used technologies, and the structure of the implementation.
5. The last chapter describes and presents experiments about the scalability of the implementation of symbolic regression. The experiments are properly evaluated.

Chapter 1

Evolutionary algorithms

Evolutionary algorithms (EAs), where genetic programming belongs, come from the family of algorithms for global optimization, which are inspired by biological evolution. These algorithms are based on a population of individuals representing candidate solutions of a given problem. During each iteration of the EA, the population is evolved to improve a fitness (quality) of candidate solutions. This process of improvement is composed of operators such as selection, crossover, reproduction and mutation. Populations in each iteration of the EA are called generations.

Essential for the design of the EA is the decision about the representation of individuals. Candidate solutions should be encoded in a way that is easily handled by computers. It is necessary due to the enormous quantity of evaluations of individuals in the population during the evolution process. A difficult representation can significantly slow down the progress toward the final solution. Individuals represented in the form of values in arrays are suitable for computer processing. The initialization of the population is performed using a random generation of these individuals. However, it can be helpful to incorporate domain knowledge about good solutions into the population [1].

The quality of candidate solutions is determined using the fitness function, which should take into account all conditions and restrictions about the final solution. In the case of more fitness objectives (e.g., the quality and the age of individuals [2]), it can be problematic. In this situation, the decision about the importance of particular goals to cover objectives appropriately in the fitness function is crucial. Furthermore, it is possible to replace fitness evaluation by fitness approximation [3]. This approach can be helpful in dealing with a high computational complexity of a particular problem.

A common setup of the EA includes selection, crossover, reproduction, and mutation. Choosing a specific type of each operator is necessary. Three major methods for selection are tournament, roulette wheel and rank selection. The crossover operator can be implemented in three basic forms, namely single-point, two-point and uniform crossover. Reproduction has only one form (i.e., copying of an individual to the next generation). Elitism, which denotes automatic reproduction of the best individual from the whole population to the next generation without any selection, can be included as well. The method of mutation depends on the individual representation and can be implemented either as an inversion of a particular bit or an addition of a value from a probability distribution.

The last step in the construction of the EA is parameters adjustment. The performance of a particular EA is closely linked to the selection of these parameters. The following list shows the core set of parameters, which should be adjusted:

- population size,
- number of generations or error threshold,
- probability of crossover,
- probability of reproduction,
- probability of mutation.

Additional specific parameters can be added to this set, depending on a particular EA.

When all steps, mentioned in the previous paragraphs, are completed, the construction of the EA is finished. In Figure 1.1 is depicted the whole process in a nutshell. At the beginning, a population is initialized, and it is directed to the wheel of evolution. In the wheel, all evolutionary operators mentioned above are used over and over again in the circle. One such circle represents one generation of candidate solutions. After a specified number of generations or when the error of the best individual in the current generation is below the selected threshold, the evolution wheel is terminated, and the best solution is provided. A significant advantage of this process is the fact that in the case where prior knowledge is not included in the initial population, the EA does not make any assumptions about the fitness landscape. Therefore EAs have a good performance in the approximation of solutions of all types of problems.

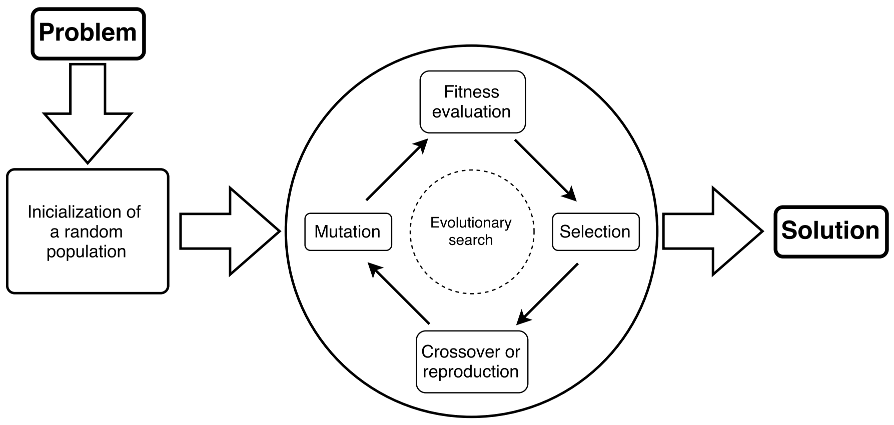

Figure 1.1: Scheme of a generic evolutionary algorithm.

Chapter 2

Genetic programming

Genetic programming (GP) belongs to the group of EAs. The unique attribute of GP is the evolution of the whole structure, the size and the shape of potential solutions [4]. The population in GP is formed from candidate solutions, which represent particular solutions in the search space for a given problem.

The whole process of evolution in GP is depicted in Figure 2.1. During evolution, GP tries to build suitable solutions step by step without any domain knowledge using the performance of random solutions (which are crossbred and mutated). No domain knowledge can be both an advantage and a disadvantage, but in any case, candidate solutions are not limited by any

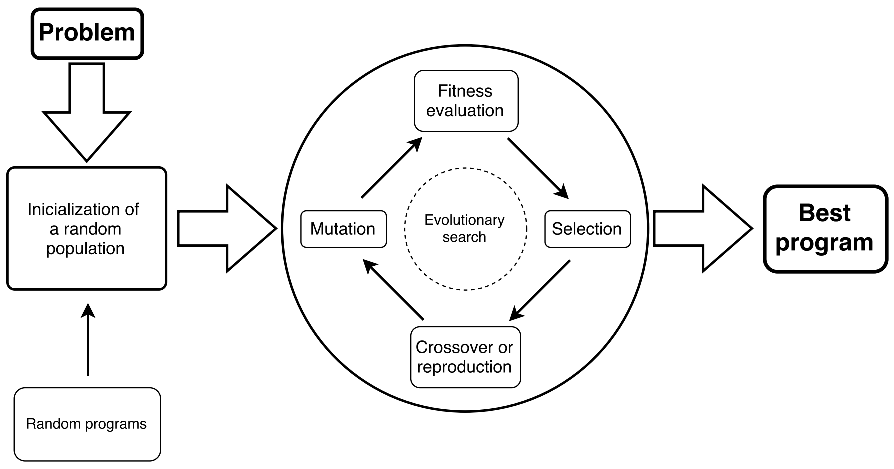

Figure 2.1: Scheme of the evolution process in GP.

predefined structure. Nevertheless, the limitation could also be dependent on a set of objects from which candidate solutions can be built.

Candidate solutions, which represent the final solutions in GP, are used to cover the search space of solutions for a particular assignment. These solutions can be derived from the random process of evolution. This process does not guarantee the discovery of the best solution. On the other hand, it generates randomness, which can be helpful in the ability to escape traps that deterministic methods may be captured by [5].

2.1 Representation of Individuals

Candidate solutions in GP are usually represented in the form of a syntax tree. However, the syntax tree is not the only possibility. Another option is to represent individuals as strings of instructions aligned into an array. This approach is used in linear GP [5]. However, this work describes the common approach, where potential solutions are represented by syntax trees.

Individuals in the population of GP are candidate solutions for a particular problem. The most of the programming languages are not suitable for direct usage in GP because individuals need to be valid after potential crossover or mutation during the evolution process. These operators are combining individuals together or changing pieces of their content, and languages such as C++ or Java are inappropriate for application in GP. There would be a high probability of invalid candidate solutions after random initialization, crossover or mutation. Functional programming languages provide the best functionalities for the representation in the form of syntax trees in GP.

A typical example of a functional programming language suitable for GP is Lisp [6]. The logic of Lisp is based on s-expressions. These s-expression are composed in the prefix notation and can be considered as functions with parameters that give results as a return value [5]. The prefix notation initially states the type of a function or operator, and subsequently, all parameters are provided. All constants and variables in parameters can be replaced by another s-expressions. Whole Lisp programs are composed using this principle, and since each s-expression can represent one subtree, Lisp programs are applicable to the tree structure of candidate solutions.

An example of the s-expression looks as follows:

$$(\min\ (-\ (/\ y\ 2)\ x)\ (+\ x\ (*\ 3\ y))),$$

where min represents the minimum function, and both x and y represents variables. This s-expression uses classical mathematic operators $+, -, *, /$ and function min. Evaluation is typically done using the *top-down* approach. This approach takes the first function or operator and tries to evaluate it. In a case some of its parameters are s-expressions, it proceeds down to find their results. This approach is applied at all levels of the Lisp program. The same instance in the infix notation would look as follows:

$$\min((y\ /\ 2)\ -\ x,\ x\ +\ (\ 3\ *\ y)).$$

Since the s-expression is equivalent to a subtree in the syntax tree, it is possible to swap some s-expressions without invalidation of the Lisp program. This fact provides an option to use the same approach with the candidate solutions of GP. In crossover, subtrees can be swapped without any invalidation. During mutation, a random part of the syntax tree is changed. If the same number of parameters is retained, it is safe for the validity of candidate solutions to perform these mutations.

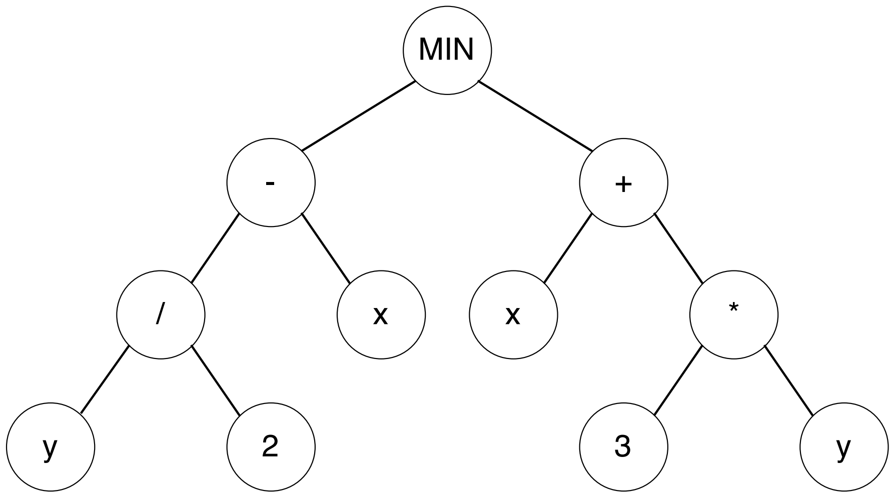

Figure 2.2: a syntax tree with an instance from the prefix notation.

A syntax tree is composed of nodes and edges. Nodes represent atomic elements of expressions such as operators, functions, variables and constants. These nodes can be further divided into terminals and non-terminals. Since constants and variables have no arguments, they are considered as terminal symbols due to their ability to terminate branching of the syntax tree. Terminal symbols represent leaves of the syntax tree. Functions and operators belong to the non-terminal elements of the syntax tree, and they are called inner nodes of the syntax tree. These inner nodes require arguments for successful execution and computation of results. The number of arguments can vary; therefore the arity of the syntax tree is not fixed on two. Edges between nodes represent a data flow. The nodes that are directly connected to a node above them can be called their children. An example of a syntax tree is depicted in Figure 2.2. The same formula as in the s-expression, discussed in the previous paragraph, is used.

Each node has to return the calculated result to its parent using the edge between them. The syntax tree evaluation is executed from the root node through his children down to terminals. Terminals return values to their parents, and this mechanism goes back up to the root node. The root node returns its return value, which represents the final result of the whole candidate solution.

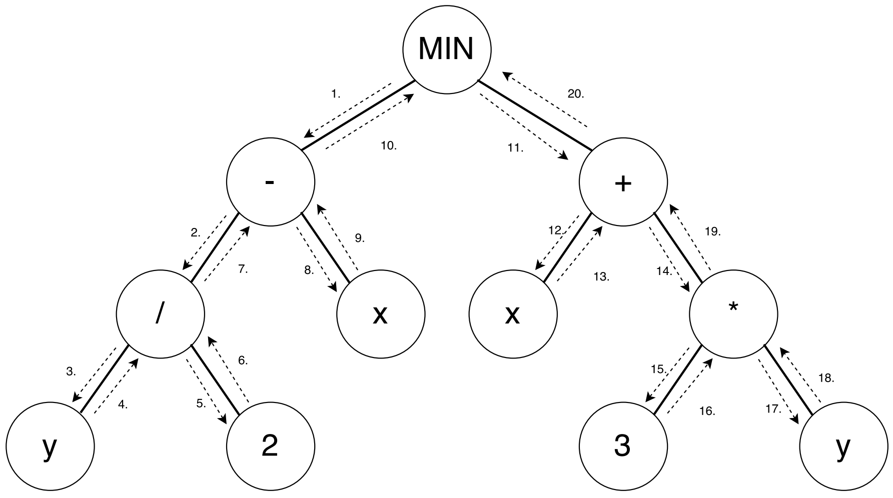

Figure 2.3: The evaluation of the syntax tree with the instance from the prefix notation.

The whole process of the evaluation is demonstrated in Figure 2.3. The evaluation proceeds through the syntax tree in the same way as the depth-first search algorithm. This algorithm always firstly goes to the first child. Secondly, it evaluates the second child and finally it provides the result to his parent. The whole syntax tree is evaluated using this approach. After passing all nodes, the result of this candidate solution is located in the root node.

To summarize, a syntax tree is an ideal structure to represent individuals. This type of representation is suitable both for computers and humans. During the evolution process, candidate solutions can be easily combined using crossover without any invalidation. As a result, the syntax tree representation of candidate solutions is one of the most used representations in GP.

2.2 Fitness Function

The choice of the fitness function is one of the most important decisions made during the setup of the GP algorithm. Based on this function, the direction of the whole evolution process toward the best solution for a particular assignment is set. The same function is used for the whole population. Hence, it defines the main objective of the evolution process.

The fitness of all individuals is evaluated by the fitness function. It should take into consideration various inputs, conditions, and environments in which candidate solutions are tested.

The importance of the fitness function is not based only on the measurement of the qualities of candidate solutions but also on the amount of the computational effort needed for its evaluation. All candidate solutions in the population have to be executed using the training dataset to obtain exact values of fitnesses. This execution is made once for each generation of the population. The effect is that most of the computation time of the whole evolution process is spent on the evaluation of candidate solutions. Therefore the set of functions used in individuals and its computational demands should be wisely considered beforehand.

2.2.1 More Objectives

Another inconvenience emerges when the consideration of more than one objective is desirable. More objectives can be combined in one fitness function. For example, candidate solutions can be handicapped in case of a higher depth of the syntax tree than some specific threshold [5]. It can be accomplished using the subtraction of the depth multiplied by some coefficient. More objectives are approached by multi-objective GP. It is divided into the usage of [5]:

- more fitness values during one generation,
- one fitness value where the fitness function is evolved during the evolution of candidate solutions.

These two types are described in the following two subsections.

2.2.1.1 Pareto Dominance

Pareto dominance is employed for evaluation of Pareto dominant solutions, which are used when more fitness values for all candidate solutions coexist within one generation. The essence of this method is to find the Pareto front of the entire population. The Pareto front is composed of the candidate solutions that are not dominated by any other solution. A solution is Pareto dominant over another solution only by being better in at least one objective and better or equal in all objectives represented by fitness values [2]. By using the Pareto front, the best solutions are easily obtained from the whole population even with the present of more objectives at the same time.

2.2.1.2 Dynamic and Staged Fitness Functions

The second type of multi-objective fitness evaluation are dynamic and staged fitness functions.

The dynamic fitness function is based on evolution of the fitness function throughout evolution of candidate solutions. At the beginning, the first objective is set to the fitness function. When the first objective is satisfied, the second objective is set to the fitness function, and this process continues until the last objective in the list [5]. Objectives should be wisely chosen because the following objectives can eliminate the previous ones. This is the main disadvantage of dynamic fitness functions.

The staged fitness function is similar to the dynamic function. It is also setting the following objectives when the previous objectives are fulfilled. However, since the preceding objectives are still checked and have to be satisfied, it does not allow to damage them [5]. Otherwise, it should return to the first objective that is not met.

2.3 Terminal Set

The terminal set is composed of elements that can be used in the leaves of the syntax tree. It means that those elements have no arguments, i.e., zero arity.

The content of the terminal set is crucial. If a too small set is used, the GP algorithm will not be able to find a good solution. On the contrary, in case of a large terminal set, the GP algorithm cannot be time effective. This issue was studied on the formula [4]: x^3+x^2+x on 20 test cases with the population size of 1000 candidate solutions. In 50 generations with one variable in the terminal set, the correct solution was found in 99.8% of runs [4]. In the case of 32 extra variables, only 35% runs were successful [4]. This experiment shows a significant difference in performance that should be taken into account.

The following sections describe two types of terminals: variables, and constants.

2.3.1 Variables

The first type of terminals are variables. Variables consist of names and values. During the evaluation of candidate solutions, the values of variables are appointed from the dataset, and results of individuals are calculated. Therefore the evaluation using any dataset would be meaningless without the presence of at least one variable.

2.3.2 Constants

Constants are divided into two groups. The first group are constants used as pre-specified, randomly generated numbers during the tree creation process or created using mutation. Pre-specified constants can be used when it is evident that a specific number (e.g., π) should be included in the population. This number can be used as a hint for the GP algorithm.

The second group of constants is called *ephemeral random constants* [5]. These constants are randomly generated using a probability distribution during the creation of the population of candidate solutions [4]. It means that a different random number is generated for each instance of this constant. However, during the evolution process, no mutation is allowed and the value of this constant stays fixed.

2.4 Function Set

The function set includes all functions that can appear in the non-terminal nodes of candidate solutions. This set should include functions that are necessary to cover the essence of a given problem.

2.4.1 Closure

Most function sets are required to have an important property known as closure [5][4], which guarantees no invalidation of candidate solutions during the evolution process. It is divided into type consistency and evaluation safety [5].

2.4.1.1 Type Consistency

Type consistency is required due to crossover and mutation operators, and also for the correct initialization of candidate solutions. Since candidate solutions can be initialized, crossed and mutated randomly, results can be set to function arguments from any other terminal or non-terminal symbol. For that reason, results from all symbols should be compatible with all their arguments [5]. Hence, it is important to be able to convert between various data types that can appear during fitness evaluation. For instance, a number can be converted to a boolean based on its positivity or negativity.

2.4.1.2 Evaluation Safety

Evaluation safety is an attribute assuring that all functions included in the function set return correct values as their results. During the evaluation process, various problems can appear, and functions should be able to handle them correctly and return a specific value in case of problems [5]. For example, if zero is used in the division as the divider, number one can be returned instead of a not a number value.

2.4.2 Types of Functions

Various types of functions can be included in a function set. The following list contains examples of functions that can be included in a function set.

- **Arithmetical functions:** addition, subtraction, multiplication, division.
- **Mathematical functions:** sine, cosine, exp, power.
- **Logical functions:** and, or, not, xor.
- **Conditionals:** if-then-else.
- **Loops:** for, while, repeat.
- **Variable assignment:** =.
- **Problem specific functions:** e.g., some kind of filter.

2.5 Initialization of Population

The population of candidate solutions is typically randomly initialized before the evolution process is started. This process can significantly influence the evolution process. Three methods of initialization are available.

2.5.1 Full Method

This method is based on random initialization of a syntax tree where all terminal nodes (leaves) are in the same depth [6]. The depth is given as a parameter of the method. At the beginning, the top-down random generation of the syntax tree is started from the root node. A random function is chosen from the function set, and the generation process expands recursively the node's children based on the number of function parameters. In the last layer, symbols from the terminal set are randomly chosen. By using this type of initialization, it is assured, that all terminals are in the same depth.

2.5.2 Grow Method

The second method for initialization is called the grow method. The process of the random syntax tree creation is similar to the full method. The only difference is that between the first and the last layer of the depth, both functions and terminals can be chosen. Hence, terminal symbols can appear in all depths except the first one. A disadvantage of this method is that the size of the function set and the terminal set can greatly affect the size of the generated syntax tree [6]. In a particular setup of these sets, the grow method can have almost the same behavior and success rate as the full method. This behavior can be adjusted using a probability of choosing a terminal or a function symbol when it has to be decided.

2.5.3 Ramped Half-and-half Method

This method is a derivation and merge of full and grow methods [4]. It uses both methods to initialize the population. The probabilities of usage of each method are 50%. The depth of the syntax tree is randomly selected from $2..maxDepth$ using the uniform distribution.

According to Koza's [4] experiments with initialization, the ramped half-and-half method is the most successful. During that experiment, all three initialization methods were tested on four problems (including symbolic regression). The ramped half-and-half method had the highest success rate in three problems. In one of these three problems, it was very close to the best solution. In all four problems, the full method had the lowest success rate. From these results can be seen that the combination of previous methods is fruitful because the advantages of both approaches are incorporated in the population and help to find better solutions.

2.5.4 Seeding

Another option that can be considered is seeding. Seeding is an addition of a non-random candidate solution to the population; therefore these candidate solutions are not created using any of the presented initialization methods. These individuals can be user-generated, they can come from another algorithm, or the best solution from any previous run of GP can be used. However, seeding does not necessarily improve the performance of GP [6].

Seeding should be wisely considered concerning other aspects of the GP algorithm. In the case of only one seeded candidate solution, this individual can be easily removed from the population in one generation of the evolution process. On the other hand, if more solutions will be seeded, they can quickly dominate the whole population, and any diversity will be lost [6]. It also depends on the selection method used in the GP algorithm. For instance, one great seeded solution can dominate the whole population. Seeding can be helpful, however, the diversity has to be ensured during the evolution process.

2.6 GP Parameters

2.6.1 Selection Method

The selection method determines which individuals will be reproduced to the next generation. It controls the selection pressure of the GP algorithm. The selected candidate solutions do not have to be only reproduced as they are, but also crossover can be made between them to create their offspring.

The aim of the selection method is to choose the best candidate solutions for a particular problem and to maintain a healthy level of the diversity at the same time. The diversity is critical for finding the global optimum instead of a local one. It is not suitable to strictly eliminate a little bit worse solutions, and some chance should be given to them to evolve in better solutions. How strict the GP algorithm is with the preservation of only the best solutions is called the selection pressure. Three most common selection methods are described in more detail.

2.6.1.1 Tournament Selection

The tournament selection method is based on a comparison of fitness values. The size of the tournament is selected by a parameter. This number of candidate solutions is randomly selected to one round of the tournament. In a single round, the individual with the highest fitness value is selected. This method does not take into consideration the scale of fitness values. It causes that any rescaling of fitness values does not change the results of tournament selection. The selection pressure can be increased by a higher size of the tournament. The commonly used size of the tournament is two. Since tournament selection is easy to implement and provides automatic fitness rescaling, it is commonly used in GP [5].

2.6.1.2 Fitness Proportionate Selection

The second important selection method is fitness proportionate selection, which is also called roulette wheel selection. This selection method is based on the actual values of fitnesses [7]. Any candidate solution can be selected with the following probability:

$$p_i = \frac{f_i}{\sum_{j=1}^{N} f_j},$$

where p denotes the probability, N is the number of candidate solutions, and f is the fitness value of the i-th candidate solution. The selection pressure is directly connected to fitness values, and rescaling can change it. It is a disadvantage of this method because in the case of high variance of fitness values, there will be a high selection pressure. On the other hand, when all fitness values are almost the same, there is a shortage of the selection pressure.

2.6.1.3 Rank Selection

The third popular method for selection is called rank selection. Compared to fitness proportionate selection, this one is not influenced by actual fitness values but by the ranking in the whole population based on fitness values [8]. Therefore, it removes the disadvantage of the previous method and preserves the same selection pressure during various variances in fitness values across the population. The only difference in the implementation compared to fitness proportionate selection is that it sorts candidate solutions and gives them rank numbers. These numbers start with the number one for the worst candidate solution and go up to N for the best solution, where N denotes the number of individuals in the population.

2.6.2 Crossover

The crossover method determines how the candidate solutions will be recombined. From this recombination process, an offspring, which takes part in the next generation, is created. The recombination is usually done using a swap of a particular pair of subtrees from the parent trees. These subtrees are selected using a crossover method, and each method applies its strategy to choose crossover points, which determine root nodes of the subtrees.

2.6.2.1 Subtree Crossover

The most commonly used form of crossover in GP is subtree crossover [5]. This method randomly selects crossover points in both parents. After that, the subtree from the first parent rooted at the crossover point is replaced by the subtree rooted at the crossover point from the second parent. Usually, only one offspring is created. However, it is possible to breed more offspring.

Crossover points can be chosen with the uniform probability distribution. However, it leads to crossover of small subtrees. In extreme cases, it results in a swap of solely terminal symbols. For that reason, the probability distribution of 90% for functions and 10% for terminals is recommended by Koza [4].

2.6.2.2 One Point Crossover

This crossover method uses the identification of a common region in both parent trees. The common region is composed of two parts in both parent trees where they have the same arity of all nodes [9]. Each of these nodes also needs to have the same type (terminal or non-terminal). Both parent trees are scanned for the common region from the root node. Subsequently, the crossover of subtrees is made using a crossover point chosen from the common region.

2.6.2.3 Uniform Crossover

This type of crossover is based on the common region as well. Crossover is made with 50% probability on each node in the common region that is inherited from the first or the second parent tree [10].

2.6.2.4 Size-fair Crossover

Size-fair crossover aims to limit the size of the offspring tree. In the first parent tree, the crossover point is randomly chosen. After that, the size of the selected subtree is calculated and using this size the crossover point in the second parent tree is selected [11]. Using this approach, both subtrees have similar size. Therefore the offspring tree has a similar share of nodes from both parent trees.

2.6.3 Mutation

Mutation is a random change in the candidate solution. This change aims to randomly search in the search space of a particular problem. Nevertheless, mutation is not necessary for GP runs and can be omitted [5]. On the other hand, its influence on overall performance depends on a particular problem and the algorithm configuration, and mutation is still widely used in GP.

2.6.3.1 Subtree Mutation

Subtree mutation is a simple mutation method, which chooses a random node. The subtree rooted at that node is replaced by a randomly generated tree [12]. There can also be included a limit for the depth of the new subtree.

2.6.3.2 Expansion Mutation

Expansion mutation selects a random terminal symbol from the candidate solution, and this symbol is replaced by a new random tree [6]. This method can be considered as a subset of subtree mutation.

2.6.3.3 Shrink Mutation

Shrink mutation can also be considered as a subset of subtree mutation. However, it works reversely in comparison with the expansion mutation method. A randomly chosen subtree is replaced by a random terminal symbol [13]. The main motivation of this method is tree size reduction.

2.6.3.4 Size-fair Subtree Mutation

Size-fair subtree mutation initially finds a random subtree. After that, this subtree is replaced by a new random subtree. The new subtree is generated based on the size of the original subtree. The size of the new subtree is chosen randomly in the range $s/2$ to $3s/2$, where s denotes the size of the original subtree [14]. Therefore the new subtree is be produced in average with the same size as the original tree.

2.6.3.5 Point Mutation

This type of mutation is also called node replacement mutation. A random node is replaced by a randomly generated node with the same arity [15]. Therefore, the new node needs to have the same number of input arguments.

2.6.3.6 Hoist Mutation

Hoist mutation is based on a random selection of a syntax tree. After that, the selected tree replaces the original syntax tree [16]. A tree that used this mutation method is always smaller than the original one; therefore this method can be beneficial in the struggle with a bloated population.

2.6.3.7 Permutation Mutation

This type of mutation performs a random permutation of arguments on a randomly selected node [4]. Since no permutation of arguments influences the results of commutative functions, this mutation method should be used only for non-commutative functions (if any effect is desired).

2.6.3.8 Constants Mutation

This method can mutate only the values of constants. Constants are mutated using the addition of random noise to the original values. The noise usually comes from the Gaussian distribution [17].

2.6.4 Population Size

The population size is a critical parameter of the GP configuration. Using this attribute, the effectivity of a search of the search space can be greatly influenced. The search space in GP is enormous, and the structure of candidate solutions is not very limited; therefore the population size should be as high as possible to provide a great diversity. The size should be at least 500 candidate solutions, and populations of thousands of candidate solutions are often used [6].

2.6.5 Number of Generations

This parameter denotes how much time, measured by the number of generations, is given to the population to evolve and find the best possible solution to a particular problem. The productive progress of the best solution is usually located in the first 50 generations [5].

However, the end of the evolution process can be indicated in a different way than using the fixed number of generations. A problem specific success indicator can be used to detect the situation when the best solution is sufficient. For instance, an error threshold can be used as an indicator for a symbolic regression problem.

2.6.6 Elitism

Elitism is a parameter that determines, if the best candidate solutions and how many of them, will be included in the next generation. It can be done in various ways. One of them is to copy m best solutions to the next generation and breed fewer candidate solutions using crossover and reproduction [6].

Another way is to breed full size of the next generation and replace m worst candidate solutions by m elitists. The number of elitists is usually two and should be chosen with caution to do not allow elitists dominate the whole population quickly and eliminate the diversity.

2.6.7 Operator Rates

Operator rates define the probabilities of operators usage during the evolution process. They can look as follows [6]:

Operator	Rate r
Crossover	$r_c \geq 90\%$
Mutation	$r_m \simeq 1\%$
Reproduction	$r_r \simeq 100\% - r_c$
Elitism	$r_e \simeq 2$

The mutation rate is performed by choosing a candidate solution with r_m rate. The individual is mutated afterward. Crossover and reproduction operators work in the same way. The elitism rate is denoted by a number of elitists that are reproduced to the next generation.

2.7 Bloat

Bloat denotes excessively large parts of candidate solutions [6]. During fitness evaluation, bloated candidates solutions require an enormous computational effort, and a run of the GP algorithm takes more time. Any implementation of GP should incorporate a tool for bloat protection. The following measures can be used [6]:

- **Maximum depth** of candidate solutions can reasonably limit the depth to prevent their syntax trees from uncontrolled growth. This measure can potentially limit the search space and restrict the performance of the best solution.
- **Suitable genetic operators** can be included. For instance, the size-fair crossover method is applicable to this case. This method produces an offspring of a size similar to the size of its parents. Subtree mutation can be adjusted to limit the depth of the mutated candidate solutions. Hoist mutation eliminates code to decrease the size of candidate solutions.

- **a penalty for deep solutions** can be performed in the selection method. Fitness values can be reduced by the depth of candidate solutions. Another way is to lower to zero the probability of the selection of the candidate solutions that are deeper than the average depth.
- In **multi-objective optimization**, the depth of candidate solutions can be used as a second fitness value. By using the Pareto front, the depth can be included in the selection process.

Chapter 3

Symbolic Regression

Symbolic regression is an application of GP, which is looking for a mathematical expression in a symbolical form [4]. This mathematical expression, which can also be considered a formula, should provide the lowest possible error on a particular dataset composed of input and output variables. It means that symbolic regression tries to find the formula that fits a given sample of data. Unlike other types of regressions, whose structure is prespecified, both the structure and the parameters of the formula are subject to search [2].

3.1 Application

Due to the unlimited structure of formulas, symbolic regression can be used to solve a variety of different problems, including the following list (taken from [4]):

- discovery of trigonometric identities,
- econometric modeling and forecasting,
- empirical discovery of scientific laws, such as Kepler's Third Law,
- symbolic differentiation yielding a function in symbolic form,
- solution of a differential equation yielding a function in symbolic form,
- solution of integral equations yielding a function in symbolic form,
- solution of inverse problems yielding a function in symbolic form,
- solution of general functional equations yielding a function in symbolic form,

- solution of equations for numeric roots,
- sequence induction,
- programmatic image compression.

3.2 Differences from the Generic GP Algorithm

Some restrictions exist in symbolic regression in comparison with the generic GP algorithm. First of all, the function set includes only arithmetical and mathematical functions. Other types of functions such as logical functions, conditionals and loops are not allowed. Therefore candidate solutions are represented in the form of mathematical expressions (formulas). These restrictions help to guarantee that only mathematical expressions are present in the population.

3.3 Types of Errors

During fitness evaluation, the fitness function is executed on each candidate solution from the population. This execution covers the calculation of the difference, which is called an error, between the results provided by the candidate solution and outputs from the dataset, by using input variables from the dataset.

More approaches are used for error calculation. All candidate solutions are executed on every single row of the dataset to calculate partial errors. The following types of partial errors can be applied to each row [18]:

1. the **absolute error:** $e = |s(X) - y|$,
2. the **squared error:** $e = (s(X) - y)^2$,
3. the **logarithmic error:** $e = \ln(s(X) - y)$,

where s denotes a candidate solution (a formula), X represents the set of input variables of a row from the dataset and y is the output variable (the correct result) of one row from the dataset.

These partial errors of all rows from the dataset have to be combined into the one fitness value (the size of the error) for each candidate solution. It can be performed in a variety of different ways:

1. the **total error:** $f(s) = \sum_{i=1}^{N} e(i)$,
2. the **mean error:** $f(s) = \frac{1}{N} \sum_{i=1}^{N} e(i)$,
3. the **minimum error:** $f(s) = \min_{\forall i \in R} e(i)$,
4. the **median error:** $f(s) = \text{median}_{\forall i \in R} e(i)$,
5. the **maximum error:** $f(s) = \max_{\forall i \in R} e(i)$,

where f denotes a fitness value, s is a candidate solution, N is the number of rows in the dataset, R is the set of rows from the dataset and i represents a row.

3.4 Dataset

The dataset represents the training data, which are used for the evolution process of symbolic regression. It is composed of rows, where each of them consists of columns with input and output variables. Input variables, which are called independent variables, are values that are set to candidate solutions during fitness evaluation. The output variable, which is called the dependent variable, is a target value for the formula result. The fitness value, which represents the error, should be as low as possible after a run of symbolic regression. Besides, the dataset may be divided into a training data and a testing data on which the performance of the best solution is validated. Potential overfitting (when a solution works only on a particular training dataset) can be discovered using this approach. Overfitting can be reduced using a limited size of candidate solutions and a sufficiently large dataset.

The values of variables in the dataset can be user generated using setting of random values to the desired formula. The dataset can also be compiled from real-world data. In the case of using this type of data, noise should be considered in the creation of the potential termination criteria.

3.5 Example Symbolic Regression

An example of symbolic regression to show the processes of initialization, crossover, mutation and evaluation of particular solutions will be performed on the following formula:

$$x^4 + x^3 + x^2 + x.$$

Table 3.1: Training dataset.

Row number	x	y
1	0.19250727	0.23807383
2	0.28349912	0.39311582
3	0.44879782	0.7811837
4	-0.044144154	-0.042277675
5	0.9041548	3.129093
6	0.61086416	1.3512108
7	-0.016869068	-0.016589222
8	-0.6885911	-0.31610864
9	0.945609	3.4848776
10	-0.1093992	-0.09859709

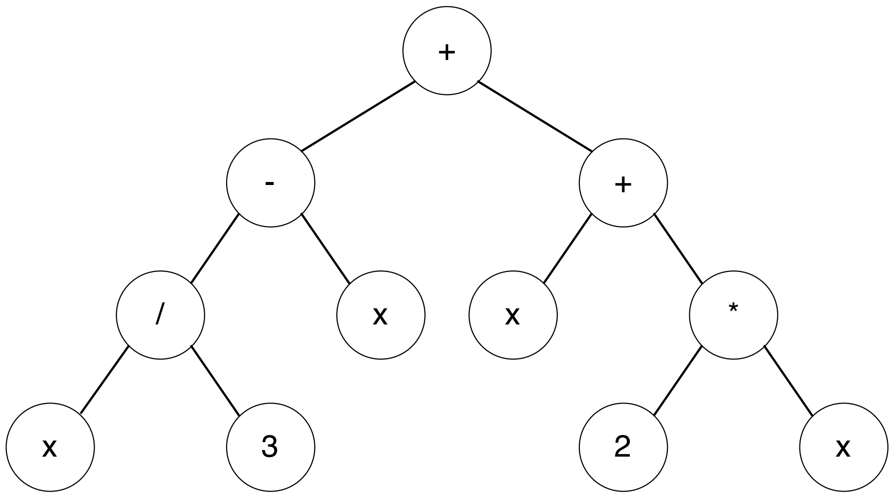

Figure 3.1: a candidate solution initialized using the grow method.

By using this formula, the training dataset in Table 3.1 was generated. The variable x represents an independent variable, and y is a dependent variable.

At the beginning of symbolic regression, candidate solutions can be initialized using the ramped half-and-half method described in 2.5.3. The ramped half-and-half method is using both the grow and the full method. The example of a candidate solution initialized using the grow method is depicted in Figure 3.1 and for the full method in Figure 3.2.

After the initialization, the evolution process is started. During this process, crossover, reproduction, and mutation are performed. An example of crossover of two candidate solutions using the subtree crossover method, which

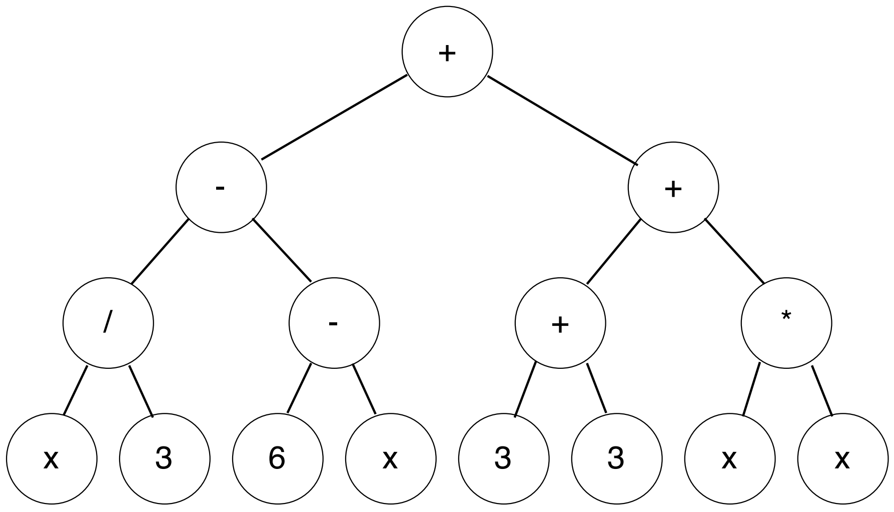

Figure 3.2: a candidate solution initialized using the full method.

is described in 2.6.2.1, is depicted in Figure 3.3. In that figure, mutation is also executed, and it is highlighted in the offspring candidate solution.

Fitness evaluation is performed on all candidate solutions in each generation of the evolution process. It is depicted in Figure 3.4, where the final candidate solution for the particular formula is evaluated using the first row of the dataset. Intermediate results of inner subtrees can be found next to non-terminal nodes. In the root node, the final result is calculated. It can be seen that the result in Figure 3.4 is the same as the y value in the first row of the dataset in Table 3.1. Therefore, the error for the first row of the dataset is zero.

3.6 Coevolution of Fitness Predictors

The coevolution of fitness predictors is a technique that replaces fitness evaluation by an approximation [3] and it can be used as an improvement of symbolic regression. This approach comprises from the following three populations [3].

1. **The population of candidate solutions** in symbolic regression includes candidate mathematical expressions.

2. **The population of fitness predictors** is evolved in a similar manner (using selection, crossover, and mutation) as the population of candid-

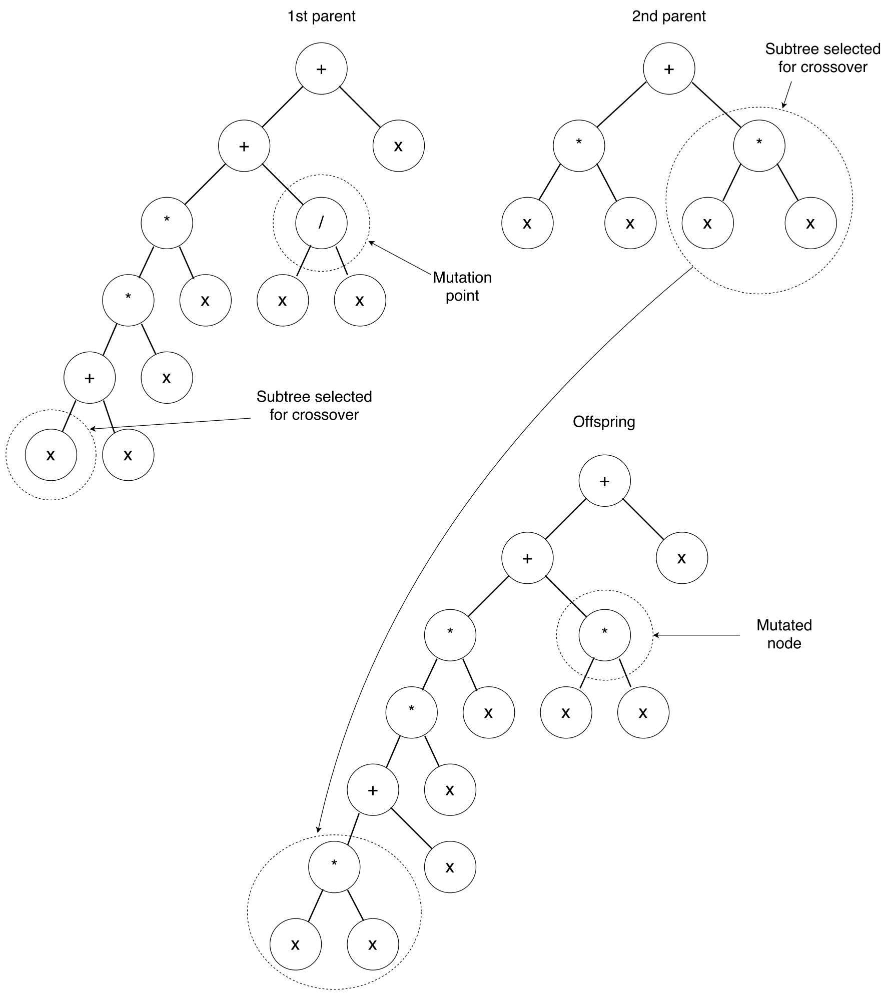

Figure 3.3: Crossover using the subtree crossover method, and point mutation.

ate solutions. Individuals of this population, fitness predictors, represent subsets of rows of a particular training dataset. At the beginning of evolution, fitness predictors are randomly initialized. Subsequently, the current best fitness predictor is used for fitness approximation of candidate solutions. Fitness approximation is performed in the same way as fitness evaluation but using the subset of the dataset specified by the best predictor and not the whole training dataset. The fitness evaluation of predictors is based on the difference between the accurate fitness of fitness trainers and the fitness approximated using a particular predictor.

3. **The population of fitness trainers** is composed of candidate solutions. These candidate solutions, which are called fitness trainers in this

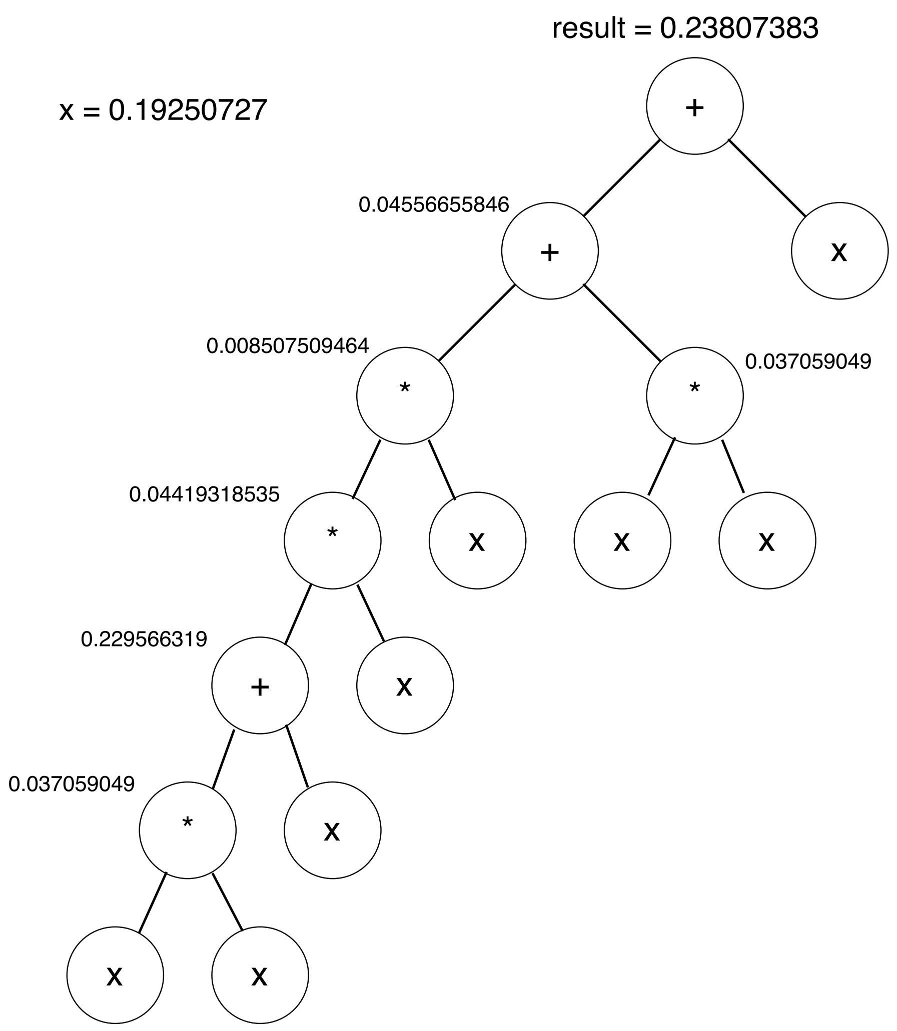

Figure 3.4: Fitness evaluation of the final solution.

population, are used to measure the qualities of fitness approximation of fitness predictors. During initialization, fitness trainers are chosen randomly from the population of candidate solutions, and their accurate fitnesses are evaluated and saved. During the evolution of fitness predictors, one new fitness trainer, which has the highest variance of the fitness approximated by all fitness predictors, is added to the population of fitness trainers. This addition is not performed each generation of fitness predictors to give them time to adapt on the current fitness trainers. The interval of this addition is usually based on the number of generations of fitness predictors. Old trainers can be gradually removed from this population to speed up the fitness evaluation of fitness predictors.

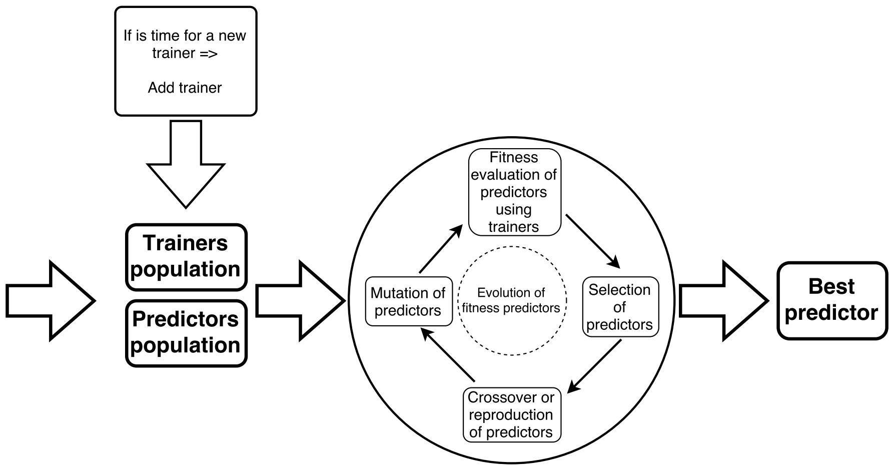
If is time for a new trainer =>
Add trainer
Trainers population
Predictors population
Fitness evaluation of predictors using trainers
Selection of predictors
Crossover or reproduction of predictors
Mutation of predictors
Evolution of fitness predictors
Best predictor

Figure 3.5: Evolution of fitness predictors.

During coevolution, these populations are cooperating and enhancing each other to find the best possible solution for a particular problem. The evolution of fitness predictors can be faster than the evolution of candidate solutions, and more generations of fitness predictors can be evolved per one generation of candidate solutions. In Figure 3.5 is depicted one round of the evolution of fitness predictors. Before the start of evolution, random initialization of predictors and trainers is performed. After that, the process of the evolution of fitness predictors in Figure 3.5 is started from the left side. Specified amount of generations of fitness predictors is evolved, and the best predictor is provided for the fitness approximation of candidate solutions. After initialization, the process in Figure 3.5 is repeated in each generation of candidate solutions.

3.6.1 Advantages

The main benefit of the coevolution of fitness predictors is the reduction of the computational effort. Since the fitness of each candidate solution does not have to be evaluated using the whole dataset, fitness evaluation (approximation in this case) takes less time, and it depends on the size of predictors, which determines the size of the subset used for fitness approximation. The evolution of fitness predictors ensures that the approximation evolves over time.

According to [3], this type of coevolution helps to destabilize local optima, therefore a higher diversification of the candidate solutions is present. It can also contribute to reduce bloat in candidate solutions and find better solutions in a shorter computational time [3].

3.6.2 Coevolution Objectives

3.6.2.1 Solutions

The primary objective is to find the candidate solution with the highest fitness for the current generation. In GP, this objective looks as follows [3]:

$$s^* = \max_{\forall s \in S} f(s),$$

where S is the population of candidate solutions for a particular problem, f is the fitness value obtained by fitness evaluation, and s^* represents the optimal candidate solution. In the coevolution of fitness predictors, fitness evaluation is replaced by fitness approximation using a particular fitness predictor. The same objective using fitness predictors is following [3]:

$$s^* = \max_{\forall s \in S} p_b(s),$$

where f was replaced by p_b, which approximates the fitness using the current best fitness predictor.

3.6.2.2 Predictors

The second objective is to find the best fitness predictor in the current generation. The fitness of predictors is evaluated based on the difference between the accurate fitness of fitness trainers and the fitness approximated using a particular predictor. The objective is to find a predictor with the lowest difference (fitness). It can be seen in the following equation [3]:

$$p^* = \min_{\forall p \in P} \frac{1}{N_t} \sum_{\forall t \in T} |f(t) - p(t)|,$$

where p^* is the best predictor, P is the population of fitness predictors, T represents the population of fitness trainers and N_t is the number of fitness trainers.

Table 3.2: Coevolution parameters

Parameter	value
Crossover rate	75%
Mutation rate	5%
Size of predictor	8 rows
Size of predictor population	8 predictors
Size of trainers population	10 trainers
New trainer interval	100 generations

3.6.2.3 Trainers

The third objective is to periodically search for the most unpredictable candidate solution and add it to the population of trainers. This is measured by the size of variance among all fitness predictors in the current generation using the following formula [3]:

$$t^* = \max_{\forall s \in S} \frac{1}{N_p} \sum_{\forall p \in P} (p(s) - \overline{p(s)})^2,$$

where t^* is the new trainer, N_p is the number of fitness predictors and $\overline{p(s)}$ represents the average predicted fitness for a particular candidate solution across all fitness predictors.

3.6.3 Algorithm Details

According to [18] and [3], algorithm parameters shown in Table 3.2 can be used in the fitness predictors coevolution.

Chapter **4**

Parallel Implementation

Parallel implementation of symbolic regression using GP is built on the foundation of the *JGAP*[1] library, written in the Java language. This library provides a rich implementation of the genetic algorithm and includes the support of GP. The parallel implementation is based on the GP part of *JGAP* and several ways of parallelization are covered as well as the sequential evaluation. Furthermore, the coevolution of fitness predictors is implemented, alongside with possible ways of its parallelization.

[1]Java Genetic Algorithms Package `http://jgap.sourceforge.net`

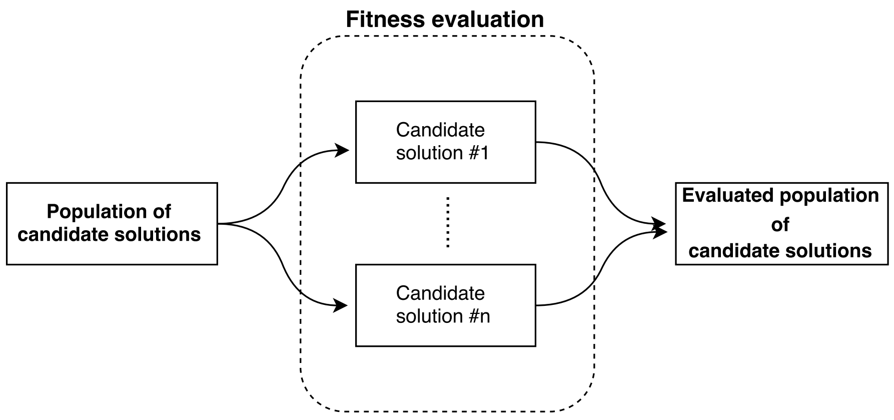
Fitness evaluation
Population of candidate solutions
Candidate solution #1
Candidate solution #n
Evaluated population of candidate solutions

Figure 4.1: Parallel fitness evaluation.

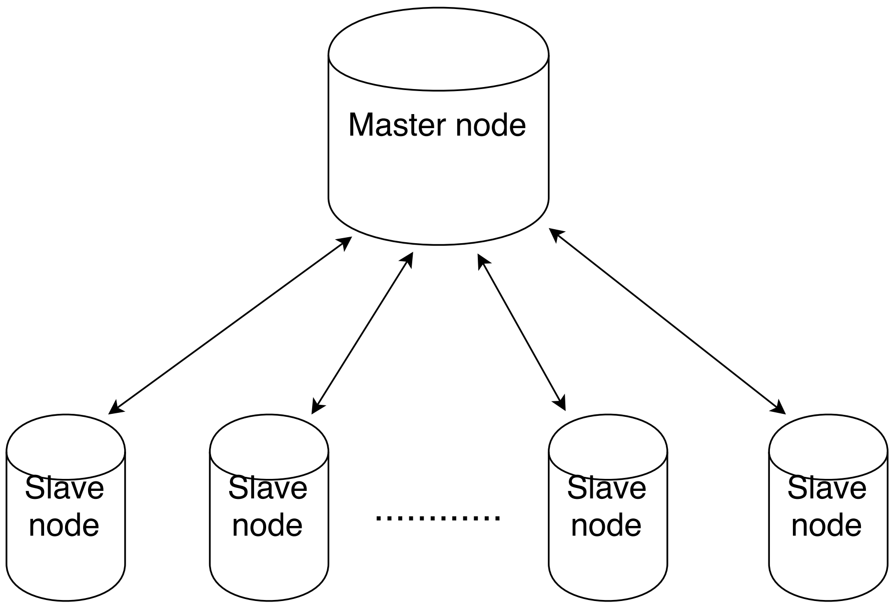

Figure 4.2: Master-slave model.

4.1 Parallelization

Since fitness evaluation can require a large computational effort [5], the parallelization of symbolic regression is established on the parallel fitness evaluation of candidate solutions. During fitness evaluation of one generation, individual solutions are independent of each other. Therefore they can be easily separated for the computation of fitness value. This approach is depicted in Figure 4.1.

Parallelization will be implemented by the master-slave model [5] shown in Figure 4.2. This model is composed of one master node, which is called the driver, and a given number of slave nodes, which are called executors.

The master node is responsible for the whole evolution process except fitness evaluation. Therefore initialization, selection, crossover, reproduction and mutation are all performed on the master node. Fitness evaluation is distributed by the master node to the set of slave nodes, which executes the actual fitness computation for each candidate solution. After evaluation, the values of fitness are collected from slave nodes and saved in particular candidate solutions as their fitness. By using this principle, fitness can be evaluated faster than using the sequential evaluation.

4.2 Technology

Apache Spark[2] is used for the implementation of master-slave model. The implementation of symbolic regression is written in the functional programming language *Scala*[3] and it is based on the *JGAP* library written in Java.

Apache Spark is an open-source engine for fast, large-scale big data processing on a cluster. It provides API[4] for effective distribution of computational tasks between the master node and slave nodes. The total number of available nodes in *Apache Spark* is equal to the number of logical cores of all installed processors. For the fitness evaluation on *Apache Spark*, a *MapReduce* model is utilizied. This model is commonly used in functional programming, and the procedure of this method is to split data, apply an evaluation on them, and combine their results. These functionalities are represented in Scala by the *map* function, which divides data into many smaller parts, and *reduce* function, which synthesizes the results. Therefore, the *MapReduce* model is suitable for parallel fitness evaluation on *Apache Spark* using the Scala language.

4.3 Classes

The implementation is divided into many classes, and the most important ones are shown in Figure 4.3. The classes with thinner frames are used during the coevolution of fitness predictors. The roles of classes are following.

- **SymbolicRegressionProblem** generates the dataset, creates a new instance of *GPGenotype* and starts the evolution process. Following attributes are determined in this class: the desired formula, the function set, the terminal set, variables, the size of the dataset and the type of parallelization. Other more detail parameters can be set using *GPConfiguration* in this class.
- **GPGenotype** includes initialization and the evolution process of candidate solutions, and in the case of the coevolution, also the evolution of fitness predictors. This class holds the population of candidate solutions, the type of the parallel evaluation, the configuration and the best-found candidate solution.
- **GPConfiguration** is used for setting various parameters such as the sizes of populations, the probabilities of crossover, reproduction, and

[2]http://spark.apache.org
[3]http://www.scala-lang.org
[4]Application Programming Interface

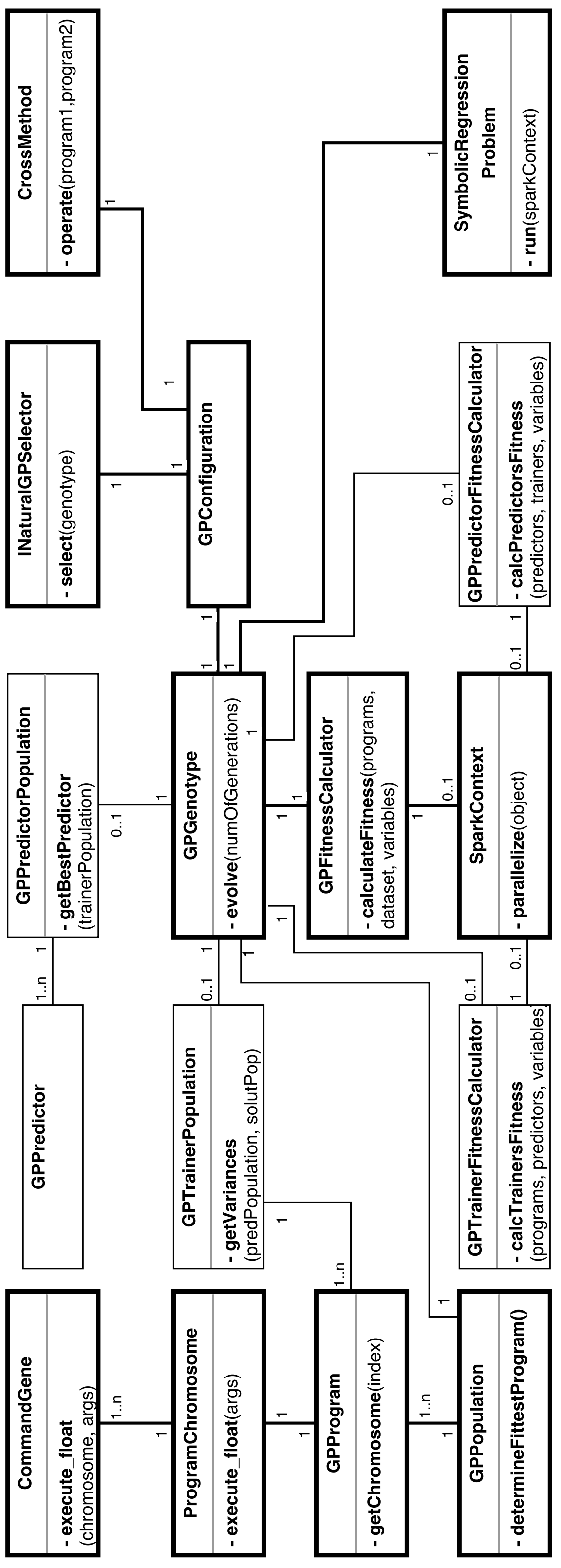
CrossMethod
- operate(program1,program2)
INaturalGPSelector
- select(genotype)
GPConfiguration
SymbolicRegression Problem
- run(sparkContext)
GPPredictorFitnessCalculator
- calcPredictorsFitness (predictors, trainers, variables)
GPPredictorPopulation
- getBestPredictor (trainerPopulation)
GPGenotype
- evolve(numOfGenerations)
GPFitnessCalculator
- calculateFitness(programs, dataset, variables)
SparkContext
- parallelize(object)
GPPredictor
GPTrainerPopulation
- getVariances (predPopulation, solutPop)
GPTrainerFitnessCalculator
- calcTrainersFitness (programs, predictors, variables)
CommandGene
- execute_float (chromosome, args)
ProgramChromosome
- execute_float(args)
GPProgram
- getChromosome(index)
GPPopulation
- determineFittestProgram()
1
0..1
1..n

Figure 4.3: Overview of classes.

mutation and the maximum size of candidate solutions. This class also holds the types of selection and crossover methods using *INaturalGPSelector* class and *CrossMethod* class.

- **GPPopulation** serves as a representation of the population of candidate solutions. It contains all candidate solutions in the form of instances of *GPProgram* class.

- **GPProgram** represents a candidate solution and holds its *ProgramChromosome*, computed fitness value and other parameters.

- **ProgramChromosome** represents the syntax tree of a particular candidate solution, and is composed of instances of *CommandGene*, which represents one node of the syntax tree.

- **GPFitnessCalculator** represents the way of fitness evaluation. The type of possible parallelization can be set by a relevant subclass of this abstract class. In the case of the parallel version of *GPFitnessCalculator*, it holds *SparkContext*.

- **GPPredictorPopulation** holds predictors in the form of intances of the *GPPredictor* class.

- **GPPredictorFitnessCalculator** has the same function as the *GPFitnessCalculator* but for predictors. It can also hold *SparkContext*.

- **GPTrainerPopulation** possesses trainers in the form of *GPProgram*s.

- **GPTrainerFitnessCalculator** is a calculator designated for the fitness evaluation of trainers. In the case of the parallel evaluation, it holds *SparkContext*.

- **SparkContext** is a class provided by *Apache Spark*, which gives access to parallel computation on a cluster using *Apache Spark* engine.

4.3.1 Fitness Calculators

Fitness calculators determine the way of fitness evaluation. They are divided into three types: fitness calculators for candidate solutions (*GPFitnessCalculator* class), for fitness predictors (*GPPredictorFitnessCalculator* class), and for fitness trainers (*GPTrainerFitnessCalculator* class). The last two types are used during the coevolution of fitness predictors. The three types of calculators exist because of optimization of the parallel evaluation. All three populations could be evaluated using the first type, however, it would be slower than using its type of calculator.

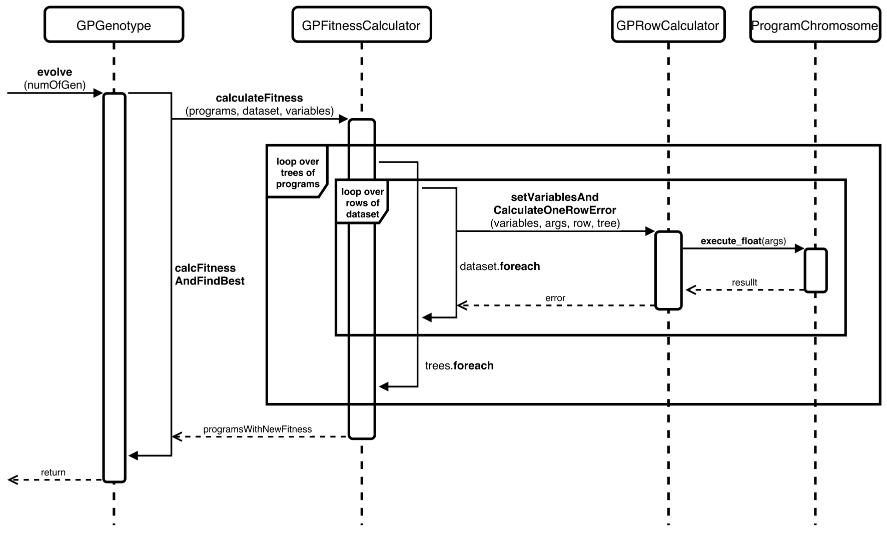

Figure 4.4: Scheme of sequential fitness calculation.

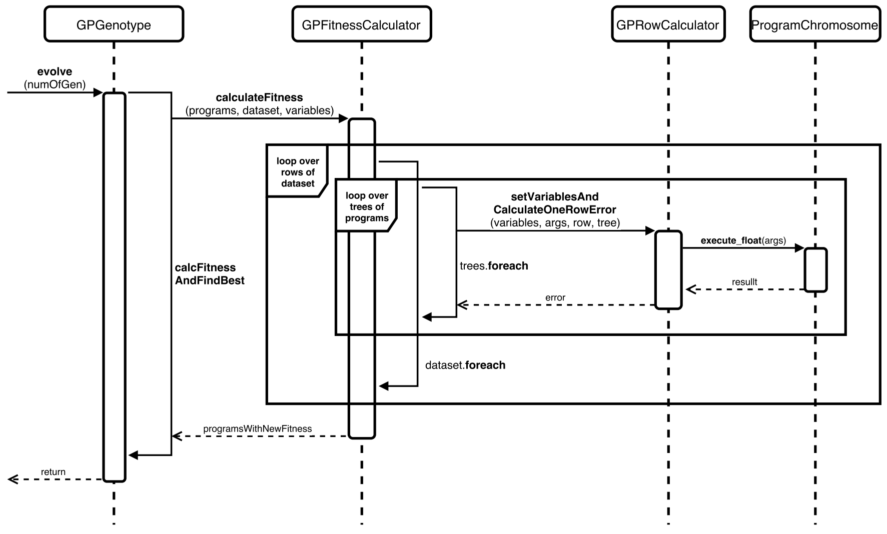

Figure 4.5: Scheme of parallel fitness calculation with a large dataset.

4.3.1.1 Fitness Calculators for Candidate Solutions

The aim of this calculator is to evaluate the fitness of candidate solutions. The parallel versions of fitness calculator on *Apache Spark* are implemented in two ways to support various parameters of the implementation. The types of fitness calculators for candidate solutions are following.

- **SeqGPFitnessCalculator** is used for sequential fitness evaluation. The structure of fitness evaluation using this calculator is depicted in Figure 4.4. In the calculator, iteration is executed over all trees of candidate solutions, and inside this iteration, each tree launches its loop over all rows of the provided dataset. At the end, results of all rows for a particular tree are combined into one value. Fitness values of all candidate solutions are evaluated by using this approach.

- **MultiThreadGPFitnessCalculator** employs the same approach as *SeqGPFitnessCalculator*. The main difference is that the trees of candidate solutions are saved in Scala's parallel collection, which supports multi-thread evaluation using all logical cores of the installed processor.

- **SparkPopGPFitnessCalculator** uses parallelization on *Apache Spark* cluster. During the initialization of this calculator, the dataset is saved to all nodes of the cluster. Therefore, since the dataset does not have to be sent to all nodes in the cluster in each round of evaluation, fitness calculation can be considerably accelerated. This calculator uses the same approach shown in Figure 4.4. The main difference from *SeqGPFitnessCalculator* is parallelization of trees of candidate solutions among the slave nodes of the cluster. Therefore, each executor calculates fitness for a subset of trees of candidate solutions.

- **SparkDatasetGPFitnessCalculator** is designed for fitness evaluation using *Apache Spark* with large datasets, which cannot fit altogether to the memory of each slave node. The structure of this calculator is depicted in Figure 4.5. It iterates over the rows of a dataset parallelized among executors, and fitness evaluation of all trees of candidate solutions for a particular set of rows is executed on single slave nodes. At the end, results for all trees using all rows are combined into one fitness value for each candidate solution.

4.3.1.2 Fitness Calculators for the Coevolution of Fitness Predictors

The second and the third group of fitness calculators are proposed for fitness evaluation of fitness predictors and trainers. Both groups have the same four types of calculators as the first group has. Therefore, there are sequential calculators, multi-thread calculators, parallel calculators parallelizing population, and parallel calculators parallelizing datasets. During execution, the same type of all fitness calculators should be utilized to ensure the same level of parallelization in all populations.

Table 4.1: Main parameters of the implementation

Parameter	Name of method
Initialization method	Ramped half-and-half method
Selection method	Tournament selection
Crossover method	Subtree crossover
Mutation rate	Point mutation
Error measure	Mean error of absolute errors

4.4 Implementation Parameters

The main parameters of the implementation are shown in Table 4.1.

Chapter 5

Experiments

This chapter describes experiments about the scalability of the proposed implementation of symbolic regression. It is composed of three sections. The first section describes the proposed experiments and their details. The second section shows and discusses the results of experiments not using the coevolution of fitness predictors. The last section presents outcomes of experiments conducted using the coevolution of fitness predictors.

5.1 Description

The experiments are divided into two main sections, with the coevolution of fitness predictors, and without it. The same setup of experiments, except the usage of the coevolution, is applied in both of these sections.

All experiments are focused on a speed-up of a parallel execution in comparison with a sequential execution. The speed-up of the parallel evaluation is defined by the following formula:

$$S = \frac{t_s}{t_p},$$

where S represents the speed-up of the algorithm, t_s is the time of the sequential evaluation, and t_p is the time of the parallel evaluation.

5.1.1 Searched Formulas

A set of five formulas, selected from the symbolic regression benchmark functions[5], is used for the experiments. The set contains the following formulas:

$$
\begin{gathered}
f(x) = 0.3 * x * \sin(2 * \pi * x), \\
f(x) = x^3 * \exp(-x) * \cos(x) * \sin(x) * (\sin(x)^2 * \cos(x) - 1), \\
f(x) = x^4 + x^3 + x^2 + x, \\
f(x) = \log(x + 1) + \log(x^2 + 1), \\
f(x, y) = \frac{1}{1 + x^{-4}} + \frac{1}{1 + y^{-4}}.
\end{gathered}
$$

The datasets for symbolic regression are generated using these formulas, where input values are random numbers uniformly distributed over the interval $\langle -5, 5 \rangle$.

5.1.2 Fixed Parameters

Many parameters are fixed during all experiments. Some parameters were mentioned in the previous chapters, in Table 4.1, and in Table 3.2 for the experiments that use the coevolution of fitness predictors. These parameters are used along with parameters in Table 5.1, where the last parameter determines the number of predictor's generations per one generation of candidate solutions.

5.1.3 Variable Parameters

Four parameters are changeable during experimentation. These parameters are stated in the following list, alongside their sets of values:

- the **size of the dataset** $\in \{1000, \underline{10000}, 100000\}$,
- the **size of the population** $\in \{100, \underline{1000}, 10000\}$,
- the **number of available nodes** $\in \{2, 4, 8, 16\}$,
- the **type of parallelization** $\in \{Sequential, SparkPopulation, SparkDataset, MultiThread\}$.

[5]`http://dev.heuristiclab.com/trac.fcgi/blog/gkronber/symbolic_regression_benchmark`

Table 5.1: Fixed parameters for experiments

Parameter	Value
Number of generations	50
Crossover probability	90%
Mutation probability	5%
Elitism	1
Maximum depth of candidate solution	17
Error	mean absolute error
Terminal set	$\{x, y$, random mutable constant$\}$
Function set	$\{+, -, *, /, \text{pow}, \sin, \cos, \exp, \log\}$
Size of tournament	2
Size of predictor's tournament	2
Size of predictor's evolution interval	3

Table 5.2: Fitness calculators used in experiments

Type of parallelization	Used fitness calculator
Sequential	*SeqGPFitnessCalculator*
SparkPopulation	*SparkPopGPFitnessCalculator*
SparkDataset	*SparkDatasetGPFitnessCalculator*
MultiThread	*MultiThreadGPFitnessCalculator*

The names of various types of parallelization are derived from the names of fitness calculators. These fitness calculators can be seen in Table 5.2. Moreover, during the coevolution of fitness predictors, corresponding fitness calculators for predictors and trainers are used as well.

The size of the dataset and the size of the population are not tested for all its combinations. During the testing of various sizes of the dataset, only the underlined size of the population is used. The same applies for tests regarding various sizes of the population where the underlined size of the dataset is used. Therefore, these two parameters are used in six combinations in total.

The number of available nodes denotes the number of available logical cores of processors. For the *MultiThread* type of parallelization, it means the number of logical cores available on a particular computer. For parallelization on *Apache Spark* it denotes the number of logical cores available in a cluster of computers. However, these experiments are conducted on a single computer, where 16 logical cores are available; therefore the *MultiThread* does not have a disadvantage regarding the number of available nodes. On the other hand, the evaluation using *Apache Spark* has a significant advantage in its possible scalability on a cluster, which the *MultiThread* using Scala's parallel collections cannot offer.

Only sixteen nodes (the last option in the set) is tested with all six combinations of sizes of the dataset and the population. All the other options of the number of nodes are examined with two of these combinations. The first combination is the largest dataset and the underlined size of the population. The second one is the highest size of the population with the underlined size of the dataset. By using this approach, $6 + 3 * 2 = 12$ combinations of sizes of the dataset, sizes of the population, and the numbers of available nodes are obtained. Each of these combinations is evaluated using the *SparkPopulation*, the *SparkDataset*, and the *MultiThread*. Since the number of nodes does not affect the sequential evaluation, the *Sequential* is evaluated only with the six combinations of sizes of the dataset and sizes of the population. It is $12*3+1*6 = 42$ combinations of variable parameters on which all five formulas are evaluated using both the coevolution of fitness predictors and classical symbolic regression without the coevolution. In total, $42 * 5 * 2 = 420$ combinations of variable parameters, formulas, and potential usage of the coevolution are conducted in the experiments. Each measurement of time (the speed-up is calculated from these values) is performed five times, and the values of the speed-up represent the average value for a particular setup of parameters.

5.1.4 Types of Experiments

In both sections of actual experiments, four types of experiments with own types of graph are conducted. In each of four graphs, four types of parallelization are shown as four groups of points in a graph, and the y axis represents the speed-up in comparison with the sequential evaluation. The *Sequential* is included as well with the value 1.0. The x axis represents a particular variable parameter on a logarithmic scale (with the base ten for the sizes of the dataset and the population, and with the base two for the number of nodes). The types of graphs and their parameters are following:

1. **Variable size of the dataset**
   - **Fixed parameters:**
     - the number of nodes = 16,
     - the size of the population = 1000.
   - **Variable parameters:**
     - the size of the dataset $\in \{1000, 10000, 100000\}$.
2. **Variable size of the population**

- **Fixed parameters:**
  - the number of nodes = 16,
  - the size of the dataset = 10000.
- **Variable parameters:**
  - the size of the population $\in \{100, 1000, 10000\}$.

3. **Variable number of nodes with the large dataset**
   - **Fixed parameters:**
     - the size of the population = 1000,
     - the size of the dataset = 100000.
   - **Variable parameters:**
     - the number of nodes $\in \{2, 4, 8, 16\}$.

4. **Variable number of nodes with the large population**
   - **Fixed parameters:**
     - the size of the population = 10000,
     - the size of the dataset = 10000.
   - **Variable parameters:**
     - the number of nodes $\in \{2, 4, 8, 16\}$.

5.2 Experiments without the Coevolution of Fitness Predictors

5.2.1 Variable Size of the Dataset

The result of this experiment is depicted in Figure 5.1. The *MultiThread* has the highest speed-up (around six) in all three sizes of the dataset, however, with the increasing size of the dataset, the *SparkPopulation's* speed-up is getting closer to the *MultiThread*. The *SparkDataset's* speed-up is also increasing with the increasing size of the dataset (up to 3.88), but it is lower in comparison with the *SparkPopulation*.

5.2.2 Variable Size of the Population

Similar results as in the previous experiment can be observed in Figure 5.2. The *MultiThread's* speed-up is the highest one. The *SparkPopulation* is slower than the *MultiThread*, but the speed-up is increasing with the size of the population. Using the *SparkDataset*, the speed-up does not increase significantly with larger populations.

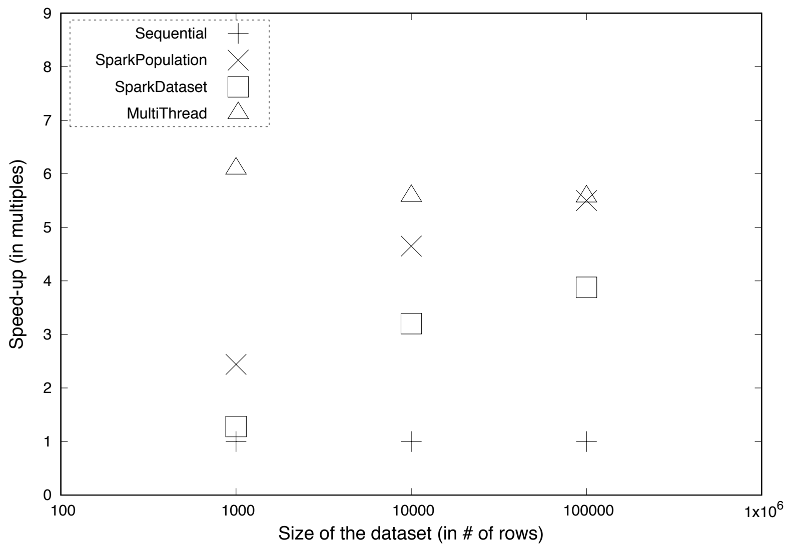

Figure 5.1: Graph showing the speed-up with a variable size of the dataset and without the coevolution.

5.2.3 Variable Number of Nodes with the Large Dataset

There is a positive correlation between the number of nodes and the speed-up. According to Figure 5.3, all types of evaluation (except the *Sequential*) evince higher speed-ups (up to 5.57 using the *MultiThread*) with the increasing number of nodes. The *SparkDataset* is again lagging behind the *MultiThread* and the *SparkPopulation* as in the two previous experiments.

5.2.4 Variable Number of Nodes with the Large Population

In this experiment, depicted in Figure 5.4, similar results as in the previous experiment with the large dataset are noticeable. However, the speed-up of the *SparkDataset* is considerably lower than in the previous experiment (3.88 vs. 2.38 using sixteen nodes).

5.2.5 Discussion

In all four experiments described above, it is evident that the parallel evaluation provides a significant speed-up in comparison with the sequential evaluation. The highest speed-up was 6.71 for the *MultiThread* in the second experiment using the largest population. In general, the *MultiThread* demonstrated the highest speed-up in all four experiments, the *SparkPopulation* has

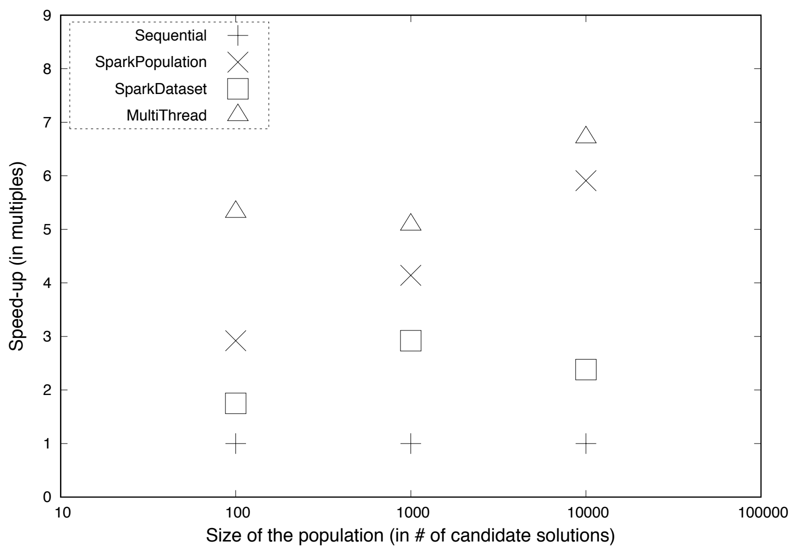

Figure 5.2: Graph showing the speed-up with a variable size of the population and without the coevolution.

the second highest speed-up (up to 5.91), and the *SparkDataset* is the third (up to 3.88). However, each of them is more suitable in different situations.

The *MultiThread* provides the best speed-up, but it utilizes only logical cores of processors in one computer. In the case of more available computers (a cluster), the *MultiThread* can be executed only on one of them. This attribute provides an advantage to the *MultiThread*, which does not have to perform extra work to support the generality of multiple nodes across the whole cluster of computers. This fact loads both the *SparkPopulation* and the *SparkDataset* with extra tasks. On the other hand, once the extra work is done, the evaluation using *Apache Spark* can benefit from the support of multiple nodes across the cluster. From the experiments can be seen that with the large dataset, the evaluation using the *SparkPopulation* has almost the same speed-up as the *MultiThread.* This indicates that the evaluation using *Apache Spark* is suitable for large amounts of data, and in comparison with Scala's parallel collections, it is scalable across the whole cluster of computers. The *SparkPopulation* provides the best performance on *Apache Spark*, however, it cannot be used with a dataset that will not fit in the memory of each node. For this purpose, the *SparkDataset* is suitable, which supports execution on a cluster with a large dataset.

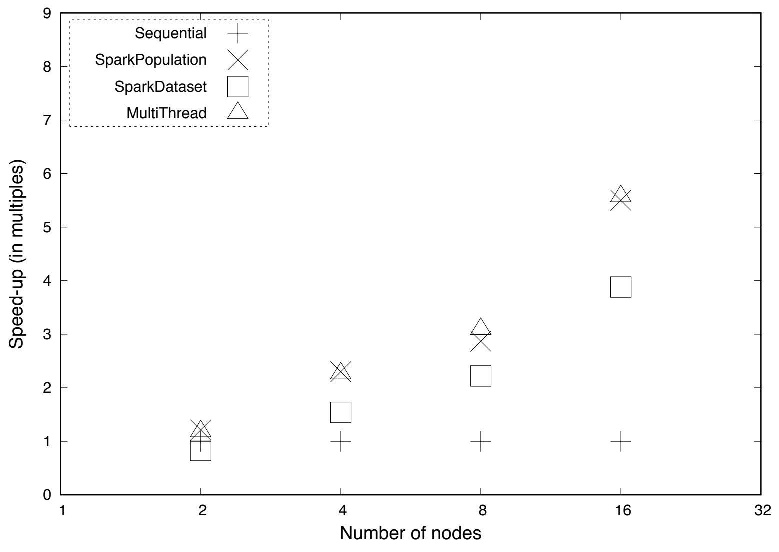

Figure 5.3: Graph showing the speed-up with a variable number of nodes using the large dataset and without the coevolution.

5.3 Experiments with the Coevolution of Fitness Predictors

5.3.1 Variable Size of the Dataset

The results of this experiment are depicted in Figure 5.5. The speed-ups of the *SparkPopulation* and the *SparkDataset* are considerably lower (the highest is 0.12) than using *Sequntial*. Therefore, with the coevolution of fitness predictors, the evaluation using *Apache Spark* is slower in comparison with the sequential evaluation. The highest speed-up achieved by the *MultiThread* is 1.53. This is a significantly lower speed-up than without the coevolution of fitness predictors.

5.3.2 Variable Size of the Population

This experiment, depicted in Figure 5.6, shows similar results as the previous experiment. Only the *MultiThread* is faster than the *Sequential*.

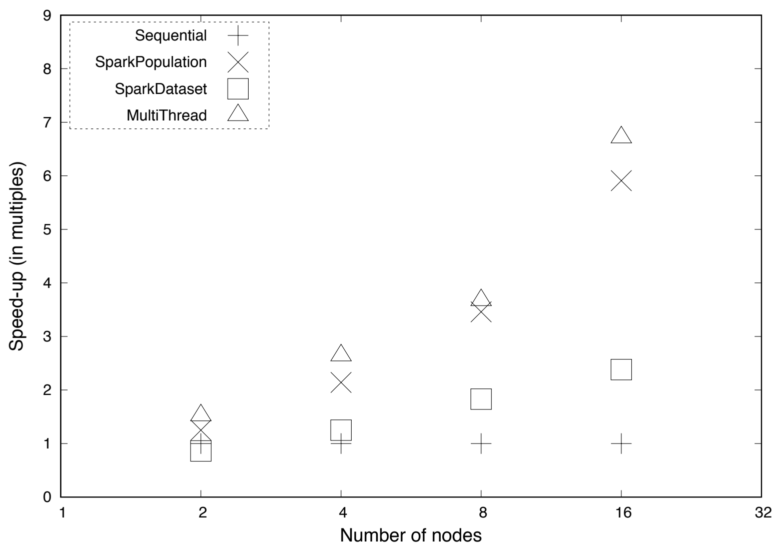

Figure 5.4: Graph showing the speed-up with a variable number of nodes using the large population and without the coevolution.

5.3.3 Variable Number of Nodes with the Large Dataset

Similar results as in the previous experiment can be seen in Figure 5.7. The highest speed-up is 1.53 using the *MultiThread* and sixteen nodes.

5.3.4 Variable Number of Nodes with the Large Population

This experiment, which results are depicted in Figure 5.8, indicates similar results as in the three previous experiments using the coevolution of fitness predictors. Only the *MultiThread* tends to be faster than the *Sequential* (in this case only using eight or sixteen nodes).

5.3.5 Discussion

According to these four experiments, it can be concluded that the coevolution of fitness predictors is not suitable for parallel evaluation (and especially on a cluster). All results using *Apache Spark* were slower than the sequential evaluation. Using the *MultiThread*, the highest speed-up was 1.53.

During the coevolution of fitness predictors, smaller amounts of data have to be evaluated, in comparison with the parallel implementation without the

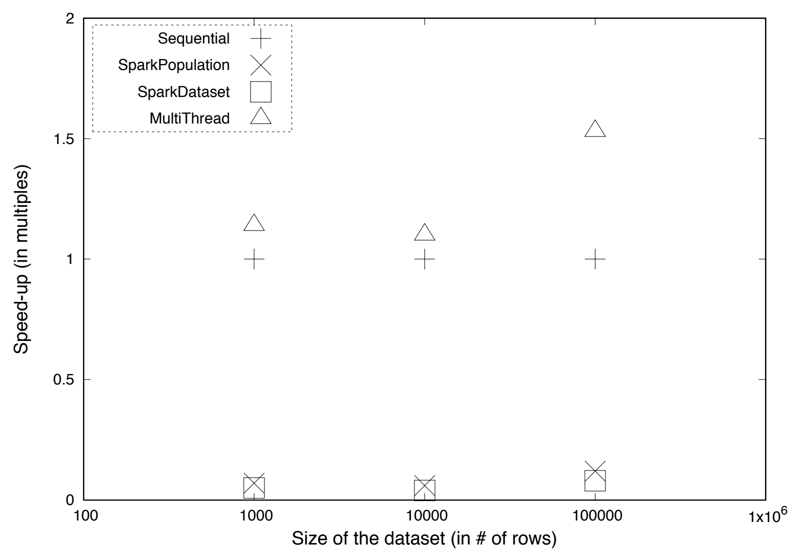

Figure 5.5: Graph showing the speed-up with a variable size of the dataset and with the coevolution.

coevolution of fitness predictors. This fact produces a high granularity of the parallel implementation with the coevolution. Besides the population of candidate solutions, also the populations of predictors and trainers have to be evaluated. These populations are of a very small size (eight predictors and ten trainers in this case), and any overhead, necessary for the parallelization, is not worth its results (especially in the case of *Apache Spark*). Also, the population of fitness predictors requires an extra time for its evolution (selection, crossover, and mutation), which is done sequentially and is difficult to parallelize. Furthermore, during the evaluation of candidate solutions, only small subsets of the dataset (based on the size of the predictor, which is eight in this case) are used for the evaluation of individuals. Thus, only a small part of the evaluation can be parallelized. All these facts, mentioned above, generates an inability of the coevolution of fitness predictors to be well parallelized.

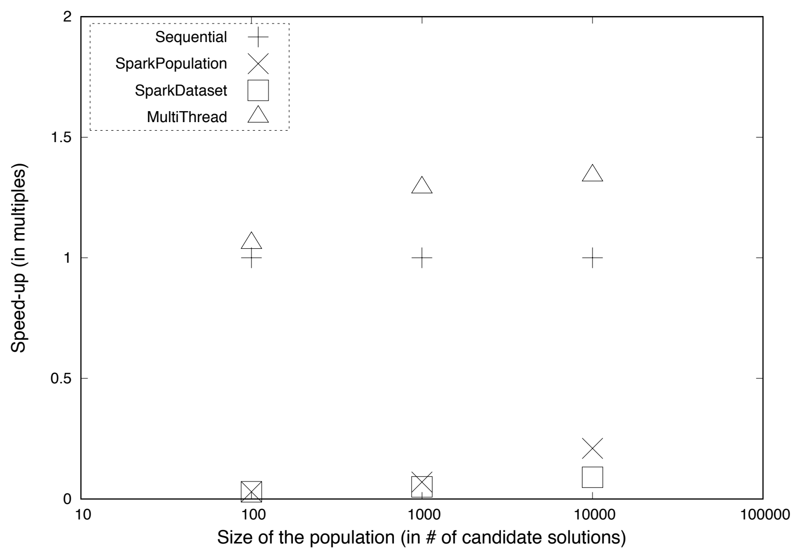

Figure 5.6: Graph showing the speed-up with a variable size of the population and with the coevolution.

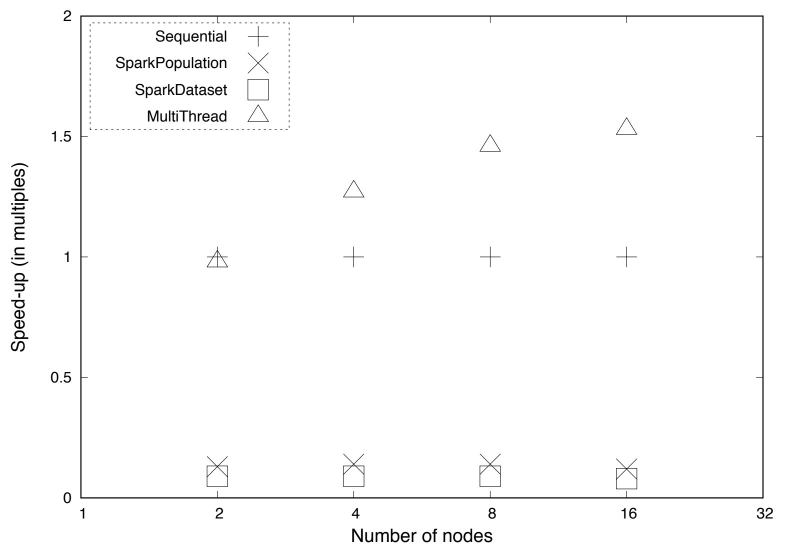

Figure 5.7: Graph showing the speed-up with a variable number of nodes using the large dataset and with the coevolution.

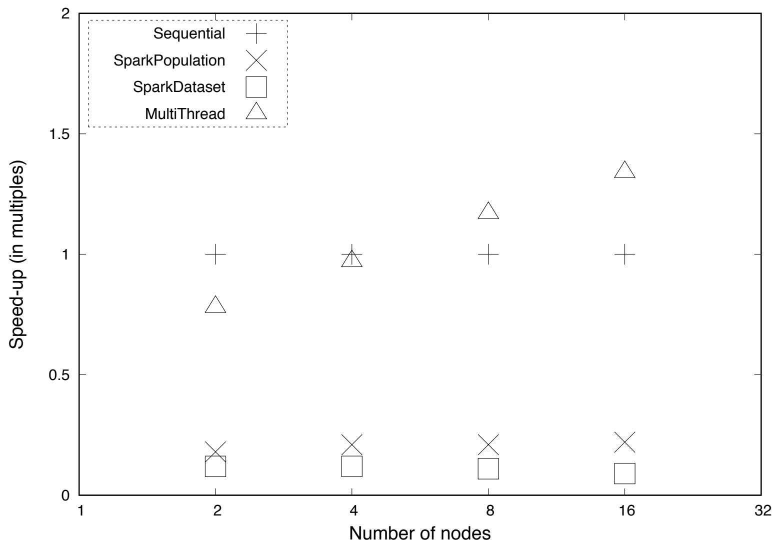

Figure 5.8: Graph showing the speed-up with a variable number of nodes using the large population and with the coevolution.

Conclusion

This thesis was focused on symbolic regression, which searches for a symbolic formula that best approximates values of the function used for generation of the dataset. Symbolic regression is typically approached by genetic programming, which belongs to the group of evolutionary algorithms. Evolutionary algorithms, genetic programming, and symbolic regression were theoretically described in detail. Based on this description, a parallel implementation of symbolic regression using genetic programming was proposed.

Symbolic regression was implemented in two versions: without the coevolution of fitness predictors, and with the coevolution of fitness predictors. The functional programming language Scala and the *Apache Spark* engine, which enables parallel execution on a cluster, were utilized for the parallel implementation. Two types of possible parallelizations using *Apache Spark* and one type using Scala's parallel collections were implemented. On both versions of the implementation, without the coevolution of fitness predictors, and with the coevolution of fitness predictors, experiments about the scalability were performed.

The experiments evaluating the scalability of the parallel implementation showed that symbolic regression not using the coevolution of fitness predictors can be significantly speeded up. The highest speed-up, 6.71, was achieved using Scala's parallel collections. However, Scala's parallel collections cannot be used on a cluster. These collections utilize only logical cores of processors in one computer. For an execution on a cluster the best performance (with the speed-up of up to 5.91) is provided by the parallelization of the population of candidate solutions using *Apache Spark*. Nevertheless, the evaluation using *Apache Spark* needs to perform extra work to support the generality of multiple nodes across the whole cluster of computers. This is the main reason for the slower evaluation using *Apache Spark* in comparison with Scala's parallel collections. Once the extra work is done, the parallel evaluation can benefit from the computational power of the whole cluster. Therefore, the scalability of the evaluation using *Apache Spark* is significantly better. Also, the results of the experiments indicated that with the large dataset, the speed-up using the parallelization of the population using *Apache Spark* was almost as high as the parallelization using Scala's parallel collections (5.50 versus 5.57, respectively). It signals that the evaluation using *Apache Spark* is suitable for large amounts of data, and in comparison with Scala's parallel collections it is scalable across the whole cluster of computers. On the other hand, the parallelization of the population using *Apache Spark* cannot be utilized in the case of a large dataset that cannot fit into the memory of each node. For this purpose, the parallelization of the dataset using *Apache Spark* was implemented. The speed-up of this type of parallelization, based on the experiments, was up to 3.88.

The experiments about the parallel implementation of symbolic regression using the coevolution of fitness predictors showed that the coevolution is not convenient for parallelization. All types of the parallelization using *Apache Spark* demonstrated slower evaluation than when using the sequential evaluation (the highest speed-up value was 0.21). This fact is caused by a higher granularity of the parallel implementation with the coevolution of fitness predictors. In addition to the population of candidate solutions, also the populations of predictors and trainers have to be evaluated. These populations are very small (eight predictors and ten trainers in this case), and any overhead, necessary for the parallelization, is not worth its results (especially in the case of *Apache Spark*). Furthermore, the fitness predictors require an extra time for its evolution which is difficult to parallelize. Also, during the evaluation of candidate solutions, only small subsets of the dataset are utilized for the evaluation of individuals. Therefore, only a small part of the evaluation can be parallelized and using *Apache Spark*, there is no speed-up at all. Only the parallelization using Scala's parallel collection was faster than the sequential evaluation. However, the speed-up is not as significant as in the experiments without the coevolution (up to 1.53).

In conclusion, the parallel evaluation of classical symbolic regression not using the coevolution of fitness predictors can significantly speed up the evaluation process and is well scalable using *Apache Spark* across the whole cluster of computers. On the other hand, symbolic regression using the coevolution of fitness predictors is not suitable for the parallel evaluation due to its high granularity.

Bibliography

[1] Schmidt, M. D.; Lipson, H. Incorporating expert knowledge in evolutionary search: a study of seeding methods. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, ACM, 2009, pp. 1091–1098.

[2] Schmidt, M.; Lipson, H. Age-fitness pareto optimization. In *Genetic Programming Theory and Practice VIII*, Springer, 2011, pp. 129–146.

[3] Schmidt, M. D.; Lipson, H. Coevolution of fitness predictors. *IEEE Transactions on Evolutionary Computation*, volume 12, no. 6, 2008: pp. 736–749.

[4] Koza, J. R. Genetic programming: on the programming of computers by means of natural selection, volume 1. MIT press, 1992.

[5] Poli, R.; Langdon, W. B.; et al. A field guide to genetic programming. Lulu. com, 2008.

[6] Simon, D. Evolutionary optimization algorithms. John Wiley & Sons, 2013.

[7] Goldberg, D. E. Genetic algorithms in search, optimization and machine learning ‘addison-wesley, 1989. *Reading, MA*, 1989.

[8] Whitley, L. D.; et al. The GENITOR Algorithm and Selection Pressure: Why Rank-Based Allocation of Reproductive Trials is Best. In *ICGA*, volume 89, 1989, pp. 116–123.

[9] Langdon, W. B.; Poli, R. Introduction. In *Foundations of Genetic Programming*, Springer, 2002, pp. 1–16.

[10] Poli, R.; Langdon, W. B. On the search properties of different crossover operators in genetic programming. *Genetic Programming*, 1998: pp. 293–301.

[11] Langdon, W. B. Size fair and homologous tree crossovers for tree genetic programming. *Genetic programming and evolvable machines*, volume 1, no. 1-2, 2000: pp. 95–119.

[12] Kinnear, K. E. Evolving a sort: Lessons in genetic programming. In *Neural Networks, 1993., IEEE International Conference on*, IEEE, 1993, pp. 881–888.

[13] Angeline, P. J. An investigation into the sensitivity of genetic programming to the frequency of leaf selection during subtree crossover. In *Proceedings of the 1st annual conference on genetic programming*, MIT Press, 1996, pp. 21–29.

[14] Langdon, W. B. The evolution of size in variable length representations. In *Evolutionary Computation Proceedings, 1998. IEEE World Congress on Computational Intelligence., The 1998 IEEE International Conference on*, IEEE, 1998, pp. 633–638.

[15] McKay, B.; Willis, M. J.; et al. Using a tree structured genetic algorithm to perform symbolic regression. In *Genetic Algorithms in Engineering Systems: Innovations and Applications, 1995. GALESIA. First International Conference on (Conf. Publ. No. 414)*, IET, 1995, pp. 487–492.

[16] Kinnear, K. E. Fitness landscapes and difficulty in genetic programming. In *Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on*, IEEE, 1994, pp. 142–147.

[17] Schoenauer, M.; Sebag, M.; et al. Evolutionary identification of macro-mechanical models. *Advances in genetic programming*, volume 2, 1996: pp. 467–488.

[18] Schmidt, M. D.; Lipson, H. Data-mining dynamical systems: Automated symbolic system identification for exploratory analysis. In *ASME 2008 9th Biennial Conference on Engineering Systems Design and Analysis*, American Society of Mechanical Engineers, 2008, pp. 643–649.

Appendix A

Acronyms

API	Application Programming Interface
GP	Genetic Programming
EA	Evolutionary Algorithm
JGAP	Java Genetic Algorithms Package

Appendix B

Contents of enclosed CD

```
│ readme.txt ........................ the file with CD contents description
├─ src ....................................... the directory of source codes
│  ├─ impl ...................................... implementation sources
│  └─ thesis .............. the directory of LaTeX source codes of the thesis
└─ text ........................................... the thesis text directory
   └─ DP_Malicek_Tomas_2017.pdf ........... the thesis text in PDF format
```