Linear Genetic Programming with Experience

A thesis

submitted in fulfilment

of the requirements for the degree

of

Master of Science (Research)

at

The University of Waikato

by

Liang Liu

THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

2015

Abstract

A novel method of using Machine Learning (ML) algorithms to improve the performance of Linear Genetic Programming (LGP) is studied. This method uses structures, which are called Experience Models (EMs), to organize the trained ML models. EMs are used for different mutate actions of the mutation operator in LGP. The purpose of using EM is to regulate the random search performed by the mutation operator. The aim of using EMs is to let the suitable candidates have higher chances to be selected.

In this study, two sources of knowledge are used to create the training sets that are used to train ML models. The first source is the pre-existing knowledge of symbolic regression. This knowledge reflects the effect of adding one math function segment to another math function segment. The second source is the knowledge generated during the evolution of LGP. This knowledge reflects the effect of using different gene components at different chromosome indexes on the overall fitness. Based on these two sources of knowledge, two types of EM are designed. They are Static Model (SM) and Dynamic Model (DM). The SM uses ML models trained with the first knowledge source. A SM tries to achieve the aim of using an EM by reducing the size of the candidate sets used by the increase action of the mutation operator. The DM uses ML models trained with the second knowledge source. A DM tries to achieve the aim of using an EM by creating distributions of gene component types, which can reflect the information in the second knowledge source, for change action of the mutation operator. In this study, SM is used only for increase action in the mutation operator; DM is used only for change action in the mutation operator.

From the experiment results, if compared with an LGP, when an LGP using a SM, it tends to need fewer generations to have a hit, at the same time achieving similar mean best fitness. In contrary, when used with a DM, an LGP do not show performance improvements.

Acknowledgements

The author owes gratitude to his supervisor Dr. Michael Mayo for support, valuable advice, and great freedom in research.

Liang Liu

July 2015

Contents

List of Figures

List of Tables

List of Algorithms

1. Introduction

1.1 Overview

This thesis studies a novel method of using Machine Learning (ML) models to improve the performance of Linear Genetic Programming (LGP) (Brameier, 2004). The idea of this method is by adding a layer of organize structure called Experience Model to organize trained ML models as classifiers. When recombination operators need to perform, the EMs will first let the ML models give predictions for given individuals, then give suggestions to recombination operators. The actual change to the given individuals by the recombination operators will be performed based on these suggestions. That is, EMs can influence the behavior of recombination operators which will then give influence to the evolution of LGP.

A novel gene representation and new recombination operators are adopted for LGP to simplify the utilization of ML models. To demonstrate the effect of using ML models with less performance influences from recombination operators, all recombination operators are designed to have only basic functionality, only one function if possible.

In this study, two sources of knowledge are used to create training sets to train ML models. They are knowledge acquired before and during an LGP run. The trained ML models are used in EMs. There are two types of EM, Static Model (SM) and Dynamic Model (DM). SM uses ML models trained with the training sets derived from the pre-existing knowledge of symbolic regression, which can be acquired before an LGP run. ML models in a SM will not change during an LGP run. On the other hand, DM uses ML models trained with the knowledge generated during the LGP evolution. The ML models in a DM will be updated at each generation. A third model, Hybrid Model (HM) is also studied; it is a model that uses a SM and a DM at the same time. Therefore, the LGP types used in this study are LGP, LGP with SM, LGP with DM, and LGP with HM.

In this study, two sets of recombination operators are used: the first set is roulette wheel selection, single-point uniform crossover, and single-action mutation; the

second set is roulette wheel selection, multi-point uniform crossover, and multi-action mutation (for more details see Chapter 3).

In this study, EMs are used only for mutation operator. In all three actions of the mutation operator, increase, change, and decrease, SM is used only for the increase action; DM is used only for the change action. Decrease action is not modified. In four LGP types, an LGP uses a mutation operator without EM; an LGP with SM uses a mutation operator with modified increase action; an LGP with DM uses a mutation operator with modified change action; an LGP with HM uses a mutation operator with both modified increase action and modified change action.

In this study, the influence of an EM is designed to be limited in one gene. The reason for this is the difficulty of creating necessary training sets to train classifiers for the SM, and the computational difficulty of utilizing complex models for the DM.

SM is used to give suggestions for increase action. In the modified increase action, new genes will be added to the last of a chromosome instead of being inserted into random indexes. The function of a SM is to reduce the size of the suggestion sets for increase action at the same time keep the suitable candidate can be selected.

DM is used to give suggestions for change action. In the modified change action, when choosing a new gene component, the choice is made based on the distribution given by the DM instead of using the uniform distribution. The function of a DM is to increase the probability of the gene components that usually appear in individuals that have higher fitness and decrease the probability of the gene components that usually appear in individuals that have lower fitness.

In this study, the symbolic regression benchmark set introduced in Uy et al. (2011) and the concrete dataset introduced in Yeh (1998) are used as benchmark sets. The experiment results are compared in two ways: mean generations needed to have a hit and mean best fitness. From the experiment results, we can see that when used with a SM, an LGP tend to need fewer generations to have a hit at the same time achieving similar mean best fitness. When used with only a DM, an LGP usually does not show performance improvements.

1.2 Motivation

Usually, two types of knowledge that can give influence to our process of solving a problem. They are static knowledge and dynamic knowledge. Static knowledge is the knowledge we learned before we encounter the problem. Usually, it is well structured and often tested. Dynamic knowledge is the knowledge about the problem. It represents how well we understood the problem. This knowledge will change (or say increase) as we learn more about the problem.

The EM is a general framework try to mimic the idea above. The Static Model represents the static knowledge and Dynamic Model represents the dynamic knowledge. In this thesis, ML models are used to learn two types of knowledge.

1.3 Chapter Arrangement

Chapter 1 is used to give an overview to this study. Chapter 2 is used to give general background information for evolutionary algorithms, LGP, symbolic regression, and machine learning. Chapter 3 is used to introduce the LGP model used in this study. Chapter 4 is used to introduce the experience model. Chapter 5 is used to introduce experiment settings and give experiment results. Chapter 6 shows the conclusion of this study.

2. Background

2.1 Evolutionary Algorithms

Evolutionary Algorithms (EAs) are optimization methods inspired by Darwinian Theory (Jones, 2002). They do stochastic optimization by creating diverse samples as individuals each represent a possible solution. At each generation, the algorithm updates a population composed of individuals through recombination operators. EA access the fitness of an individual in the population using problem specific fitness function. If the algorithm reaches the maximum generation, it will pick the best individual recorded as a solution. Alternatively, if any individual satisfies the stop criterion during the evolution, the algorithm will stop and present it as a solution. The abstract of evolutionary algorithms can be seen in Figure 2.1.

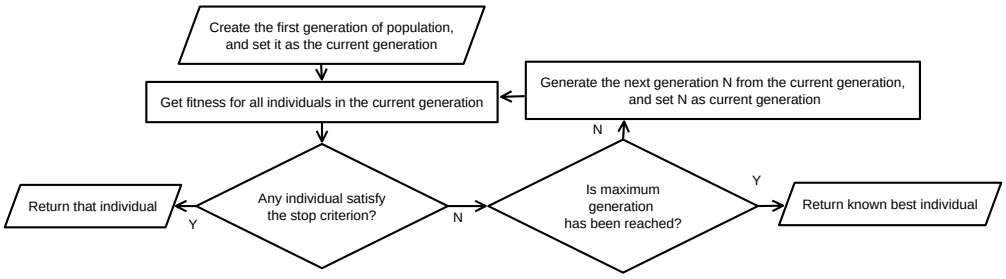


Figure 2.1 Abstract of evolutionary algorithms

Several branches of EA exist (Cantú-Paz and Kamath, 2001; de Castro, 2007), including Evolutionary Strategy (ES), Evolutionary Programming (EP), Genetic Algorithms (GA) and Genetic Programming (GP). Many theories and frameworks were also created based on EA. For example, Estimation of Distribution Algorithms (EDA) (Pelikan et al., 2002; Hauschild and Pelikan, 2011), Evolutionary Dynamic Optimization (EDO) (Nguyen et al., 2012), Genetics-Based Machine Learning (GBML) (Kovacs, 2012), Search Based Software Engineering (SBSE) (Harman et al., 2012) and Learning Classifier System (LCS) (Bacardit et al., 2008).

2.2 Linear Genetic Programming

In Poli et.al (2008), genetic programming (GP) is described as:

> "… an evolutionary computation (EC) technique that automatically solves problems without requiring the user to know or specify the form or structure of the

solution in advance. At the most abstract level GP is a systematic, domain-independent method for getting computers to solve problems automatically starting from a high-level statement of what needs to be done."

Linear Genetic Programming (LGP) is a form of GP (Brameier, 2004). The differences between them mainly lie in the chromosome representation (Brameier and Banzhaf, 2007). As the name of LGP implies, LGP uses linear representation. Figure 2.2 gives a comparison between linear representation and tree representation.[1]

Both tree representation and linear representation are the logical structure of their genetic contents. In tree representation, the atomic structure is node. One node contains one non-terminal or one terminal. The term non-terminal is used for all operators which can be set as inner nodes and cannot be set as leaf nodes (consider the situation when a + is placed at the leaf node). Terminal is the name for all variables and constants, they can be set as leaf nodes and cannot be set as inner nodes (consider the situation when a constant 1 is an inner node and a variable x is the descendent of 1). Therefore, the logical organization of gene components in GP with tree representation can be naturally split into two sets, a non-terminal set and a terminal set. In linear representation, because a list is used instead of a tree, one node has to have all the necessary components to be able to function alone (consider the situation when a list with two × and an 1). Therefore, the terminology of non-terminal and terminal cannot be used to describe nodes in LGP. However, they can be used to describe the components within a gene, which is the atomic structure of the LGP. A gene used in LGP has to contain at least two parts, one terminal and one non-terminal to be able to represent a math function properly.

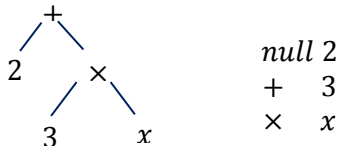


Figure 2.2 Comparison between tree representation and linear representation. The figure on the left is tree representation. The figure on the right is linear representation. They both represent the same polynomial $2 + 3 \times x$.

[1] Figure 2.2 is only used to demonstrate the differences between these representations.
In Brameier (2004) and Brameier and Banzhaf (2007), the basic gene structure used is $opr, regD, reg1, reg2, \ldots$ opr is one of the commands in the command set, reg^* is the register used. $regD$ is the register used to store calculated result. $reg1, reg2$ and the following reg^* are the terms of opr. Here, we can think opr as a non-terminal and all the regs as terminals.
In the right graph, $null$ means the non-terminal does not exist at this index.

Unlike tree-based GP, genes in LGP do not have direct relationship between each other. This makes LGP very versatile which has several meanings. The first is the capability of reorganizing itself and the capability of exchanging genetic materials between individuals. Because there are no direct relationships between any two genes, linear chromosomes are robust. We do not need to worry about the index of a particular gene in the chromosome. Therefore, any number of genes between any two indexes can be safely deleted from one chromosome or exchanged between two individuals. Such action cannot be done with the tree representation (a leaf node cannot be swapped with an inner node). The second is LGP can use recombination operators from different sources. This study borrows uniform crossover from genetic algorithms (see Section 3.2.2).

The recombination operators in LGP are selection, crossover, and mutation. Selection and crossover works together. Selection is used to select individuals from the population. Crossover is used to exchange genetic contents between selected individuals. Mutation is used to replace existing genetic materials with new elements of an individual. Workflow of recombination operators in one generation is shown in Figure 2.3.

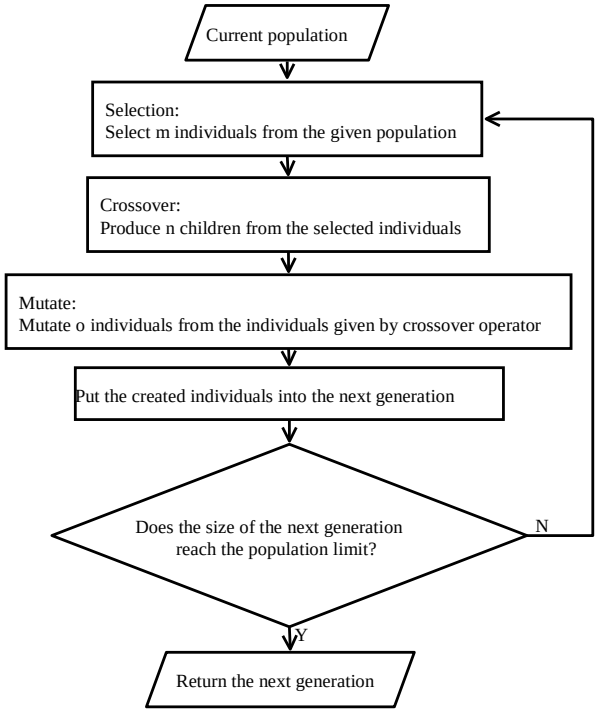


Figure 2.3 Workflow of selection, crossover, and mutation operators in one generation

Elitism is a method that tries to keep the best population for the next generation. When using elitism, the best individuals from the current generation and the individuals in the next generation will be merged together and compete for survival and the worst individuals will be removed from the next generation. The evolution of LGP with elitism can be seen in Figure 2.4.

Many variations and applications based on LGP have been published. Oltean and Groşan (2003) compared several LGP variants used for symbolic regression. Oltean et al. (2009) presented a review for GP variants with linear representations. Recent developments of LGP are listed Table 2.1 and 2.2.

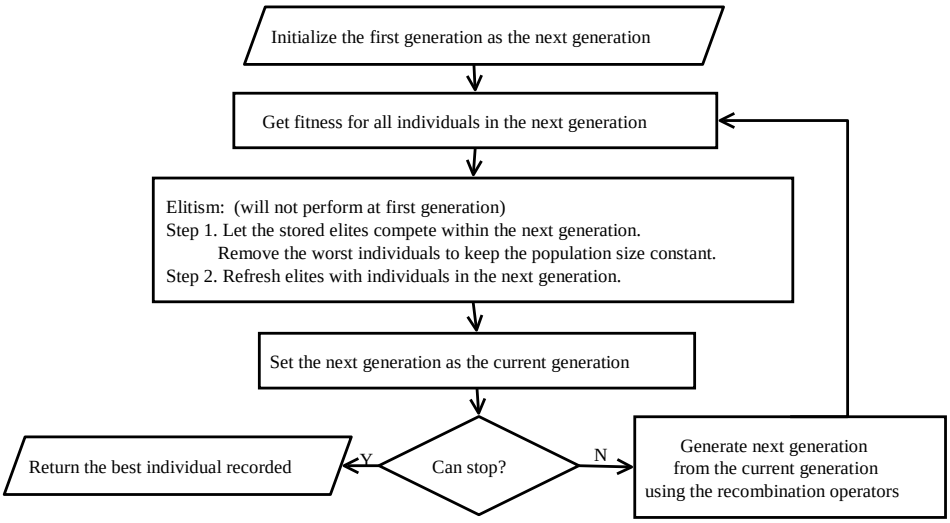


Figure 2.4 Evolution with elitism

Table 2.1 Recent developments on theory of LGP

Article	Topic
Hu and Banzhaf (2009), Hu et al. (2011) and Hu et al. (2013)	Study on various aspects of evolvability in LGP system.
Wilson and Banzhaf (2008)	Gives comparison between Cartesian Genetic Programming and LGP
Watchareeruetai et.al (2011)	Studies the effect of redundancies in LGP.
Harwerth (2011)	Discusses the effect of applying islands in LGP.
Gaudesi et al. (2013)	Proposes a new distance metric for LGP
Downey et al. (2010)	Introduces one child selective crossover for LGP for multiclass object classification.
McPhee and Poli (2008)	Introduces how to use soft assignment in LGP.

2.3 Symbolic Regression

In Genetic Programming (GP) (Koza, 1992), symbolic regression is defined as:

"… finding a mathematical expression, in symbolic form, that provides a good, best, or perfect fit between a given finite sampling of values of the independent variables and the associated values of the dependent variables. That is, symbolic regression involves finding a model that fits a given sample of data."

There are two essential parts in symbolic regression: a terminal set and a function set. The terminal set can have variables, constants, and function calls without any input (Poli et.al, 2008). The function set can also be called as non-terminal set. It usually includes algorithm operators ($+, -, \times, \div$) and mathematical functions like exponent, logarithmic, sine and cosine (Poli et.al, 2008). In this study, the former is called Double Variable Non-Terminals (DVNT), and the latter is called Single Variable Non-Terminals (SVNT).

Table 2.2 Recent studies on applications of LGP

Article	Topic
Wilson et al. (2011)	Uses LGP to produce trade action for stock trading with multiple time frames.
Wilson and Banzhaf (2009)	Uses LGP and development GP with soft memory for stock trading.
Wilson and Banzhaf (2010)	Uses LGP for foreign exchange trading.
Shavandi and Ramyani (2013)	Uses LGP to predict solar global radiation.
Zahiri and Azamathulla (2012)	Uses LGP for the prediction of flow discharge in compound channels.
Zahiri and Azamathulla (2014)	Studies the performance of LGP and M5 tree for predicting flow discharge in compound channels.
Mehr et al. (2013)	Uses LGP for the prediction of streamflow.
Mehr et al. (2014)	Uses LGP for successive-station monthly streamflow prediction.
Guven and Kişi (2011)	Uses LGP to model daily pan evaporation.
Guven and Kişi (2013)	Studies the application of monthly pan evaporation modeling using LGP.
Gandomi (2010)	Uses LGP for the formulation of elastic modulus of concrete.
Gandomi et al. (2014)	Uses LGP for the prediction of shear strength on reinforced concrete beams without stirrups.
Saeed et al. (2013)	Uses LGP to predict the strength of concrete under multiaxial compression.

2.4 Machine Learning

Machine Learning (ML) is a major research field. It includes many mature algorithms such as linear regression, Artificial Neural Networks (ANN), decision trees, Bayes networks etc.

EA and ML can complement each other. Models of ML algorithms like ANN, decision tree, rule-based classifier, Bayesian networks, and support vector

machines can be generated through EA (Cantú-Paz and Kamath 2001; Larrañaga et al., 2013; Kumar and Beniwal, 2013). On the other hand, ML models can also be used in EA. Examples are learnable evolution model (Wojtusiak, 2009; Cervone et al., 2000) and cultural algorithms (Reynolds, 1994; Coello and Becerra, 2002; Reynolds and Peng, 2005). Zhang et al. (2011) gave a review on recent applications of using ML algorithms in GP.

In general, when using ML algorithms, there are four steps:

1) The generation or acquisition of training set

In this step, the training set used to train ML models is acquired. The class of each instance in the training set is labeled. Training set is a dataset contain rows of instances. Each instance contains several attributes, each attribute occupy one column of the training set. Usually, the last attribute, or the last column is the class attribute. However, sometimes this can be changed. The class value in the class attribute of an instance is the learning target for the ML algorithm.

2) Preprocessing

The aim of this step is to make the training set more suitable for training. If the training set is already appropriate, this step is not necessary. Actions in this step may include removing or replacing inappropriate values in the training set, e.g. null values; standardizing or normalizing the training set; and reducing the dimension of the training set.

3) Training of the ML models

In this step, a ML model is trained with the training set. During this step, usually the performance of several ML algorithms needs to be compared in order to find the algorithm that can give the highest accuracy. Often, a ML algorithm needs to be tuned (changing its parameters) to make the training more efficient. There are many methods to evaluate the trained models. The most usual one of them is the 10-fold crossover[1]. The training set can also be used for evaluation purpose, especially when the training set is small. The usual desired state for the trained models is to be neither overfit (give good prediction accuracy on the training set

[1] 10-fold crossover is a method which use 10% of the training set as the test set to evaluate the training performance. The training accuracy is the mean accuracy on all 10 test sets.

but give bad prediction accuracy on the test set[1]), nor underfit (cannot give desired prediction accuracy at all).

4) The use of trained models

After having a ML model, it can be used to give predictions on new samples. However, these samples might need to go through step 2 to make them suitable.

[1] Test set have the same composition as training set. Usually, test set is a part of the training set that is purposely left out to test the training performance. Sometimes, test set is a separate data set, which may or may not have the same origin as the training set, and the sole purpose of it is to test the training performance.

3. Linear Genetic Programming in This Study

In this study, a novel gene representation is adopted for LGP. The reason of using it is to make the use of EM more convenient. The crossover and mutation operators adopted in this study have only basic functions. The single-point uniform crossover and multi-point uniform crossover can only exchange genes between two individuals and cannot cause chromosome length changes. The mutation operators separate their different functions into different actions (increase action, decrease action and change action), so that each action has only one function (increase the length of a chromosome, decrease the length of a chromosome and mutate one gene in the chromosome). The aim of using these operators is to minimize the influence from the recombination operators on the evolution of LGP to make the analysis on the performance improvements of using EM with LGP more straightforward. Roulette wheel selection is used in this study. The reason of using it is that the crossover and mutation operators used are weak operators that only introduce minor changes into individuals. In order to keep the diversity in population[1], roulette wheel selection is used.

3.1 Gene and Chromosome Representation[2]

When considering a standard math function, it is simple to separate it into segments as shown in Figure 3.1.

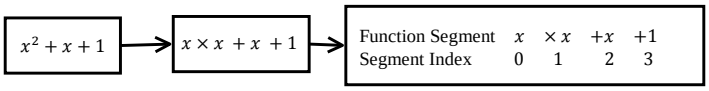


Figure 3.1 Separate math function into segments

If we write these math function segments vertically, we can have an LGP program as shown in Figure 3.2.

[1] Less surviving pressure can keep the diversity in the population to some degree.
[2] The structure of this representation is designed with only the benchmark set I (see section 5.1) in mind. Although this gene representation can represent other functions in other benchmark sets, it is not the author's intention to create a general representation.

LGP program	
Index	Gene
0	x
1	$\times x$
2	$+x$
3	$+1$

Figure 3.2 An LGP program

If we include parentheses, this gene representation can be written as:

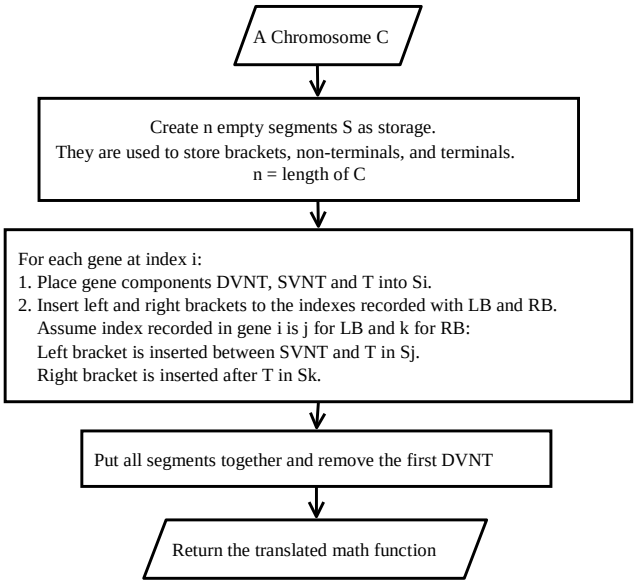
LB RB DVNT SVNT , (3.1)

where LB and RB are the indexes in the chromosome. They represent where the brackets will be placed in the translated math function. DVNT stands for double variable non-terminals, including $+, -, \times, \div$. SVNT stands for single variable non-terminals, including $null$[1], sin, cos, exp and log. T stands for terminals. It can be seen as a set of variables and constants. In this study, a left bracket will always be placed between a SVNT and a T; a right bracket will always be placed after a T. If we have a chromosome as shown in Table 3.1, we can translate it into $sin(x) - exp\big(x \times (x)/(1) + cos(1)\big)$ with Algorithm 3.1.

Table 3.1 A chromosome

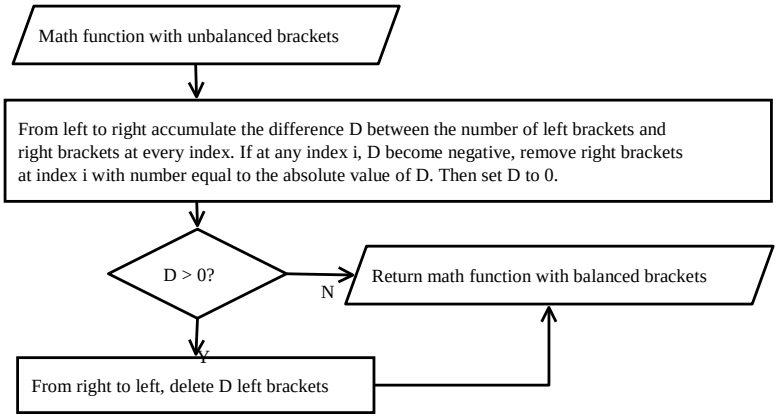
Index	LB	RB	DVNT	SVNT	T
0	1	4	$+$	sin	x
1	2	3	$-$	exp	x
2	3	0	$\times$	$null$	x
3	0	2	$/$	$null$	1
4	5	5	$+$	cos	1

[1] If we want to represent a math function properly, it is necessary to have a symbol that can let situations like $+x$ exist. Therefore, $null$ is included.



Algorithm 3.1 Convert chromosome into math representation

It is possible to have situations when brackets are not balanced.[1] Algorithm 3.2 is used to rebalance the brackets.



Algorithm 3.2 Rebalance brackets by deletion

To explain Algorithm 3.1 and 3.2 clearly, consider the following example:

If we have a chromosome as shown in Table 3.1, the math function of it can be translated by using the following steps:

[1] Although genes are created with balanced brackets, there are various ways to introduce unbalance. The first is the change action of the mutation operator. In it, the LB and RB components are chosen randomly and do not try to keep the balance of brackets. The second is the effect of crossover. A gene that can let an individual have balanced brackets might not able to let another individual have balanced brackets. The third is the decrease action of the mutation operator. This action can delete genes that have brackets paired with other genes.

1) Write down DVNT, SVNT, and T combinations by the order of the gene they belong.

Storage Segments	$+sinx$	$-expx$	$\times x$	$/1$	$+cos1$
Chromosome Index	0	1	2	3	4

2) Insert left and right brackets as LB and RB of the gene at the 0th index.

Storage Segments	$+sinx$	$-\exp(x$	$\times x$	$/1$	$+cos1)$
Chromosome Index	0	1	2	3	4

3) Insert left and right brackets as LB and RB of the gene at the 1st index.

Storage Segments	$+sinx$	$-\exp(x$	$\times (x$	$/1)$	$+cos1)$
Chromosome Index	0	1	2	3	4

4) Insert left and right brackets as LB and RB of the gene at the 2nd index.

Storage Segments	$+sinx)$	$-\exp(x$	$\times (x$	$/(1)$	$+cos1)$
Chromosome Index	0	1	2	3	4

5) Insert left and right brackets as LB and RB of the gene at the 3rd index.

Storage Segments	$+\sin(x)$	$-\exp(x$	$\times (x)$	$/(1)$	$+cos1)$
Chromosome Index	0	1	2	3	4

6) Insert left and right brackets as LB and RB of the gene at the 4th index. Because they are larger than the largest valid gene index, they will be placed at the last valid chromosome index, 4.

Storage Segments	$+\sin(x)$	$-\exp(x$	$\times (x)$	$/(1)$	$+\cos(1))$
Chromosome Index	0	1	2	3	4

7) Remove the first DVNT.

Storage Segments	$\sin(x)$	$-\exp(x$	$\times (x)$	$/(1)$	$+\cos(1))$
Chromosome Index	0	1	2	3	4

8) Write down the first row. We get $sin(x) - exp(x \times (x)/(1) + cos(1))$.

Then consider the chromosome show in Table 3.2. It is the chromosome show in Table 3.1 with an additional gene.

Table 3.2 A chromosome with six genes

Index	LB	RB	DVNT	SVNT	T
0	1	4	+	sin	x
1	2	3	−	exp	x
2	3	0	×	$null$	x
3	0	2	/	$null$	1
4	5	5	+	cos	1
5	5	0	×	$null$	x

To create a function, 1) write down DVNT, SVNT and T combinations by the order of the gene they belong.

Storage Segments	$+sinx$	$-expx$	$\times x$	$/1$	$+cos1$	$\times x$
Chromosome Index	0	1	2	3	4	5

2) Insert all the brackets.

Storage Segments	$+\sin(x))$	$-\exp(x$	$\times (x)$	$/(1)$	$+\cos 1)$	$\times ((x)$
Chromosome Index	0	1	2	3	4	5

3) Delete first DVNT and write down the function.

$$\sin(x)) - \exp(x \times (x)/(1) + \cos 1) \times ((x)$$

Now we have an unbalanced function. We can apply Algorithm 3.2 to rebalance it.

1) Remove excessive right bracket from left to right, we get

$$\sin(x) - \exp(x \times (x)/(1) + \cos 1) \times ((x)$$

2) Remove excessive left bracket from right to left, we get

$$\sin(x) - \exp(x \times (x)/(1) + \cos 1) \times (x)$$

The gene representation described in this section is designed to ease the difficulty on using the Experience Models. There are benefits of using this representation with LGP.

First, it is simpler for Dynamic Model (see Section 4.2) to find less biased relationships between gene components at a chromosome index and the overall fitness when linear representation are used and with all genes are functional (guaranteed by Algorithm 3.1). If using the gene representation introduced in Brameier (2004) or Brameier and Banzhaf (2007), because of the introns (non-functional genes. See Brameier (2004) or Brameier and Banzhaf (2007)) are included, it is not possible to find an unbiased relationship between a gene

component and the overall fitness. The same is true for tree representation because of the bloat.

Second, the insertion index for Static Model (see Section 4.1) is much simpler to be determined. The insertion index is the chromosome length in this study (the counting start from 0, therefore, the index after the last chromosome index is chromosome length). With tree representation, when bloat does not exist, static model has to determine which inner node or leaf node is the most suitable location. If bloat exists, this becomes much more difficult because the index found might not able to let the inserted gene cause the expected effect, as the structure might have changed after the insertion, the nodes in the bloat block might become active. It is the same for LGP with introns. As introns are segments of code that are purposely marked as non-functional (permanent) or functionally jumped through, the introns functionally jumped through might become active after the insertion because some conditions might have changed.

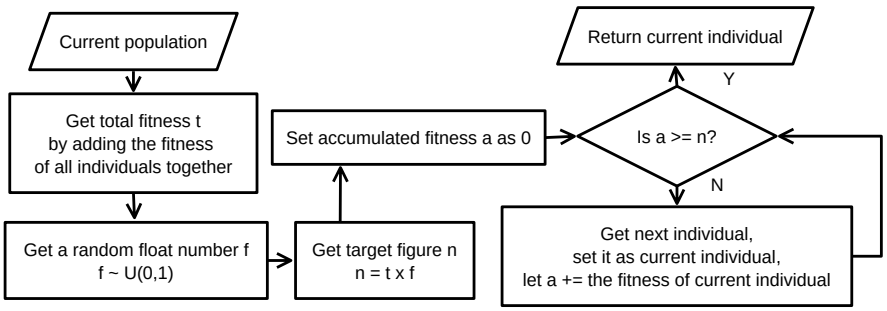
3.2 Initialization

In this study, the genes in the initial generation are generated randomly. For these genes, their LB and RB components are sampled
from $[0, initial\ chromosome\ length - 1]$, their SVNT, DVNT, and T components are chosen randomly from the corresponding non-terminal set or terminal set.

3.3 Recombination Operators

3.3.1 Selection

Roulette wheel selection is used in this study. It is also called as Fitness-Proportionate selection (Luke, 2009). It is a select with replacement method. The algorithm is shown in Algorithm 3.3. The selection operator is used with crossover operator. If crossover operator needs n individuals, roulette wheel selection has to run n times to choose n individuals from the population.



Algorithm 3.3 Roulette wheel selection

3.3.2 Crossover

In this study, the uniform crossover operator described in Luke (2009) is used with two modifications. The first is, instead of exchanging genes with same indexes, genes will be exchanged with random indexes. [1] The second is, instead of making exchanging decisions for all indexes, only one or a percentage of genes will be exchanged between two individuals. [2]

The workflow of the crossover operator is shown in Figure 3.3. The crossover rate is a predefined float number between [0,1]. It is used to control how often the crossover operator should perform.

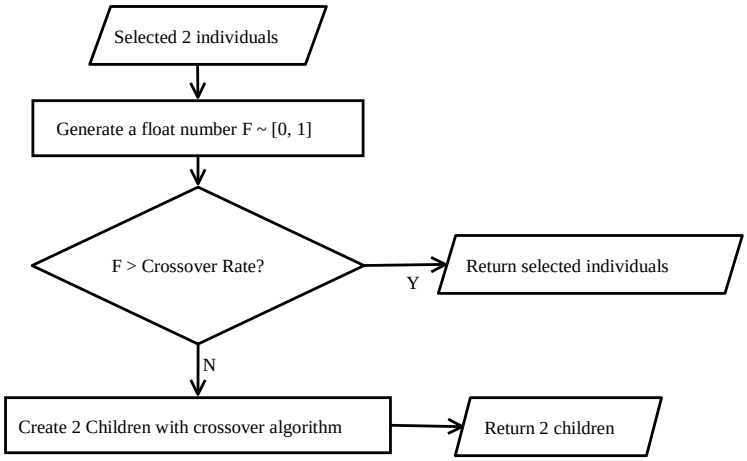


Figure 3.3 Workflow of crossover operator

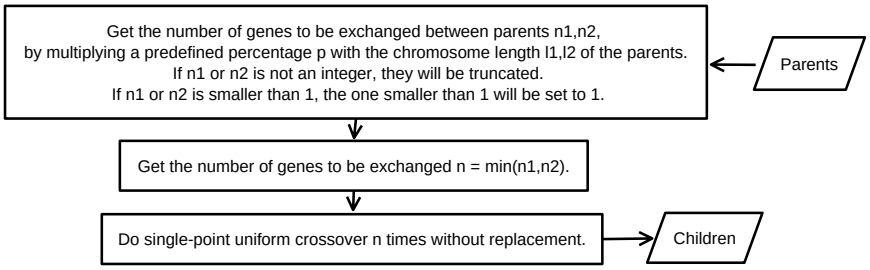
[1] We cannot do the same, because the length of the chromosomes might be different.
[2] As genes are exchanged randomly, we can use this equivalent method. A random float number between (0, 1) is used in the uniform crossover algorithm in Luke (2009) to limit the number of genes exchanged between two individuals. In this study, single-point uniform crossover limits the amount of genes exchanged between two individuals to one; multi-point uniform crossover limits the amount of genes exchanged between two individuals to a fixed percentage of total genes of the shorter chromosome of the two individuals.

1) Single-Point Uniform Crossover

This crossover algorithm uses two individuals. One gene is chosen randomly from each individual, then the selected genes will be swapped between two individuals to create two children.

2) Multi-Point Uniform Crossover

This is an extended version of single-point uniform crossover. Instead of only exchanging one gene, a percentage of genes will be exchanged between two individuals. The algorithm can be seen in Algorithm 3.4.



Algorithm 3.4 Multi-point uniform crossover

3.3.3 Mutation

Mutation is a recombination operator used to change an individual asexually. The workflow of the mutation operator used in this study is shown in Figure 3.4. The mutate rate is a predefined float number between [0,1]. It is used to control how often the mutation operator should perform.

In this study, the functions of mutation operators are separated into three aspects. They are increase action, which will add new genes into a chromosome, decrease action, which will delete existing genes from a chromosome, and change action, which will change the content of an existing gene with different element.

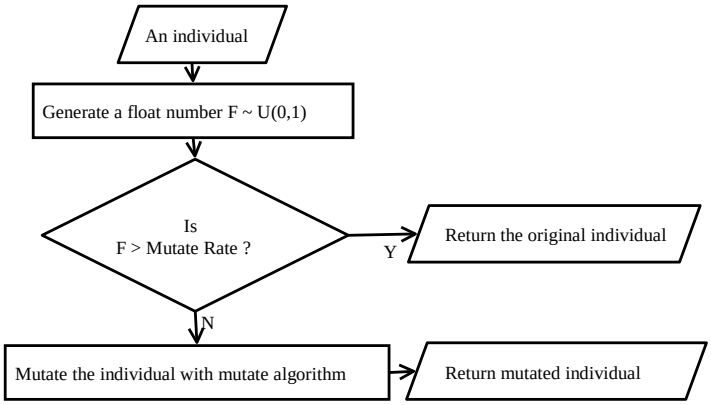
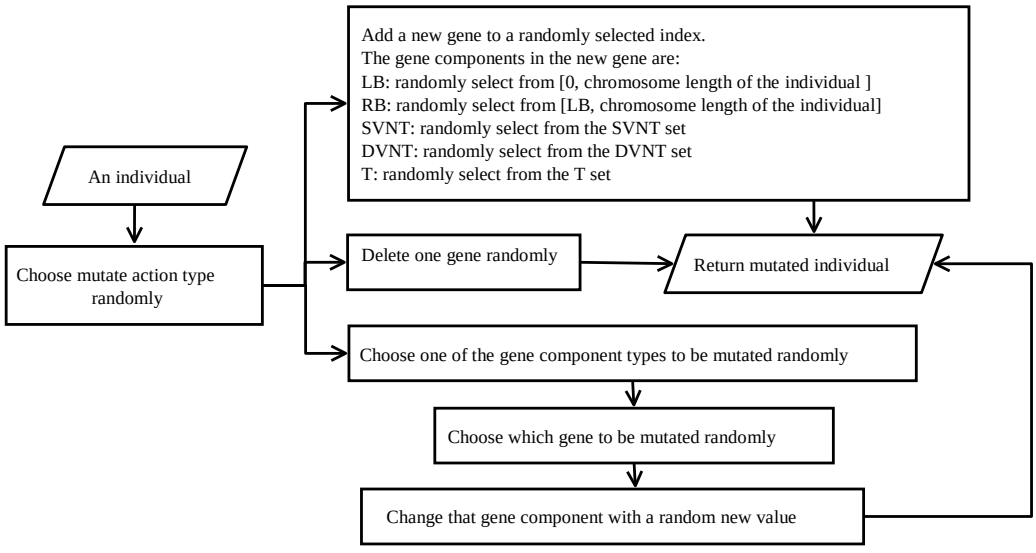


Figure 3.4 Workflow of mutation operator

1) Single-Action Mutation

Single-action mutation operator is shown in Algorithm 3.5. It mutates a candidate only one time with one mutate action. When a candidate has a chromosome length of 1, no decrease action will be performed on it. When a candidate has chromosome length of the maximum chromosome length, no increase action will be performed on it.

In the increase action, the LB component will be acquired by sampling from $[0, cl]$, where cl is the chromosome length of the candidate, the RB component will be acquired by sampling from $[index\ in\ current\ LB, cl]$, the SVNT, DVNT and T components are acquired by choosing randomly from the corresponding non-terminal set or terminal set. In the change action, if there are values other than the existing value, new values will be acquired by choosing randomly from them. For the LB and RB components, new values will be acquired by sampling from $[0, cl - 1]$. If the new value of the LB or RB component is equal to the existing value, it will be minus with 1 to be different from the existing value. In addition, if the new value is 0, it will be increased by 1. In the decrease action, one gene will be deleted randomly from the candidate.



Algorithm 3.5 Single-action mutation operator

2) Multi-Action Mutation

The algorithm for multi-action mutation is shown in Algorithm 3.6. It can be seen as an extended version to the single-action mutation operator. Instead of only

mutate with one action, this operator can mutate an individual multiple times with multiple action types. The maximum mutate times is an integer used to control how many mutate actions can be performed. The maximum chromosome length is a predefined integer to control the maximum chromosome length of a chromosome.

This mutation operator is designed with the emphasis on decrease action to limit the changes in an individual. The aim is to make the mutation operator do not introduce too many new genes into an individual, and therefore, make the population have a more gradual evolution.

3.4 Fitness

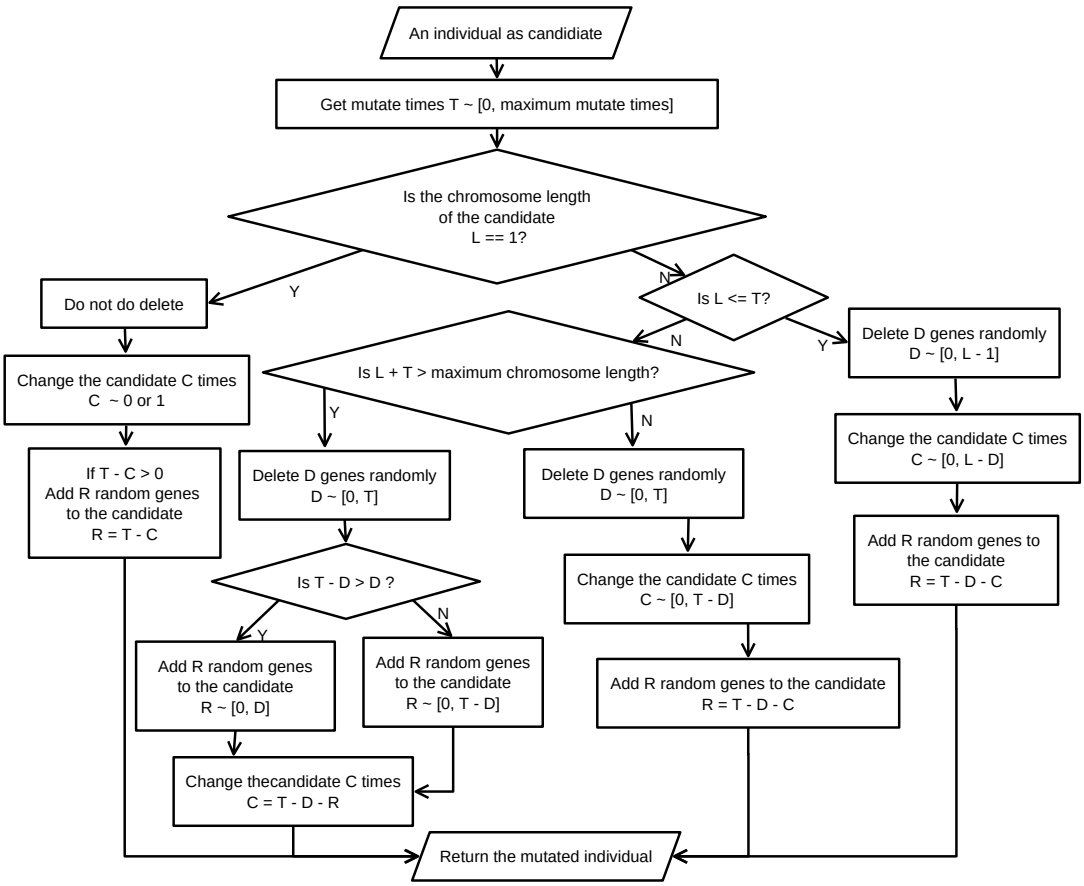
In this study, mean absolute error (MAE) is used to measure the error between the target values and the values of an individual at the training cases of the benchmark problem. It is shown in Equation (3.1):

$$MAE = \frac{1}{n}\sum_{i=1}^{n} abs(e_i), \tag{3.1}$$

where $e_i$is the error at position i. The fitness is calculated by Equation (3.2):

$$fitness = {}^{1}/_{(1+MAE)} \tag{3.2}$$

From Equation (3.2), we can see that the maximum fitness is 1.0.



Algorithm 3.6 Multi-action mutation operator

4. Experience Model

Experience Model (EM) is a technique used to regulate the random search performed by recombination operators. As a population based metaheuristic algorithm, when an LGP is used to acquire the solution of a task, its recombination operators continuously sample the solution space. During their search, some regions of the space that cannot bring gain for fitness are repeatedly visited. The effort of search in these regions is wasted. With regulated recombination operators, the number of such action can be reduced. In this study, EMs are used to regulate the search activities of the mutation operator in LGP.

In this study, the structures used to organize trained machine learning (ML) models are the EMs. There are two types of information used for training: the first type is the pre-existing knowledge that can be acquired before an LGP run; the second type is the information generated during an LGP run. In this study, the EM model uses the ML models trained by the first information type is called as Static Model (SM); the EM model uses the ML models trained by the second information type is called as Dynamic Model (DM). ML models in a SM will not change after their creation; ML models in a DM need to be updated at each generation. A third EM model, Hybrid Model (HM), is also used. Its purpose is to test the performance improvements on LGP when an LGP uses both SM and DM.

SM and DM, when used in mutation operator, only one instance of them can exist at any LGP run. In LGP with SM, only one SM will be used for increase action. In LGP with DM, only one DM will be used for change action. In LGP with HM, one SM will be used for increase action and one DM will be used for change action. The workflow of single-action mutation operator with HM is shown in Figure 4.1. For multi-action mutation operator, as its functions are separated as single-action mutation operator, the usage of EMs in it is the same as the usage of EMs in single-action mutation operator. In both mutation operators, if a SM is used, when three new genes are needed, the increase action will asks the SM three times for support. Decrease action is not modified to keep an important source of randomness.

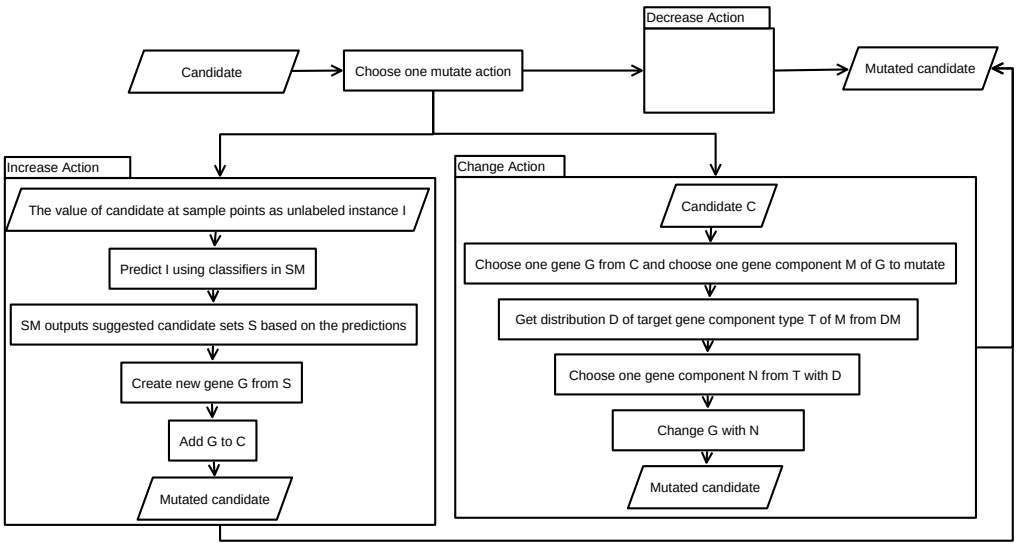


Figure 4.1 Workflow of single-action mutation operator with HM

4.1 Static Model

In this study, Static Model (SM) is the EM model used to learn the pre-existing knowledge of symbolic regression. To use a SM we have to go through 4 steps: 1) acquire a dataset; 2) create training sets from the dataset acquired; 3) train classifiers with the training sets created; 4) organize the trained classifiers to form a SM.

The abstracted workflow of the usage of a SM can be seen at Figure 4.2. The SM works as a per instance basis. That is, for each new gene needed, an unlabeled instance is generated for the SM; then, based on the given instance, the SM generates the suggestions sets for mutation operator; after that, based on the provided suggestion sets, the new gene is created.

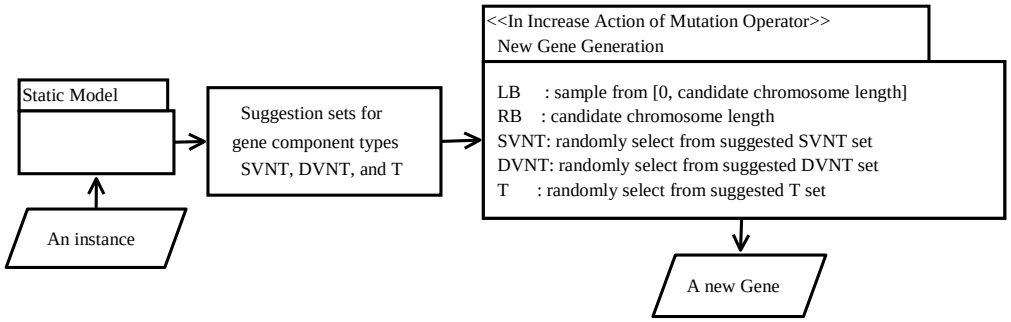


Figure 4.2 Abstracted workflow of the usage of SM

The function of a SM in a mutation operator is to reduce the amount of candidates in candidate sets. This can redirect the search activities to more proper regions as

candidates that are not accepted by the SM cannot be selected by the increase action. The ML models used in SM are classifiers. They are used to give predictions on which gene components should have the chance to be selected and which are not. As EM is a general framework, the ML algorithm used to create ML models for SM can be any ML algorithms as long as it can satisfy the following conditions: 1) it can deal with numeric attributes[1]; and 2) the outputs should be numeric value[2].

4.1.1 Pre-existing Knowledge of Symbolic Regression

In this study, the pre-existing knowledge of symbolic regression is the relationship between math function segments with brackets. In other words, the knowledge considered is the effect of adding a math function segment to another sequence of segments. In this study, because of the limitation of compute capability, only relationships between two math function segments are considered.

If we have a function x, we want to know the effect of $+1$, we can do the following calculation $x + 1 - x$. Here, x and $+1$ can be seen as math function segments. We can acquire the effect of $+1$ by minus the first segment from the combination of the two.

In this study, these differences are called as signatures. Because most of the math function segment sequences have different signatures, this means signatures can be used for creating training set to train ML models.

4.1.2 Dataset

In this study, the values of signatures at the training cases of the benchmark problems form the datasets. Therefore, to acquire the datasets, the enumeration of math function segments has to be found. In this study, because only relationships between two math function segments are considered, brackets have only two valid positions, 0 and 1. When considering bracket positions for creating the enumeration, only brackets in the first segment will be considered. Under this situation, there are only two choices for the brackets combinations. They are shown in Table 4.1.

[1] This condition is listed here because the attributes in training set and the unlabeled instances need to be predicted are numeric values.
[2] This condition is required because the predictions are considered as confident values.

Table 4.1 Combination of brackets

Type	LB	RB	Example
A	0	0	$sin(x) + 1$
B	0	1	$sin(x + 1)$

If we write a signature in gene representation, we can have Table 4.2. LB and RB is not included in Table 4.2.

Table 4.2 Signature in gene representation

Gene Index	DVNT	SVNT	T
0	D1[1]	S1	T1
1	D2	S2	T2
2	–	S1	T1

Templates of signatures can be acquired by adding brackets with indexes show in Table 4.1 into the translated math function of Table 4.2. These templates can be seen in Equations (4.3) and (4.4):

$$S1\ (T1)\ D2\ S2\ T2 - S1\ T1 \tag{4.3}$$

$$S1\ (T1\ D2\ S2\ T2) - S1\ T1 \tag{4.4}$$

The type A combination of brackets is used in Equation (4.3) and the type B combination of brackets is used in Equation (4.4).

In this study, some signatures are ignored. This is because either they are redundant or they cause conflict in training set[2]. They are: relationships between constants, like $\sin(1) + \cos(1)$; 0 conditions, like $\log(1) +$, $\log(1) -$, $+\log(1)$, and $-\log(1)$; when relationships between same variable and different constants appear, only one will be used. For example, if we have $x + \sin(1)$, $x + \cos(1)$, $x + \exp(1)$, and $x + 1$, only $x + \sin(1)$ will be used. Signatures with $\times 1$ or $\div 1$ are also ignored; when signatures with the type B bracket combination are mathematically equivalent with signatures with the type A bracket combination, signatures with the type B bracket combination will be ignored.

If we have a DVNT set as $+, -, \times, \div$, a SVNT set as $null$, and a T set as x, we can have signatures shown in Table 4.3 by using Equations (4.3) and (4.4).

[1] D1 is not important here, as it will be ignored.
[2] It is necessary to remove redundancy from the beginning. It is much easier to do so at this stage.

Table 4.3 Signatures created from DVNT set of $+, -, \times, \div$, SVNT set of $null$, and T set as x

$(x) + x - x$	$(x) - x - x$
$(x + x) - x$	$(x - x) - x$
$(x) \times x - x$	$(x) \div x - x$
$(x \times x) - x$	$(x \div x) - x$

After ignoring certain enumerations, we can have signatures that will be used for creating the dataset. They are shown in Table 4.4.

Table 4.4 Signatures used for creating the dataset

$(x) + x - x$	$(x) - x - x$
$(x) \times x - x$	$(x) \div x - x$

A dataset can be acquired by having the values of the created signatures at the training cases of a benchmark problem. Because the training cases are real values, the attributes in the dataset are numeric attributes.

If we have training cases of $(-1, -0.5, 0.5, 1)$, we can have the dataset show in Table 4.5 from signatures shown in Table 4.4.

Table 4.5 Dataset without information for labeling

Signatures	Dataset				
	Row Index	Attributes			
		A1	A2	A3	A4
$(x) + x - x$	1	-1	-0.5	0.5,	1
$(x) - x - x$	2	1	0.5	-0.5	-1
$(x) \times x - x$	3	0	-0.25	-0.25	0
$(x) \div x - x$	4	2	1.5	0.5	0

In order to create a training set, the dataset has to be labeled. In this study, each row of the dataset corresponds to one signature. Because the cause of change on a math function is what we want the classifiers to predict, the labeling information is the DVNT, SVNT, and T combination in the second math function segment used to create the signatures.

When we have Table 4.5, we can mark it to create the dataset used for creating training sets. It is shown in Table 4.6.

Table 4.6 Dataset with information for labeling

Row of Dataset	Information for Labeling		
	DVNT	SVNT	T
$(-1, -0.5, 0.5, 1)$	$+$	$null$	x
$(1, 0.5, -0.5, -1)$	$-$	$null$	x
$(0, -0.25, -0.25, 0)$	$\times$	$null$	x
$(2, 1.5, 0.5, 0)$	$\div$	$null$	x

4.1.3 Training Set

In this study, the class attribute is numeric binary class. That is, only two possible values for class attribute in the training sets, they are 0 and 1.[1] For each classifier, one training set will be created for it. The accuracy of the classifier is its accuracy on its training set.[2]

The first step to create a training set is to standardize every row in the dataset to$[-1,1]$.[3] The second step is to label all the instances with one of the labeling methods. The third step is to remove duplicates.[4] When all duplicated instances have same class value, only one of them will be kept. If any duplicated instances have class labeled as 1, instances with class labeled as 0 will be discarded and only one instance with class labeled as 1 will be kept.

In this study, two labeling strategies are used. The first is to label only one row of the dataset as 1, all the other rows are labeled as 0. Under this labeling method, one classifier will be trained to discriminate one instance from the rest, and the labeling information associated with the instance will be transferred to the corresponding classifier. Therefore, the amount of classifiers trained is equal to

[1] In this study, the predictions are seen as confidence values. This is possible because all the predictions are numeric values with range of $[0, 1]$.

[2] Other methods cannot be used here because of the size of the dataset can be small (several instances). Further, we wish the classifiers to learn as much as possible. The reason for this is that signatures are created from the combination of two segments. In the evolution of LGP, the chromosomes can easily grow over that size. From this, we can see that the dataset is not sufficient for the real situations. Therefore, further separates a training set into a training set and a test set might lead to lowered accuracy on all cases.

[3] This is a standard procedure on training artificial neural networks (ANN). Its purpose is to increase the prediction accuracy. The MLPRegressor algorithm (Pentaho, n.d.) used in this study is a multilayer perceptron (MLP) with one hidden layer. It can deal with numeric attributes and can output numeric predictions (the reason of using regressor). It is one of the WEKA implementation of the multilayer perceptron algorithm.

[4] Duplicates have to be removed. First, they can cause conflicts as instances have same attribute values but have different class values. Second, duplicates can cause unnecessary high weight on one instance because of multiple appearances.

the number of the instances. This method is called as Type I labeling method. The concept of Type I labeling method is shown in Figure 4.3.

If we have the dataset show in Table 4.6, we can use Type I labeling method to create the training set used to train the classifier with labeling information of $+, null, x$. This training set is shown in Table 4.7.

In the second method, the training sets are created based on the labeling information in the dataset. For each gene component of a gene component type, a classifier will be created to discriminate it from the rest of its type. Therefore, the total amount of classifiers trained is equal to the total number of gene components in all gene component types. This method is called as Type II labeling method. The concept of Type II labeling method is shown in Figure 4.4.

If we add another row to the dataset show in Table 4.6[1], we can have a new dataset as shown in Table 4.8. We can use Type II labeling method to create the training set shown in Table 4.9 from Table 4.8. It is used to train classifier responds to give predictions for DVNT+.

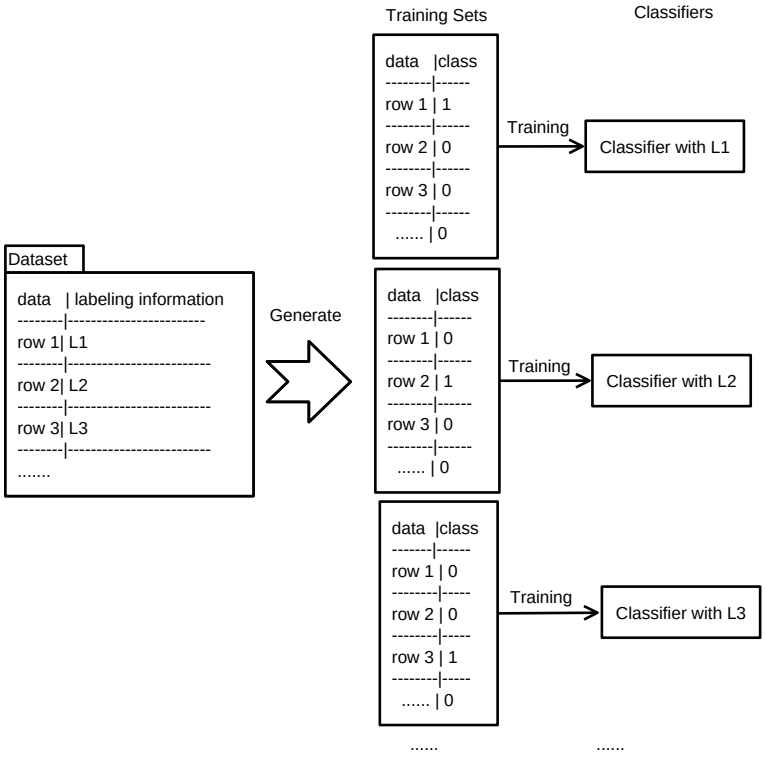


Figure 4.3 Concept of Type I labeling method

[1] We do not need to care how we acquired this dataset and the exact labeling information. This example is just a demonstration of how a training set is created using Type II labeling method.

Table 4.7 A training set created with Type I labeling method

Row Index	Attributes				
	A1	A2	A3	A4	Class
1	−1	−0.5	0.5	1	1
2	1	0.5	−0.5	−1	0
3	0	−0.25	−0.25	0	0
4	2	1.5	0.5	0	0

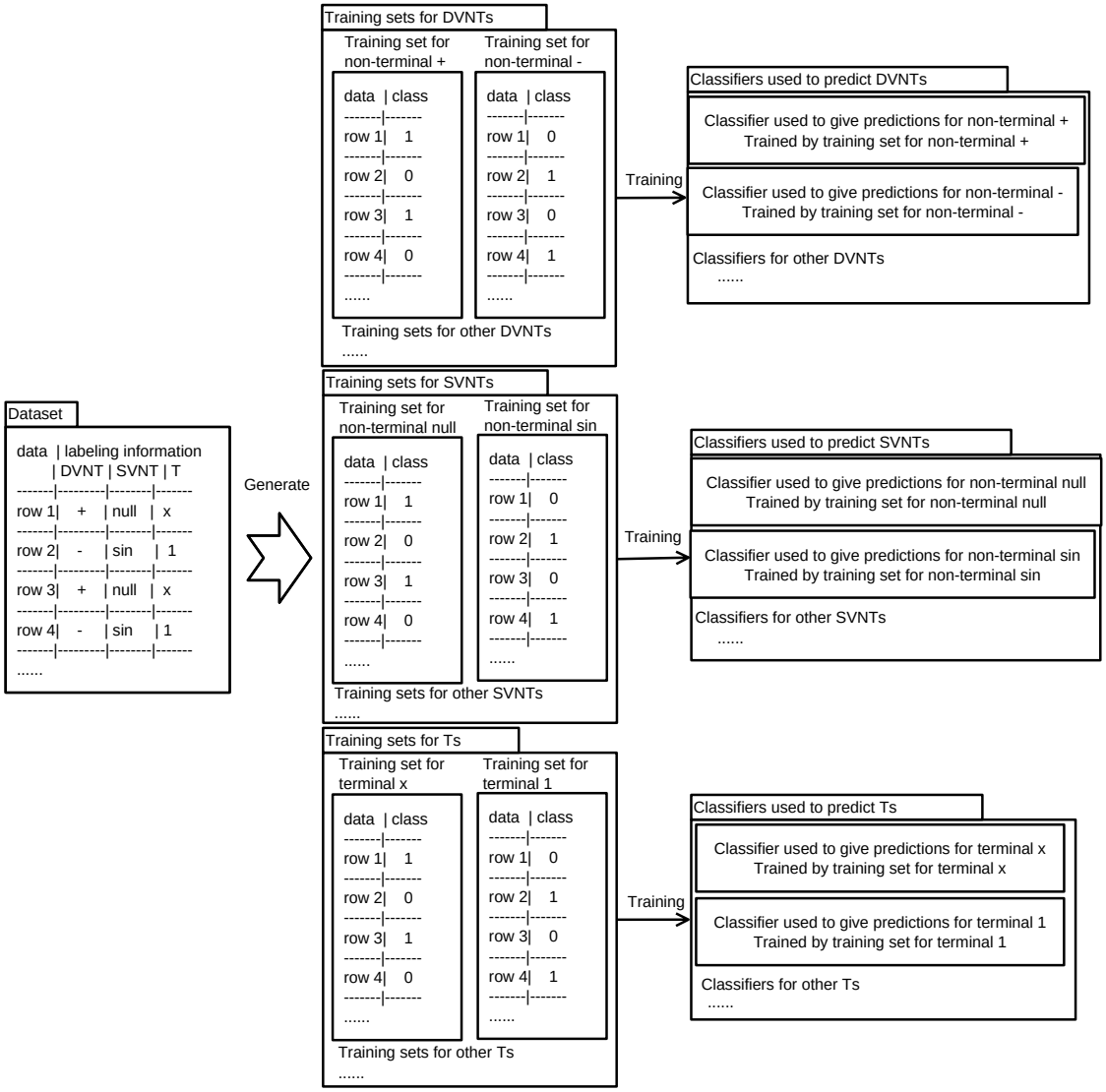


Figure 4.4 Concept of Type II labeling method

Table 4.8 Dataset with one more row

Row of Dataset	Information for Labeling		
	DVNT	SVNT	T
$(-1, -0.5, 0.5, 1)$	$+$	$null$	x
$(1, 0.5, -0.5, -1)$	$-$	$null$	x
$(0, -0.25, -0.25, 0)$	$\times$	$null$	x
$(2, 1.5, 0.5, 0)$	$\div$	$null$	x
some data other than the first row	$+$	$null$	x

Table 4.9 A Training set created with Type II labeling method

Row Index	Attributes				
	A1	A2	A3	A4	Class
1	-1	-0.5	0.5	1	1
2	1	0.5	-0.5	-1	0
3	0	-0.25	-0.25	0	0
4	2	1.5	0.5	0	0
5	some data other than the first row				1

4.1.4 Samples for Predictions

In this study, the samples used for the classifiers to give predictions are acquired from the differences between the target values and the values of the candidate individual on the training cases of the benchmark problem. The difference at position i is shown in Equation (4.1):

$$s_i = t_i - c_i, \tag{4.1}$$

where s_i is the difference at position i, $t_i$is the target value at position i, $c_i$is the value of the candidate individual on training case i. If we put all the s_i together, we can have the unlabeled sample S as shown in Equation (4.2):

$$S = \bigcup_i s_i \tag{4.2}$$

We can see that sample S does not have a class attribute. Therefore, before sending it to a SM, a pseudo class attribute will be added to S to make it become a complete instance.

The reason of using Equation (4.1) is that the direct way to increase the fitness of an individual is to minimize the differences between the target values and the values of the individual at the training cases. As signatures can be seen as descriptions of the effect of one math function segment added to the end of another math function segment, if a given sample is similar with the values of a signature at the training cases, then it is possible that the gene components correspond to the signature can reduce these differences.

4.1.5 Structure of Static Model

In this study, two types of SM structure are used. The first type is based on the Type I labeling method and is called as Type I structure. With this structure, a SM

uses the same set of classifiers to give the predictions for all gene components. When an unlabeled sample is transferred to a SM with Type I structure, the SM will let each classifier give its prediction on the sample, and then these predictions will be used to make the decision of whether a gene component can be included in the suggestion set. For any gene component, if the labeling information of a classifier contains it, then the prediction of that classifier will be used on the evaluation of whether this gene component should be included. Figure 4.5 shows the workflow of Type I structure.

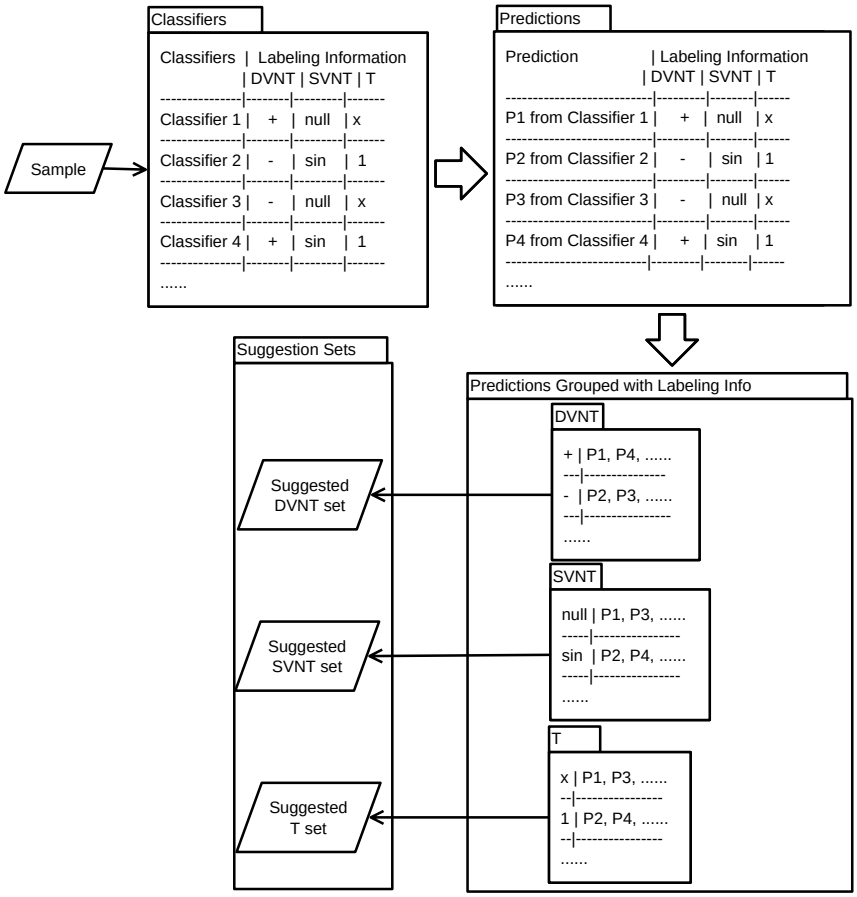


Figure 4.5 Workflow of Type I structure

The second type is based on the Type II labeling method and is called as Type II structure. With this structure, a SM uses different groups of classifiers to give suggestion sets for different gene component types. Whether or not a gene component can be added to its corresponding suggestion set is based on the prediction given by the classifier trained to predict for this gene component. That is, whether a gene component will be accepted or not only rely on the prediction

of one classifier instead of the predictions from a group of classifiers as in Type I structure.[1] The workflow of Type II structure is shown in Figure 4.6.

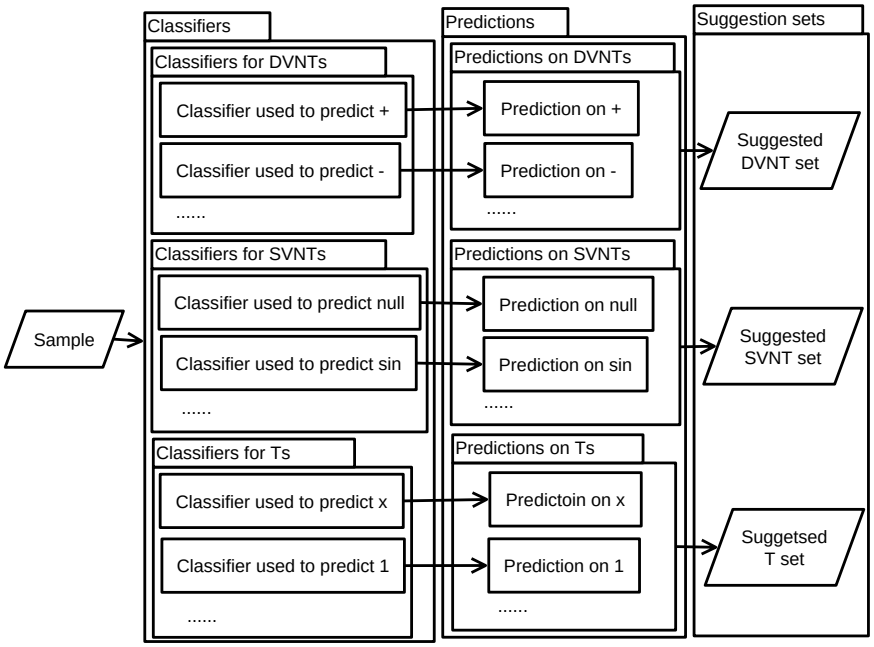


Figure 4.6 Workflow of Type II structure

In this study, because the trained classifiers are underfit in real situations[2], we cannot rely on one classifier to decide which gene component is the best one in its type, as the predictions of the classifiers may not be accurate. To overcome the low accuracy of a single classifier, collections of classifiers are used. In Type I structure, a single collection of classifiers is used to provide the predictions needed. In Type II structure, for each gene component type, a collection of classifiers is used to provide predictions, each classifier in it responds to give prediction for one gene component. In this way, instead of predicting which gene component is the best one directly from the sample, the suggestion sets are created by considering the collected predictions from classifiers. To make the suitable candidates can be chosen by the mutation operator while rejecting the candidates with low confidence values, a strategy of only minimizing the false negative rate

[1] In SM with Type II structure, training multiple classifiers for one gene component is not necessary. This is because only one training set can be acquired. Training multiple classifiers with the same training set while using same settings for ML algorithm usually will not give significantly different classifiers.

[2] The number of math function segments of the signatures used in this study is two. The number of math function segments of the benchmark problems in benchmark set I (see section 5.1) are larger than three. We can see this difference makes the classifiers trained with the training sets underfit in real situations.

is adopted. This is because when the false negative rate is low, the suitable candidates have high chances to be included into the suggestion sets.

The two structures of SM have different abilities in reducing the false negative rate. For the first type, SM with Type I structure, when a gene component needs to be decided on whether it should be included into a suggestion set, all the predictions from the classifiers with labeling information that contain this gene component will be considered. If any prediction has higher value than the minimal acceptable value, this gene component will be included. However, this type of structure cannot adapt to the situations when many variables are needed. In order to make a SM able to work in a situation when many terminals are used, the SM with Type II structure is designed. In it, whether one gene component can be added into the suggestion set is decided by only one classifier. Therefore, its risk on having a false negative is higher than the Type I structure.

4.1.6 Static Model in Mutation Operator

The SM, in this study, is only used for increase action of the mutation operator. In the modified increase action, instead of inserting randomly, new genes will be added to the last of chromosomes. When an individual needs a new gene, a sample will be created with the standardized differences between the target values and its values at the training cases of the benchmark problem. Then, the SM will give suggestion sets for gene component types of DVNT, SVNT, and T based on this sample. For a gene component, if it has at least one prediction with value higher than the minimal acceptable value, the SM will add it to its corresponding suggestion set. For the new gene, its DVNT, SVNT, and T component will be chosen randomly from these sets. Its RB component will be the index of the new gene and LB component will be sampled from $[0, RB]$. Figure 4.7 shows the usage of a SM in a mutation operator.

It is possible to have empty suggestion sets returned by a SM.[1] When this situation happens, the corresponding full set of that gene component type will be used. To reduce the number of empty suggestion sets returned by the SM and to minimize the false negative rate, the minimal acceptable prediction value is set to 0.1.

[1] This might happen, as it is possible that no prediction has a value higher than the minimal acceptable value.

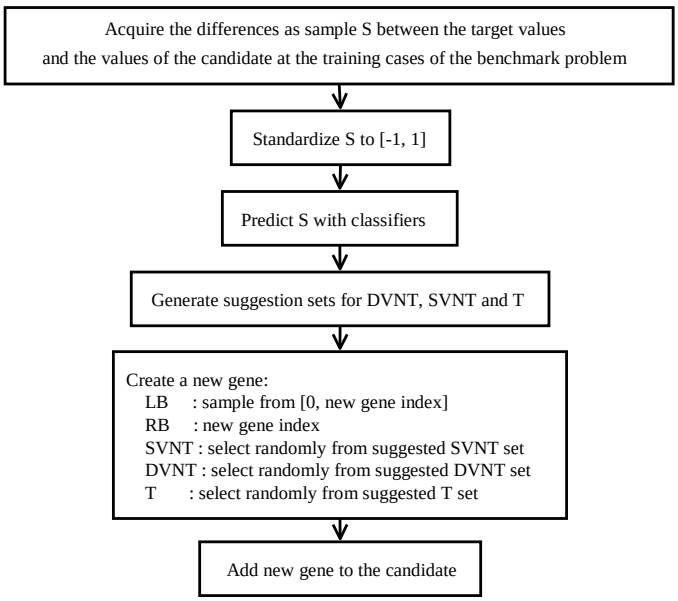


Figure 4.7 Workflow of increase action with SM

4.2 Dynamic Model

The function of a DM in a mutation operator is to change the uniform distribution used in the change action into a distribution that can reflect the effect of having a gene component on the overall fitness. In the distributions generated by a DM, the gene components that usually in individuals with better fitness have higher chance to be selected and vice versa.

If we consider the evolution of LGP as a stream of data, we can see this data stream is composed of chunks of fitness (one chunk of fitness contains the fitness of all individuals in one generation) and the associated chromosome information. In a DM, the information it records is the relationships between gene components and the overall fitness at all existing chromosome indexes. In this study, this relationship takes the form of ratios between accumulated fitness of all individuals that have a particular gene component at a particular chromosome index and the total occurrences of that gene component at that chromosome index. For each chunk (or a generation), the contents of the DM will be updated. The abstracted workflow of the usage of a DM is shown in Figure 4.8.

As in SM, any ML algorithm can be used to create the ML model used for DM. The requirements are: 1) the algorithm can give outputs that can be seen as distribution[1]; 2) the trained model can be updated at runtime[2].

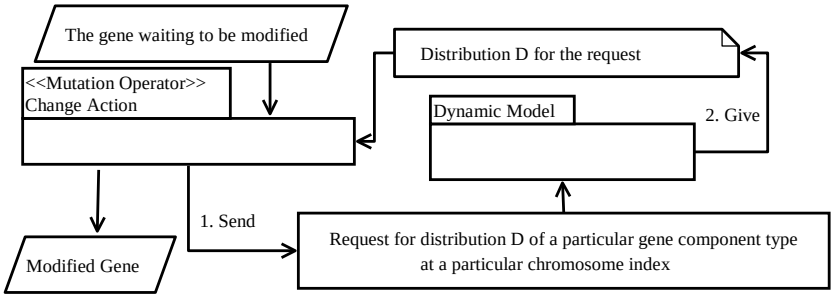


Figure 4.8 Abstracted workflow of the usage of DM

4.2.1 The Generation of Knowledge

In LGP runs, fluctuation of fitnesses, change of genetic contents, and change in chromosome lengths happens along the evolution. They are the source of knowledge that will be used to train classifiers used in the DM.

During the evolution of LGP, it is common that the fitness of an individual goes up or down just because the change of one gene component. If the change of only one gene component can lead to the change of the fitness, then when a chromosome needs to be changed it is beneficial to choose the gene component that might lead to a fitness increase.

We can know which gene component in a gene component type has higher possibility to make a fitness increase for an individual at chromosome index i by comparing ratio r_i^t between all gene components in that gene component type. This ratio is shown in Equation (4.4):

$$r_i^t = {\sum f_i^t}\Big/{c_i^t}, \tag{4.4}$$

where f_i^t is the fitness of an individual that has gene component t at chromosome index i, $\sum f_i^t$ is the accumulated fitness of all individuals in all generations that have gene component t at chromosome index i, c_i^t is the accumulated total occurrences of t at index i.

[1] This is because the change action has to select gene components based on them.
[2] This is because the ML model needs to learn from the population.

This thought is naive. However, if a gene component at a particular chromosome index is good for fitness in general, the confidence to predict that it can give a fitness increase for an individual can be higher than the one not as good as it.

In this study, instead of giving prediction on which gene component should be used, the DM is used to change the distributions of different gene component types used in the change action of the mutation operator. For each gene component of each gene component type at every chromosome index, the DM records the accumulated fitness of every individual that has it and the total occurrences of it. In the modified change action, the selection on gene components is based on the distributions given by DM.

4.2.2 Distributions from Dynamic Model

In this study, the ratio r_i^t shown in Equation (4.4) is used to form new distributions that are used in change action. To form the new distribution of a gene component type at index i, we should combine all the ratios of the gene components of the required gene component type, and then normalize them to form a proper distribution. The new formed distribution D_i^T is acquired by using Equation (4.5) :

$$D_i^T = \bigcup_t \frac{r_i^t}{\sum_t r_i^t}, \tag{4.5}$$

where D_i^T represents the distribution used to influence the choice of the gene component of gene component type T at chromosome index i.

4.2.3 Structure of Dynamic Model[1]

Based on Equation (4.4), the structure shown in Figure 4.9 is used in DM to record the accumulated fitness and the total occurrences of one gene component.

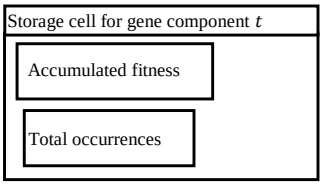


Figure 4.9 DM storage cell

[1] In this study, the ML algorithm used in DM is built directly into the DM, therefore, it become a part of the DM. In other situations, the chosen ML model can replace the algorithm described in this section directly. That is, in update process, population is the input of DM and will be converted into training set for ML models; in usage phase, the outputs of ML models should be grouped into one distribution and then output by DM.

If we combine the storage cells of every gene component of all gene component types at a chromosome index, we can have the storage structure used to generate the modified distribution for all gene component types at that chromosome index. This structure is shown in Figure 4.10.

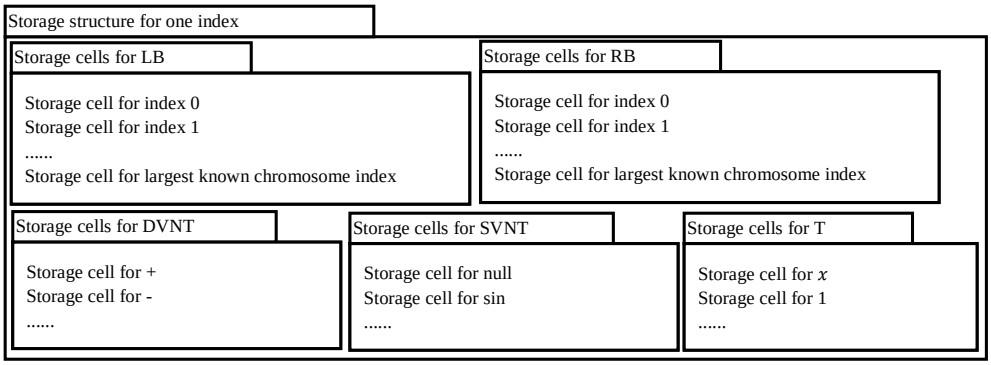


Figure 4.10 DM Storage structure used to generate modified distribution for one index

We can have a DM when the storage structure shown in Figure 4.10 is created for every chromosome index. The structure of DM is shown in Figure 4.11.

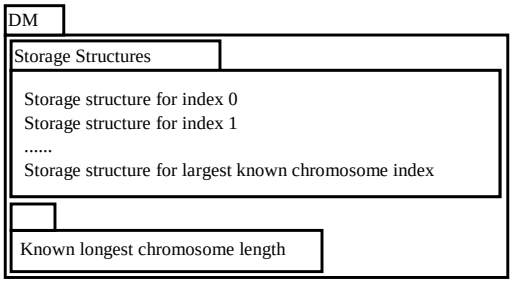


Figure 4.11 Structure of DM

4.2.4 Usage of Dynamic Model

In this study, the DM is used only for change action of a mutation operator. The only purpose of using DM in a mutation operator is to generate distributions for change action to influence the choices on gene components. A DM needs to be updated to reflect the changes in the current population. The workflow of DM is a two-step process. The first step is to update a DM with the current population. The second step is to use the DM to generate distributions for the change action.

During the update process, every individual in the current population will be used for update. For an individual, if its gene components have corresponding storage cells, then these corresponding storage cells will add the fitness of this individual

to their accumulated fitness and add one to their total occurrences. When one gene component cannot find corresponding storage cell in the storage structure at a chromosome index, the DM will create a new storage cell for it and add the new storage cell into the storage structure at that chromosome index. The DM will initialize the new storage cell with accumulated fitness as the fitness of this individual, and the total occurrences as one. If any individual has chromosome length longer than the recorded longest chromosome length, the longest chromosome length will be refreshed. The update process is shown in Figure 4.12.

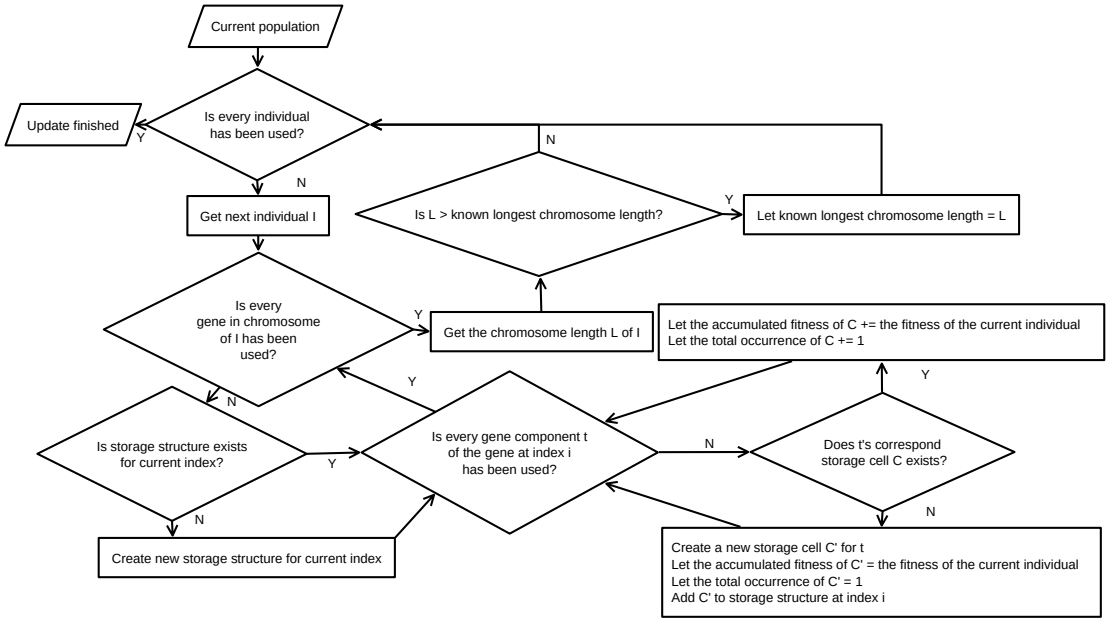


Figure 4.12 Update process of DM

When a distribution D_i^T is needed, a DM will first generate the ratio r_i^t for every recorded gene component of the required gene component type at the required chromosome index. Then these ratios will be normalized to form a proper distribution. If this distribution can represent all the gene components of the required gene component type, it will be used in change action directly. If not, this distribution will be completed by using a reduced probability of half the average possibility of the known gene components as the probability for the gene components that do not exist.[1] Then, as the sum of all probability in the distribution is larger than 1, this distribution will be normalized again to become

[1] A penalty for non-existing items is necessary. First, non-existing items need to be included as they may be suitable. Second, it is too strong to give too much credit to non-existing items, as they may not be suitable. Therefore, a penalty for non-existing items should be seen as a compromise.

proper. Then, it will be used in change action. The workflow of creating a distribution is shown in Figure 4.13.

The recorded largest chromosome length is used to generate new distributions for LB and RB. The number of valid gene components (chromosome indexes) in these gene component types increases as the known longest chromosome length increases. When not all the indexes are recorded, a DM will set the unrecorded indexes with the reduced probability as stated above.

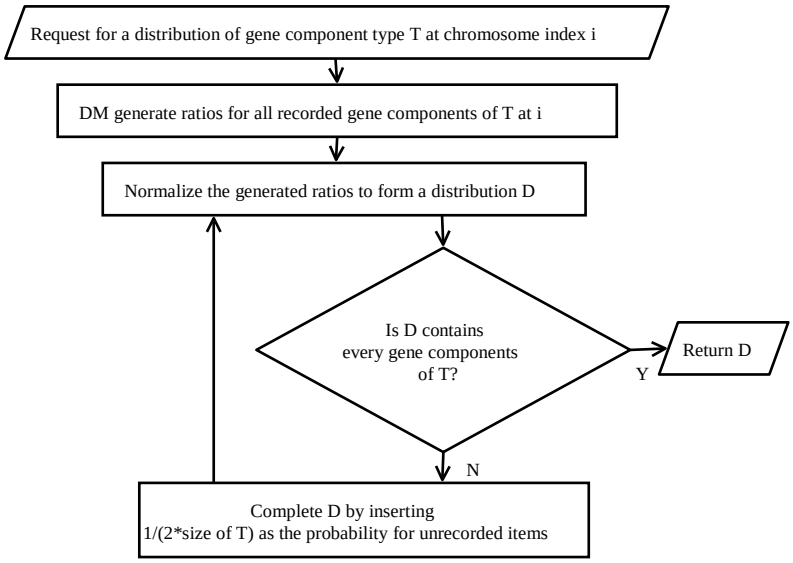


Figure 4.13 The process of generating distributions with a DM

5. Evaluation

5.1 Benchmark Problems

In this study, two benchmark sets are used. The first is benchmark set I, it is the benchmark set introduced in Uy et al. (2011). The functions in this set are shown in Table 5.1 with training cases and test cases. In this study, uniform interval is used instead of random interval used in Uy et al. (2011). The test cases are created for comparison purposes for this study. The second set is benchmark set II, this benchmark problem is the concrete dataset introduced in Yeh (1998). In this study, this set is first standardized column-wise to reduce the value range of the attributes, then randomly separated into two parts, training cases (75% of all instances) and test cases (25% of all instances).

Table 5.1 Benchmark set I

	Functions	Training Cases	Test cases
1	$x^3 + x^2 + x$	20 points ⊆ [-1,1], interval 0.1 with exception of (0,0)	20 points ⊆[-2,-1)∪(1,2] with interval 0.1
2	$x^4 + x^3 + x^2 + x$		
3	$x^5 + x^4 + x^3 + x^2 + x$		
4	$x^6 + x^5 + x^4 + x^3 + x^2 + x$		
5	$\sin(x^2)cos(x) - 1$		
6	$sin(x) + sin(x + x^2)$		
7	$log(x + 1) + \log(x^2 + 1)$	20 points ⊆(0,2] with interval 0.1	20 points ⊆(2,4] with interval 0.1
8	$\sqrt{x}$	20 points ⊆(0,4] with interval 0.2	20 points ⊆(4,8] with interval 0.2
9	$sin(x) + sin(y^2)$	100 points ⊆[-1,1]x [-1,1], interval 0.2 with exception of x=0 and y=0	100 points ⊆ [-2,-1)∪(1,2]x[-2,-1)∪(1,2] with interval 0.2
10	$2sin(x)cos(y)$		

Table 5.2 Data Characteristics of benchmark set II

Number of Instances	1030
Number of Attributes	9
Attribute Type	Numeric
Missing Attribute Values	0
Class Attribute	The last attribute

5.2 Algorithm Implementations

This study uses Java to implement all the algorithms described in Chapter 2 to 4. Two types of non-terminal and terminal settings are used. The first type is the

minimal setting that is shown in Table 5.3. In the minimal setting, only $null$ will be included in the SVNT set and no constants will be included in the terminal set. The second type is the full setting as used in Uy et al. (2011). The full setting is shown in Table 5.4.

Table 5.3 Minimal setting for non-terminal set and terminal set

Terminal	x for function 1 to 8 in benchmark set I x and y for function 9 and 10 in benchmark set I 8 variables for benchmark set II
Single Variable Non-Terminal	$null$
Double Variable Non-Terminal	$+, -, \times, \div$

Table 5.4 Full setting for non-terminal set and terminal set

Terminal	x and 1 for function 1 to 8 in benchmark set I x and y for function 9 and 10 in benchmark set I 8 variables for benchmark set II
Single Variable Non-Terminal	$null, sin, cos, exp, log$
Double Variable Non-Terminal	$+, -, \times, \div$

5.2.1 Machine Learning Algorithm for Static Model

In this study, WEKA version 3.7.11 (Hall et al., 2009) is the underlying ML framework. The MLPRegressor algorithm is used to create the classifiers for SM. It is downloadable through the package manager within WEKA. MLPRegressor (Pentaho, n.d.) is an ANN algorithm that is a multilayer perceptron[1] (MLP) with one hidden layer. It is designed to handle numeric attributes and can output numeric predictions. Detailed information of it can be found in the WEKA software. The parameters of MLPRegressor for training are 60 for numFunctions, 8 for numThreads and poolSize, 1×10^{-10} for ridge and tolerance, true for useCGD. The target accuracy on training set is 0.95. To acquire the best accuracy, for each training set, 100 classifiers will be trained with it, if any classifier reaches the target accuracy, it will be used; otherwise, the classifier with highest accuracy will be used.

[1] Multilayer perceptron is a feedforward network model of ANN.

5.2.2 Variances of Static Model

In this study, SM with Type I structure does not give suggestion set for terminal.[1] When LGP uses a SM with Type I structure, terminals for new genes will be chosen randomly from terminal set as in the unmodified increase action.

5.2.3 Method to Acquire Fitnesses and Values at Training Cases

In this study, Java reflection is used for both tasks. After converting a chromosome into a mathematical function, the mathematical function is translated into the Java expression with functions in java.lang.Math. For each function translated, a pair of brackets is added for it to keep the priority in translated Java expression. For example, if the math function is $\sin\big(x + (x)\big) + x$, the translated Java expression will be $Math.sin((x + (x))) + x$. Then the translated Java expression will be inserted into a template to create a Java file for compilation. The compiled Java object is created though reflection and is used to acquire fitness and the value of that individual at the training cases. The template is shown in Figure 5.1. The "Hashcode" is the hashcode of the string that store the mathematical function. Two error situations are considered in the template. The first situation is arithmetic exception, as a divide by zero exception, it means the function is wrong in mathematical means, therefore, the positive infinity is returned. As we can see from Equations (3.2), this will make the fitness of that individual become zero. The second situation is the calculated values are not real numbers. To make the individual can survive from such situation (the function might have correct values at other training cases), 0 is returned.

```
public class _Hashcode_of_INSERTED_FUNCTION{
    public static double Get(double x){
        try{
            double value = INSERTED_FUNCTION;
            // Protect Mode
            if(Double.isNaN(value) || Double.isInfinite(value)) {
                return 0;
            }
            else{
                return value;
            }
        }
        catch(ArithmeticException ex){
            return Double.POSITIVE_INFINITY;
        }
    }
}
```

Figure 5.1 Compile file template

[1] SM with Type I structure will not be used on benchmark set II. As the chances are 50% to have the suitable terminal selected in benchmark set I, not predicting terminals can simple the design of SM with Type I structure.

5.3 Experiments

5.3.1 Experiment Settings

Two settings of LGP are used in this study. In Type I setting, the LGP uses single-point uniform crossover operator and single-action mutation operator. In Type II setting, the LGP uses multi-point uniform crossover operator and multi-action mutation operator. SM with Type I structure only used in Type I setting, SM with Type II structure used only in Type II setting. Because of the complexness of the solution space, experiments on benchmark set II only use Type II setting.[1] For both settings of LGP, experiments are performed with minimal and full setting of non-terminal and terminal sets. Therefore, there are four experiment settings for benchmark set I. They are Type I setting with the minimal setting, Type I setting with the full setting, Type II setting with the minimal setting, and Type II setting with the full setting. There are two experiment settings for benchmark set II, Type II setting with the minimal setting, and Type II setting with the full setting. The parameters for LGP are shown in Table 5.5.

Table 5.5 Parameters for LGP

Population Size	500
Maximum Generations	2000
Initial Chromosome Length	1
Maximum Chromosome Length	150
Selection	Roulette Wheel Select
Crossover and Mutation Operators	Type I setting : Single-Point Uniform Crossover and Single-Action Mutation; Type II setting: Multi-Point Uniform Crossover and Multi-Action Mutation
Crossover Rate	0.9
Mutate Rate	0.1
Crossover Percentage in Multi-Point Uniform Crossover	0.2
Maximum Mutate Action Times in Multi-Action Mutation Operator	4
Hit Condition	Fitness $\geq$ 0.99
Total Runs	100 (random number generated by java.util.random with seed 12345)
Use Elitism	Yes

[1] The reason for only using Type II setting on benchmark set II is SM with Type I structure is not suitable for large amounts of variables. For benchmark set I, under the full setting, SM with Type I structure only uses several hundreds of classifiers. However, for benchmark set II, this number will be increased to several thousands. The creation of classifiers will become a dominant factor for LGP uses a SM with Type I structure.

5.3.2 Experiment Results

For all experiments, Wilcoxon test is used for significant tests with $p < 0.05$.[1]

1. Experiment Results on Benchmark Set 1

Figure 5.2 – 5.5 are the boxplots of mean generations needed to have a hit of 100 runs on four experiment settings. The horizontal line in the graph is the median and the diamond is the mean. Table 5.6 – 5.9 show the mean generations needed to have a hit of 100 runs on four experiment settings. A value of 2000 in these tables means there are no hits happening in any single run. Table 5.10 – 5.13 are the mean best fitness of 100 runs of four experiment settings on training cases. Table 5.14 – 5.17 are the mean best fitness of the last turn of 100 runs of four experiment settings on test cases.

In Table 5.6 – 5.17, if a block is marked with superscript L, D, S, or H, this means the value of that LGP type is significant in compare to the value of the LGP type marked by the superscript of the same row. L means LGP, D means LGP with DM, S means LGP with SM, and H means LGP with HM. In Table 5.6 – 5.9, a value with a superscript means fewer generations needed to have a hit than the marked type. In Table 5.10 – 5.17, a value with a superscript means higher mean best fitness than the marked type.

[1] The reason for using Wilcoxon test instead of using student t-test is because the shape of histograms. In nearly all cases, the shapes of them are not bell shaped. They usually have shapes of several spikes or have heavy left tail with no right side.

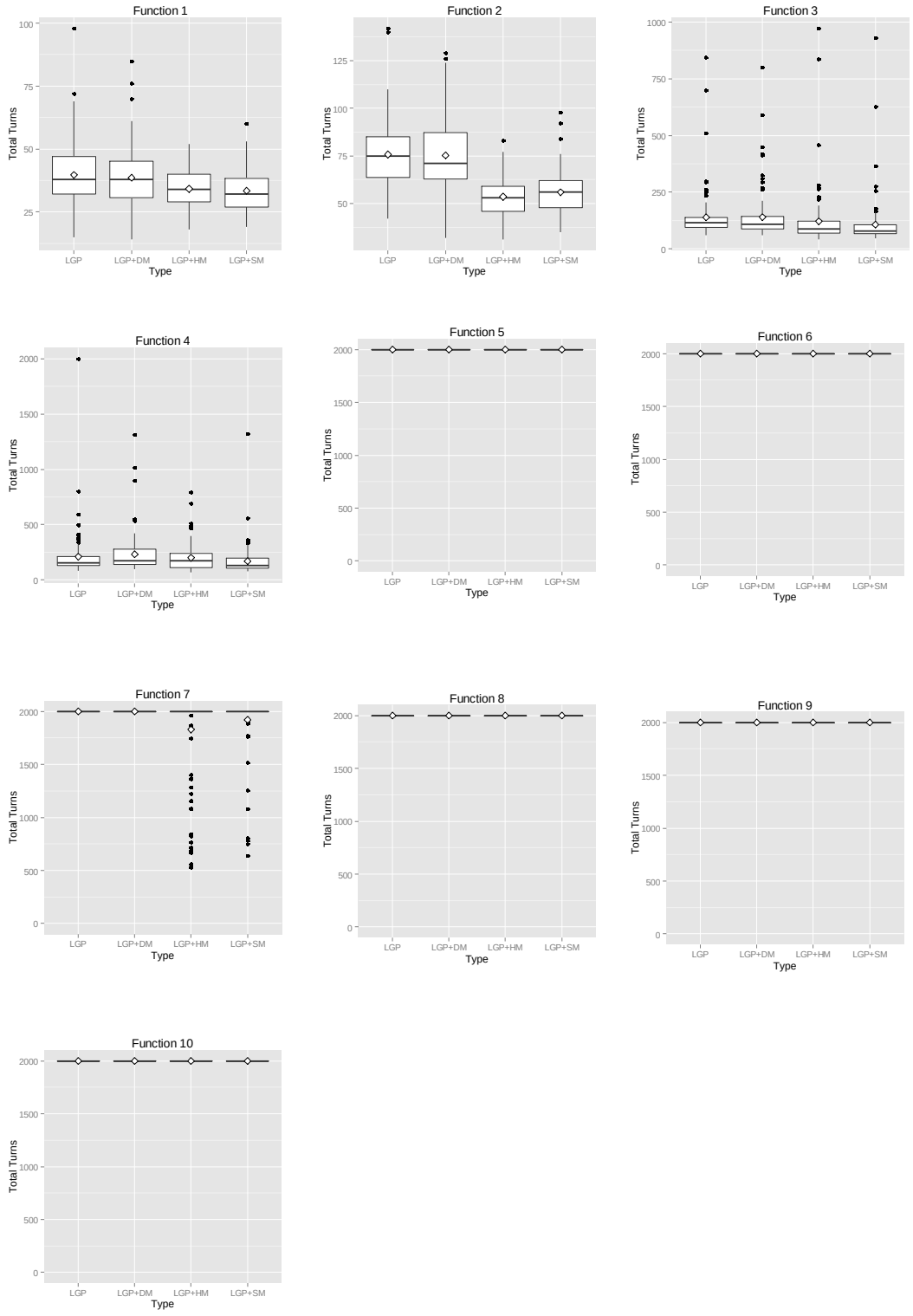


Figure 5.2 Boxplot of the generations needed to have a hit on Type I setting with the minimal setting for non-terminal and terminal sets on benchmark set I

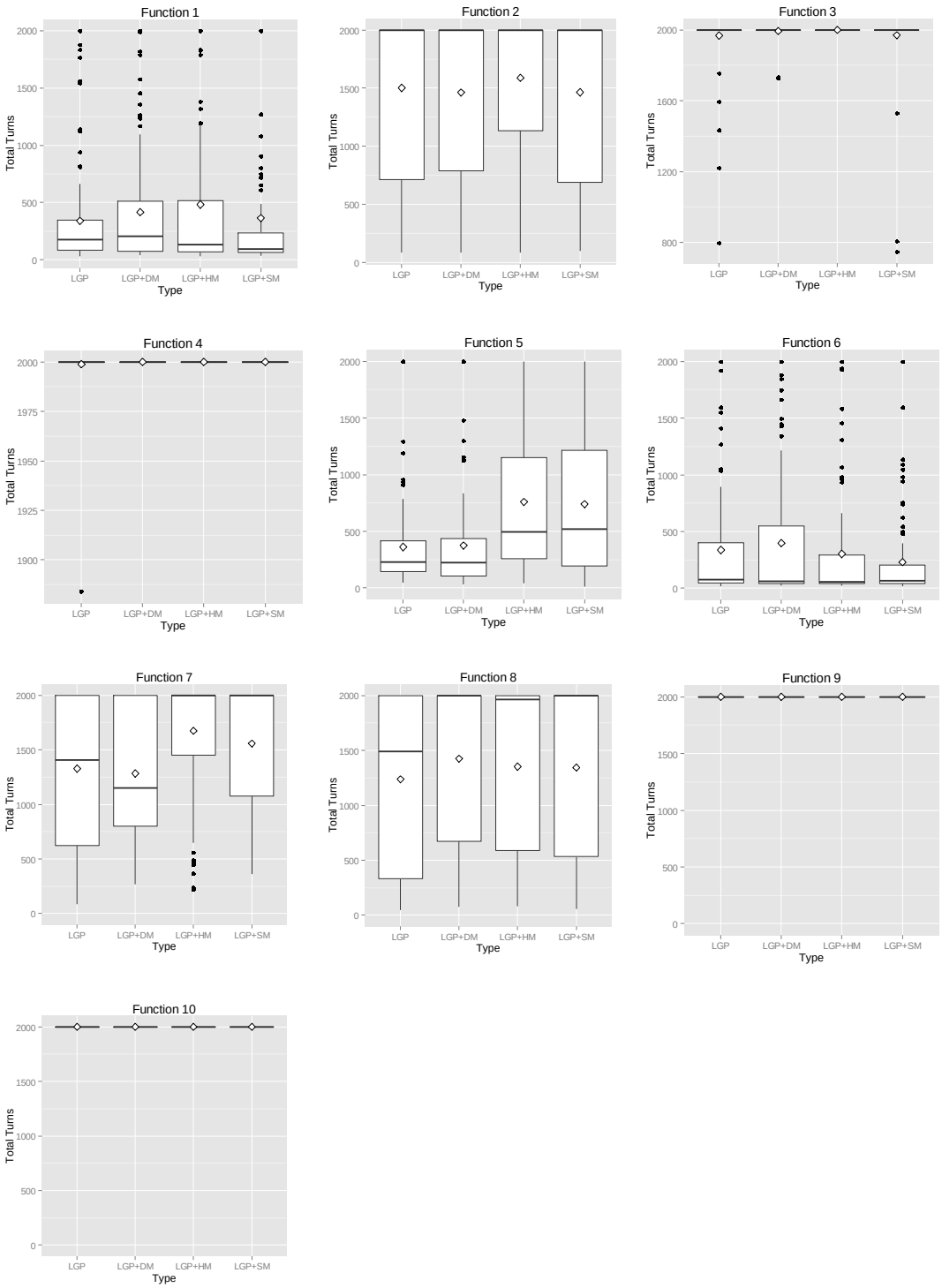


Figure 5.3 Boxplot of the generations needed to have a hit on Type I setting with the full setting for non-terminal and terminal sets on benchmark set I

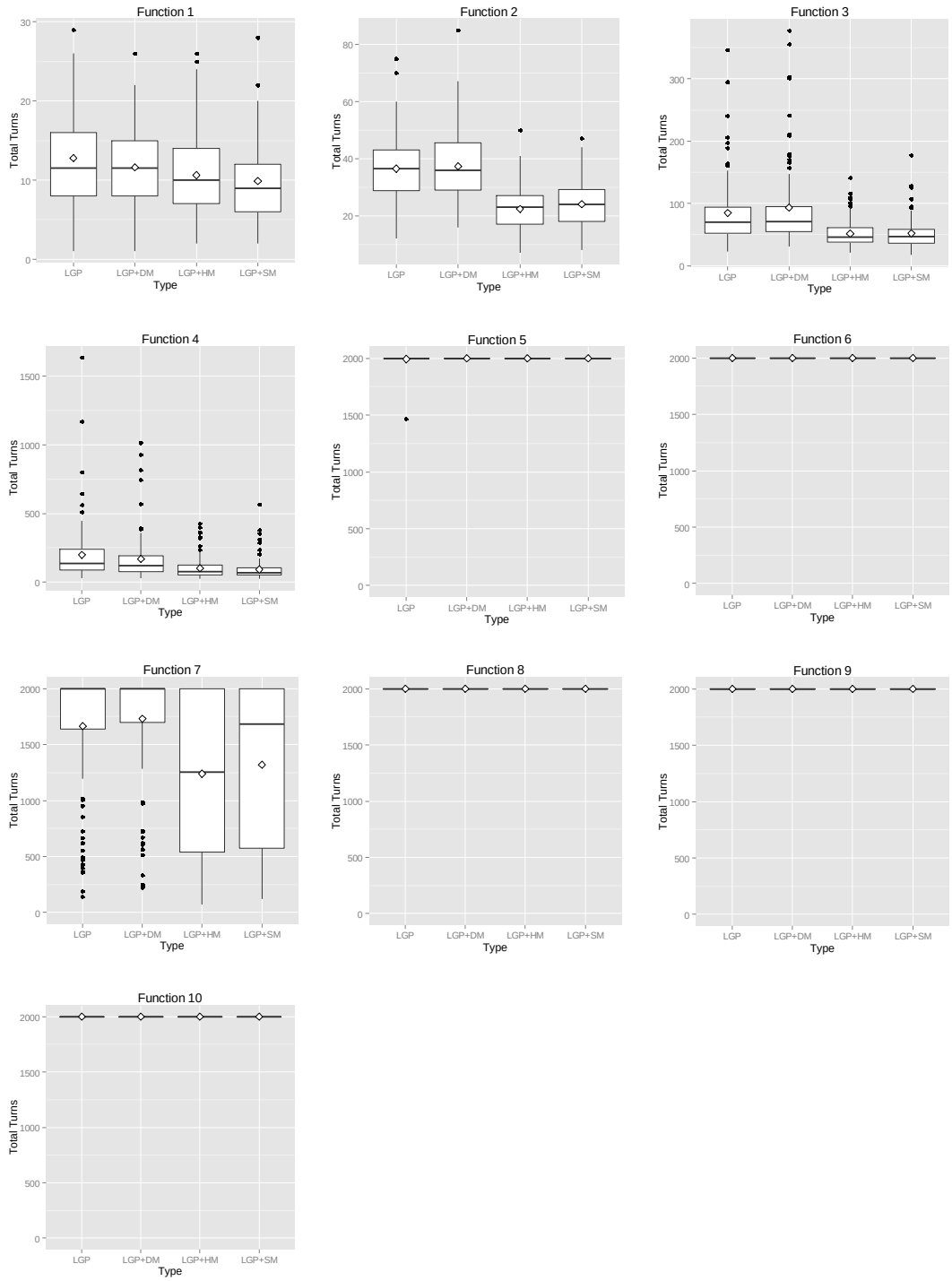


Figure 5.4 Boxplot of the generations needed to have a hit on Type II setting with the minimal setting for non-terminal and terminal sets on benchmark set I

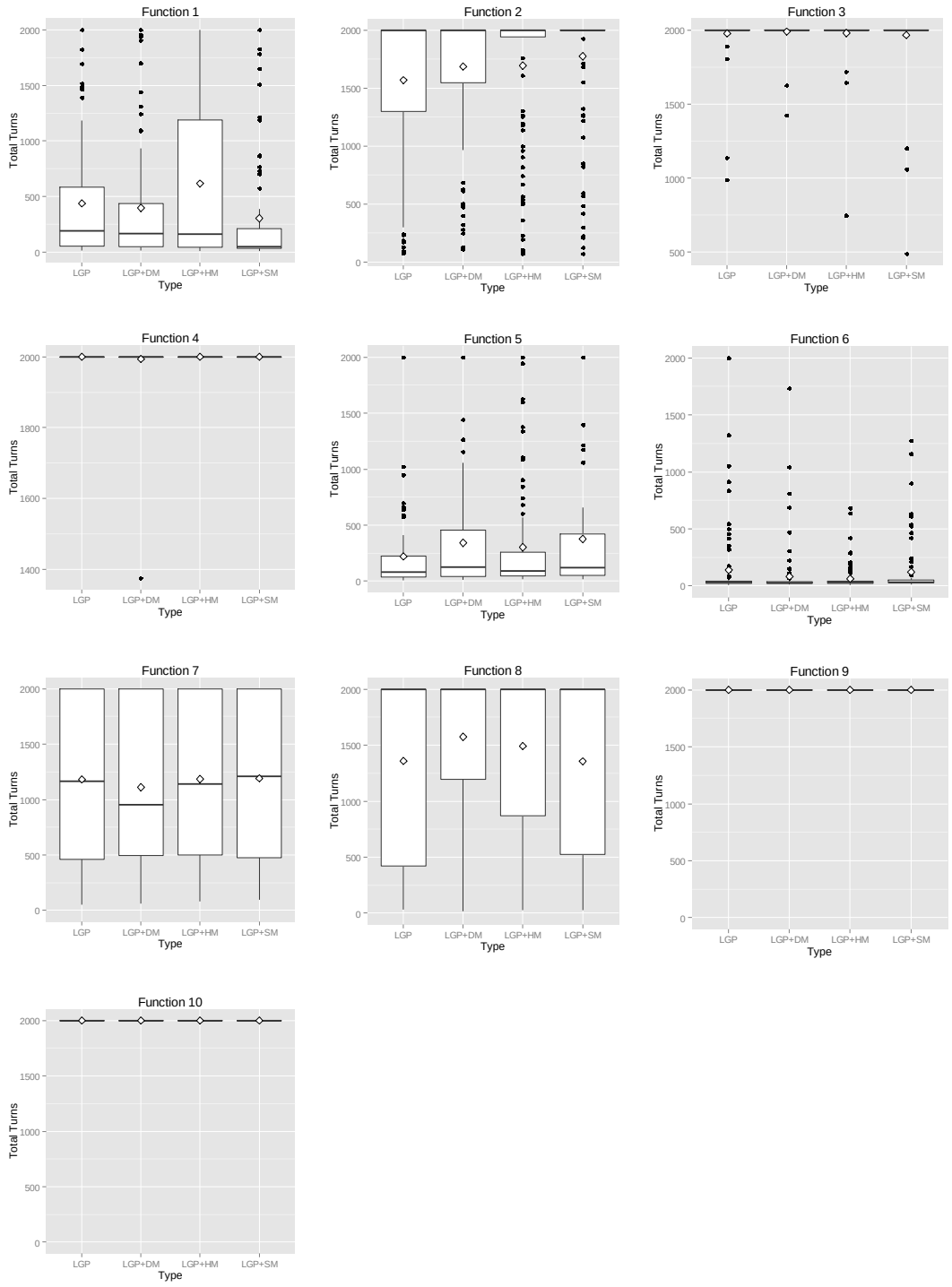


Figure 5.5 Boxplot of the generations needed to have a hit on Type II setting with the full setting for non-terminal and terminal sets on benchmark set I

From Figure 5.2 – 5.5 and Table 5.6 – 5.9, we can see that, for LGP, with the minimal setting, usually, fewer generations are needed to have a hit under Type II setting than Type I setting. With the full setting, this only happens in function 5 to 7. For other LGP types, in experiments with the minimal setting, the same performance increases like LGP can be observed. For LGP with SM, the performance increases in some cases are far greater than LGP. For example, in function 4, the mean generations needed is reduced about 50% in compare to about 17% in LGP. In experiments with the full setting, the performance between LGP types is not obvious. However, some LGP types tend to receive more benefit from Type II setting than LGP. For example, in function 1, the mean generations needed for LGP with SM are decreased about 16% in contrast to about 30% increasing in LGP. If we compare the performance on having a hit between four LGP types, we can see that, in experiments with minimal settings, this performance increases when more powerful recombination operators are used. In experiments with full settings, this performance decreases. However, LGP with SM and LGP with HM are less affected, and sometimes can have performance increase. We can also see that, in experiments with minimal setting, LGP with SM and LGP with HM are nearly always need less mean generations to have a hit than LGP and LGP with DM. In experiments with full setting, under Type I setting, LGP can frequently perform better than LGP with SM and LGP with HM. LGP with DM can perform better than LGP with SM and LGP with HM on function 5 and 7. Under Type II setting, on function 1, LGP with SM performs better than other LGP types. On function 2 and 5, LGP performs better than LGP with SM. On function 8, it is better than LGP with DM. LGP with DM performs better than LGP with SM. Overall, under Type II setting with full setting, the performance differences between LGP types are less significant than the difference in Type I setting with full setting.

From Table 5.10 – 5.13, we can see that, by using multi-point crossover operator and multi-action mutation operator, the mean best fitness are increasing for all LGP types on nearly all functions. With the minimal setting, on function 1 to 4, four LGP types can always have perfect hit except the experiment on function 4 of LGP under Type I setting. Under Type I setting with the minimal setting, LGP with SM and LGP with HM are better than LGP and LGP with DM on function 7 to 10. LGP and LGP with DM have higher mean best fitness than LGP with SM

and LGP with DM on function 5 and 6. Under Type II setting with the minimal setting, same situation appeared on function 5 to 8. On function 9, the differences between four LGP types become not obvious. On function 10, the situation is reversed. LGP and LGP with DM have higher mean best fitness than LGP with SM and LGP with HM. Under the full setting, the differences on mean best fitness between four LGP types become less distinctive as minimal setting. With Type I setting, LGP with DM have higher mean best fitness than LGP with SM and LGP with HM on function 5 and 7. LGP have higher mean best fitness than LGP with SM and LGP with HM on function 5, and is better than LGP with HM on function 7. On function 3, LGP with SM and LGP with HM have better mean best fitness than LGP and LGP with DM. LGP with SM has better performance than LGP with DM on function 1 and show higher mean best fitness than LGP with HM on function 2. With Type II setting, LGP has better mean fitness than LGP with SM and LGP with HM on function 2. It also show better performance than LGP with SM on function 5 and 10, and has better mean best fitness than LGP with HM on function 8. LGP with DM has better mean best fitness than LGP with SM and LGP with HM on function 10. LGP with SM has better mean best fitness than all other LGP types on function 1, it also show better mean best fitness than LGP on function 7. It and LGP with HM show better mean best fitness than LGP and LGP with DM on function 9.

Table 5.6 Mean generations needed to have a hit on Type I setting with the minimal setting for non-terminal and terminal sets on benchmark set I

Function Index	LGP	LGP with DM	LGP with SM	LGP with HM
1	39.71	38.56	**33.39** ^{L,D}	34.15 ^{L,D}
2	75.69	75.3	55.88 ^{L,D}	**53.58** ^{L,D}
3	138.83	139.95	**107.04** ^{L,D}	121.41 ^{L,D}
4	210.19 ^{D}	232.24	**170.25** ^{L,D,H}	199.3
5	2000	2000	2000	2000
6	2000	2000	2000	2000
7	2000	2000	1921.41 ^{L,D}	**1830.08** ^{L,D}
8	2000	2000	2000	2000
9	2000	2000	2000	2000
10	2000	2000	2000	2000

Table 5.7 Mean generations needed to have a hit on Type I setting with the full setting for non-terminal and terminal sets on benchmark set I

Function Index	LGP	LGP with DM	LGP with SM	LGP with HM
1	**340.07** ^{S}	416.75	365.35 ^{D}	483.62
2	1502.22	**1463.09**	1464.86	1588.75
3	**1968.02** ^{H}	1994.62	1970.82	2000
4	**1998.84**	2000	2000	2000
5	**361.43** ^{S,H}	375.02 ^{S,H}	740.6	759.88
6	337.27	398.46	**230.15**	302.28 ^{L}
7	1328.46 ^{S,H}	**1284.15** ^{S,H}	1558.59	1674.98
8	**1238.23**	1425.69	1344.16	1352.85
9	2000	2000	2000	2000
10	2000	2000	2000	2000

Table 5.8 Mean generations needed to have a hit on Type II setting with the minimal setting for non-terminal and terminal sets on benchmark set I

Function Index	LGP	LGP with DM	LGP with SM	LGP with HM
1	12.78	11.61	**9.89** ^{L,D}	10.63 ^{L}
2	36.47	37.34	24.08 ^{L,D}	**22.38** ^{L,D}
3	84.66	93.45	51.8 ^{L,D}	**51.52** ^{L,D}
4	199.01	169.92	**92.82** ^{L,D}	102.04 ^{L,D}
5	**1994.68**	2000	2000	2000
6	2000	2000	2000	2000
7	1665.46	1730.48	1320.6 ^{L,D}	**1238.82** ^{L,D}
8	2000	2000	2000	2000
9	2000	2000	2000	2000
10	2000	2000	2000	2000

Table 5.9 Mean generations needed to have a hit on Type II setting with the full setting for non-terminal and terminal sets on benchmark set I

Function Index	LGP	LGP with DM	LGP with SM	LGP with HM
1	439.51	397.3	**304.89** ^{L,D,H}	617.39
2	**1568.89** ^{S}	1686.32	1774.54	1694.67
3	1978.26	1990.55	**1967.49**	1981.11
4	2000	**1993.74**	2000	2000
5	**220.33** ^{S}	342.73	376.99	303.35
6	138.86	82.21 ^{S}	121.71	**62.23**
7	1181.82	**1111.47**	1191.24	1186.13
8	1359.57 ^{D}	1575.21	**1356.94**	1492.81
9	2000	2000	2000	2000
10	2000	2000	2000	2000

Table 5.10 Mean best fitness on Type I setting with the minimal setting for non-terminal and terminal sets on training cases of benchmark set I

Function Index	LGP	LGP with DM	LGP with SM	LGP with HM
1	1	1	1	1
2	1	1	1	1
3	1	1	1	1
4	0.9995	1	1	1
5	0.9253 ^{H}	**0.9355** ^{S,H}	0.9164	0.907
6	0.9454 ^{S,H}	**0.9461** ^{S,H}	0.9307	0.9303
7	0.7388	0.7457	0.8391 ^{L,D}	**0.8626** ^{L,D}
8	0.9037	0.9067	0.9206 ^{L,D}	**0.9278** ^{L,D}
9	0.5672	0.5668	**0.5693** ^{L,D}	0.5687 ^{L,D}
10	0.6454	0.6443	0.6454 ^{L,D}	**0.6463** ^{L}

Table 5.11 Mean best fitness on Type I setting with the full setting for non-terminal and terminal sets on training cases of benchmark set I

Function Index	LGP	LGP with DM	LGP with SM	LGP with HM
1	**0.9969**	0.9964	0.9964 ^{D}	0.9956
2	0.9783	0.9808	**0.983** ^{H}	0.9777
3	0.966	0.9654	**0.9706** ^{L,D}	0.9694 ^{L,D}
4	0.9662	0.9659	0.9654	**0.9668**
5	**0.9938** ^{S,H}	0.9934 ^{S,H}	0.9922	0.9916
6	**0.9993**	0.9989	0.9989	0.9989
7	0.9879 ^{H}	**0.9882** ^{S,H}	0.987	0.9865
8	**0.9885**	0.9876	0.9881	**0.9885**
9	0.5778	0.5768	0.578	**0.5779**
10	**0.6376**	**0.6376**	0.6328	0.6329

Table 5.12 Mean best fitness on Type II setting with the minimal setting for non-terminal and terminal sets on training cases of benchmark set I

Function Index	LGP	LGP with DM	LGP with SM	LGP with HM
1	1	1	1	1
2	1	1	1	1
3	1	1	1	1
4	1	1	1	1
5	**0.9867** ^{D,S,H}	0.9849 ^{S,H}	0.9803	0.982
6	**0.9511** ^{S,H}	0.9492 ^{S,H}	0.9418	0.943
7	0.9841	0.9843	0.9867 ^{L,D}	**0.9869** ^{L,D}
8	0.9168	0.9181	0.9417 ^{L,D}	**0.942** ^{L,D}
9	0.5682	0.5677	**0.5687** ^{D}	0.5684
10	**0.6483** ^{S,H}	0.648 ^{S,H}	0.6474	0.6473

Table 5.13 Mean best fitness on Type II setting with the full setting for non-terminal and terminal sets on training cases of benchmark set I

Function Index	LGP	LGP with DM	LGP with SM	LGP with HM
1	0.9962	0.9968	**0.9981** ^{L,D,H}	0.9958
2	**0.9802** ^{S,H}	0.977	0.9731	0.9755
3	0.9683	0.9689	0.9695	**0.97**
4	0.9639	**0.9655**	**0.9655**	0.9646
5	**0.9944** ^{S}	0.9941	0.9935	0.9941
6	0.9997	**0.9999**	0.9996	0.9996
7	0.9896	0.9894	0.9906 ^{L}	**0.9907**
8	**0.9879** ^{H}	0.9865	0.9877	0.9866
9	0.5752	0.576	**0.5829** ^{L,D}	0.5822 ^{L,D}
10	**0.6368** ^{S}	0.6359 ^{S,H}	0.6288	0.6308

From Table 5.14 – 5.17, we can see that, the mean best fitness is lower at test cases for all LGP types. Only differences are the polynomials, as exact solution can usually be found, the mean best fitness on these benchmark problems are not drop. We can also see that, the differences on mean best fitness on all benchmark problems are less significant as the difference in Table 5.10 – 5.13. If compare the times of having significant mean best fitness to other LGP types between mean best fitness under training cases and test cases, we can see that, LGP with SM and LGP with HM keep more significant mean best fitness under test cases than LGP and LGP with DM. This may mean the solutions given by these two types are more stable.

Table 5.14 Mean best fitness on Type I setting with the minimal setting for non-terminal and terminal sets on test cases of benchmark set I

Function Index	LGP	LGP with DM	LGP with SM	LGP with HM
1	1	1	1	1
2	1	1	1	1
3	1	1	1	1
4	0.9929	1	1	1
5	0.8537 ^{H}	**0.8544** ^{S,H}	0.8513 ^{H}	0.8489
6	**0.685** ^{S,H}	0.6726	0.6627	0.6675
7	0.6357	0.6446	0.7606 ^{L,D}	**0.7914** ^{L,D}
8	0.8141	0.8159	0.8081	**0.835** ^{S}
9	0.5509	**0.5541**	0.5501	0.5483
10	0.6435	0.6426	0.6453	**0.6439** ^{D}

Table 5.15 Mean best fitness on Type I setting with the full setting for non-terminal and terminal sets on test cases of benchmark set I

Function Index	LGP	LGP with DM	LGP with SM	LGP with HM
1	0.8721	0.8307	**0.8771**	0.8364
2	0.5098	0.5576	**0.5616** ^{L,H}	0.5034
3	**0.214**	0.1944	0.192	0.1928
4	0.1051	0.1033	**0.1089**	0.1088
5	0.8688	0.8848 ^{L}	0.8998 ^{L,D}	**0.9095** ^{L,D}
6	**0.9885**	0.9811	0.9773	0.9801
7	0.9074	**0.9169**	0.9129	0.9089
8	**0.9687**	0.968	0.9676	0.9673
9	0.5597	0.5606	0.5611 ^{L}	**0.5614** ^{L}
10	**0.6049**	0.6016	0.6014	0.6038

Table 5.16 Mean best fitness on Type II setting with the minimal setting for non-terminal and terminal sets on test cases of benchmark set I

Function Index	LGP	LGP with DM	LGP with SM	LGP with HM
1	1	1	1 ^{L}	1
2	1	1	1 ^{L,D}	1 ^{L,D}
3	1	1	1	1
4	1	1	1	1
5	0.858	**0.8612** ^{S}	0.8537	0.8542
6	0.7083	**0.7117**	0.7055	0.7084
7	**0.9231**	0.9224	0.9185	0.9168
8	0.8092	0.8224	0.8498 ^{L,D}	**0.8566** ^{L,D}
9	0.5541	0.551	**0.5553**	**0.5553**
10	0.6445	0.6469	0.6469 ^{L}	**0.6473** ^{L}

Table 5.17 Mean best fitness on Type II setting with the full setting for non-terminal and terminal sets on test cases of benchmark set I

Function Index	LGP	LGP with DM	LGP with SM	LGP with HM
1	0.8363	0.8419	**0.9328** ^{L,D}	0.8581
2	**0.5264** ^{S,H}	0.4626	0.4501	0.4606
3	**0.2138**	0.1942	0.2061	0.2074
4	0.1067	0.1047	0.1041	**0.1069**
5	0.8925 ^{D}	0.8813	0.8943 ^{D}	**0.9013** ^{L,D}
6	**0.994**	0.998	0.9908	0.9906
7	0.9075	**0.9229** ^{L,H}	0.9205	0.9134
8	0.9656	0.9665	**0.9678**	0.9675
9	0.5597	0.5599	**0.5616** ^{L,D}	0.5614 ^{L,D}
10	**0.6043**	**0.6043**	0.6034	0.6016

From the experiment results, we can see that, under the minimal setting, LGP with SM usually converge significantly faster and tend to have significantly higher mean best fitness; under the full setting, LGP usually has faster convergence speed and higher mean best fitness. However, under the full setting, four LGP types usually do not show significant performance differences.

LGP with DM usually has similar performace on both convergence speed and mean best fitness as LGP. LGP with HM usually has similar performance on both convergence speed and mean best fitness as LGP with SM.

From the experiment results, we can also see that with the minimal setting, the exact solutions for polynomials (function 1 to 4) in the benchmark set I can always be found. Under this setting, the exact solution of other functions cannot be found because of the insufficient of expression power. With the full setting, the exact solution of the simpler polynomials, like function 1 to 3, can usually be found. However, with the increase of expression power, the exact solution of other benchmark problems, like function 5 to 8, can also be found occasionally. The best solutions of all runs under benchmark set I can be found in Table A.1.3 to A.1.6 at Appendix 1.

For the overall performance of the crossover operators used in this study, because of their aggressiveness on introducing changes in chromosome is low, their performance cannot be compared with any crossover operators described in Uy et al. (2011).

If we consider the mean best fitness in the experiment results of this study, we can see the LGP with gene representation used in this study cannot have good results on function 9 and 10, the algorithm in Uy et al. (2011) also cannot always achieve high mean fitness on these functions, so this can be seen as a common issue. LGPs used in this study usually cannot have the same amount of hits as the GP algorithm introduced in Uy et al.(2011). The reason for this can be thought as they are designed for different purposes. In this study, the gene representation is designed for the direct usage of EM, the crossover and mutation operators are designed to have minimized ability to let the performance analysis be more straightforward.

From the experiment results, we cannot see obvious performance improvements to an LGP by using a DM. The reason of this can be seen from Figure 5.6 and 5.7. These figures show the typical changes on the probability of DVNTs given by the DM during the evolution. Figure 5.6 shows the changes on the probability of DVNTs at chromosome index 1. Figure 5.7 shows the changes on the probability of DVNTs at chromosome index 2. The training cases for LGP is from function 8 in benchmark set I. In Figure 5.6 and 5.7, legend "*" means ×, "/" means ÷.

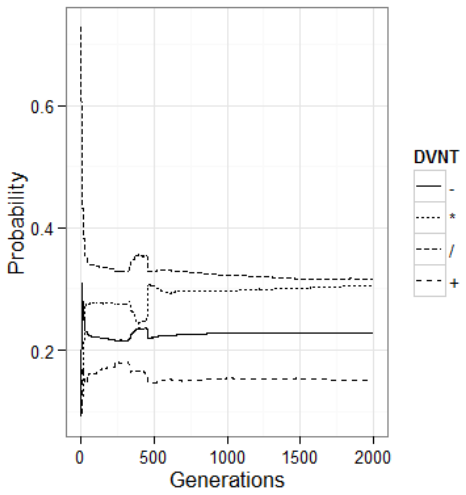


Figure 5.6 The changes of the probability of DVNTs at chromosome index 1

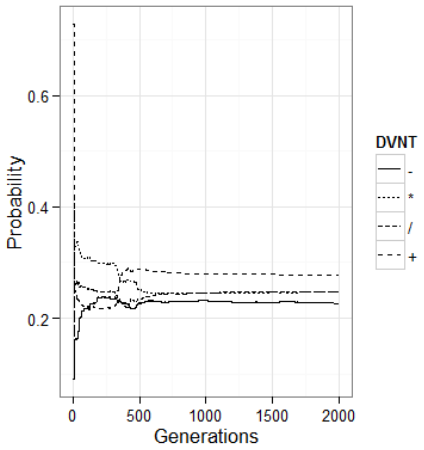


Figure 5.7 The changes of the probability of DVNTs at chromosome index 2

From these figures, we can see the probability of DVNTs changes as the DM gets more and more information and they eventually become stable. The problem is when the changes stop, the differences between them are not large enough. We can see that the values of these probabilities are very near to 0.25, which is the probability of the DVNTs in uniform distribution. This suggests that the DM

algorithm suggested is not able to create the necessary differences between the probability of DVNTs.

Table 5.18 shows the mean values of the amount of full sets in the suggested DVNT sets returned by SMs, and the amount of appearances of +, −, × and ÷ in these suggested DVNT sets. These sets are acquired by using SMs giving suggested DVNT sets for all individuals in the generation before the last generation. The values are the average value of 100 LGP runs. For each SM type, a SM only gives suggestion sets on the individuals evolved with the experiment setting that uses it. The target function of these LGP runs is the function 2 in benchmark set I. The type of LGP is LGP with SM.

From Table 5.18, we can see that under the minimal setting, SM with both types of structure usually do not give full sets. This may suggest that the prediction accurracy is acceptable under the condition of only minimizing the false negative rate. Under the full setting, the accurracy of the classifiers in SM with Type I structure drops greatly, as nearly all the suggested DVNT sets are full sets. This makes LGP with SM usually uses the same DVNT set in increase action as LGP.

For SM with Type II structure, under the full setting, the amount of full sets returned is not increased as greatly as SM with Type I structure. However, if we also consider the amount of DVNTs appeared in the suggested DVNT set, we can see average appearance times of + and × decreased, at the same time, the average appearance times of − and ÷ increased. We can also see increased conflict situations (return both + and − or × and ÷). This suggests that the predict accuracy of the classifiers used in SM decreases under the full setting.

Table 5.18 Mean times of suggested DVNT sets given by different types of SM

			Type I Setting		Type II setting	
			Minimal Setting	Full Setting	Minimal Setting	Full Setting
Total Appeareances	Full Sets		15.68	480.41	16.48	35.35
	+		421.65	487.15	423.94	281.77
	−		173.71	495.78	147.57	297.8
	×		316.24	495.5	367	131.22
	÷		173.97	499.29	147.57	202.16
	Total Times Appeared Together	+ and ×	268.99	483.55	320.45	97.02
		− and ÷	173.6	495.27	147.57	111.53
		+ and −	121.97	483.51	97.99	124.98
		× and ÷	36.32	494.94	36.55	60.86

2. Experiment results on Benchmark Set II

For benchmark set II, no hit can be made for runs of all LGP types. From Table 5.19 and 5.20, we can see that under the minimal setting, LGP has significantly higher mean best fitness on training cases than LGP with SM and LGP with HM. LGP with DM has significantly higher mean best fitness on training cases than other LGP types. Under the full setting, four LGP types do not show significant differences on mean best fitness at training cases. From Table 5.21 and 5.22, we can see that, under the minimal setting, LGP and LGP with DM have significantly higher mean best fitness on test cases than LGP with SM and LGP with HM. Under the full setting, at the test cases, the differences between four LGP types are not significant.

From all four tables, we can see that, under the minimal setting, LGP with DM has the best performance; under the full setting, all LGP types have similar performances. If compared with the experiment results in Yeh (1998), the accuracy acquired by four LGP types is lower than the accuracy given by an ANN (about 0.8 versus over 0.9).

The best solutions of all runs on benchmark set II can be found in Table A.1.1 and A.1.2 in Appendix 1.

Table 5.19 Mean best fitness of Type II setting with the minimal setting for non-terminal and terminal sets on training cases of benchmark set II

LGP	LGP with DM	LGP with SM	LGP with HM
0.8091 ^{S,H}	**0.8103** ^{L,S,H}	0.8047	0.8054

Table 5.20 Mean best fitness of Type II setting with the full setting for non-terminal and terminal sets on training cases of benchmark set II

LGP	LGP with DM	LGP with SM	LGP with HM
0.8225	0.822	0.8209	0.8207

Table 5.21 Mean best fitness of Type II setting with the minimal setting for non-terminal and terminal sets on test cases of benchmark set II

LGP	LGP with DM	LGP with SM	LGP with HM
0.8187 ^{S,H}	0.8172 ^{S,H}	0.8075	0.8107

Table 5.22 Mean best fitness of Type II setting with the full setting for non-terminal and terminal sets on test cases of benchmark set II

LGP	LGP with DM	LGP with SM	LGP with HM
0.8165	**0.817**	0.8145	0.8133

6. Conclusion

6.1 Summary of The Work

In this study, EMs are used to improve the performance of LGP. EMs use trained ML models to generate suggestions for different mutate actions to influence the course of evolution. Two types of EMs are designed. The first type is SM. The source of knowledge of a SM is the pre-existing knowledge of symbolic regression. It is the knowledge about the effect of adding one math function to the end of another math function. In this study, these effects are called as signatures. The ML models used in SM are trained with training sets created from the values of signatures at the training cases of the benchmark problem. The output of a SM is the suggestion set on different gene component types. They are based on the predictions given by the ML models. To increase the chance of including the suitable candidates into the candidate sets, a SM only tries to minimize the false negative rate when making the decisions on whether to include a candidate in a candidate set or not. The second type is DM. Its source of knowledge is the fitness and chromosome of all individuals of all generations of an LGP run. The ML models in a DM are used to capture the effect of using a particular gene component at a particular chromosome index on the overall fitness, and they update at each generation. The output of a DM is the distribution of different gene component types. A third model HM is also used. The purpose of using it is to test the performance improvements when both SM and DM are used. In this study, all types of EM are used only in mutation operator. The SM is used only for increase action, the DM is used only for change action.

The gene representation used in this study works fine on nearly all benchmark problems. It does not work well on function 9 and 10 in benchmark set I. This suggests that the gene representation might need further improvements. However, if we consider the experiment results on the same functions in Uy et al. (2011), we can find that these functions also have low mean best fitness under GP. This suggests that the gene representation may not be the sole source of this performance issue.

In this study, two sets of recombination operators are used. The first set contains roulette wheel selection, single-point uniform crossover, and single-action mutation. The second set contains roulette wheel selection, multi-point uniform crossover, and multi-action mutation. From experiment results, we can see that an LGP using the second set of recombination operators usually have higher mean best fitness than an LGP using the first set of recombination operators. This may suggest that, by using more powerful recombination operators, the performance of the LGP can be improved. However, when using the second set of recombination operators, an LGP usually needs more generations to have a hit. On this aspect, when using with an EM, especially using with a SM, the performance of an LGP tends to be less affected. This may suggest that, when using a SM, an LGP can acquire more performance improvements from using more powerful recombination operators.

When compared with the LGP, the LGP with DM usually do not show significant performance improvements in both benchmark sets. However, if we consider the experiment results under the minimal setting on benchmark set II, we can see that using DM can make LGP have higher mean best fitness than other LGP types. The reason of why DM does not always do this might be the design of DM cannot create necessary differences between the probabilities of different gene components.

On the other hand, when used with a SM, an LGP can have significant performance improvements on convergence speed. When compared with an LGP, an LGP with a SM usually needs significantly fewer generations to have a hit under the minimal setting. Under the full setting, an LGP with a SM usually has similar convergence speed as an LGP. Under the minimal setting, an LGP with SM can have higher mean best fitness than an LGP at benchmark set I. Under the full setting, LGP with SM and LGP tend to have similar mean best fitness. With the test cases, we can see LGP and LGP with SM usually do not show significant differences on the mean best fitness. The reason for LGP with SM cannot have better performance than LGP under the full setting should be the drop of accuracy of the classifiers. The cause of it might be the insufficient training sets.

Two different types of SM did not show distinctive performance differences when used with LGP on benchmark set I. There are performance differences between

them; however, if we also consider the performance differences of LGP under different sets of recombination operators, we can see that the trend of the performance differences between LGP with different SM types are similar to the performance differences between LGPs. Therefore, the differences between different SM types might be caused by the recombination operators.

For HM, its effects on LGP are similar to the SM. When an LGP with SM performs significantly better than an LGP, an LGP with HM also tends to perform significantly better than an LGP and vice versa. The inability of DM might be the cause of this.

6.2 Contributions and Future Research Directions

In this thesis, EM is studied. It is a general framework. In EM, different ML algorithms can be used to model the same knowledge, if they can learn the dataset (generated from the knowledge) and give same type of output. In this study, SM needs ML models with outputs that can be seen as confident values and can learn from data set with numeric attributes; DM needs ML models with outputs that can be seen as distributions and can be updated at runtime.

New recombination operators are adopted for LGP in this study. The crossover operator adopted cannot make change to the length of a chromosome. This function is moved to the mutation operator, which now have three functions, increase the length of a chromosome (increase action), decrease the length of a chromosome (decrease action), and change the genetic contents of a chromosome (change action). The benefit to have this design is it is easier to make control on the course of evolution. If we use old style crossover-mutation operator combination, the change in chromosome is often have multiple effects, that is the length of a chromosome can be increased or decreased while the genetic contents of two chromosome exchange. When using the combination adopted in this thesis, a chromosome can only be changed one effect a time. That is, crossover operator only do exchange, in mutation operator, increase action only add new genes to a chromosome, decrease action only delete existing genes from a chromosome, change action only changes genetic content of a gene.

The future research should be focused on 1) explore better interactions between SM and DM; 2) expend EM into other recombination operators and the

initialization phase; 3) find better ML algorithms for EM on symbolic regression task. The reason to have 1) and 2) is the basic idea behind EM is to control the evolution, which can be thought like human selection for plants and animals.

Appendix

Appendix 1 Best Solutions

Table A.1.1 and A.1.2 show the best solutions of different LGP types of 100 runs under benchmark set II. Table A.1.3-A.1.6 show the best solutions of 100 runs of the four LGP types under different experiment settings under benchmark set I. In these tables, "*" means ×, "/" means÷. To save space, "Math." is removed to save space. In Table A.1.1 and A.1.2, variables are represented with array notation a[i], where i means attribute i in the data set, $i \in (0,1,2,\dots,7)$.

Table A.1.1 Solutions with highest fitness of 100 runs on Type II setting with the minimal setting for non-terminal and terminal sets of benchmark set II

Type	Solution	Fitness
LGP	(((((a[7]*((((((((((((((((a[7])))))))))))))))-a[3]*a[7]-a[3]))*a[7])*a[7]*a[7]-a[0]))*a[4]*a[7]*a[7]*a[7]*a[7]*a[7]*a[7]))*a[7]*a[7]-a[0]*a[7]*a[7])*a[7]	0.8200
LGP with DM	(((((((a[7]*a[7]))*a[7]-(((a[3]/a[7]/(a[7])))-a[0])))))*a[7]/(a[7]+a[7])*(a[7])*a[7]*a[7]*a[7])	0.8254
LGP with SM	((((((a[3]+a[4])))))*((a[7])*(a[7])*(a[0]+((a[1])*a[1])*a[7]*a[7]*a[7])-a[3]))*a[7]	0.8169
LGP with HM	a[7]*((((((((((((a[7]))))))))))*a[7]*(a[4])-a[0])*a[7]*a[7]*a[7]*a[7]*a[7]*a[7]*a[7]*a[7]-a[0]))*a[7]*a[7]	0.8151

Table A.1.2 Solutions with highest fitness of 100 runs on Type II setting with the full setting for non-terminal and terminal sets of benchmark set II

Type	Solution	Fitness
LGP	sin(((a[7])))*cos((((((a[7])+cos(a[7]))+cos(a[7]))*cos(a[1])+cos(a[7])+sin(a[0]))*cos(a[4])+cos(a[7])))	0.8534
LGP with DM	sin((a[7]-cos((((((a[4])))))))*sin(((a[0]*sin(a[7])-cos(a[4]))*cos(a[1])+a[7]*sin(a[7])))	0.8493
LGP with SM	(a[7]*((((a[7]*(a[7])))))-cos(a[4])*(a[7]*cos(a[1])*exp(a[0])))	0.8462
LGP with HM	sin((a[7]-cos(a[3])*sin(a[0])*sin((a[7]))))+cos(a[4])*sin(((a[7]*sin(a[7])*cos(a[6])-a[7]-((((a[7]))))))	0.8469

Table A.1.3 Solutions with highest fitness of 100 runs on Type I setting with the minimal setting for non-terminal and terminal sets of benchmark set I

Type	Index	Solution	Fitness
LGP	1	((x))+((x)+x*x)*x	1.0
	2	((x))+(((x*x)+x)*x+x)*x	1.0
	3	((x)*(((x)))+(x))*(x+x*x*x)+x	1.0
	4	(((x)))+((x)+(((x*x)*x+x))*(x+x*x))*x	1.0
	5	((x-((((((x)))))/((x+x))*x*x*x))*x-x/x	0.9888
	6	x+(((x)+(x)*x))/((((x*((x)*x)))+x)*x*x*x*x+x)*x	0.9709
	7	x/(((((x)/(x+x)+(((x)))/x)))/x+x)+x	0.9895
	8	(x)/(((((((((x+x+x+x)))))))))+x/((x+x+x+x)/x)+x/(((x+x/x)))	0.9872
	9	(x*((((y+x)))))/((y+((x+x+(x)))/y/y))/y	0.5767
	10	((((((x*(((y)))))-((((x))-y/((((y)-x)))*x))))*(x/x-y*y)*x))*y*(x*x-y*x)-y/y	0.6537
LGP with DM	1	(x)*((x)+x*x)+x	1.0
	2	((x))+x*(((x*x)+x)*x+x)	1.0
	3	(((x*((x))+x)))*((x*x*x)+x)+x	1.0
	4	(x)*((x))*((((x*x+x)*x+x)*x+x))+x*x+x	1.0
	5	x*((((((((((x))*(x*(x))))))))-x-x/((x)+x)*(((x))))*x)*x*x*x+x*x-x/x	0.9895
	6	(x+((((((((((x))))))))))+(x*x)-x*x/(((x/x-(x+x)))/x+x+x)*x*x	0.9694
	7	(x)/((((((x/x/(x+x)))+x/x/x)))+x)+x	0.9895
	8	x/(((((x+((x+x+(x))))))/x))+(x)/(x/(((x+x))+x/x/x)/x+x)	0.9760
	9	x*((((((((((((((x)))*y*y))))))))))*y/(y-(x+(y*x*y*y)*x*x*x*x)*x*(x*x)*x*x*x)	0.5739
	10	((((((y))))))*(((((x-(y)))/y/y/(y*y-(((y))*x))*y/(((x)+x))*(y*x*y))*x*y))/y-y/y	0.6522
LGP with SM	1	(x)+((x)+x*x)*x	1.0
	2	x+(((x)*(((x+x*x)))+x)*x)	1.0
	3	((x+(((((x)))*x))))*(x+x*x*x)+x	1.0
	4	(((((x*(x*(x)))+x)*(((x)+x*x)))+x)*x+x)	1.0
	5	(((x-(x)))-x/(((x)*x*x)+x)	0.9842
	6	x+(((((((((x))+x*x)))))))*(x/(x*x*x*x*(x+x)+x))	0.9651
	7	((x))+((x))/(((x/(x+(x)/x/(x+x))/x+x/x)/x)+x)	0.9917
	8	x/((((x/((x)))+x)))+x/((((x+x+x+x)))/(x+x/x))	0.9872
	9	(y)*((y))*(((y*((x-x*y*x))*((((y+x))))*((((y*(((((x+x))))))+x*(x))+x*y)*y)*y-y*y)*y)*y)*y)	0.5738
	10	((y-(x)))/(x-y)	0.6477
LGP with HM	1	(x)+(((x))+x*x)*x	1.0
	2	(x)+(((x))+(x+(x*x))*x)*x	1.0
	3	((x)+(((x)*(x))*(x)))*(x+x*x)+x	1.0
	4	x*((((((x)))*(x)+x))*((x*x*x+x))+x)+(x)	1.0
	5	((((x-(x)))))-x/((x)*x*x+x)	0.9842
	6	x+(((((((x)))))))+(x/x)))/((((x/x)/(x*(x*x)))+x*x*x+x)*x*x)	0.9709
	7	x+((((x))))/(((((x+x+(x/(x)))/((x+(x+x+x))+x)))/x+x/x/x+x))	0.9952
	8	x/(((((x/(x))+x))))+x/(x*((((x+(x))+x+x)/(x+x/x)/(x))))	0.9872
	9	(((y-x*y*y)))*(((((((((y+((x)))))))))*x*((x+x))-x))*y*((y*(y+y)*y*y))*y*y*(x+y)	0.5769
	10	(((x)-y))/(y-x)	0.6477

Table A.1.4 Solutions with highest fitness of 100 runs on Type I setting with the full setting for non-terminal and terminal sets of benchmark set I

Type	Index	Solution	Fitness
LGP	1	(((x)))*((x)*x+x)+x	1.0
	2	x*(x*(x)+((1)))*((x+1))	1.0
	3	(((x*((((x))))+x)))*(x*x*x+x)+x	1.0
	4	(((1+x)*exp(((x)*sin(x)))/cos((x))/((((x))))*sin(x)*sin(x))*1	0.9923
	5	cos((x*(((x)))))-cos(x)-cos(x)	0.9965
	6	sin(x)+sin((((x))+(x)*x))	1.0
	7	sin((1))*sin(((((x*sin((1*sin(1)))))*sin(((1)*sin(x))))*sin(1)))+x	0.9957
	8	exp(((1)*log((x))/(1+1)))	1.0
	9	sin(((y/sin(((((y*sin((y*(y+sin(x)))))))*sin((x*(y)))*sin((y*y/sin(y)))))))*y	0.6059
	10	sin(((((y*cos((((x/sin((((y)-x)*exp(x)))))/y)))*exp(x))-cos(x)*sin(y))*sin(y)))-cos((x))	0.6713
LGP with DM	1	x+((x))*(x+x*x)	1.0
	2	x*(((x)))+x+x*((x+x*x))*x	1.0
	3	((x)+((((1)))))*(x*x*x+x)*x+x	1.0
	4	(((((x))))/cos((((((x)))))+exp(x)/exp(1))/exp(1)*(((x)*x))/cos((x/exp(1)*sin(1)))*x+exp(x)*x/cos(x)	0.9894
	5	cos((((x))*x))-cos(x)-cos((x))	0.9965
	6	sin(((x+((x))*x)))+sin(x)	1.0
	7	sin((((1*sin(((1)+sin(x))))*((((((x))*sin(1)*sin(1))))))*sin(1)))*sin(1)*x/sin(1)+sin(x)	0.9967
	8	log((x))*log(((((x)))+exp(1)+log(x)/exp(1)))/exp(((1)))+1	0.9924
	9	cos(((x*x/cos(((((y*exp(y)))/cos((y))*exp(x)))))/sin(y)))*x*(x)	0.5927
	10	cos(((((y/sin((((((x/x-cos(y)))))))))/sin((x-y))/sin(x))-sin((y))*y))*cos(y)-cos(y)	0.6851
LGP with SM	1	(1+(((x))*x+x))*(x)	1.0
	2	x+x*(((1+((((x))))*x+x)))*x	1.0
	3	((((((x))))))+(((x)))*x*(x*x+x*x*x+x+1)	1.0
	4	((x*((x)))+(x))/cos(x)*exp(((x*sin(x)/exp((1*cos(1)))))))	0.9875
	5	cos((x*(((x)))))-cos(x)-cos(x)	0.9965
	6	sin(x)+sin(((((x)))+x*x))	1.0
	7	log(((1+((x)))*(x*x+1)))	1.0
	8	exp(((1+1/exp(1))*log(x)/exp((((1))))))	0.9960
	9	sin((y/cos(((y)))/log((y*sin(y)-sin((x))))-exp(((((y)))))*exp(y)-exp(y)*cos(x)))	0.6027
	10	cos(((((y/sin(((((((x-y))*sin(y)/sin(x)))))))))+sin((y))))*sin(y)*sin(y)-cos(y)-(y*cos(y))*sin(y)	0.6685
LGP with HM	1	((((x*((x))+x)*1*x)))+x	1.0
	2	((x+(((1)))))*(x*x*x+x)	1.0
	3	sin(x)*exp(((((((x)))))))+x*((x+sin(1)+sin(1)*x*x))*x*(x)	0.9862
	4	exp((((x))))*((((x)))/cos(x)*sin(x))+x+(x-sin(x))+x*sin(x)*sin((x*x*x))*sin(((((x)))))	0.9890
	5	sin(((x*(x))))*cos(x)-cos(((x-sin(x))))	0.9974
	6	sin(((x)+((x))*x))+sin(x)	1.0
	7	exp(((1+cos(x))/exp(((x)))/exp(1)))*x/exp(1)*sin((x))+((x))	0.9951
	8	exp((1*log((x))/((1+1))))	1.0
	9	sin((((((y*cos((((y))))))/cos(x)/sin(y))/sin(y)/sin(((y)))))))*cos(y)*cos(((y/sin((x)))*cos((y+cos(x)*cos(x)))/cos(y)*exp(x)))*exp(x)	0.5921
	10	sin((((((y-x))))/(y-x)))*sin(((y-x)))-cos((x*y*exp(((x-sin(y)*exp(x))))))	0.6689

Table A.1.5 Solutions with highest fitness of 100 runs on Type II setting with the minimal setting for non-terminal and terminal sets of benchmark set I

Type	Index	Solution	Fitness
LGP	1	(((x))*((x))+x)*x+x	1.0
	2	x+((((((x))))))*x+x+x*x*x)*x	1.0
	3	(x+((((x)))*x))*(x*(x*x)+x)+(x)	1.0
	4	(((((x*(((x)))))+x)*((x)*x*x+x)+x)*x)+x	1.0
	5	(((((x))))*((((x)))))/(x-((x*x*x-x-x)*x)*x*x*x)*x-x/x	0.9921
	6	((x))*(x+(((x)*(x+x))+x))/((x)/(((x))+x*x*x)*((((x))))+x*x*x)*((x/x))-(x)*x	0.9743
	7	x/(((((((x))+((x)))))/x/(x*x+x+x)))+x+x/x/x)+x	0.9924
	8	x/(((x+(x+x)/(((x)/((x+x))+(x))/x)))/(((x-x/x))))+x/x	0.9721
	9	y/((((y/x+(x+x+x)/(((((((((y*x)))))))))/(y)))))*y*(y+x)*y/y/y	0.5759
	10	(((((((((x)))))))*x))/(y*y-x-((x-x))/x/((y-x))-x-(x))*y-x/x	0.6557
LGP with DM	1	(((x))+x*(x))*(x)+x	1.0
	2	((((x*(((x)))))+x)*x+x)*x+x	1.0
	3	(x+(((x)))*x)*((x*(x))*x+x)+x	1.0
	4	((x+(((x)))*x)*((x+x*((x))*(x)))+x)*x+x	1.0
	5	((x))*(((((((x)))))))-x/(x+x)*x*(x)*x*x-x/x	0.9888
	6	(x)-(((((((x)))))))/((x+x)*x+(((x))+x)+x)+x)*x)*x*x+x+x*(x)	0.9652
	7	(x)/(((((x+x/x))/((((x+(x)))+x/x)))+x/x)/x+x)+x	0.9923
	8	x/((((x))+(x)+((((((x+x))/x+(x)+x))))))*(x+x+x*x+((x))+x+x)/x	0.9855
	9	x*((((((y))))))*y/(y+(((x+(x+x*y))))/y+x)/y*y*y*y	0.5742
	10	((((y-((x)))*x/(((x))+(((x)-y))-y)*x/x)+(x*y*y)))-x/(x)	0.6530
LGP with SM	1	(x*(x*((x))+(x))+x)	1.0
	2	x+((x)+(x)*((x)+((x))*x))*x	1.0
	3	(((((x*((x)))+x)))*(x+x*x*x)+x)	1.0
	4	((((x+(x*((x)))*(x))*(x+x*x))+(x))*x)+x	1.0
	5	((((x)*x-(((x)))/x)))-x*x*x/(((x/(x/x)+x)/x)/x)	0.9888
	6	(x+((x*((x)+x*(((x))))))/(x*(x*x)*x*((x)+x)+x))	0.9651
	7	(x)+x/((x+(((x))+x)/((x)+((((x+x))))/x))/x/x+x)	0.9924
	8	(((x+x)))*(((((((((x)+x))+x/x))))/((((x+x+x*x+x)+x+x)))+x)-x)	0.9845
	9	(y/((((((((((y))))))))))*(((y)))/y/(y*(y)*y*y)+(y+y+y)*x*y*y*y*y)*x*(y*(y))*y)	0.5771
	10	((((x)))/(((x)-(y))/x/(((y-x)))-x)*(x)*y*x-x)/x	0.6525
LGP with HM	1	((x*(x)+(x))*x)+(x)	1.0
	2	((x+((((x)*x))))*(x)+x)*x+x	1.0
	3	((((x*(x))+x)*((x*(x*x))+x)+x))	1.0
	4	((((x+((((x)))*(x))*x)*(x*x+x)+x)))*x+x	1.0
	5	(((((x*(x)))))-(((x)/x))-x*x*x*x*x*x/(x+x)	0.9888
	6	x+((x))*((((x))*x+x))/((x+((((x*x*x+x)))))*((x*x))*x*x))	0.9709
	7	x+(((((((x)))))))/((((x)/x)+x/(x))/x+x/(x+x/x/(x)/x)*x)	0.9928
	8	((((((x+x))))))/((x/x+x/((x/x/x/x+x+(x))/(x+((x+x))/x))))	0.9752
	9	((x-(((x)))*y*((x))*((y*(y)))))*((x+y))*y*y*(y*(((y+(y)+y)*x)*y-y)-x)	0.5744
	10	((y)-x)/((x)-y)	0.6477

Table A.1.6 Solutions with highest fitness of 100 runs on Type II setting with the full setting for non-terminal and terminal sets of benchmark set I

Type	Index	Solution	Fitness
LGP	1	x*((((((x))))))+x+x*x*x	1.0
	2	((1+x))*(x*(((x*x)))+x)	1.0
	3	x+(((x)))*(x*((((1+(x)*x+x)*x)))+x)	1.0
	4	(((((((x)))))/cos(x)+((x))*sin((x))+cos(1)))*x*x*x+exp(x)*x	0.9867
	5	sin(((x*((x)))))*cos(x)-1	1.0
	6	sin((x))+sin((x*(x)+x))	1.0
	7	((x/exp((x))/exp((((1))-cos((x))))+x)*sin((x)))/exp(1)+x	0.9953
	8	exp(((1+cos((1))/((1-1-cos(1))-cos((1))))*log((x))))	1.0
	9	cos((((x*sin(((y))))*exp(((x))))))*cos(((x*x/sin(y))))/exp(y)*x*x	0.5886
	10	sin((((y))-cos(y)/((y)-x)))*sin(((x)-cos(((y)/sin(y)))*cos(x)))-cos(y)	0.6669
LGP with DM	1	((((x*x))))*x+x+x*x	1.0
	2	x*(((x))*(1+x+((x)*x))+sin(x)/sin((x)))	1.0
	3	sin((((((((((((x)))))))-cos(x)*sin(x))))*sin(((x*((x))+log(1)-cos(x)+exp(x))))))*x*sin(x)+log(1)*x*x+exp(x)/cos(x)*x	0.9944
	4	cos((((((((((1)))))))*sin((1+exp(x))))))*sin(x)/cos(x)*sin(x)*sin((((1))))/cos(x)*sin(1)*sin(x)*sin(x)+exp(x)/cos(x)*(x)	0.9906
	5	cos(((x)*((x))))-cos(x)-cos(x)	0.9965
	6	sin(x)+sin(((x*x)+x))	1.0
	7	log(((((1)))+((x+x*x))*x+x))	1.0
	8	exp((x/((((x))/log(x)+1/log(x)*(x)))))	1.0
	9	cos((((x/y)-cos((x)))/sin(y)/y/sin((y+sin(y)))))*((x)*((x)))	0.5899
	10	cos((y/sin((((((y-x)))))/sin(((y)))*sin((y-cos(y)-x))))*cos(y)-sin(y)/sin(y)	0.6733
LGP with SM	1	((x)+(((x)))*x)*x+x	1.0
	2	((x+(((x*x+(x)+x*((x*x))))*x))	1.0
	3	(((((1)/cos(x))))+x)*((((((x*x))))))*x+x*x-sin(x)+x+x	0.9912
	4	x*(((((((1)))))+x*(((x*sin(x)*exp(x)+x))))+x*x*sin(x)*x*sin(x)+x)	0.9892
	5	cos((x))*sin((((x)*x)))-1	1.0
	6	sin((x*((x))+(x)))+sin(x)	1.0
	7	sin(1)*sin((x*sin(1)*sin(((((1*sin((((1)*sin(x)))))))))*sin(1)))+x	0.9970
	8	exp((((1*log(x))/(1+1))))	1.0
	9	sin((y/sin(y)-log((((y/sin(((y/cos(y)/sin(((x-sin(y)))))/exp((((((x/sin(x)))/y/sin(y)))))))))*sin(x)))/cos(y)))	0.6025
	10	sin(((y*((x)*(y)))*sin(((y*sin(y)/(y-x))/y))))-y/((y))	0.6607
LGP with HM	1	x+((((x))))*(x+x*x)	1.0
	2	x*(((((((x)))))))+x+x*x*(x*x+x)	1.0
	3	((((x+x*(((x)*x))))*(x+1)*x))+x	1.0
	4	(((x+1)))*x*exp((((x+(x))*cos(1))*x))	0.9813
	5	sin(((x)*((x))))*cos(x)-1	1.0
	6	sin(x)+sin((((((1))))*x*x+x))	1.0
	7	x+((x))/exp(((((x))/exp(1)*cos(1)*sin(x))))*cos(((1)))*sin(x)	0.9949
	8	exp(((1*sin((1)))*log(x)/exp(((((1)/exp(1))/sin(1)/sin(1))))))	0.9994
	9	sin(((((x*x*cos((x*(x/sin(y)/cos(((y/cos(((y)*cos(x))))/cos((y)))))*x))))/cos((y)))/exp(y)))	0.5936
	10	sin((y*sin((y))/y*sin(((((x*sin(y))))/x-sin(y)/(((x)-y))))))-cos((((((y*sin(x)*sin(x)))*sin(x)*sin(x)))*sin(x)))	0.6664

References

Bacardit, J., Bernadó-Mansilla, E. & Butz, M.V. (2008). Learning classifier systems: looking back and glimpsing ahead. In Bacardit, J., Bernadó-Mansilla, E., … Takadama, K. (Eds), Learning Classifier Systems. (pp 1-21). Berlin Heidelberg: Springer. DOI: 10.1007/978-3-540-88138-4_1

Brameier, M. (2004). On linear genetic programming. PhD thesis. Dortmund University of Technology. Retrieved from http://d-nb.info/1011533146/34

Brameier, M., & Banzhaf, W. (2007). Linear Genetic Programming. Springer. Retrievable from http://www.springer.com/us/book/9780387310299

Cantú-Paz, E., & Kamath, C. (2001). On the use of evolutionary algorithms in data mining. In Abbass, H. A. and Sarker, R. A. and Newton, C. S. (Eds), Data Mining: A Heuristic Approach. (pp-48-71). Idea Group Publishing.

Cervone, G., Michalski, R. S., & Panait, L. A. (2000). Combining machine learning with evolutionary computation: Recent results on LEM. In Michalski, R. S. & Brazdil, P. B. (Eds), Proceedings of the Fifth International Workshop on Multistrategy Learning. (pp-41-58).

Coello, C. A. C., & Becerra, R. L. (2002). Adding knowledge and efficient data structure to evolutionary programming: a cultural algorithm for constrained optimization. In Langdon, W. B., Cantú-Paz, E., … Jonoska, N.(Eds), proceeding of the Genetic and Evolutionary Computation Conference. (pp-201-209). New York, USA.

De Castro, L. N. (2007). Fundamentals of natural computing: an overview. Physics of Life Reviews. 4(1), pp-1-36. DOI:10.1016/j.plrev.2006.10.002

Downey, C., Zhang, M., & Browne, W. N. (2010). New crossover operators in linear genetic programming for multiclass object classification. In Pelikan, M. & Branke, J. (Eds), proceeding of GECCO 2010. ACM. 885-892.DOI:10.1145/1830483.1830644

Gandomi, A.H., Alavi, A.H., & Arjmandi, P. (2010). Formulation of elastic modulus of concrete using linear genetic programming. Journal of Mechanical Science and Technology. 24(6), pp-1273-1278.DOI:10.1007/s12206-010-0330-7

Gandomi, A. H.,Mohammadzadeh S. D., & Alavi, A. H. (2014). Linear genetic programming for shear strength prediction of reinforced concrete beams without stirrups. Applied Soft Computing. 19, pp-112 – 120. DOI:10.1016/j.asoc.2014.02.007

Gaudesi, M., Squillero, G., & Tonda, A. P. (2013). An efficient distance metric for linear genetic programming. In Blum, C. and Alba, E. (Eds), proceeding of GECCO 2013. (pp-925-932). ACM. DOI:10.1145/2463372.2463495

Guven, A., & Kişi, Ö. (2011). Daily pan evaporation modeling using linear genetic programming technique. Irrigation Science. 29(2), pp-135-145. DOI:10.1007/s00271-010-0225-5

Guven, A., & Kişi, Ö. (2013). Monthly pan evaporation modeling using linear genetic programming. Journal of Hydrology. 503, pp-178-185. DOI:10.1016/j.jhydrol.2013.08.043

Hall M., Frank E., … Witten I. H. (2009). The WEKA Data Mining Software: An Update; SIGKDD Explorations, Volume 11, Issue 1. WEKA software can be downloaded at http://www.cs.waikato.ac.nz/ml/weka/downloading.html

Harman, M., McMinn, P., & Yoo, S. (2012). Search based software engineering: techniques, taxonomy, tutorial. Empirical Software Engineering and Verification. In Meyer, B. & Nordio, M. (Eds), Lecture Notes in Computer Science. (pp-1-59).Springer Berlin Heidelberg. DOI:10.1007/978-3-642-25231-0_1

Harwerth, M. (2011). Experiments on Islands. In Silva, S., Foster, J. A., & Giacobini, M. (Eds), proceeding of EuroGP 2011. (pp-239-249). Springer. DOI:10.1007/978-3-642-20407-4_21

Hauschild, M., & Pelikan, M. (2011). An introduction and survey of estimation of distribution algorithms. Swarm and Evolutionary Computation, 1(3), pp-111-128. DOI:10.1016/j.swevo.2011.08.003

Hu, T., & Banzhaf, W. (2009). Neutrality and variability: two sides of evolvability in linear genetic programming. In Rothlauf, F. (Ed), proceeding of GECCO 09. (pp-963-970). ACM. DOI:10.1145/1569901.1570033

Hu, T., Payne, J. L., & Moore, J. H. (2011). Robustness, Evolvability, and Accessibility in Linear Genetic Programming. In Silva, S., Foster, J. A., & Giacobini, M.(Eds), proceeding of EuroGP 2011. (pp-13-24). Springer. DOI: 10.1007/978-3-642-20407-4_2

Hu, T., Banzhaf, W., & Moore, J. H. (2013). Robustness and Evolvability of Recombination in Linear Genetic Programming. In Krawiec, K., Moraglio, A., & Hu, B. (Eds), proceeding of EuroGP 2013. (pp-97-108). Springer. DOI:10.1007/978-3-642-37207-0_9

Jones, G. (2002). Genetic and Evolutionary Algorithms. In Encyclopedia of Computational Chemistry. DOI:10.1002/0470845015.cga004

Kovacs, T. (2012). Genetics-Based Machine Learning. In Rozenberg, G., Bäck, T., & Kok, J. (Eds), Handbook of Natural Computing. (pp-937-986). Springer Berlin Heidelberg. DOI: 10.1007/978-3-540-92910-9_30

Koza, J. R. (1992). Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press.

Kumar, D., & Beniwal, S. (2013). Genetic algorithm and programming based classification: a survey. Journal of theoretical and applied information technology. 54(1), pp-48-58

Larrañaga, P., Karshenas, H., & Santana Roberto. (2013). A review on evolutionary algorithms in Bayesian network learning and inference tasks. Information Sciences. 233, pp-109-125. DOI:10.1016/j.ins.2012.12.051

Luke, S. (2009). Essentials of Metaheuristics. Lulu. Retrievable at http://cs.gmu.edu/$\sim$sean/book/metaheuristics/, online version 1.3

McPhee, N. F., & Poli, R. (2008). Memory with memory: Soft assignment in Genetic Programming. In Keijzer, M. (Ed), proceeding of GECCO '08. (pp-1235-1242). ACM. DOI:10.1145/1389095.1389336

Mehr, A. D., Kahya, E., & Olyaie, E. (2013). Streamflow prediction using linear genetic programming in comparison with a neuro-wavelet technique. Journal of Hydrology. 505, pp-240-249. DOI:10.1016/j.jhydrol.2013.10.003

Mehr, A. D., Kahya, E., & Yerdelen, C. (2014). Linear genetic programming application for successive-station monthly streamflow prediction. Computers & Geosciences. 70, pp-63-72. DOI:10.1016/j.cageo.2014.04.015

Nguyen, T. T., Yang, S., & Branke, J. (2012). Evolutionary dynamic optimization: A survey of the state of the art. Swarm and Evolutionary Computation, 6, pp-1-24. DOI:10.1016/j.swevo.2012.05.001

Oltean, M., & Groşan, C. (2003). A comparison of several linear genetic programming techniques. Complex systems. 14(4), pp-285-313

Oltean, M., Groşan, C., & Mihăilă, C. (2009). Genetic programming with linear representation : a survey. International Journal on Artificial Intelligence Tools. 18(2), pp-197-238. DOI:10.1142/s0218213009000111

Pelikan, M., Goldberg, D. E., & Lobo, F. G. (2002). A survey of optimization by building and using probabilistic models. Computational Optimization and Applications. 21(1), pp-5-20. DOI:10.1109/acc.2000.879173

Poli, R., Langdon, W. B., & McPhee, N. F. (2008). A field guide to genetic programming. Published via http://lulu.com and freely available at http://www.gp-field-guide.org.uk

Pentaho. (n.d.). MLPRegressor. Retrieved from http://wiki.pentaho.com/display/DATAMINING/MLPRegressor

Reynolds, R. G. (1994). An introduction to cultural algorithms. In Sebald A.V.and Fogel L.J (Eds), Evolutionary Programming --- Proceedings of the Third Annual Conference. (pp-131-139). World Scientific Press. San Diego, CA, USA.

Reynolds, R. G., & Peng, B. (2005). Cultural algorithms: computational modeling of how cultures learn to solve problems: an engineering example. Cybernetics and Systems. 36(8), pp-753-771. DOI: 10.1080/01969720500306147

Saeed K. B., Amir H. G., …& Amir H. A. (2013). Numerical modeling of concrete strength under multiaxial confinement pressures using linear genetic programming. Automation in Construction. 36, pp-136-144. DOI:10.1016/j.autcon.2013.08.016

Shavandi, H., &Ramyani, S. S. (2013). A linear genetic programming approach for the prediction of solar global radiation. Neural Computing and Applications. 23(3-4), pp-1197-1204. DOI: 10.1007/s00521-012-1039-6

Uy, N.Q., Hoai, N.X., & Galván-López, E. (2011). Semantically-based crossover in genetic programming: application to real-valued symbolic regression. Genetic Programming and Evolvable Machines. 12(2), pp-91-119. DOI:10.1007/s10710-010-9121-2

Watchareeruetai, U.,Takeuchi, Y., & Ohnishi, N. (2011). Redundancies in linear GP, canonical transformation, and its exploitation: a demonstration on image feature synthesis. Genetic Programming and Evolvable Machines. 12(1), pp-49-77. DOI:10.1007/s10710-010-9118-x

Wilson, G. C., & Banzhaf, W. (2008). A Comparison of Cartesian Genetic Programming and Linear Genetic Programming. In O'Neill, M.,Vanneschi, L., & Tarantino, E. (Eds), proceeding of EuroGP 2008. (pp-182-193). Springer. DOI: 10.1007/978-3-540-78671-9_16

Wilson, G. C., & Banzhaf, W. (2009). Soft memory for stock market analysis using linear and developmental genetic programming. In Rothlauf, F.(Ed), proceeding of GECCO 2009. (pp-1633-1640). ACM. DOI:10.1145/1569901.1570119

Wilson, G. C., & Banzhaf, W. (2010). Interday foreign exchange trading using linear genetic programming. In Pelikan, M., & Branke, J.(Eds), proceeding of GECCO 2010. (pp-1139-1146). ACM. DOI:10.1145/1830483.1830694

Wilson, G. C., Leblanc, D., & Banzhaf, W. (2011). Stock trading using linear genetic programming with multiple time frames. In Krasnogor, N., & Lanzi, P. L. (Eds), proceeding of GECCO 2011. (pp-1667-1674). ACM. DOI:10.1145/2001576.2001801

Wojtusiak, J. (2009). The LEM3 System for Multitype Evolutionary Optimization. Computing and Informatics. 28(2), pp-225-236

Yeh, I-C. (1998). Modeling of strength of high performance concrete using artificial neural networks. Cement and Concrete Research. 28(12), pp-1797-1808. The dataset is downloaded from: http://archive.ics.uci.edu/ml/machine-learning-databases/concrete/compressive/

Zahiri, A., & Azamathulla, H. Md. (2012). Flow discharge prediction in compound channels using linear genetic programming. Journal of Hydrology. 454–455, pp-203-207. DOI:10.1016/j.jhydrol.2012.05.065

Zahiri, A., & Azamathulla, H. Md. (2014). Comparison between linear genetic programming and M5 tree models to predict flow discharge in compound channels. Neural Computing and Applications. 24(2), pp-413-420. DOI:10.1007/s00521-012-1247-0

Zhang, J., Zhan, Z.H., … Shi Y. (2011). Evolutionary Computation Meets Machine Learning: A Survey. Computational Intelligence Magazine, IEEE. 6(4), pp-68-75. DOI:10.1109/mci.2011.942584