Universiteit Leiden

Master Computer Science

A GE Benchmark and an automated GE comparison system

Name: Yitan Lou

Student ID: s1996177

Date: 29/10/2019

Specialisation: Computer Science and Advanced Data Analytics

1nd supervisor: Dr. Hao Wang
2st supervisor: Prof. Dr. Thomas Bäck

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

Grammatical Evolution (GE), an application of the Evolutionary Algorithm (EA), is usually used to produce computer programs automatically. In the past decade, many GE and GE-variants systems have been implemented by researchers in the GE community. It brought us a simple problem, which GE system is the better one? However, there is no universal GE benchmark available over a long time, and most researchers were testing their systems in their way without a guideline.

In this work, we are proposing a highly-flexible benchmark for GE systems and applied this benchmark into an automatic GE comparison system, which can be used to compare the performance of different GE and GE-variant systems. Benefit from the design of the 'Kernel-Interface' structure of this proposing benchmark, it is a cross-language benchmark, which means it can be used to test GE systems implemented in different computer languages. Meanwhile, this benchmark also incorporated an automatically hyper-parameter tuning algorithm with the name of MIP-EGO [1] as a module, which can help systems to find out the most suitable configuration and thus reduce the external effect from hyper-parameter settings as much as possible. As a test of this proposing benchmark, two GE systems (PonyGE2 and SGE) are automatically tested, and the difference in the performance of different systems, as well as some interesting patterns on the choose of hyper-parameters, were founded. For instance, all tested systems are in favor of large population size on all problems.

Acknowledgements

This master thesis was done at LIACS, Leiden under the supervision of Prof. Dr. Thomas Bäck and Dr. H. Wang. I want to thank them for their generous help, advice, and patient guidance during this project. Furthermore, I would like to thank my girlfriend and all my friends in Leiden for providing spiritual support and for distracting me from stress. The accomplishment of the thesis would never be possible without any of you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 What is Grammatical Evolution?

Automatic programming is a popular topic in the field of artificial intelligence and computer science today. Many methodologies to this topic have enjoyed quite much success in this field. One of them, Grammatical Evolution, which is usually written in abbreviation as GE, is an application of Evolutionary Algorithm (EA) that inspired by the evolutionary process in biology. GE has the ability to automatically generate mathematical expression, string sequence, or even program fragment in any computer language [3][4].

Grammatical Evolution is generally a framework that allows a computer to automatically generate an executable program. Under such a frame, every individual is a bit string, which can be derived into a program fragment with the help of concrete derivation rules. And a large number of individuals consist of the population. The evolutionary process is performed on the population, which means that new individuals can be produced by "elder" individuals, and those individuals can also be eliminated in accordance with the "healthy" level.

1.2 Dealing Problems

Although the concept of GE has been published for only a few years, the community has developed a considerable number of GE systems and GE variants. These squandering flowers all claim that they are the most advantageous system or methodology in some aspects. Such a phenomenon is beneficial for the development of Grammatical Evolution since it keeps the diversity in this field. But it comes to a problem for many ordinary users of GE: 'Which system should I use for my specific problem?'.

Similar to the situation in the brother field Genetic Programming (GP) [5], which is using the Evolutionary Algorithm as the core to evolve the program, the GE community is also facing the problem of no availability of universal benchmark to test GE systems. When a new GE system is proposed, usually several test problems are necessary to test and demonstrate the performance of the coming system. For a long time, the test problems are usually selected based on the experience of developers in this field, making the test not that meaningful since everyone may use diverse problems to test. On the other side, this status is not that friendly for unprofessional users who want to use Grammatical evolution on their own applications, since they do not have sufficient experience to choose a suitable problem set to test several GE systems on hand.

Another problem along with the testing GE systems is every problem may have its unique search space landscape. It leads to the fact that the same configuration can have total different performances over different problems. Since the hyper-parameters of most systems (e.g Evolutionary parameters) for most GE systems are manually controlled by users, the performance of a GE system may show some instability among a problem set, and also makes the task of comparison between several GE systems a tricky problem. These two problems are the main issues to deal with in this project.

1.3 Project Overview

In this work, we are trying to deal with abovementioned problems by designing an automated GE system comparison system. To achieve this target, a Benchmark for Grammatical Evolution systems with expandability is designed and implemented. Meanwhile, an automatic hyper-parameter tuning method is used in this work combining with the benchmark to reduce passive influences from the interplay between systematic configuration space and problem space at most.
Generally, this automated GE system comparison system can be observed in two parts, the benchmark and the comparison system. The benchmark part is actually a collection of problems, in a way of their implementations and supportive documents. These benchmark problems are collected from other works in the field of Grammatical Evolution and some other related fields, such as Genetic Programming. Considering many GE systems are implemented in a different environment, benchmark problems are implemented in C, combining with a high-level language interface for systems implemented in high-level programming language calling. In such a way, the efficiency of the program and the unity of the problem are ensured.
Another part of this project is the comparison system. This part is used to apply the benchmark to make the comparison over different Grammatical Evolution systems. The designed system can automatically read in the predefined hyper-parameters list for every GE systems and control them to run benchmark problems. As for the problem of every benchmark problem with different best-suited configuration caused by their unique landscape in search space, a global optimization algorithm called MIP-EGO [6] is used to tune the configuration of hyper-parameters of GE systems, which can reduce the configuration-caused influence on the performance of GE system as much as possible.

1.4 Structure of this work

This rest of this thesis work is structured as follows. In Chapter 2, the background of evolution in both biology and computation aspects is introduced, which is the underlying mechanism of Grammatical Evolution. On the meanwhile, the primary representation method in Grammatical Evolution, Backus-Naur Form (BNF) is illustrated in this part to help the reader to understand how it works. In chapter 3, we expound the details of working scheme of grammatical evolution and tested GE systems. On the other side, we briefly discuss some problems that Grammatical Evolution is mainly facing. Furthermore, a global optimization method MIP-EGO, which was used to optimize the hyper-parameters of tested systems in this project is also included in this chapter. In chapter 4, we dive into the introduction of benchmark part and the implementation of the automatic testing system. Chapter 5 gives the result of our test results and they are analyzed and discussed in chapter 6, combining with our recommendation of future works.

Chapter 2

Background

2.1 Evolution in Biology

2.1.1 Gene and Molecular Basis

In the late 19th century, an Austrian scientist Gregor Mendel, observed the existence of "trait inheritance" of pea plants, which means that the traits of pea plants follow the statistical laws. In his long-term experiment, the traits of pea plants were recorded and tracked. In his work[7], the concept "discrete inherited units" was used to explain the characteristics observed in his previous experiments.

Figure 2.1: Gregor Mendel. Figure courtesy to A&E Television Networks[1].

Later in 1889, in the book "Intracellular Pangenesis"[8] from Hugo de Vries, an assumption was made that different physical characters have corresponding individual hereditary carriers. He named the hereditary carriers as "pangenes". In the year 1905, the term "gene" was firstly introduced in the work[9] from a Danish botanist Wilhelm Johannsen, to address the fundamental unit of heredity inside organisms. Until the mid 20th century, scientists finally found out that Deoxyribonucleic acid (DNA) is the substance people are looking for, which stores the genetic information[10][11], and the structure of DNA was proved to be simply linear, according to the experiment result from Benzer[12]. By combining with the observation in laboratory, Francis Crick proposed "the central dogma" of molecular biology in his work [13], which is often stated in short as "DNA makes RNA and RNA makes protein" [14], and the modern study of discrete inherited units of gene on a molecular level is usually known under the name of "Molecular biology" or "Genetics".
The mechanism of DNA carrying the genetic information has been unveiled with the development of molecular genetics[15], in which the chemical structure of DNA plays an important role. DNA is usually a big chain-shape molecule composing of four different kinds of basic nucleotide sub-units. The composition of each subunit was similar, a Deoxyribose, a phosphate group, and the most important, one of the four nucleobases, which includes adenine (A), cytosine (C), guanine (G), and thymine (T). Numerous deoxyribonucleic acid molecules connect to each other one by one with

[1]Gregor Mendel. (2019, April 17). Retrieved from `https://www.biography.com/scientist/gregor-mendel`

the help of Phosphodiester bond, which is the bond between the phosphate group and Deoxyribose sugar, and on the other side, nucleobases were also arranged into a string, on which the genetic information is stored.
"Two chains of DNA twist around each other to form a DNA double helix with the phosphate-sugar backbone spiralling around the outside", as it claimed in [15], and two hydrogen bonds can be formed by adenine (A) and thymine (T) when they are aligned, whereas three hydrogen bonds can be formed by cytosine and guanine (G), these two strands in a double helix must be complementary to each other. This structure can be easily understood with the help of Figure 2.2.

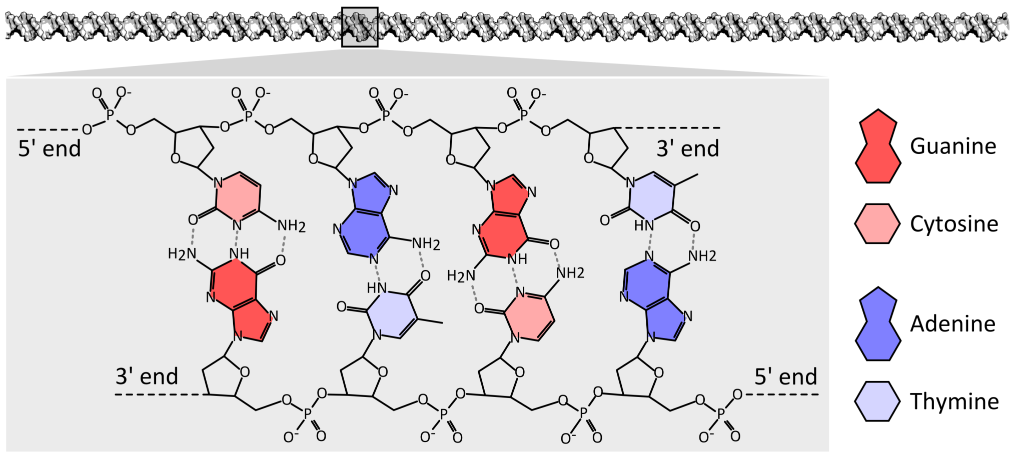


Figure 2.2: DNA's chemical structure. Non-covalent hydrogen bonds between the pairs are shown as dashed lines. Figure courtesy to T., Shafeee[3].

"A chromosome consists of a single, very long DNA helix on which thousands of genes are encoded"[15], and all chromosomes inside the cell core carry all genetic information unit, which is usually called 'Gene'. For human's instance, every person has 23 pairs of Chromosomes, including 22 pairs of autosomes and one pair of sex chromosomes. They record all genetic information a person has. In the field of molecular biology and genetics, the word 'Genome' is used to refer to all genetic material of an organism[16][17].

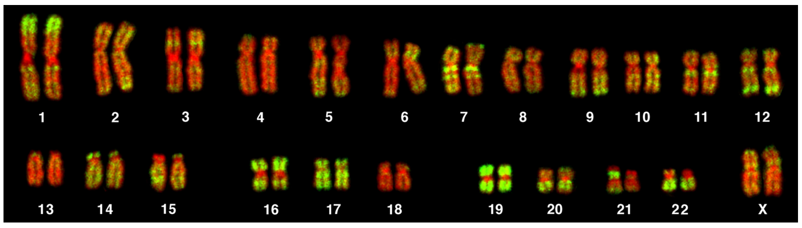


Figure 2.3: Fluorescent microscopy image of a human female chromosomes. Figure courtesy to Bolzer et al[5].

2.1.2 Expression of Gene

RNA is the abbreviation of Ribonucleic acid, which is also a kind of nucleic acid and shares a similar structure with the DNA molecule, and both of them have four different kinds of nucleobases. However, in most cases, the RNA molecule has only one stranded, and the length of RNA is much shorter than DNA molecule [18]. Meanwhile, The complementary nucleobase for adenine (A) is uracil (U) in RNA, rather than thymine (T), which plays a significant role in DNA structure[19].

[3]T., Shafee. (2015, April 17). The chemical structure of a four base pair fragment of a DNA double helix. Retrieved June 20, 2019, from `https://upload.wikimedia.org/wikipedia/commons/b/b2/DNA_chemical_structure_2.svg`

[5]Bolzer et al., (2005) Three-Dimensional Maps of All Chromosomes in Human Male Fibroblast Nuclei and Prometaphase Rosettes. PLoS Biol 3(5): e157 DOI: 10.1371/journal.pbio.0030157, Figure 7a

This structure allows RNA molecule to be complementary to a stranded of DNA molecule and record all the information it has. Therefore RNA plays an essential role in the whole process of the gene's expression.
In Eukaryote, the expression of genetic follows the central dogma2.4, and starts with transcription, in which an RNA copy of a gene's DNA sequence is produced. This process is performed by enzyme RNA polymerase (RNAP), which can locally open the double-stranded DNA and use one of them as a template to produce an RNA molecule, which is complementary to the template DNA segment[20]. The process of transcription makes an RNA copy of a gene's DNA sequence and it is called Messenger RNA (mRNA), which plays a role of an information carrier and it can be used to produce protein in the process of translation, a later phase of gene expression.

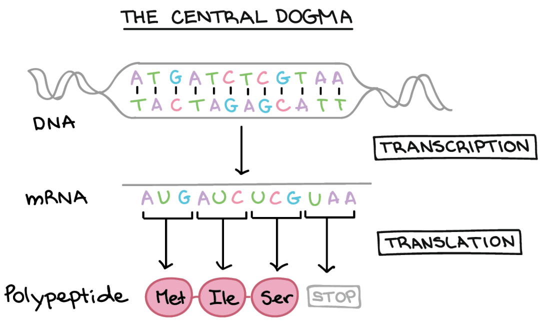
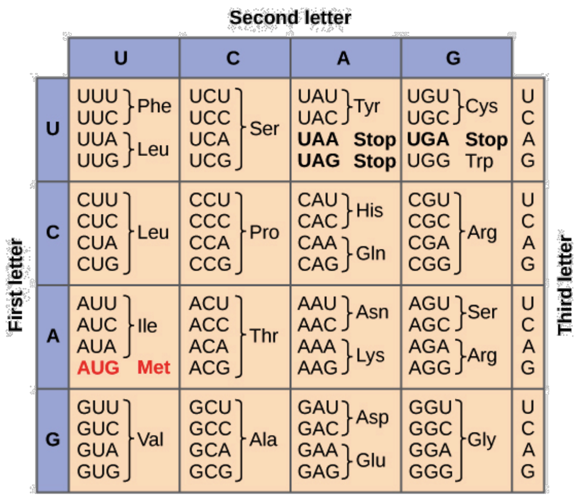


Figure 2.4: The Central Dogma. Figure courtesy to Khanacademy[6].

A codon refers to a sequence of three successive nucleotides on DNA or RNA sequence, in accordance with one specific amino acid or stops flag in the production of protein. [21] In the process of translation, the ribosome reads the mRNA sequence in and links the amino acid according to the codon series on mRNA with the help of tRNAs, who carry amino acids to the ribosome and match every codon with the amino acid it codes for[22]. Since there are four different kinds of nucleotides and each codon has three nucleotides, there are 64 ($4^3 = 64$) possible codons in total. Each codon can be translated to one specific amino acid except for three "stop" codon mark, and the rules of representation are summarized into so-called "The genetic code", as it shows in Figure 2.5, based on the work from Nirenberg et al. [23]



First letter	Second letter: U	C	A	G	Third letter
U	UUU, UUC Phe	UCU, UCC, UCA, UCG Ser	UAU, UAC Tyr	UGU, UGC Cys	U, C
	UUA, UUG Leu		**UAA Stop**, **UAG Stop**	**UGA Stop**; UGG Trp	A; G
C	CUU, CUC, CUA, CUG Leu	CCU, CCC, CCA, CCG Pro	CAU, CAC His	CGU, CGC, CGA, CGG Arg	U, C
			CAA, CAG Gln		A, G
A	AUU, AUC, AUA Ile	ACU, ACC, ACA, ACG Thr	AAU, AAC Asn	AGU, AGC Ser	U, C
	AUG Met		AAA, AAG Lys	AGA, AGG Arg	A, G
G	GUU, GUC, GUA, GUG Val	GCU, GCC, GCA, GCG Ala	GAU, GAC Asp	GGU, GGC, GGA, GGG Gly	U, C
			GAA, GAG Glu		A, G

Figure 2.5: The genetic code. Figure courtesy to OpenStax[7].

[6]Intro to gene expression (central dogma). (n.d.). Retrieved June 20, 2019, from `https://www.khanacademy.org/science/biology/gene-expression-central-dogma/central-dogma-transcription/a/intro-to-gene-expression-central-dogma`

[7]OpenStax, Biology. OpenStax CNX. Sep 29, 2015 `http://cnx.org/contents/185cbf87-c72e-48f5-b51e-f14f21b5eabd@9.87`.

2.1.3 Genetic Variation

Genetic variation is one of the most important motive powers in the natural world to keep diversity, which refers to the process of changes in the sequence of DNA, RNA or other cellular molecules across generations. Germ cells, such as sperm and egg, is the central place which genetic variation happens, but it can also be found in somatic or all other cells. It has several kinds of sources for genetic variation, such like mutation and genetic recombination. The variation that arises in germ cells can be inherited from individual to another one, or in other words, variation is unidirectional. This caused the affection of population and ultimately the whole evolution process.[24]
Specifically, Mutation refers to the change on the nucleotide sequence of the genome, which can belong to any organism, such like animals, plants, virus, or even extrachromosomal DNA[25]. Mutations can be caused by different reasons, includes errors during DNA duplication, any type of damage to DNA molecule, error during replication, the error-prone repair process of error-prone and countless unpredictable reasons[26]. The result of mutation may or may not be observable, for example in the case of one codon was changed to another one, but both of them produce the same amino acid, then this mutation may not have any effect to the phenotype of the organism.
Changes in DNA molecule caused by mutation could cause errors in protein production, which lead to the production of partially or completely non-functional proteins. If a mutation affects the production of a protein which plays a critical role in the organism, a medical condition can result and it becomes a harmful mutation. But on the other side, the effect of mutation may be positive in a given environment. For example, a mutation enables the organism better environmental stress than wild-type organisms. According to the research from SW Doniger et al. in [27], 7% of point mutations in noncoding DNA of yeast and 12% in coding DNA is harmful mutations. Other than that, mutations are either neutral or beneficial for the organism to some extent.

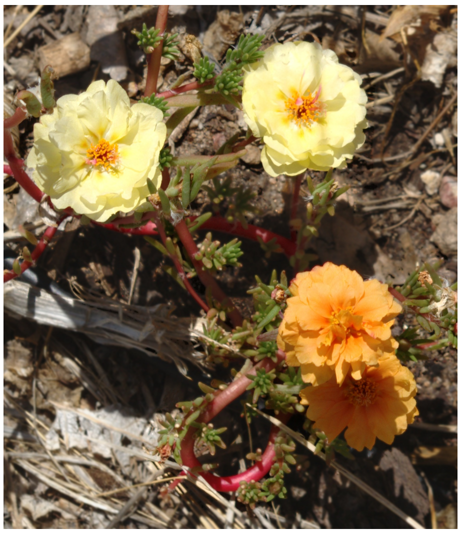


Figure 2.6: Different colors of flower produced by mutation. Figure courtesy to Friedman, J[8].

Apart from mutation, genetic recombination is another important category of genetic variation, which describes the process, that genetic material from different organism exchanges when offsprings are produced. This lead to the fact that gene fragment of offspring can be found on either of its parents, but differs from its parent when treating gene as a whole.
For prokaryotes (such as bacteria), recombination can trigger between individuals through transfer, or via the transmission of viruses (such as phage), and use genetic recombination to combine these genes into their own inheritance. But for more complex organisms, recombination is usually due to the interchange of homologous chromosomes. The exchange of genetic material between two homologous chromosomes has the name of chromosomal crossover, which is firstly described by Thomas Hunt Morgan[28], based on the discovery of Frans Alfons Janssens [29].

[8]Friedman, J. (2019, July 13). Moss rose or rose moss, Portulaca grandiflora, with flowers of two colors as a result of a mutation. The orange is probably the mutant, as it's closer to the purple wild type. Retrieved July 22, 2019, from `https://en.wikipedia.org/wiki/Mutation#/media/File:Portulaca_grandiflora_mutant1.jpg`

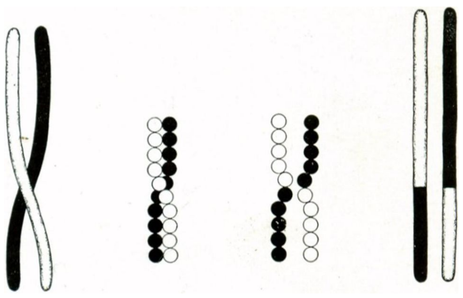


Figure 2.7: Thomas Hunt Morgan's illustration of crossing over. Figure courtesy to Chamary, J[10].

As a result of the chromosomal crossover, a new arrangement of a maternal and paternal allele can be seen on the same chromosome. Even though the genes on chromosomes appear in the same order, some alleles may have disparity. Since the existence of such a process and its result, it is theoretically possible to get offsprings with any combination of parental alleles, and those alleles appear on one offspring does not have any influences to each other. In other words, all alleles are independently inherited. This important principle of "independent assortment" is known as one of the fundamental principles of genetic inheritance[30]. Despite crossover is typically between homologous regions of chromosomes, the crossover also has the possibility to be a Non-homologous one. In normal cases of DNA replication, each strand of DNA is used as a template to produce a new strand with the help of the principle of complementary base pairing, two identical and paired chromosome should be created if it works properly. But sequence mismatch in this process, which is theoretically rare but still possible to happen, may lead to unequal exchanges. This could result in the deletion or insertion of genetic information to the chromosome, and be considered to be a general resource of mutation within a genome[31].

2.1.4 Natural Selection

The term natural selection was popularized by English naturalist Charles Darwin, used to refer the difference on phenotype performs influence on survival and reproduction of individual under natural conditions[32]. it is known that natural variation occurs within all populations of organisms, some of the variations may lead to some differences in traits (phenotype). These characters may reflect better resilience to the environment, which could possibly become a reproductive and heritable advantage since the gene controls these trait is also inheritable. For example, some difference may enlarge an individual's chance of surviving in a specific environment. Then this individual could have a higher chance to inherit its gene to offspring since it has a higher reproductive rate. Even though such heritable advantage from a trait to others is very slight in one generation, it could become dominant over many generations. In this way, nature "select for" those individual with reproductive advantage, and finally result in change is evolution, as Darwin described[33].
Formally, the process of natural selection could be described in the following part[34]:

- Variation. Variation in any sense of organism, which may involve body size, the number of offspring, resistance to disease, etc.
- Inheritance. Some traits are passed to offspring from parents with consistency, while others are prone to be affected by environment or weakly inheritable.
- Higher rate of population growth. The most population can generate more offspring than the local environment can support. High mortality could be a shared experience for every generation of the organism.
- Differential survival and production. Individuals with traits, which can better suit the local environment, have a better chance to produce more offspring to the next generation.

[10]Chamary, J. (2016, May 30). Modern Biology Began In The New York 'Fly Room'. Retrieved June 22, 2019, from `https://www.forbes.com/sites/jvchamary/2016/03/18/the-fly-room/#319d281306d5`

An powerful evidence of natural selection was observed in Britain in the process of the industrial revolution. The peppered moth has two colors in Great Britain, light and dark specifically. Due to the industrial revolution, the air was severely polluted and many of trees became blackened. This gives dark-colored moth a better chance to survive and produce offspring, as they had the advantage of hiding from their predators. In around 50 years from the first dark moth being caught, almost all newly caught moth in area of industrial Manchester was dark-colored. Only after the air quality starts to improve by the Clean Air Act in 1956, dark moth becomes rare again and light moth re-dominated in peppered moth population. The term "Fitness" is used to evaluate individual adaptation to the environment, which plays a central role in the concept of natural selection. For the former case, it can be concluded that dark moth has higher fitness than light one during the industrial revolution, and the light moth has better fitness in other time verse visa.

2.2 Evolutionary Computation

2.2.1 Introduction and Brief History

In the area of Computer science, there is a family of the algorithms, which is inspired by the natural biological evolution process and mainly used for global optimization problems, is called Evolutionary Computation (EC). Due to the source of its inspiration, sometimes people also call it Natural Computing, Evolutionary Algorithm, etc. Technically, Evolution Computing is a sub-field of Artificial Intelligence (AI) and, a family of the generic population-based meta-heuristic optimization algorithm.

Surprisingly, the start-point of applying the evolutionary computation can be traceable to 1940s, which is even earlier than the breakthrough of computers [35]. In the late 1940s, the idea of "genetical or evolutionary search" was firstly proposed by Alan Turing. And by the end of the 1960s, an actual computer program for "optimization through evolution and recombination" was implemented by Bremermann, according to [2], whereas in the decade, three different streams of the basic idea were developed separately. In the united states, Fogel et. al. proposed the idea of "evolutionary programming" [36][37], while Holland named his algorithm as "Genetic Algorithm (GA)" [38][39]. Simultaneously, the term of "Evolutionary Strategies" was used to refer the algorithm from Schwefel and Rechenberg in Germany. For a long time after that, these different ideas are developed independently until the last decade of the last century. Since the early 1990s, these three genres have been seen as different representatives for one technology, which has come to be a well-known field under the name of "Evolutionary Computation". Also, a new idea in this area called "Genetic Programming (GP)" was put forward by Koza [40][41] in the age of 1990s. Today, the term "Evolutionary Computing" is denoted for the whole field, meanwhile, the term "Evolutionary Algorithm" is representing the algorithms involved in [2]. Grammatical Evolution, which is the main topic of this work, can be seen as a branch of the GP since they share the same philosophy to evolve program fragment.

As it has been discussed before, the field of Evolution Computation was generally inspired by the natural process of evolution theory, so even if the existence of differences between several streams in this field, they eventually share a same basic scheme. The common underlying idea is similar to the evolution in the biological sense: A population of individuals is living in an environment with limited resources, therefore the competition for these limited resources is inevitable. This process will cause a rise in the aspect of fitness of the population over a long time. During this process, those individuals with better fitness value have more chance to pass their 'gene' to the next generation. Variation such as mutation and recombination may happen, which is the motive power of diversity, and a man-designed selection mechanism is playing the role of nature does in the real world, to evaluate and select the individuals can survive in the artificial environment. Figure 2.8 demonstrate such process in a flowchart.
As it has been showd in this flow chart, to define a complete evolutionary algorithm, a list of components is necessary to define, since EA is simulating the natural process artificially:

- Individual and Population
- Evaluation function (or fitness function)

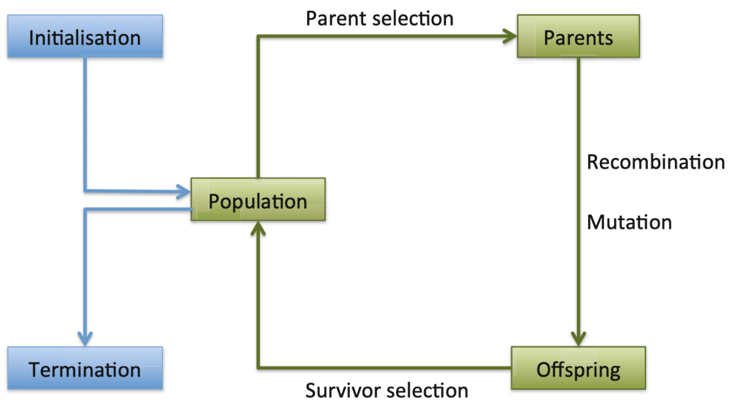


Figure 2.8: The general scheme of an evolutionary algorithm as a flowchart. Figure courtesy to Eiben et al. [2].

- Parent selection mechanism
- Variation operators (recombination and mutation)
- Replacement

2.2.2 Individual and Population

Different from the natural process, in the filed of EC, almost everything is defined by people since it is an artificial simulation of the natural world. In the field of Evolutionary Computation, the objects forming potential solution candidates within the original problem context are referred to as phenotypes, and their encoding form, which is used in the process of the evolutionary algorithm is known as genotypes. The word representation is saying the mapping from phenotypes to its corresponding genotypes. Using the simplest example to illustrate this, two different species of animal are represented by two binary code, 0 and 1. Here, these two species of animals are the phenotypes and 0 and 1 are the genotypes representing the genotypes. What needs to pay attention to is, the land space of genotypes may be very different from it of phenotypes, and EA's work mostly happens in the space of genotypes. Meanwhile, the word candidate solution, or individual are synonyms of phenotype, they usually denote the possible solution in the space of original problem. Wheares on the side of EA, people usually the term genotype,chromesome and again individual are used to represent points in genotype space,which is the EA takes place, according to [2].
All individual composed the population, who shapes the unit for evolution. A single individual is static because their chromosome won't change to adapt to the environment. But the population can hold many candidate solutions, and together with the mechanism of variation ensure the diversity of it. Therefore, the population is dynamic in the evolution, and we say the population can self-adapt to the external environment. In almost all evolutionary algorithm, the size of the population does not change during the evolution, which is also a simulation of the natural world, that is, the resources are limited and the environment can only support a number of individuals to survive. In the case of the population is larger than the environment can bear, selection would happen to keep those currently better-fitted individuals and eliminate those worst part until the size of the population is equal or smaller than the limitation. Thus, the population size is a pre-defined parameter in most of EAs.

2.2.3 Evaluation function

As the name suggests, the evaluation function is a function to evaluate how fitness an individual is, it basically defines what is good or bad for an individual in the context of original problem space by giving an indicator, which is called fitness value, as the result of this function. It is the basis of selection process since all selection are based on the fitness level of individuals. In the field of Evolutionary computation, evaluation function also has aliases like fitness function, objective function, etc. One instance for helping to understand this concept: The task is to evolve a string "*Hello World!*".Here, the string "*Hello World!*" is the target string s^*, and every phenotype for this problem is also a string with the name of 'evolved string' s. For this problem, we have an evaluation function is the edit distance between the evolved string and the target string, which is formally written by $F = Edit\_Distance(s, s^*)$. The smaller the fitness value is, the better fitted the individual is. This also indicates this problem as a minimization problem. Mathematically, the transformation between the minimization problem and maximization problem is simple by modifying the fitness function. For example in this string evolution case, it can be easily changed into a maximization problem by using the reciprocal of previous fitness value as the new fitness value, which is $F^{'} = Edit\_Distance(s, s^*)^{-1}$.

2.2.4 Variation (Crossover and Mutation)

Variation is a mechanism to generate new individual by modifying genotypes spontaneously. As a result of this, the new individual with never appeared genotypes and correspondingly new have phenotypes may be created, which probably have different fitness levels from previously had. Similar to variation in the biological world, variations are non-oriented, which means it can either created better-fitted individuals or worse. Since it is the motive power of new genotypes and phenotypes, we say it is the insurance of diversity of the population. Similar to it in the biological world, Variation operators in EC can be divided into two categories: mutation and crossover.
Crossover, or recombination, is the variation method used to merge genetic information from two parents individuals into their offspring genotype(s). The principle behind crossover is that, simulating the mating process of organics with different but desirable features to inherit these desirable features to their offsprings. Human is the most obvious example for this, the child usually has some desirable or undesirable traits from their parents as the result of recombination. The biology of this planet has proved that in a long-term, the recombination is a superior form of reproduction and can improve the characteristics of species in a long-term no matter it is sexual or asexual reproduction. The actual work of crossover in EC is generally the same with it in biology. Think of the genotype as the DNA in biology, and every codon is the basic unit for crossover. Figure 2.9 illustrates the simplest way of crossover in EC: one point crossover. Only one crossover point is randomly located and tails of its two parents are swapped to get new off-springs. Each rectangle in figure is representing a codon in genotype. Apart of this, there are several other type of crossover in EC, such like two points crossover (two crossover points) and uniform crossover (every codon is randomly seperatedly chosen to build new genotype).

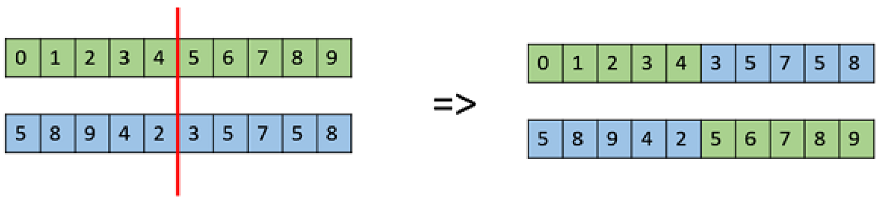


Figure 2.9: Example of One-point crossover. Figure courtesy to Tutorialspoint.com[11].

Different with it in the crossover, the mutation is the change of genotype applied on a single individual rather than two, and mutation is usually very slight on the level of genotype, even though it can cause a severe change in the aspect of phenotype. A mutation operator is usually stochastic, the mutation on any codon should be usually unoriented and unbiased, the result of mutation could be any possible value. For those changes who are oriented as it is known that can result in a better-fitted phenotype, it is improper to classify them into mutation despite the fact that they are unary operations. Mutation has several kinds of categories, Figure 2.10 demonstrate the simplest bip-flip mutation, to randomly choose a mutation position on a bit string chromosome

[11]Tutorialspoint.com. (n.d.). Genetic Algorithms Tutorial. Retrieved July 27, 2019, from `https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_crossover`

and flip the codon bit. Historically, mutation operator is responsible for various role in different EC branch, according to [2]. In genetic programming (GP), it is often not used at all, whereas in genetic algorithm (GA), it is responsible for creating new genotype fragment to enhance the diversity of gene for the whole population.

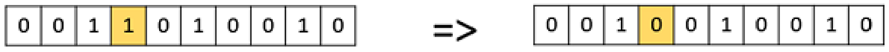


Figure 2.10: Example of bit-flip Mutation. Figure courtesy to Tutorialspoint.com[12].

2.2.5 Selection Machanism

The basic idea of selection is simple, to keep those better-fitted individuals and remove those worse ones from the population. By performing selection, the overall fitness of the whole population would gradually improve and converge to the (local) best solution in the problem context. The main reason to have the selection mechanism in EC is the limitation of population size. Of course, we can observe this in nature, in which the limitation of population size normally comes from the limited resources, as previously mentioned. Selection in EC includes parents selection and survivals selection.
Parent selection is used to choose the better individual in the sense of fitness as the parents for the next generation. However, it does not means that the better individual can always inherit their genetic information to the offsprings. In EC, the parents selection mechanism is typically probabilistic, which means those individuals with higher fitness level have a higher chance to become a parent than those have relatively low fitness values. Those less-fitted individuals still have the chance to be a parent and pass their genetic information fragment to the next generation. The main reason is that sometimes less-fitted individuals still carry genetic information fragment to build the global optimum chromosome, and on the other side, the current better-fitted individuals have the possibility to be local optimum. This probabilistic parents selection can avoid algorithm to converge to the local optimum to some extent since some chromosome fragments, which is only can be found in the less-fitted individual but crucial for the global optimum, still have the chance to stay in population.
As for the survivals selection, the general idea is to eliminate those individuals with relative lower fitness value, since the size of the whole population exceeds the limitation of it after the creation of the offsprings by selected parents. In many works, survival selection is also called replacement, which means the new-generated offsprings take over the positions of relatively less-fitted individuals in the population.

2.3 Backus–Naur form

Backus–Naur Form or Backus Normal Form (BNF) is a formal mathematical shape to describe a language. It is often used in the aspect of computer science to describe the syntax of a language, such as programming languages and communication protocols. According to the theory of Chomsky Hierarchy[42], Backus–Naur form is a kind of context-free grammar (CFG, type 2 in Chomsky hierarchy), which has a relation with pushdown automata. Theoretically, type 0 grammar is related to Turing machines can describe any computable problem, whereas type 2 can only represent a proper subset of problems which type 0 grammar can describe. However, Chomsky type 0 grammar has enormous difficulty in the aspect of either design or parse. Therefore context-free grammar is usually used to formally represent high-level programming language in computer science, even though Chomsky type 0 grammars have the advantage of expressive power[43]. Formally[44],a Backus-Naur Form grammar is consist by tuple <T,N,P,S>, where

- T is a set of terminal symbols
- N is a set of non-terminal symbols

[12]Tutorialspoint.com. (n.d.). Genetic Algorithms Tutorial. Retrieved July 27, 2019, from `https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_mutation`

- P is a set of production rules
- S is a start symbol

Terminal symbols are those symbols that cannot be further derived, whereas a series of terminals or non-terminals could replace non-terminal with the help of production rules. Also, the start symbol is the start point of derivation. The production rules are written in the following shape:

$$\texttt{<symbol>} ::= \texttt{alternative1} \; [| \; \texttt{alternative2}]... \tag{2.1}$$

For the left-hand side, the symbol is a non-terminal, and every alternative on the right-hand side of derivation rules consist of one or more sequence of non-terminal or terminals. Alternatives are separated by a vertical bar "|". The "::=" means that the symbol on the left-hand side must be derived into one alternative on the right-hand side. Any sequence of terminals which is derived by using the production rules is said to be syntactically correct, and the syntax correctness of a derived sentence can be verified by building a parse tree, which reflects the derivation process.
As an example, considering underlying BNF can be derived into a simple mathematical expression:

```
(1)<expr>      ::=  <expr><op><expr>  (A)
               |    (<expr><op><expr>)  (B)
               |    <pre-op> (<expr>)  (C)
               |    <var>  (D)

(2)<op>        ::=  +  (A)
               |    -  (B)
               |    /  (C)
               |    *  (D)

(3)<pre-op>    ::=  sin  (A)
               |    cos  (B)
               |    tan  (C)

(4)<var>       ::=  x  (A)
```

Based on this BNF, assume that mathematical expression `sin(x*x)+x` is derived. Following parse tree demonstrates how this expression is derivated.

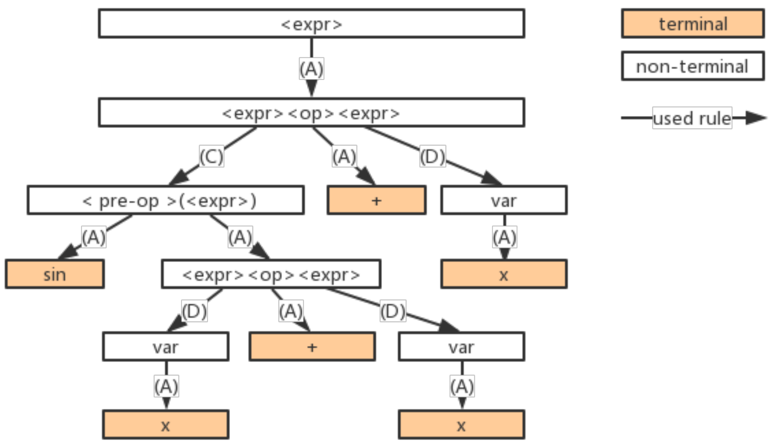


Figure 2.11: Parse tree for example BNF.

Chapter 3

Grammatical Evolution

3.1 Introduction to GE

In this chapter, Grammatical Evolution (GE)[45], which can evolve computer program in any computer language, will be disscussed in detail. Grammatical Evolution was firstly introduced by Ryan et al.[45][46], who explored a unique way of using grammars to evolve programs in the aspect of automatic programming.
To describe the working scheme in Grammatical Evolution briefly, GE uses an evolutionary algorithm to evolve variable-length binary strings, which are considered as the genome of individuals and used to represent corresponding integer string codons. At the same time, these integer codons can determine which derivation rule is going to be used to produce mathematical expression, string, or even program segment needed. Moreover, all these codons work together to form a valid solution. The details of this working mechanism of GE will be discussed later.
GE is set up so that the EA component is an independent module out of the outputted program by taking the virtue of the genotype-phenotype mapping mechanism. And the BNF, like the search algorithm, is a plug-in part of the system that in charge of the outputted language and syntax. Based on these characters, GE theoretically has the ability to evolve programs in any computer language.

3.2 GE Mechanism

3.2.1 Expression of gene in GE

Similar to other Evolutionary Algorithms, Grammatical Evolution got inspiration from the biological process in nature. It is simulating the process of production of protein from the genetic material of an organism, and protein is the fundamental material for an organism to maintain basic live operation and expression of heritable traits[47].
In the last chapter, the process of expression of genetic material was briefly introduced, every group of three consecutive nucleotides, which was called codon, are used to build protein as a group. The corresponding relation between codons and amino acid produced is called 'the genetic code'. Therefore, the sequence of these amino acids is determined by the sequence of codons on the DNA molecule. These amino acids are basic blocks to construct protein, as they are connected one by one with each other to become a protein molecule, and it can be concluded that the sequence of codons determines the production of protein. The result of the expression of gene interacting with the environment the organism lives is so-called phenotype.
The process of gene expression is similar in Grammatical Evolution, Figure 3.1 shows a comparison between the process of expression of the gene in Grammatical Evolution and the natural world.
In grammatical Evolution, a binary string with variable length was treated as a "DNA" molecule to store inheritable genetic information of an individual. Moreover, the binary string is usually a thing all individuals have. So in most cases, an individual is represented by a binary string. Usually,

[2]O'Neill, M. (Ed.). (1998, October 02). Grammatical Evolution. Retrieved May 25, 2019, from `http://www.grammatical-evolution.org/papers/gp98/node2.html`

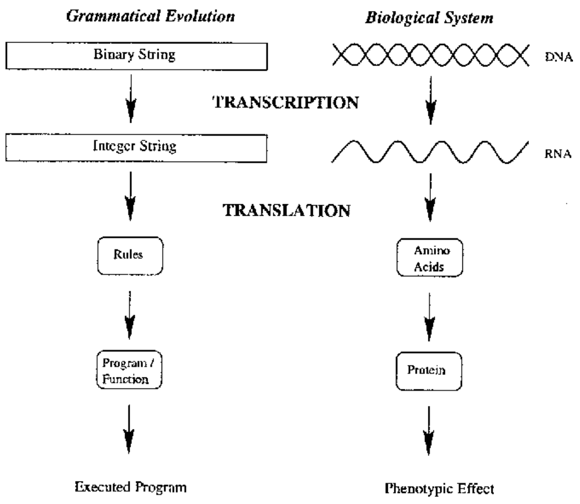


Figure 3.1: A comparison between Grammatical Evolution and natural biology. Figure courtesy to O'Neill, M.[2].

binary string is easy for a computer to read, but not that convenient for a human to understand. Before the expression, a process called transcription is necessary that the binary string will be converted to an integer string. This process is used to simulate the process of passing information from DNA to RNA before translation in the organism. The integer string is used in the mapping process, in which the string can be converted into program segments or any other terminals, with the help of the BNF file, which stores all production rules in it. Every integer on the string will perform a "mod n" calculation, on which n is the number of possible derivation rules, and the result of such calculation is the order of derivation rule. For every integer in the integer string, such process will perform once until terminals have replaced all non-terminal. It is noteworthy that in most grammatical systems (includes the original paper), this process is depth-first, even though it is possible to do this in breadth-first. As a result of this operation, the binary string will be 'translated' into a string, a mathematical expression or a program segment called phenotype. This operation is also a simulation of the process from RNA to amino acid, and finally, protein. To illustrate this process in GE better, an example from [44] gives a great insight into how it works: Considering an individual with a start symbol `<expr>`:

11011100	11001011	00110011	01111011	00000010	00101101	...

Before the mapping/translation process, convert this binary string into a integer string and we can get:

220	203	51	123	2	45	...

Here, the BNF at the end of the last chapter will still be used. For the start symbol `<expr>`, we have four derivation rules to choose from:

```
(1)<expr>  ::=  <expr><op><expr> (A)
           |    (<expr><op><expr>) (B)
           |    <pre-op> (<expr>) (C)
           |    <var> (D)
```

So, for the first codon 220, we can get the order of selected derivation rule by 220 mod 4 = 0, so rule (A) will be chosen, and the expression `<expr>` will be extended to

`<expr><op><expr>`

Then taking the next codon for the first non-terminal in the last expression. Due to that 203 mod 4 = 3, the last of four production rules will be selected to replace the first `<expr>`. And the expression becomes:

<var><op><expr>

Since <var> involves only one choice, it can be directly mapped into X and the expression becomes

X <op> <expr>

Then we can deal with the <op> in the expression. We can see that the <op> has the derivation rules of:

(2)<op>	::=	+	(A)
	|	-	(B)
	|	/	(C)
	|	*	(D)

Since 51 mod $4 = 3$, the option (D) is selected to replace <op>, and the expression will be modified to

X * < expr>

This depth-first expansion will continue until all non-terminals are replaced by terminals. And for this example we are discussing, the expression would finally become:

$$X * X$$

The whole process of this example can be summarized into Figure 3.2.

1	2	5	7		
220	203	51	123	2	45

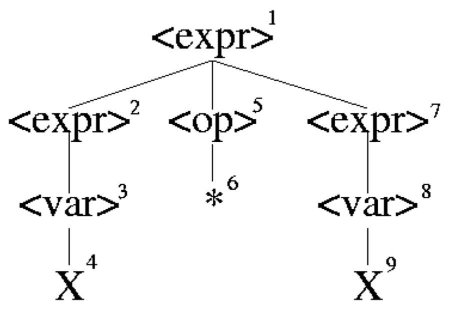


Figure 3.2: derivation tree of example. Figure courtesy to Ryan, Conor[44].

In Grammatical Evolution, it is not necessary that all codons have to be used. In this example, the last codons are not used ,and the last two codons are remained and have totally no influence for the phenotype. On the other side, if the length of the genome is not long enough to derive all non-terminal, this individual will be regarded to be invalid. A technique called 'Wrapping' can be used to relieve the influence of problem. While wrapping is used and the genome of an individual is not long enough to derive a phenotype, the codons are used in a circle, just like the genome are wrapped up. After the last codon is used, the first codons of the genome are going to be orderly used from the first codon again. In addition, it is likely some slight differences may exist between different GE variants. For example, in some GE variants or systems, when a non-terminal has only one production rule (<var> case in our example), '*mod*' operation is performed on the corresponding codon. In other words, all non-terminal are treated in the same way, regardless of the number of production rules it has.

3.2.2 Architecture of GE

Although the expression of the gene in Grammatical Evolution is novel, the global structure of Grammatical Evolution still follows some basic disciplines of Evolutionary Algorithm. To build a complete Grammatical Evolution system, every part in Problem, Grammar and Search Algorithm is indispensable. To look at this in high level, GE provides a skeleton to solve a specific problem, Grammar is an approach to describe the method to solve the problem in a formal way (maybe a program segment), and the Search Algorithm provides the methodology to get access to the final answer. Figure 3.3 shows the architecture of a Grammatical Evolution system.

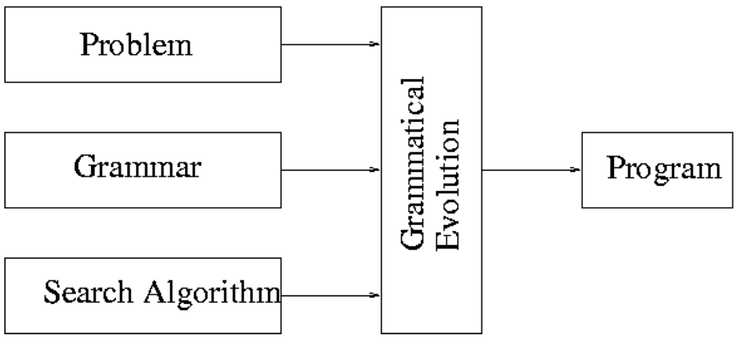


Figure 3.3: Architecture of GE. Figure courtesy to Ryan, Conor[44].

The search algorithm is a method to approach one of the local optima or global optima of the corresponding problem. It provides an effective way to continuously optimize the solution closer to the optimum value in the search space. In standard Grammatical Evolution[46][45][3], Evolutionary Algorithm (usually Genetic Algorithm) is used as the core of it. As an optimization technique, this technique starts to find the optimum at any random point, optimize it continuously, and can always find at least one local optimum. Theoretically, the Search Algorithm can be replaced with other techniques to approach the same target as GE does. For example, T. Stützle et al. use another stochastic local search heuristics called 'Irace' to replace the standard EA method in their work[48]. Grammar is the way to express a problem formally, or in other words, a mapping mechanism of a problem to map genotypes into phenotypes. In standard GE and most of GE-variants, BNF is used as the expression method of mapping mechanism. As a context-free grammar, BNF is a type 2 Grammar in Chomsky hierarchy with good expression power and relatively easy to use. Similar to the Search algorithm, grammar can be seen as a module in Grammatical Evolution and it is replaceable to change BNF to other options. Ortega A. et al. [43] developed Christiansen Grammar Evolution (CGE) by replacing context-free grammar by Christiansen grammars (which is a type 0 grammar) and it makes that GE-variant have a better ability to describe more complex problems. The 'problem' in Figure 3.3 refers to the puzzle system that is going to dealing with, for example, to build a program that can sum up an Arithmetic progression. To describe a problem in Grammatical Evolution, one fitness function is the crucial thing, which is critical to evaluate an individual.

3.2.3 Working Mechanism

With the knowledge of how the genome is expressed, the working mechanism of Grammatical Evolution can be seen more clearly. In a Grammatical Evolution system, the first step to evolve a population is always to initialize one, on which all evolutionary manipulations from Grammatical Evolution system will be performed. In GE, the initialization technique can be either similar or different with a normal evolutionary algorithm. For the first situation, the initialization is performed on the basis of the genome. On the other side, it is also possible for GE to perform the initialization on the basis of Derivation Tree. Formally, derivation tree (or phase tree) is an ordered, rooted tree representing how a context-free grammar is derived into a specific syntactic structure, just like Figure 2.11 shows. In the field of GE, researchers have invented several tree-based initialization techniques except for the simplest random derivation tree initialization. For example, Ramped Half-Half initialization (RHH) was firstly introduced by Ryan et al. [49] to initialize, and Fagan et al. [50] further developed Position Independent Grow (PI Grow) technique on the base of traditional RHH technique.

After the population has been initialized, a step of evaluation is necessary for the population. For the evaluation, due to the fact that the phenotype in GE is usually a program segment, it is necessary to make certain the judge criteria about how to evaluate the phenotype program before the system starts to run. The indicator of this is called fitness value, which represents how good the phenotype program is. The system will always be given a fitness value for every individual in the population. And the function to calculate the fitness value has the name of the evaluation function, which is pre-defined by the user or stated problem setting.
The number of iteration of evolution is called generation. In each generation, a similar operation is performed on the whole population and to generate a new population to replace the older one. This process is also an imitation of the behavior of the real organism in the natural environment. All the operations on each generation can be categorized into several parts; they are selection, crossover, mutation, evaluation, and replacement, which come from the evolutionary algorithm. They will be iteratively performed in every generation :

- **Selection** here means the parents selection, which is based on the fitness value of each individual. Those individuals with better fitness have a higher possibility to be selected to pass their genome to the next generation.
- **Crossover** and **mutation** are similar to what they are in the evolutionary algorithm since the core of GE is still an EA. Both crossover and mutation are only performed on the genotype of individuals with given crossover rate and mutation rate.
- **Evaluation** is performed for the new generation.
- **Replacement** is the final step in every generation; those individuals with worse fitness value will be replaced by those with better fitness value to keep the population size stable. This step will replace the old generation with the new population.

Algorithm 1 describe this whole process formally. Since GE is an application of evolutionary algorithm (EA), the workflow of GE is the same with other evolutionary algorithms.

Algorithm 1 Grammatical Evolution

```
Require: $Termination\_condition, Mapping\_scheme$
  BEGIN
  $n \Leftarrow 0$
  INITIALIZATION of the first population
  EVALUATE the first population
  while $Termination\_condition$ NOT satisfied do
    $n \Leftarrow n + 1$
    PARENTS SELECTION
    CROSSOVER population
    MUTATE population
    EVALUATE population
    REPLACEMENT
  end while
  END
```

3.3 Discussion of GE

Grammatical Evolution is such a method to theoretically solve almost any kinds of problem in the way of optimization consistently if the definition of the problem is precise and adequate. But as a matter of fact, still, many hindrances are placed on the way towards that possibility. The limit of computation power causes some of the hindrances, and some of other hindrances are the result of the structure of GE itself. In this section, the main problem GE is currently facing will be discussed, and some immature personal ideas are declared as well.

The first hindrance, which is also the least important one for GE is the limitation of computation power. As it has been declared in the previous part of this chapter, Grammatical Evolution still uses EA as its core to evolve its population. Due to the design of Evolutionary Algorithm, it always needs to maintain a relatively significant population to keep those 'potential' gene fragments for the global optimum or even local optimum, which may be dispersed in many different individuals throughout the entire evolution process. This design demands more computation resources for sure, if we compare this to those strongly oriented searching methods. However, it is also the essences of EA as well as nature, that the composed of several simple parts can sometimes produce surprising results.

The Grammar file, which is used to indicate how genotype is mapped into phenotype in grammatical evolution, also has a significant influence on the performance of GE. Different from our intuition, the grammar file is not merely an external file for grammatical evolution system. It plays one of the essential roles in the whole process of evolution. Figure 3.4 illustrates the mechanism of grammatical evolution from another perspective. Mapping, search mechanism, and evaluating mechanism include almost all manipulation we have to solve our problem. Among these, the grammar file defines every rule of mapping process wheres the design of GE algorithm controls the mapping mechanism. Any tiny modification in the grammar file can cause a considerable difference in the GE process. However, in the field of GE, such vital files have to be purely written by people. This lead to the fact that the great performance of Grammatical Evolution has great reliance on an expert-written and well-designed grammar file for most problems. This problem is not severe in some widely-used test problems since many different grammars have been tested in the community for millions of times and many researchers have done much work for these, and these grammar files are acceptable for these problems. But in the more general case, especially for those applications or user-customized problems from non-expert users, the grammar file from them may become the roadblock toward the better performance of GE. Dirk Schweim et al. [51] studied the structure of grammar for GE and advised the average branching factor, which is the expected number of non-terminals chosen in mapping one genotype codon to a phenotype tree code, should be as close to 1 as possible to help with the efficiency of GE. However, it is still uneasy about writing a proper and efficient grammar for a specific problem, since the average branching factor is the only evaluation of an existing grammar .

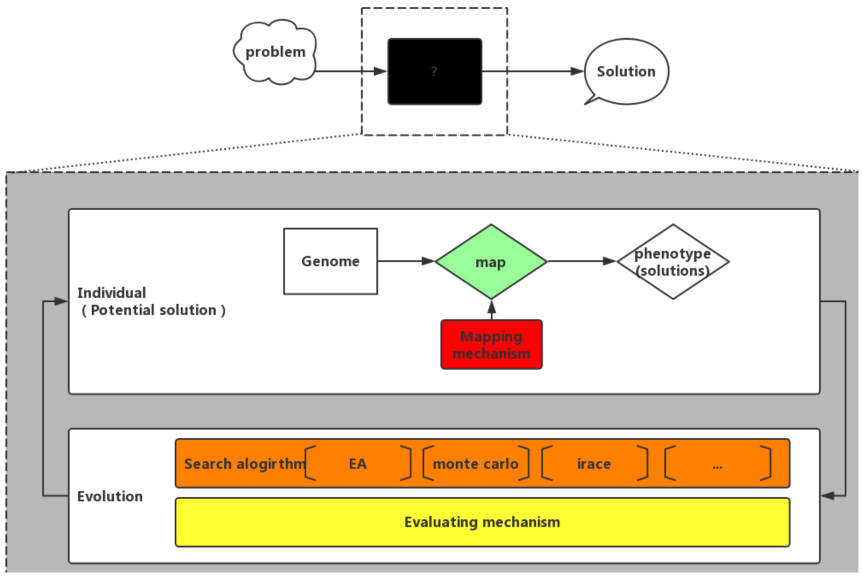


Figure 3.4: Another perspective of Grammatical Evolution

Meanwhile, the design of GE itself also brought some problems. In canonical grammatical evolution, the selection of non-terminal is revealed by doing a *mod* calculation over the codons. Since the number of available derivation rules is usually small, it is easy to get the same result in this calculation, even if the codons are different. This mapping mechanism of grammatical evolution implies an N to 1 relationship between the genotype and phenotypes. That is, every phenotype has a large number of corresponding genotypes. In theory, a phenotype in solution space is accessible to be found in the case that it has at least one correspondent point in the search space. The N to 1 relationship between genotypes and phenotypes in GE is highly redundant, as it can have a number of points in search space, it actually needs to locate the global optimum or local optimum we are searching for. This character may sometimes increase the possibility for GE to get the optimum point. However, it also decreases the efficiency of the search process, since many candidates genotypes tested in the search process may finally point to one same phenotype, and the evaluation of phenotype may be computation costly in some problems.

On the other side, this mapping mechanism also has the problem of low locality. The derivation of genotype to phenotype is a repeated nesting loop since the selection of one derivation rules can influence the later derivation. This derivation way may causes a phenomenon different from the expression of the gene in the natural world. Because in GE, a neighborhood genotype may have a phenotype with no similarity. This character diverses from our intuition that neighboring genotypes should usually correspond to neighboring phenotypes. In other words, if we visualize the landscape of fitness of all possible points in search space, what we get is a rugged space full of ravines and spikes. The term of low locality is used to represent this character in the community of GE. It can cause the search process much harder since the direction of evolution is hard to find for either local or global optimums.

In fact, many variants of GE have tried to solve the problem of high redundancy and low locality. For example, the SGE system, which was tested in this project, is a great example that made efforts in this regard. However, in theory, a GE variant algorithm with lower redundancy and higher locality does not mean the change of search space. These two characters are only helpful for the GE system to improve its efficiency in the searching process. This leads to the fact that an 'advanced' GE variant system cannot make sure that it always performs better than a canonical GE system since they are usually dancing on the same stage.

3.4 GE systems

3.4.1 PonyGE2

PonyGE2[52] is a python implemented Grammatical Evolution system, which is developed by UCD's Natural Computing Research and Application Group. PonyGE2 provides a modular-based implementation of GE, which is the most significant advantage when it is compared to other GE implementations and also its author's first python-based GE- implementation PonyGE[53], which allows users to modify almost every part in Grammatical Evolution. These merits lead to the fact that PonyGE2 may be a "rapid-prototyping medium for any python workout", as it said in their work.

Before the publication of PonyGE2, Grammatical Evolution has been implemented in many computer languages, including C[54], java[55], R[56], and even Ruby[57]. Due to some historical reasons, most of the previous work cannot reach a good balance among functional integrity,good acceptance of implemented language, cleanness and compactness of code as well as structure. To address these problems, PonyGE2 merged the characters of modular design and feature-rich aspect from GEVA (java implemented GE system)[55] on the basis of PonyGE, and reconstructed all codes into a package structure[52]. With their efforts, a new python-implemented GE system with much newly added functionality package is born, which is PonyGE2.The code structure of PonyGE2 can be seen in Figure 3.5. This clear structure allows users to understand what every modular is doing and to work on a specific part of this system without reviewing dozens of irrelevant codes. Meanwhile, with such a structure, users have great flexibility to modify any modular of original work, for example, to use a user-written fitness function library to replace the original one.

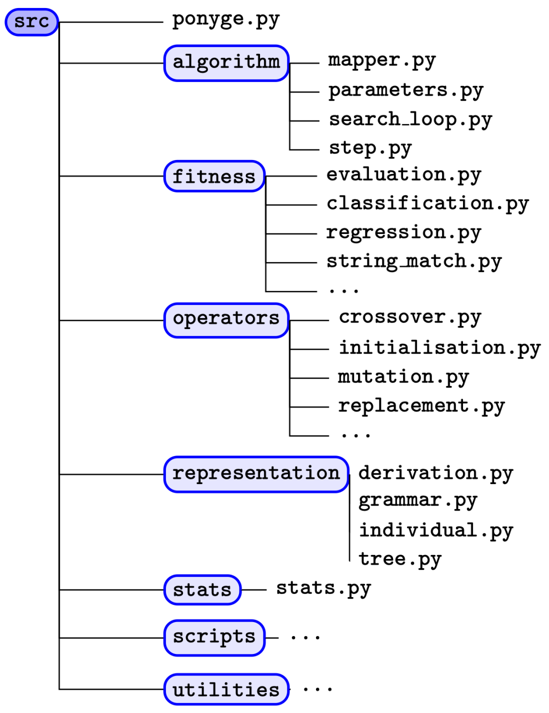


Figure 3.5: Organizational structure of the PonyGE2 Codebase. Figure courtesy to Fenton et al. [52].

PonyGE2 can be referred to as a canonical GE system, as it can be set to follow all basic disciplines of standard Grammatical Evolution [46][45][3], even it has great flexibility to do other experimental GE-variants experiments. The general workflow in PonyGE2 for typical can be seen in Figure 3.6, which is also the default setting of this PonyGE2 system.

One of the most significant advantages of Grammatical Evolution is that PonyGE2 integrates the ability to mix and matches representation tree, which means not only a genome is kept in the evolving process, but also a full derivation tree which corresponds to the genome. This character allows more operators to perform in PonyGE2, as some operators can only be used in the derivation tree of individuals. With the potential to be one of the most powerful systems in the aspect of Grammatical Evolution, PonyGE2 has a significant number of options for users to choose in its parameter list. In almost every step of Evolution, several options are available. These options are also the guarantee of the high flexibility of this GE system to some extent.

Most of the available options in PonyGE2 are represented in an abbreviated form for convenient considerations, for example, when users need to use Position Independent Grow Initialization, it is necessary to feed the configuration with command parameter *'– initialization PI_grow'* or modifying the parameters file. Meanwhile, some of these options still have sub-parameters, which are only useful when a specific option is selected. Using the previous example to demonstrate, when *'PI_grow'* initialization method is used, the minimum and maximum initialization depth are activated to control the initialization process in GE. In the case of no other value specified by the user, the system will automatically use its default value. Table 3.1 depict the relation of most configurable options and their sub-parameters, which is summarized from the Wiki document of PonyGE2 project. Since it is redundant to explain all parameters of PonyGE2, this part can be easily found on their project Wiki. In the matter of fact, much more parameters are defined in PonyGE2 system, but no detailed explanation and description are specified on their either published paper [52] and their website document[2]. In the actual test of this project, some of these parameters are disabled for safety reason, as some specific parameter can easily cause the error of project. For this part, a detailed description can be found in chapter 4.

[2]https://github.com/PonyGE/PonyGE2/wiki

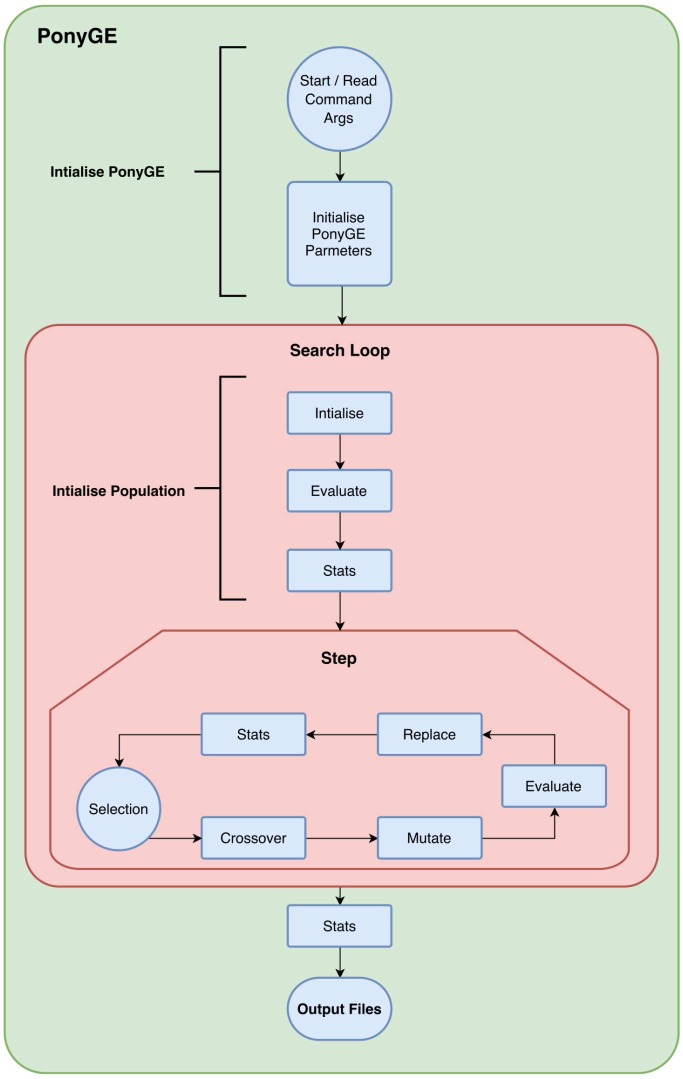


Figure 3.6: PonyGE2 control flow diagram for typical GE/GP setup. Figure courtesy to Fenton et al. [52].

Parameter Name	Options	Related sub-parameter	sub-parameter range
initialization	uniform_genome	init_genome_length	INT
	uniform_tree	max_init_tree_depth	INT
	rhh		
	PI_grow	min_init_tree_depth	
selection	tournament	tournament_size	INT
	nsga2_selection		
	truncation	selection_proportion	[0,1]
crossover	tournament fixed_twopoint	-	-
	variable_onepoint		
	variable_twopoint		
	subtree		
mutation	int_flip_per_codon	mutation_probability	[0,1]
	int_flip_per_ind	mutation_events	[INT]
	subtree		
replacement	generational	elite_size	integer in [1,100]
	steady_state	-	-
	nsga2_replacement	-	-

Table 3.1: (Partial) Parameters list of PonyGE2

On the other side, every option is implemented in the way of a submodule, so users can write their own option/extension module to become a part of PonyGE2 system.

3.4.2 Structured Grammatical Evolution (SGE)

Structured Grammatical Evolution is a recent variant of canonical grammatical evolution, which was firstly published in the work [58] by Lourenço et al. from the University of Coimbra. One point to note is that, since they have also named their system as Structured Grammatical Evolution (SGE), the word SGE can represent the algorithm as well as the corresponding system. In this section, the main difference between SGE and canonical GE will be introduced, and meanwhile, some important information about the corresponded SGE system will be delivered.

As it has been introduced before in section 3.2, GE uses a context-free grammar to realize the target of mapping genotype into the phenotype. Due to the mechanism of it works, one of the problems it comes with is the problem of high redundancy and low locality, which could be potentially harmful to the efficiency of GE [59][60]. SGE is proposed to relieve the issues of locality and redundancy of canonical grammatical evolution by replacing the context-free grammar to a structured mapping method. Different from the situation in GE, a one-to-one mapping mechanism between the genotype and the non-terminals are used in the SGE. To archive this target, a pre-processing procedure is required. By such a procedure, the standard context-free grammar can be translated into SGE-used grammar. In SGE, every genotype is represented by several sets of integers, rather than a long integer string in canonical grammatical evolution. Here, one example is used to demonstrate this difference of representing method between SGE and canonical grammatical evolution. Considering we have following context-free grammar:

`<start>`	::=	`<expr><op><expr>` (0)
	|	`<expr>` (1)
`<expr>`	::=	`<term><op><term>` (0)
	|	`(<term><op><term>)` (1)
<op>	::=	`+` (0)
	|	`-` (1)
	|	`/` (2)
	|	`*` (3)
<term>	::=	x_1 (0)
	|	`1` (1)

One individual with the genotype of $[27, 7, 55, 22, 3, 4, 30, 16, 203, 24]$ can be finally be derived into the phenotype $(1/1) + x_1 * x_1$. And it is obvious that this phenotype can have many potential genotype because of the working mechianism of GE, such like $[7, 7, 55, 22, 3, 4, 30, 16, 203, 24]$.

However, in SGE, one phenotype can only have one genotype. For this case, the genotype in SGE is written in [[0][1,0][2,0,3][1,1,0,0]]. Each bracket here is representing one non-terminal in order. In the first bracket we have only a 0, it means that for the first non-terminal, rule (0) of first non-terminal (`<start>`) will be used for derivation from the `<start>` to `<expr><op><expr>`. And for the second bracket, we have two value 1 and 0, which means rule (0) and rule (1) of second non-terminal (`<expr>`) will be used for derivation respectively to `(<term><op><term>)<op><term><op><term>`. This process continues until the translation for all four non-terminals end. Figure 3.7 demonstrate this process in a more intuitive way. Due to the reason that codons controlling different kinds of non-terminals are separated, even they are still mapping in a depth-first way, there is no different to map all codons belongs to the same non-terminals into terminals according to the order of non-terminals at once, just as a layer structure does.
This new mapping mechanism is the main difference between GE and SGE, which has also brought several characteristics as results of that, which are different from canonical GE:

- Because of the way SGE deal with grammar, no recursion in Grammar is permitted for the

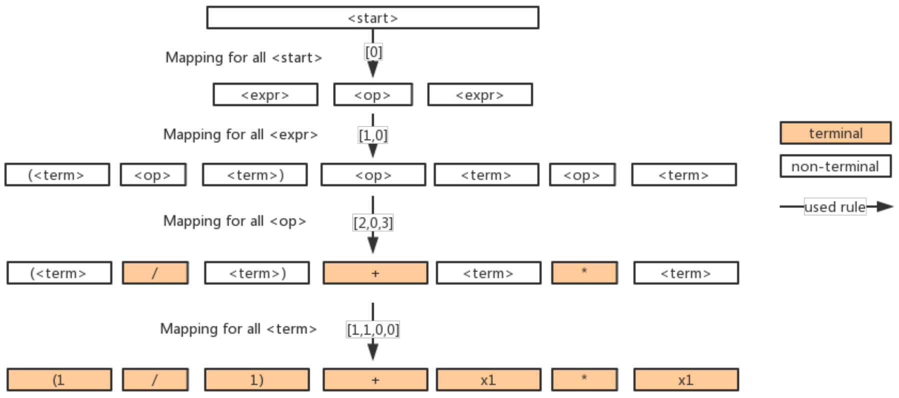


Figure 3.7: Example of mapping process in SGE

grammar file for SGE. Pre-processing is mandatory to translate standard context-free grammar to SGE-used grammar. The maximum recursion level must be pre-defined for transferring a context-free grammar to a grammar without any recursion, to limit the genotype size.

- All integers in genotype are bounded by the number of possible options of the corresponding non-terminals, as a result of every integer is representing. A derivation rule to use. However, in Grammatical Evolution, integers in genotype could theoretically be any natural number, since it uses a 'mod' calculation to choose which derivation rule to use. The SGE's structure ensures that one variation on one codon would not affect the derivation of other non-terminals, and this characteristic lead to the high locality in theory.
- In SGE, the relation between genotype and phenotype is always one to one, since every codon is directly referring to a derivation rules, whereas in GE, the relation between genotype and phenotype is usually N to 1. This design reduces the redundancy of canonical GE and does the search for optimum more efficient in theory.
- Since the shape of genotype in SGE is restricted to a set of list with the sizes of occurrence number of corresponding non-terminals, the variation operation in the evolution process has less diversity than canonical GE. For example, the crossover in SGE can only be performed on candidates with the same structure of genotype. Some advanced operation technique for GE (e.g., derivation-tree based crossover technique) is not permitted in SGE.

SGE system[3] is implemented in python2 by the same team who designed the Structured Grammatical Evolution. It is relatively a smaller system when it is compared with the previously mentioned PonyGE2 system. This system does not have so many options to choose as PonyGE2 does, only basic evolutionary configuration and another unique parameter for SGE, which is the maximal recursion level for grammar. All configurable options for SGE and explanation are listed in the following list:

1. POPULATION_SIZE: The size of the maximal population in each generation. In the case of the population still have a vacancy for the candidate, a new individual will be created by the existing population until the population size reaches the limitation.
2. ELITISM: The number of candidates kept at the end of each generation, which means these candidates can survive from one generation to the next generation.

[3]`https://github.com/nunolourenco/sge`

3. TOURNAMENT: The parameter "TOURNAMENT" is used for selecting a parent individual for generating new individuals. In this process, a "TOURNAMENT" size of candidate parents will be randomly selected from the population to build a pool, and the individual with the best fitness value will be chosen as the parent to generate new individual .together with another parent. Each parent is selected independently.
4. PROB_CROSSOVER: The probability of crossover in generating a new individual.
5. PROB_MUTATION: The probability of mutating at each mutable single codon. Every mutation independently happens.
6. MAX_REC_LEVEL: The maximal recursion level to transfer context-free grammar to no-recursion grammar in SGE.

3.4.3 Grammar-Guided Evolutionary Search (GGES)

Grammar-Guided Evolutionary Search (GGES) is a system used to support the experimental work of [61] to examine the performance of canonical GE and context-free grammar genetic programming (CFG-GP). And in their later work, the system also has included an implementation of Structured Grammatical Evolution (SGE), which is the method mentioned in the previous section 3.4.2.

According to the introduction of CFG-GP in the work [62], both GP and CFG-GP are a tree-based technique with a little difference in the way of representation. Here, a small example (6-bit multiplexer) is used to demonstrate the difference between GP and CFG-GP. Table 3.2 is the representation of example problem, and the number after GP non-terminals is the number of necessary components number of corresponded non-terminals to achieve so-called "closure." Only if "closure" is achieved, the produced function will not cause errors. On the other side, Table 3.3 is the grammar file of the example problem in the shape of BNF. Figure 3.8 shows the process that GP and CFG-GP produce a same binary function: `(a0 or a1) and (not d0)`.

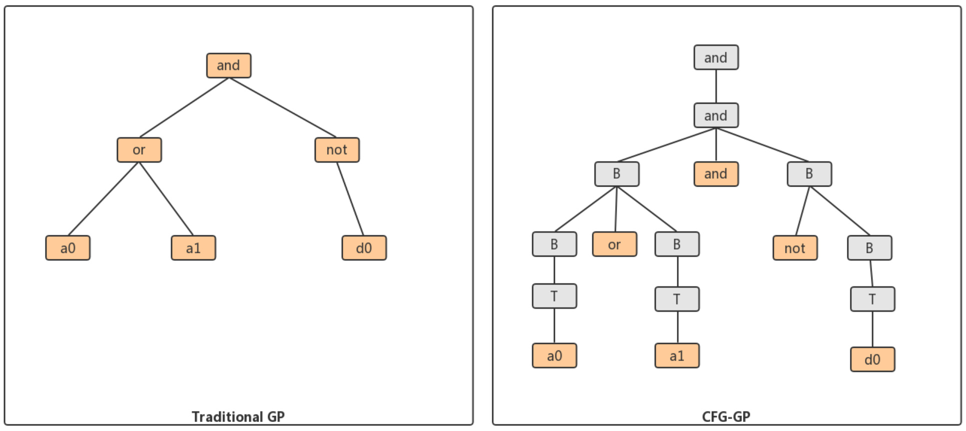


Figure 3.8: The difference between GP and CFG-GP. One same program produced by GP and CFG-GP respectively.

From this example we can get some feeling about that even though GP and CFG-GP are both tree-based technique, the grammar of CFG-GP implicit contains the grammar constraint for produced program since all production must follow the rules defined in the grammar files, whereas in GP the concept of closure is necessary to avoid invalid production of programs which is result from the

GP terminals	`a0,a1,d0,d1,d2,d3`
GP non-terminals	`and(2), if(3), or(2), not(1)`

Table 3.2: GP representation of a 6-multiplexer problem

```
<start>  ::=  <b>
<b>      ::=  <B> and <B> | <B> or <B> | not <B> | if <B> <B> <B> | <T>
<T>      ::=  a0 | a1 | d0 | d1 | d2 | d3
```

Table 3.3: CFG-GP representation of a 6-multiplexer problem

mixed usage of component with different roles. On other words, the CFG-GP uses grammar solved the problem of a valid structure (representing a valid phenotype) in program production, which is achieved by importing a rule of "closure" in GP. As for the system of GGES, the current version of GGES system includes totally three different techniques, includes CFG-GP, GE (standard) and SGE. One point worth to mention is, the SGE inside the GGES cannot deal with grammar file with any recursive rules, whereas in original SGE system, users can define a maximal recursion depth to transfer an ordinary grammar file into a grammar file automatically can be accepted by the core of SGE. This makes most grammar files we are using cannot be accepted by the SGE module in GGES system.

The control of hyper-parameters of GGES system is accomplished by using a configuration file. The list of hyper-parameters and their default values in GGES system can be found in Table 3.4.

Name	**Range and (Default Value)**		
Representation	***CFG-GP***	***GE***	***SGE***
Population Size	INT,(1000)		
Generations	INT,(50)		
Tournament Size	INT,(3)		
Elitism Count	INT,(1)		
Crossover Rate	[0,1],(0.9)		
Mutation Rate	[0,1],(0.05)		
Max. Initialisation Depth	INT,(2)	-	
Min. Initialisation Depth	INT,(6)	-	
Max. Depth	INT,(17)	-	
Wrapping	-	Boolean, (False)	

Table 3.4: List of hyper-parameters in GGES system.

3.5 Hyper-parameter Tuning

Even though most of the Grammatical Evolution systems have the ability to evolve executable computer program fragments or mathematical expression automatically, some parameters are still necessary to control the process of the evolution process. As these parameters are on a high-level aspect, they are usually called hyper-parameters to distinguish from parameters in low-level aspects, such as those parameters in evolved programs. These hyper-parameters usually has a great influence on the performance of the Grammatical Evolution system. For example, when PonyGE2 system is being tested, several different hyper-parameter sets as follows are tested on the same problem[4]:

[4]Only a part of hyper-parameters of PonyGE2 system are selected for this test here, all other hyper-parameters are not mentioned remains as the default value of PonyGE2 system.

Problem:	StringMatch Problem (symbolic regression)
Target:	Good Morning
Fitness call limit:	10000
System:	PonyGE2

Group	init tree depth	max tree depth	crossover rate	tournament size	Best in 5 runs
1	13	19	0.75	8	**0.500**
2	13	19	0.75	**5**	**1.486**
3	**11**	**15**	0.75	8	**1.976**
4	13	19	**0.70**	8	**0.670**

Table 3.5: Several Hyper-parameters groups tested on same problem showed great influence on the performance of system.

As the result shows in 3.5, a big difference can be caused by different groups of hyper-parameters despite only small modification between them. In this test, we modify only one hyper-parameter from the default value (group1) for each group, but the performance of the final result from different groups of parameters expresses an absolute relative error of 295.2%. This result implies the fact that, the value of hyperparameters on the Grammatical Evolution system.

The setting of Hyper-parameters is usually done manually based on the experience of users, which is usually uncontrollable for the performance and sometimes time-wasting to search for a suitable hyper-parameter setting for one specific problem. Due to this reason, an automatic hyper-parameter tuning method is necessary to help tested grammatical evolution system to perform as good as possible. Here we see this problem as an optimization problem, in order to find the most suitable hyper-parameter setting for GE systems and their testing problems. In this project, an algorithm called Mixed Integer Parallel Efficient Global Optimization1(MIP-EGO)[1] is used for hyper-parameter tuning over Grammatical Evolution system. MIP-EGO is a global search strategy that is designed for black-box functions, according to this work from Dr. H. Wang from Leiden University, which is also a supervisor of this work.

For this complex optimization problem, considering the search space of hyper-parameter is represented by $\mathcal{C}$, and what we are looking for is one best-performed hyper-parameter setting $c^* \in \mathcal{C}$. To solve this, a statistical model is constructed on the basis of several randomly selected starting points on the search space of hyper-parameters. The EGO algorithm relies on a so-called meta-model, which will construct n sample points in the space of hyper-parameter space: $X = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$. For the case of MIP-EGO, the random forest is selected to be this meta-model. The sampling method is based on the Latin hypercube sampling (LHS) [63]. And simultaneously, the performance of tested system by sample configurations is denoted by $Y = \{F(\mathbf{x}_1), F(\mathbf{x}_2), \ldots, F(\mathbf{x}_n)\}$. This performance is represented by a metric value, which is obtained by averaging the result of testing the GE system on one specific problem with given hyper-parameter configuration x_n over several times. In this project, this metric is the average value over 5 tests on a given configuration. After acquiring all data pairs of X and Y, meta-modeling will product a predictor $\hat{F}$ of the performance metric, and a metric s^2, which is used to measure the uncertainty of the prediction(e.g., mean square error). Meanwhile, other unobserved configuration will be quantified by the so-called Moment-GeneratingFunction of Improvement (MGFI) [6, 64], which has the ability to balance the trade-off between prediction and uncertainty from observed hyper-parameter configurations, defined as $\mathcal{M}(\mathbf{x}; \hat{F}, s^2, t) = \Phi\left(\frac{F_{\min} - \hat{F}'}{s}\right) \exp\left((F_{\min} - \hat{F} - 1)t + \frac{s^2 t^2}{2}\right), \hat{F}' = \hat{F} - s^2 t$. Formally, this algorithm can be described in Algorithm 2.

Algorithm 2 Mixed Integer Parallel EGO for Hyper-parameters Optimisation (MIP-EGO)

```
Target: MIP-EGO($\mathcal{C}, F, q, t_0$)
  sample the initial data set $(X, Y)$
  evaluate $Y \leftarrow \{F(\mathbf{x}_1), F(\mathbf{x}_2), \ldots, F(\mathbf{x}_n)\}$
  train random forest: $\hat{F}, \hat{s}^2 \leftarrow (X, Y)$
  while stopping criteria are not fulfilled do
    for $i = 1 \rightarrow q$ do
      $t \leftarrow t_0 \exp(\mathcal{N}(0, 1))$
      $\mathbf{x}' \leftarrow \arg\max_{\mathbf{x} \in \mathcal{C}} \mathcal{M}(\mathbf{x}; \hat{F}, \hat{s}^2, t)$
      compute $y' \leftarrow F(\mathbf{x}')$
    end for
    $X \leftarrow X \cup \{\mathbf{x}'\}$
    $Y \leftarrow Y \cup \{y'\}$
    re-training: $\hat{F}, \hat{s}^2 \leftarrow (X, Y)$
  end while
```

Chapter 4

Method

4.1 Benchmark

When different Grammatical Evolution systems are compared, what is going to be done usually is to test these systems on many problems and compare their performance on that. Most of GE and variants systems are using similar principles to evolve solutions for specific problems, and these GE systems usually ask for some exogenous information which is connected with the problem it deals with. This exogenous information that GE systems asking for includes an evaluation function, a grammar file, and sometimes a training/testing dataset in the cases of it is a supervised learning problem. For example, when the PonyGE2 system is used to evolve a target string, a grammar file and the evaluation function are prerequisites for the system. Moreover, since we are meant to evolve a specific string, the training data is the target string itself.

In order to make a comparison between different GE systems, a benchmark is quite essential to evaluate the performance of systems, and also make sure that every tested system is sharing the same information for every specific problem. The benchmark for GE typically composed of many independent problems, and the performances of tested systems were typically represented by the fitness level of tested systems on all benchmark problems. Each problem in the benchmark should have its grammar file, an evaluation function(or fitness function), and training data or dataset if the problem is a supervised learning problem.

One problem we are facing with is, which problem could be a candidate of benchmark problems. In the work of [5], James McDermott et al. suggested that problems which are mostly used in testing Genetic programming systems mainly come from historical reasons. Lourenço et al. imply that a similar situation is also for the field of Grammatical Evolution in their work [60] too. Considering the fact that on the one hand, most of the work in the field of Grammatical Evolution is still using some problems for historical or empirical reasons. And on the other hand, researchers in the community tend to add their own problems to test. In this work, we choose some most-widely used problems in other works in this field as benchmark problems, and also leaves a method to add new problems into the benchmark. In such a way, we believe the benchmark can have both great expansibility and historical continuity to previous works in this community.

4.1.1 Default Benchmark problems

In this work, several problems were widely used in the filed of Grammatical Evolution and Genetic programming are considered to become benchmark problems. The selection of benchmark problems also refers some idea from [5], since GP and GE are neighbor fields and many problems are shared in these two fields. One point worth to mentioned is, all problems in this benchmark are considered to be minimization problems since we believe it helps to simplify configurations over different systems.

1. **String Match problem**

String Match problem can be seen as an instance of symbolic regression problems. The target of string match problem is a purely a string. As a simple but classic symbolic regression problem, this String Match problem is collected from the demo problems of PonyGE2 system, with some modification on the calculation of fitness. With a pre-defined target string s^*, GE system is asked to evolve a string s. In the case of perfect evolution, there should be no difference between s and s^*, which could be expressed as $s = s^*$.

The BNF file for this problem can generate both vowel letters and consonant letters in Uppercase and lowercase. On the same time, several simple characters are included in the BNF file, such as question mark, Exclamation mark, etc. The specific definition of the BNF file for this problem can be found in Appendix A.

The object function of String Match problem here is using the edit distance, which is different from it in PonyGE2's demo problem. In compare with the method used in the original problem in PonyGE2 (the distance of string in ASCII code), edit distance (also known as Levenshtein distance) is a more intuitive method to evaluate the distance of two strings. Therefore, the objective function of this problem is defined as the minimal edit distance between the target string and candidate string. In the case of a perfect evolved candidate, the fitness will become zero since the candidate is exactly the same with the target string. On the other side, For the worst case, the fitness value can be the length of the target string L. Formally, the objective function (or fitness function) can be defined by:

$$f(s) = LevensteinDistance(s, s^*) \to \min, \tag{4.1}$$

2. **Regression problem (Vladislavleva4)** Regression problem is also a kind of symbolic regression problem, but with a target of a mathematical expression. For the case of Vladislavleva4[5], the target mathematical expression is a 5-variable real-value function:

$$g^*(x) = \frac{10}{5 + (x_0 - 3)^2 + (x_1 - 3)^2 + (x_2 - 3)^2 + (x_3 - 3)^2 + (x_4 - 3)^2} \tag{4.2}$$

In order to evolve this mathematical expression, several primary mathematical characters and variables are included in its BNF file, e.g., $+, -, *, /, sin, tan$. In the meanwhile, a limitation for evolved constant also exists, which only allow generating constant between 0 and 100 with maximally two digits. Detailed definition of this BNF can be found in Appendix A. On Vladislavleva4 problem, the quality of the candidate mathematical expression g is represented by the RMSE(root mean squared error) value to the target formula g^* on a data set, which is consist of 5000 points $\{\mathbf{x}_1, \ldots, \mathbf{x}_{5000}\}$. The objective function can be written as:

$$f(g) = \sqrt{\frac{1}{5000} \sum_{i=1}^{5000} (g^*(\mathbf{x}_i) - g(\mathbf{x}_i))^2} \to \min. \tag{4.3}$$

3. **Regression problem (Keijzer6)** Keijzer6 is also known as Harmonic curve regression problem. Similar with previous mentioned Vladislavleva4 problem, Keijzer6 problem [65] is also a symbolic regression problem with a target of one variable function. A smaller variable number also lead to the fact that the search space for Keijzer6 is smaller than Vladislavleva4. The BNF of Keijzer6 problem shares similar non-terminal with other symbolic regression with the target of a mathematical expression, and the detailed bnf file can be found in Appendix A. The target fucntion of Keijzer6 problem is

$$g^*(x) = \sum_{i=1}^{x} \frac{1}{i} \tag{4.4}$$

As a supervised problem, every candidate is trained on a train set with a size of 50 and tested on a test set, which includes 120 samples. And the fitness level of this problem is represented by the root mean squared error on the base of the test set.

$$f_3(g) = \sqrt{\frac{1}{120} \sum_{i=1}^{120} (g^*(\mathbf{x}_i) - g(\mathbf{x}_i))^2} \to \min. \tag{4.5}$$

4. **Regression problem (Pagie polynominal)** As a regression problem, the Pagie problem has a reputation for being a relatively hard problem[66][5] even it has only two variables and a smooth searching space. The target function to approximate for this problem is:

$$g^*(x,y) = \frac{1}{1+x^{-4}} + \frac{1}{1+y^{-4}} \tag{4.6}$$

   The training set for this function is sampled on the interval of $[-5, 5]$ with a step $s = 0.4$ for both variables respectively, who has a total of 676 (26*26) sample points. The production rules allow GE system to use eight kinds of operations, which can be found in Appendix A. Assuming the evolved function candidate as g, the fitness value f of this problem is defined by the RMSE of g on the test set T, which is sampled by the target function g^* on the interval of $[-5, 5]$ with the step $s = 0.1$, with a total of 10000 (100*100) instances. Formally written, the objective function for the Pagie problem is:

$$f(g) = \sqrt{\frac{\sum_{i=1}^{10000}(g(x_i,y_i) - g^*(x_i,y_i))^2}{10000}} \to \min. \tag{4.7}$$

5. **Classification problem (Banknote)** The classification problem is a kind of problem to identify which category one element belongs based on a group of training data, which contains some observations of characteristics from know categories. Here, the banknote problem is included in this benchmark. The Banknote problem is to evolving a formula to identify whether a banknote is fake or not based on four numerical indicators extracted from its image of that banknote. The GE system is necessary to evolve a decision function $g\colon \mathbb{R}^4 \to \{1, -1\}$ to assign an indicator (1 represents real, and -1 means fake) to classify its authenticity. As a supervised learning problem, the Banknote problem owns a training data set with 372 instances, and all evolved individuals will be tested on a test set with 1000 instances.

   The BNF file of Banknote problem allows to use only six mathematical operators(+,-,*,/, square root and logarithm), detailed deviation rules can be found in Appendix A. Also, the fitness value of this problem is defined as the F1 score of the results of classification on the test set, which is formally written in:

$$f = 1 - F1(g) = 1 - \frac{2 \times \text{precision} \times \text{recall}}{\text{precision} + \text{recall}} \to \min, \tag{4.8}$$

6. **Predictive problem (Boston Housing)** The target of Boston housing problem is a housing price prediction model, which is built on a data set composed by the real housing prices of Boston area and their corresponding demographic data. This problem is collected from UCI repository [67]. This problem can also be seen as a supervised, regression problem without a known best answer. Similar to the regression problem, the GE system needs to evolve a formula g to predict housing prices on the basis of 13 variables in the data set. The size of the training set is 354, and, 152 instances are used for testing the evolved candidates. Apart from 13 variables, six kinds of basic mathematical operations $(+, -, *, /, log, sqrt)$ and eight different constants are allowed to be used in evolving process. The detailed description is listed in its BNF file A. The fitness level is represented by the RMSE on the tested set, officially written in:

$$f(g) = \sqrt{\frac{1}{152}\sum_{i=1}^{152}\left(g^*(\mathbf{x}_i) - g(\mathbf{x}_i)\right)^2} \to \min. \tag{4.9}$$

7. **Constructed problem (Max[py])** Max problem is an instance of constructed problems, according to the classification in [5]. The target of this problem is quite simple, to produce a program fragment which can produce a return value as big as possible with several limited statements in computer programming language and constants. Considering the candidate here is an executable computer program fragment and the way we evaluate it is to run it, we choose python as the program language for this problem due to its character of it being an

interpreted language, and no compilation is necessary for the execution. The grammar rules for this problem can be found in Appendix A, which is modified based on Pymax problem from PonyGE2 system to avoid some loop-caused rules since it cannot be dealt with by some GE systems.

The evaluation of this problem is direct and straightforward. Since the target of this problem is to make the produced number as large as possible, the reciprocal of the produced number will be used as the fitness value and make the problem into a minimization problem. Moreover, we also multiply a constant 1000000 to make the fitness value more intuitive for users. Formally written, the candidate program will be noted by character P, and the fitness of this problem can be calculated by:

$$f = 1000000 * \frac{1}{\text{output}(P)} \to \min . \tag{4.10}$$

8. **5-Parity problem** Parity is a classical problem in the community of GP and GE. The target of this problem is to find out a boolean function that takes an input of binary string, and returns an indicator about whether the input is even(0) or odd(1). In the case of the length of the input is 5, it is called 5-parity problem. Four different binary operations are allowed to use To evolve such a boolean function, specific rules for the evolving process can be found in Appendix A The evaluation of evolved candidate is intuitive, all possible inputs are used as the test set, and the candidate phenotype would run through the whole test set. The fitness function is represented by the miss-computed instances in the test set, formally written by formula, in which g and i are representing the phenotype of the evolved candidate and the instances in test set respectively, and f_i is the correct result of instances:

$$f(g) = \sum_{i=1}^{2^5} (g(i) - f_i) \to \min . \tag{4.11}$$

9. **Multiplexer problem (11-bits)** The Multiplexer problem was firstly introduced in the work of Koza [41], whose target is to simulate a multiplexer in the field of electronics. For this test problem, an 11-bit multiplexer is used. It has an input of 3 'address' bits and 8 ($2^3 = 8$) data registers, totally 11 bits. Every single input is either 0 or 1. The 11-bits multiplexer function should have the ability to select the particular data bit that is singled out by the three address bits. For example, when we have three address bits with value 110, the value stored in the seventh data register (i_9) should be the output, just like fig 4.1 shows.

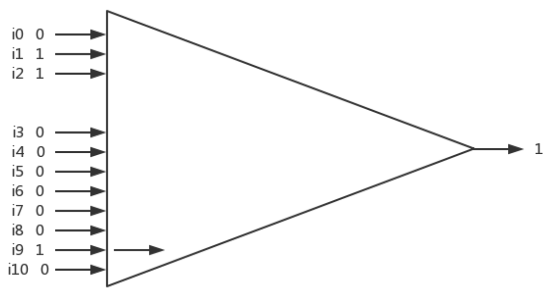


Figure 4.1: Example of the 11-Bit Boolean Multiplexer with the input 11000000010. The first three bits are the address arguments whose binary value indicates the data bit i_9 as the output.

The BNF file of this problem defines as it shows in Appendix A. With this BNF file, the GE system can generate a multiplexer simulator in the form of a Boolean expression

$g(i_0, i_1, ..., i_{11})$, whose output would be boolean value, i.e either $True(1)$ or $False(0)$. A perfect multiplexer boolean expression can always give out the value stored in the correct data register (between $i3$ and $i10$) according to the corresponding 3-bits address (from $i0$ to $i2$).

Theoretically, the 11-bit multiplexer problem has a total of 2048 (2^{11}) possible combination of all 11 inputs. For this problem, all 2048 possible combinations are used as the test set to test every evolved candidate. Meanwhile, to make this problem as a minimization problem, the fitness value of this problem is defined as the number of mismatched output in the 2048 trails, which can be formally defined by the following formula:

$$f(g) = 2^{11} - \sum_{i=1}^{2^{11}} g(i_0, i_1, ..., i_{11}) \rightarrow \min . \tag{4.12}$$

10. **Santa Fe trail problem (Artificial Ant Problem)** The artificial ant problem is a kind of Path Finding and Planning problem, according to the categorization in [5]. The target of this problem is to evolve a set of logic for an artificial ant G_{ant}. This set of logic can help the ant to find all food lying along an irregular path on a square plane with a width of 32 grids, according to the work[41]. There are 89 food pellets on the plane in total, and the ant's starting point is at the upper-left cell of the plane with the coordinate (0,0) and facing to the east (right). Figure 4.2 shows the position of food pellets on the plane.

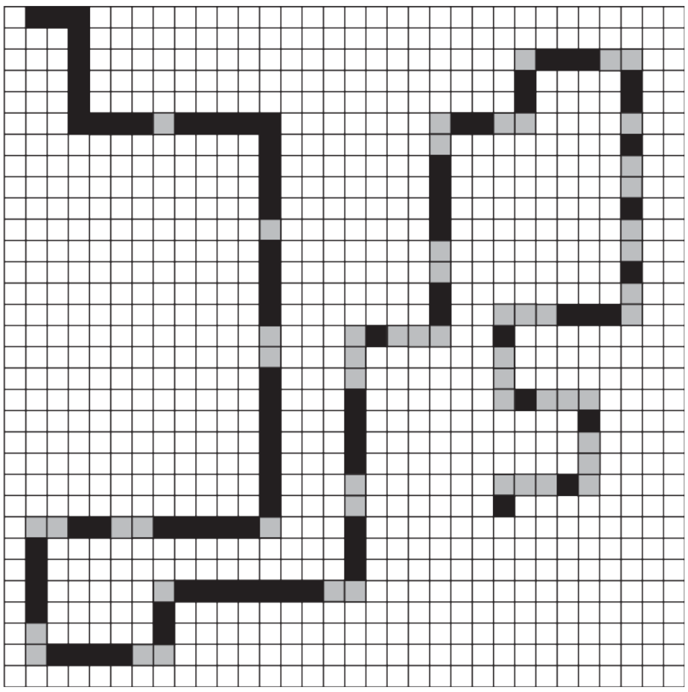


Figure 4.2: Santa Fe food trail for the Artificial Ant problem. Black cells are the food pellets and gray cells are the gaps in the trail. [2]

The artificial ant has only a very limited vision of its world. Specifically, it can only perceive whether there is a food pellet in the adjacent cell in front of or not. And the ant can only available for only three different operations:

- Turn Left: The ant turns left for 90 degrees without moving.
- Turn Right: The ant turns right for 90 degrees without moving.

[2]Controlling bloat : individual and population based approaches in genetic programming - Scientific Figure on ResearchGate. Available from: `https://www.researchgate.net/figure/Santa-Fe-food-trail-for-the-Artificial-Ant-problem-Black-cells-are-the-food-pellets_fig4_277165074/actions#reference`[accessed 10 Jul, 2019]

- Move Forward: The ant move 1 step toward the direction it is facing. In the case of there is food in the cell the ant entering, it eats this food pellet.

On the same time, the artificial ant can have the logic of IF-ELSE conditional operators. Apart from previously mentioned operations, no further operation is available for the ant. The BNF file in Appendix A defines all these operations and other supportive non-terminals. It also has the limitation of the maximum basic operation number, 543. Which means the evolved artificial ant can only turn left, turn right, or move forward for a maximum of 543 times.

The fitness of this problem is easy to describe, which is the number of rest food pellet on the plane. That is, the more food pellets the evolved ant can eat, the better fitness it has. Formally, the objective function can be written as:

$$f(G_{ant}) = 89 - exec(G_{ant}) \rightarrow \min\,. \tag{4.13}$$

4.1.2 Implementation of Benchmark and its Structure

The benchmark designed in this work is mainly used for testing different GE and GE variant systems. However, every system has its implementation environment. For example, PonyGE2[52] only support a python environment with version number is higher than 3.5, whereas SGE[60] system is only available for python2.7 at most. Some other GE variant systems may also be implemented in C, Java, or any other computer languages according to the habit of its author. This reality has caused the difficulty for the implementation of this benchmark since it is not easy to ensure the broad applicability for different GE systems come with different implementation languages.
In order to ensure the applicability of benchmark for GE and GE variant systems, a method in implementation is used here. That is, those parts need to be programmed, i.e., objection functions for every benchmark problems, are implemented in a relative underlying language, C-language. Meanwhile, interfaces for different languages are provided to adapt to different run-time environments. By such a design to ensure the broad applicability, and on another side, to reduce repetitive work on coding object function in different languages, the overview of this benchmark can be seen in Figure 4.3:

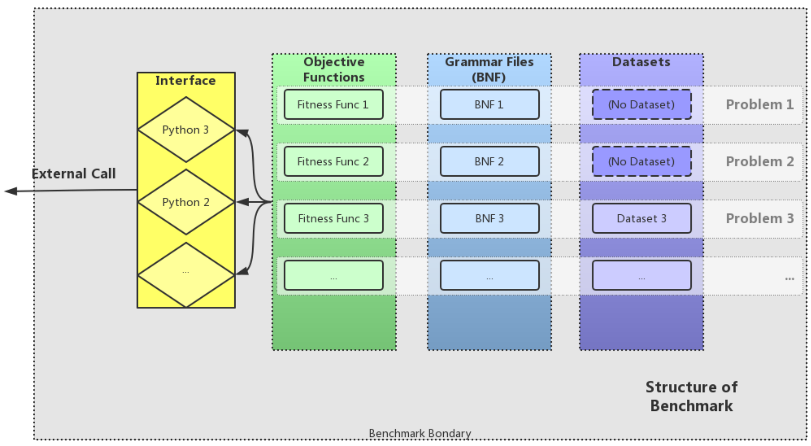


Figure 4.3: Benchmark Structure

Benefit from the great portability and adaptability of C language to other computer languages, this C-based implementation of the objective function is an insurance of easy-calling for other language implemented systems. In the case of a new GE system is necessary to test, the work of programming all test problem is simplified to write an interface for the corresponding language. In this work, an interface for Python is implemented with the help of Cython package since two system are going to be tested in this work (PonyGE2 and SGE) are implemented in Python2 and Python3 respectively. And of course, for C and C++ implemented systems, no further interface is necessary, as they can call the objective function directly. As for other computer languages, the interfaces for them needs some efforts from users and it may also be our future work and released in later versions. As for the detail of usage for this benchmark, it comes with the application of this benchmark and will be discussed in detail in later sections.

4.2 Application

4.2.1 Automated comparison over GE systems

In this work, the designed benchmark is used to compare the performance of different Grammatical Evolution systems, and an automated comparison system is implemented. They compared systems include PonyGE2 and SGE. Preliminary, GGES system is also included in this project for a broader comparison. However, since a technical problem we found in the test phase, we finally disabled it and excluded GGES system in this work. The specific reason will be discussed in Chapter 6.

The comparison over different GE systems is revealed in such a way, that for every tested GE system g, the tested system will be controlled to run every problem on a given problem set $P = (p_1, p_2, ..., p_n)$ (by default, every problem in benchmark will be tested). Here, for every independent problem p_n, a fitness value f will be given as an output of the GE system, and the hyper-parameter configuration of the GE system is denoted as c. The fitness value f can be represented by $f_{g,n} = G(c, p_n)$, in which G means to run problem p_n on system g with configuration c. Here, we can consider each benchmark problem as an optimization problem, and the search space is constructed by the hyper-parameter configuration. MIP-EGO will be used to find a best hyper-parameter configuration c^*, which is corresponding to the best fitness value(minimum value) $f^*_{g,n} = G(c^*, p_n)$. Due to the problem that, the global optimum of different problems may locate at a different position since the landscape of the search space of different problems could differ from each other. So for every independent problem, the optimization needs to be executed separately. Similar to this, even for the same problem, the optimization process for two systems needs to be done separately because of their hyper-parameters are totally different. After iterate this process on both test problems and systems, the best possible configurations for given system g on each problem $F^*(g, P) = (f^*_{g,1}, f^*_{g,2}, ..., f^*_{g,n})$ would be found. Moreover, when the hyper-parameter tuning process for different systems $\{g_1, g_2, ..., g_n\}$ is finished, their performance over different problems can be compared by using their fitness value generated on different test systems with 'best-suited' hyper-parameter configurations. Algorithm 3 formally describe this process. In which q represents the iteration number of MIP-EGO algorithm. In this project, the value of q is set to 100.

Algorithm 3 Automated GE systems Comparison Test

```
Target: $F^*(g, P)$ for all $g$ in $g' = \{g_1, g_2, ..., g_n\}$
  for $g$ in $g' = \{g_1, g_2, ..., g_n\}$ do
    Initialization for System Controller for system $g$
    Interface build for objective functions calling
    for $p$ in $P = (p_1, p_2, ..., p_n)$ do
      MIP-EGO$(c, F, q)$ $\{F = f_{g,n} = G(c, p_n)\}$
    end for
  end for
```

The implementation of this automated comparison test completed by Python3, and the usage of this system will be introduced in a later section. Here, Figure 4.4 shows an overview of abstract structure for this work.

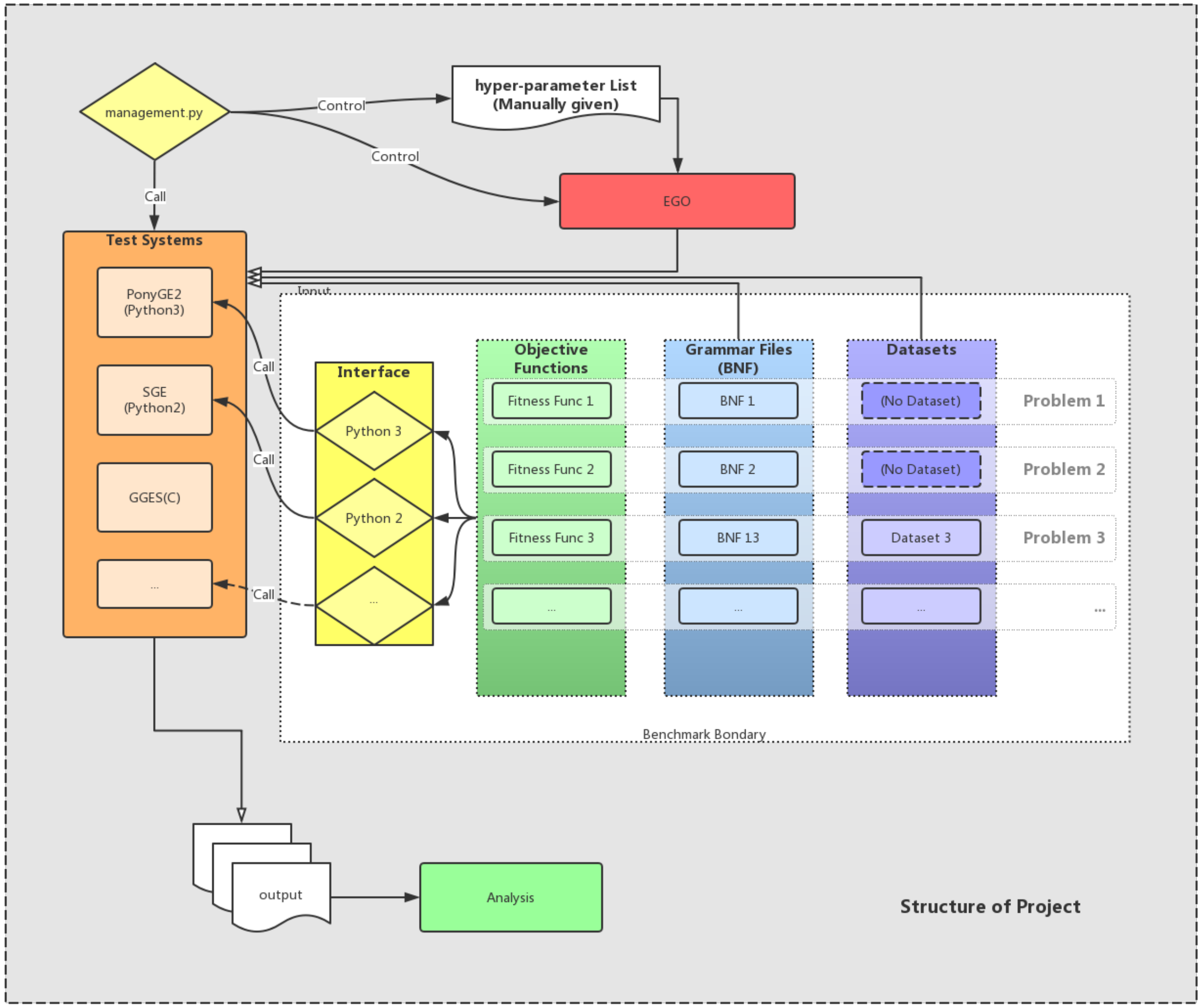


Figure 4.4: Structure of This project

To use the GE comparison system and compare different grammatical evolution systems, several **input** elements is necessary to prepare before the test starts, which are:

- **Hyper-parameter list for test systems**: Every system has different names and ranges of their hyper-parameters, so it is important to tell the comparison system previously, what hyper-parameters every test systems have and the tuning range of them. In this project, this work is achieved by reading a JSON file which includes a list hyper-parameters. Thus for every test system, a JSON file is necessary to specify all names of system and ranges of it. The specific form of this can refer to an example in Appendix B.1.
- **A name list of tested systems**: Before running a GE system comparison automatically, it is necessary to tell the computer what systems are going to compare. A list of name of all test systems needs to be specified. In the meanwhile, the code of the tested system must be reachable for this project. The GE comparison system will automatically call systems to test problems.
- **A name list of tested problems**: By default, all problems in the benchmark will be tested for the comparison, but this is not mandatory. User has the authority to test only part of it or specify their problem. The name list of the tested problems is specified. If the new problem is added for the test, the name list of tested prorblem must be implemented first. For detailed information, please refer to the manual of this project[3].

Correspondingly, the **output** of this comparison system is not simply an indicator of "who is the best". It includes a wide range of information and files as the output for the test to help users get to know about the tested system and their relative performance better. They include:

- **Formatted data file for best-founded hyper-parameter configuration:** For each system, the best-founded configuration for every problem will be stored in CSV file.
- **Formatted data file for best-founded fitness value:** For each system and benchmark problem, the fitness value in each iteration of the tuning process is stored in a CSV file.
- **Problem-based fitness curve:** The change of fitness value of tested systems on all benchmark problems over the whole hyper-parameter tuning process. Both the average value and standard deviation are shown in this graph in the case of multiple time test.
- **Distribution of Tuned hyper-parameter:** After the hyper-parameter tuning process, the best-founded configuration are stored. In the case of the test has tested for multiple times, the distribution of hyper-parameter is shown in parallel coordinates and may reveal some interesting patterns.
- **Log files:** In this system, all original log files generated by either tested systems and comparison systems are kept to avoid any loss of important information. The categories of log files will be introduced later in the following part.

For the reason that the comparison between different systems can be in many aspects, and it is usually tough for us to predict all requirements in the phase of implementation, different types of original log files are recorded in this project. By such a way, it is believed that most useful information can be decently stored to cover the potential needs as much as possible. On the other side, the standard output, which is the output on the console of the testing machine, also covers innumerable critical information. In this project, all standard outputs on the console are stored for further analysis, and it is strongly advised to do so in customized usage too. All these log files can be categorized into four parts:

1. **Global Output logs:** This kind of log file records the standard output which is printed to the console of the test machine, which is the most valuable type of log files. This type of log includes the output of interface compiling, information produced by hyper-parameter tuning process (MIP-EGO package), objective value for every iteration, best objective value until know for every iteration, etc. The analysis is mainly based on this kind of log files. This log is

[3] `https://github.com/dabingrosewood/MasterThesisProj/`

generated by *'nohup'* in command line[4], every execution of *'management.py'* produces such a log file.

2. **Configuration records:** For each process of hyper-parameter tuning, the best configuration for the current problem and system will be recorded in this type of log file. Similar, the best objective value will be stored here, too. The name of this type of log follows the schema of *out_[SYSTEM_NAME]_[PROBLEM_NAME]_[TIMESTAMP][MACHINE_NAME].txt.*

3. **Fitness value records:** Every hyper-parameter tuning process last for many iterations (100 in this project). Also, for every configuration generated in one iteration, it is tested for several times (5 in this project). Every test produces a corresponded objective value, and it will be stored in this type of files. For those all objective value belongs to the same problem and same system, are stored in one file. The name of them follow the schema *summary_of_[SYSTEM_NAME]_[PROBLEM_NAME]_[MACHINE_NAME].log* and located at the root folder of systems separately.

4. **Test system logs:** Every tested system has its run time logs, which is usually stored in their system folder. This part of log files was temporarily not used in this project.

4.2.2 Basic Usage method

The way of implementation follows the way we mentioned in the previous section. In this section, the primary usage method of our automatic GE comparison system is introduced. In the case of a more detailed description of software description or document is necessary, please refer to the Appendix C.
According to different sub-targets of this system, the system is composed of two parts apart from the benchmark itself, test part and analyzing part. The test part covers the work before the output file in Figure 4.4, and the analyzing part does the rest of the work. Test part aims to use the designed benchmark to GE system candidates and record all information in different kinds of log files. And correspondingly, the target of analyzing part is to extract meaningful information from generated log files and give out some meaningful statistical data. The layout of the file structure for this software can be found in Appendix C.
As has been mentioned previously, it is necessary to specify some information as the input of this test system. The list of hyper-parameters for every test system must be manually completed in JSON files and be placed in directory *'util'*. An example of the list of hyper-parameters in a JSON form can be found in Appendix B.1. For the rest of the input, it is necessary to specify them in the main program of *"management.py"*. Following codes is an excellent example of it.

```python
if __name__ == "__main__":
    global_log_cleaner() # clean previous test result

    #here to define the problem for the comparison
    problem_set=['ant','string_match','vladislavleva4','mux11']

    #shared parameters
    n_step=10
    n_init_sample=5
    eval_type='dict'
    max_eval_each=50000
    test_sys=['SGE','PonyGE2']
    parameter_list_dir='/util'

    test=TesterManager(test_sys, problem_set, n_step, n_init_sample,
                       eval_type, max_eval_each, parameter_list_dir)
    test.run()
```

[4]In this work, command like *'nohup python3 management.py >logs/output_[MACHINE_NAME]_[TEST_NUMBER].txt 2>&1 &'* is used to store standard output which is print to console, which generate this kind of log files.

By running *"management.py"*, the system will execute the test part of the GE comparison system. And it is recommended to call this script by using command like ***"nohup python3 management.py >logs/output_[MachineName].txt 2>&1 &"***, since this command tells the computer to automatically stores produced standard output into corresponding log file under "logs" directory, which is the global output logs mentioned before.

On the other side, the analyzing part is responsible for data extraction from generated log files and analyzing work. By default, the file *"post_test/paramerter_extractor.py"* is the main program for the analyzing work. Following codes specify how to use it.

```
if __name__ == "__main__":
    default_log_dir = "../logs/"

    # The problem set used in test. The orders of given problems matters.
    dealing_problem_set = ['ant','string_match','vladislavleva4','mux11']

    # extract information for original log files.
    # It produces two type of files:
    #      1. For each problem and system, a .csv file records all tuned
        hyper-parameters."Configuration_[SYSTEM]_[PROBLEM].csv"
    #      2. For each problem and system, a .csv file records all fitness value in each
        iteration of hyper-parameter tuning.
    extractor = PARAMETERS_EXTRACTOR(default_log_dir, dealing_problem_set)
    extractor.run()

    # comparison between different GE systems on different benchmark problems.
    # This method will produce
    system_analyzer(target_dir='tmp/', show=True)

    # Draw a coordinate parallel to show the distribution of hyper-parameters.
    conf_analyzer(target_dir='tmp_para', show=True)
```

This script can extract information from the original global log file, transform data into '.csv' formatted files and produced the default outputs, including the graph of (average) fitness change over the hyper-parameter tuning process, the parallel coordinate of distribution of tuned hyper-parameters as well as a formatted data file for these graphs. Following Table 4.1 demonstrated the descriptions and locations of generated files of automated GE comparison system:

Name	**Location**	**Description**
[Problem].jpg	post_test\	Problem-based fitness curve.
[Sys]_[Problem].csv	post_test\tmp\	Formatted best-founded fitness values file.
configurations_*[Sys]_[Problem]*.csv	post_test\tmppara\	Formatted best-founded hyper-parameter configurations file.
configurations_*[Sys]*.png	post_test\distr_conf\	Distribution of Tuned hyper-parameter.
output_*[machine]_**.txt	logs\	Global Output logs.
out_*[Sys]_[Problem]**.txt	logs\	Configuration records.

Table 4.1: The list of Produced File and Location of them.

In this project, both the changed of fitness from different systems and distribution of their tuned configuration will be analyzed to compare systems and their reasons for that. In Chapter 5, the result of the analysis will be discussed in detail.

4.2.3 Extend the benchmark

For most Grammatical Evolution system users, no matter their target is to compare different system or to run their applications, benchmark problems cannot fully satisfy their demand. For this reason, the expansibility of this benchmark must be considered. Before adding a new problem into this work, several prerequisites must be satisfied to keep the general schema of the benchmark. That is,

- Since every test problem could be called for millions of times during the period of test, the efficiency of the program must be taken into consideration. Therefore, the newly added fitness function should be written in C language to keep the running efficiency and the adaptability for test systems.

- If the newly added problem is a supervised learning problem and has training and testing datasets, they must be named by Train.txt and Test.txt respectively.

- The final main function for calling the objective function for the newly added problem should be named `type eval_[ProblemName]()` and the source file should be included in 'fitness.h,' which is as a composite of objective functions of this benchmark.

- In the case of the new problem is a supervised probelm, it is necessary to store the training set and the test set in two seperated files. Each line represents an instance of data.

After satisfying all these prerequisites, we still need to implemented an interface for the environment of the corresponding GE system. Here, one example for adding new test problem to PonyGE2(Python) is used to demonstrated here. In the case of a new problem is going to added into the benchmark for test:

1. Coding the evaluation of problem into '.h' or '.c' file, name of function for calling it as `int evaluate_[problem_name](argv[])` and include it in `' \ cython\ fitness.h'`

2. Adding the declaration of calling evaluation function for new problem in `' \ cython\ interface.pyc'`

3. Copy the whole cython folder into `'PonyGE2 \src \fitness '` and build it.

4. Adding a fitness class in PonyGE2 system, under `'PonyGE2 \src \fitness '`, use "from cython.interface import evaluate_[problem_name]" to call coded evaluation function.

5. Copy necessary files (grammar, dataset) into corresponding place in test system.

6. Run the test with parameter files or command line.

For the detail description of how to add new problem into the benchmark, please review the Manual in Appendix C or the 'README.md' file.

4.3 Test settings

Two GE systems, PonyGE2 and SGE, are compared by using previous mentions automatic GE comparison system in this work as a test of this benchmark. In this section, the detailed setting in our comparison test is explained to ensure that readers can repeat our test easily. The Global setting is to tell the computer what system and problems to test and control the hyper-parameter tuning process of them. Iteration Number is the number of iteration of MIP-EGO process for each problem and system. Initial Point Num. represents how many sample points to construct in the initialization of MIP-EGO algorithm, which is the n if that algorithm. Maximum objective function calling number defines the limitation for calling fitness function in every run of the GE system. Meanwhile, the test is going to be executed for 20 times independently to limit the influence of the exception case. All results in chapter 5 will be using the average value of valid results of these 20 runs.

[4] `https://github.com/dabingrosewood/MasterThesisProj`

Name	Value
Number of independent test	20
Test systems	PonyGE2,SGE
Test Problems	Ant,String_Match,Mux11,...
Iteration Num.	100
Initial Point Num.	5
Maximum Objective Function Calling Num.	50000

Table 4.2: Global Settings

Apart from the global settings, the tuning range for hyper-parameters of GE systems is also predefined before the test. In the test phase of this project, we found that different kinds of error can easily accompany the tuning of some hyper-parameter. Especially for the PonyGE2 system, when several hyper-parameter is open to be tuned, it is almost for sure that error would occur. We looked into these problems and analyzed the reasons for them, classify these problems into three categories of reason.

- The first type of error comes because of the way PonyGE2 defines its systematic parameters. A sub-parameter may have a relation with several high-level parameters, even if they are representing different things. For example, one parameter with the name of *"mutation_ event"* is a sub-parameter of options for two parameters simultaneously. In the case of mutation method is chosen to be *"subtree"* or *"int_flip_per_ codon"*, *"mutation_ event"* is enabled to represent how many times this kind of mutation could happen. Because of these two mutation method are totally different, a totally error-free number for *"int_flip_per_ codon"* can easily cause error for *"subtree"* mutation[5]. To deal with this kind of problem, we spilled the parameter *"mutation_ event"* into two in the process of hyper-parameter tuning, each one is connected with one specific way of mutation, and they are tuned independently as two different hyper-parameters.

- The second type of error is the so-called *time-out error.* In PonyGE2 system, some hyper-parameter is in control of the depth of the derivation tree. It is easy to understand that, the deeper the tree is, the bigger the search space is. In the case of the search space is too big for the current computer, the program will constantly run without ending or with a huge time cost. In order to deal with this problem, the hyper-parameters is controlling the depth of the tree will be remained as default to balance the trade-off between time cost and performance of the system.

- The third type of error comes from the limitation of options themselves. Some options have strict pre-requisite to use. For example, *"nsga2 selection"* method is only available for multiple objective optimization problems. For this kind of problem, these unavailable options will be excluded from the option list of tuning.

Similar with the PonyGE2 system, SGE system can also suffer from the second type of problem of PonyGE does, as SGE has a hyper-parameter with the name *MAX_ REC_ LEVEL* to control the maximal recursion level in transferring context-free grammar to non-recursion grammar. Therefore, a similar method is used in SGE to avoid the error: *MAX_ REC_ LEVEL* will remain as the default value (5) in most of the problems. For a special case string_match problem, this value is manually increased because of default value limited the performance of system[6] Moreover, the calculation of fitness for this problem is not computation costly, so increasing the value of *MAX_ REC_ LEVEL* will not cause time-out error.
Table 4.3 and 4.4 specified ranges of tuning for hyper-parameters in PonyGE2 and SGE system respectively. The detailed explanation of mentioned hyper-parameters can be found in the introduction of these GE systems in chapter 3.4.

[5]int_flip_per_codon mutation can have a large range of mutation number that subtree mutation in PonyGE2 since the way subtree works, it will generate a new tree to replace the subtree on the mutation point, too many times of this kind of mutation could make the tree exceed the limitation of tree depth.

[6]The structure of grammar for this problem requires many recursions to generate even a short string. Because of this complex structure of grammar file definition for this problem, a "small" value of *MAX_ REC_ LEVEL* is unsuitable for this problem.

Name of Hyper-parameter	Type	Value/Range
INITIALISATION	Nominal	"PI_grow", "rhh", "uniform_tree"
CROSSOVER	Nominal	"variable_onepoint", "variable_twopoint", "fixed_twopoint", "fixed_onepoint"
CROSSOVER_PROBABILITY	Continuous	$[0, 1]$
MUTATION	Nominal	"int_flip_per_codon", "subtree", "int_flip_per_ind"
MUTATION_PROBABILITY	Continuous	$[0, 1]$
MUTATION_EVENT_SUBTREE	Ordinal	$[1, 5]$
MUTATION_EVENT_FlIP	Ordinal	$[1, 100]$
SELECTION_PROPORTION	Continuous	$[0, 1]$
SELECTION	Nominal	"tournament", "truncation", "variable_onepoint"
TOURNAMENT_SIZE	Ordinal	$[1, 50]$
ELITE_SIZE	Ordinal	$[1, 100]$
CODON_SIZE	Ordinal	$[200, 1000]$
MAX_GENOME_LENGTH_SIZE	Ordinal	$[100, 500]$
POPULATION_SIZE	Ordinal	$[100, 1000]$

Table 4.3: Hyper-parameters Tuning Range for PonyGE2

Name of Hyper-parameter	Type	Value/Range
POPULATION_SIZE	Ordinal	$[100, 1000]$
ELITISM	Ordinal	$[50, 500]$
TOURNAMENT	Ordinal	$[1, 50]$
PROB_CROSSOVER	Continuous	$[0, 1]$
PROB_MUTATION	Continuous	$[0, 1]$

Table 4.4: Hyper-parameters Tuning Range for SGE

Chapter 5

Evaluation

5.1 Problem found in Test

As it has been mentioned previously, GGES was firstly included in the project to compare its performance with other GE systems. The problem we found is that, in the test phase, an unstable problem about acquiring standard output from GGES system for python was found, by which the management script is written. This could lead to the result that the extracted test result for GGES is not correct, so the part of GGES problem is disabled in the last version of this work. This part will become our future work, to solve the problem of acquiring blocked, standard output from GGES system.

In the test phase, we found that the execution of the Max problem cost a massive amount of time on our test machine under our testing settings. According to the grammar file and generated candidates, the main reason for this vast time cost comes from the design of grammar and the target of this problem. It is allowed to generate a nested-loop structure in this problem. In order to generate a number as large as possible, GE system tends to build a very deep nested loop since the depth of loop is almost the synonym of the large number under the setting of this problem. However, the cost for that is a huge time cost to calculate the function includes this deeply nested loop. In order to ensure the integrity and consistency of this test, this extremely time-costly problem was excluded in this test to make this test could be finished in an acceptable time.

5.2 Result of Test

The application of the proposed benchmark frame is used to compare two grammatical evolution system in this project. Even though all GE and variant systems follow the same general idea of GE, the performance of two systems shows some difference, and their characters diverge.

In many symbolic regression problems (StringMatch, Keijzer6, and Vladislavleva4), PonyGE2 system seems reached a performance wall after a very limited number of iteration of hyper-parameter tuning. The standard deviation is also tiny, which proves this it is stable for PonyGE2 system to reach such a performance wall under current configuration tuning range. The situation of the SGE system is different. In StringMatch problem, SGE system continually increases its performance throughout the whole hyper-parameter tuning process. Also, in the case of keijzer6 problem, PonyGE2 approach the value 0 very early and remain stable in all runs, whereas SGE was still struggling to eliminate the error between evaluated individuals and target sample. As for the Vladislavleva4 problem, SGE's progress is much slower than PonyGE2 system. Moreover, for all these problems, the SGE system never performs better than the PonyGE2 system, even it is approaching the fitness value of the PonGE2 system. The change of fitness value throughout the hyper-parameter tuning for both systems on symbolic regression problem can be found in Figure 5.1 to 5.3.

The result of the banknote problem share some characters from previously mentioned problems. As

Figure 5.4 shows their performance on this classification problem. In the whole process of hyper-parameter tuning, the error value of the SGE system remains bigger than it is from PonyGE2 system. On the same time, the standard deviation for both systems shrink throughout the whole process and become relatively small at the end of tuning, which is considered to be an evidence that both systems are approaching their global or local optimum under the tuned configurations.

The SGE system turns back its situation in the Housing problem, a supervised learning problem with a relatively larger number of variables. Even though SGE PonyGE2 owns a better start point at the first iteration, SGE surpasses PonyGE2 very soon and keep it leading until the end of the tuning process with stability. A similar situation also exists in problem Pagie and 5-Parity. As Figure 5.6 shows, SGE outperformed PonyGE2 in the whole process and, it hits the performance wall very early. For the 5-Parity Problem, both systems improved their performance throughout the hyper-parameter tuning process, but SGE got a relative better result at the end of it.

Multiplexer 11 is a problem with higher complexity when we compare this to other problems in this benchmark. One point to mention is, the time cost for running this problem is much higher than others. Considering the core of SGE and PonyGE2 systems remain unchanged, the most time cost is used in evaluating evolved individuals. For this problem, SGE and PonyGE2 perform almost equally well at the start of hyper-parameter tuning, as it shows in 5.8. But with the number of iteration goes up, PonyGE2 system opens the gap with SGE system, the fitness value of PonyGE2 decrease slightly faster than SGE system. What different with other problems is, the standard deviation of the SGE system is smaller than PonyGE2, which proves that SGE's performance is a little bit more stable than its competitor on this problem.

In the test of the artificial ant (Santa Fe train) problem, the performance of the SGE system is worse than PonyGE2 still. At the beginning iteration of hyper-parameter tuning, we say the performance of SGE system is volatile since a relatively high standard deviation was observed at the first 20 iterations. PonyGE2 reached the optimum value after around 30 iterations and SGE cost almost 50 iterations. During the whole tuning process, PonyGE2 shows better stability with very limited standard deviation.

One interesting phenomenon that attracts our intention is, there is a large difference of performance at the start iteration over several problems, as it shows in Figure 5.1 to Figure 5.4 and Figure 5.9. In all these problems, PonyGE2 always has a much better start point than SGE. We thought of two reasons, and these reasons may also interact with each other to cause this phenomenon. One possible reason is that the SGE system is more sensitive to the hyper-parameter than PonyGE2. In compare with PonyGE2, the SGE system has a much shorter parameter list, and it makes this system more sensible than PonyGE2 since they are following a similar evolving mechanism. The result of the test also supports this idea since SGE has bigger improvements in more problems than its opponent with the help of hyper-parameter tuning. This sensibility of SGE may cause the worse result than PonyGE2 at the start of the hyper-parameter tuning process since both systems may not have proper configurations on hand, and SGE is much more sensitive to that. The second

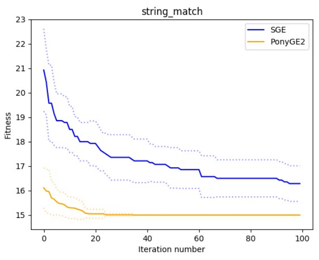


Figure 5.1: StringMatch Problem

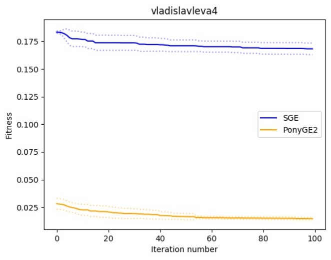


Figure 5.2: Vladislavleva4

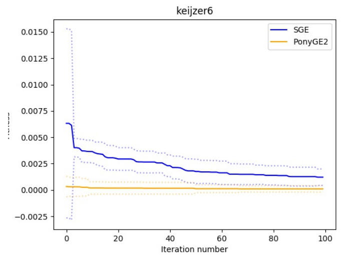


Figure 5.3: Keijzer6 Problem

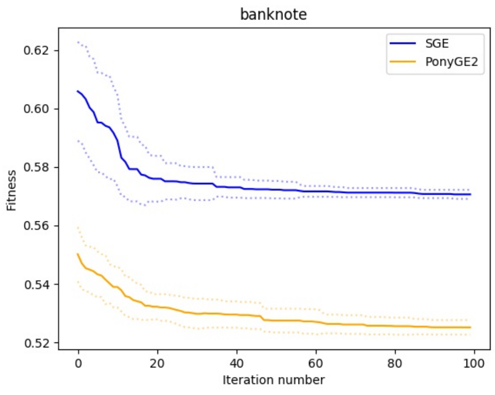


Figure 5.4: Banknote Problem

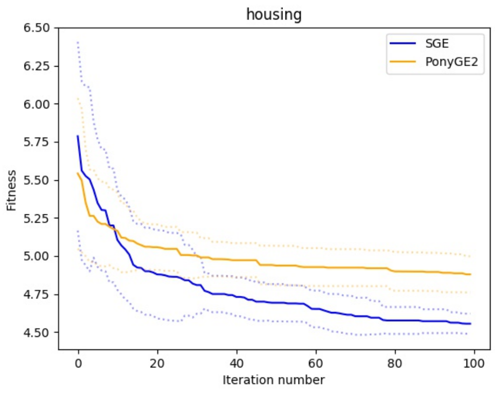


Figure 5.5: Housing Problem

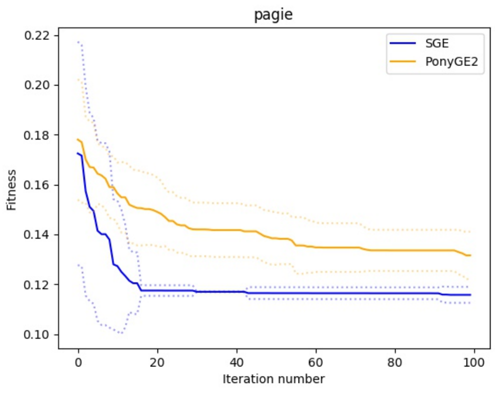


Figure 5.6: Pagie Problem

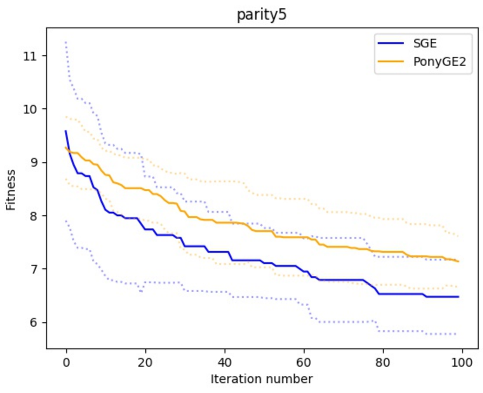


Figure 5.7: 5-parity Problem

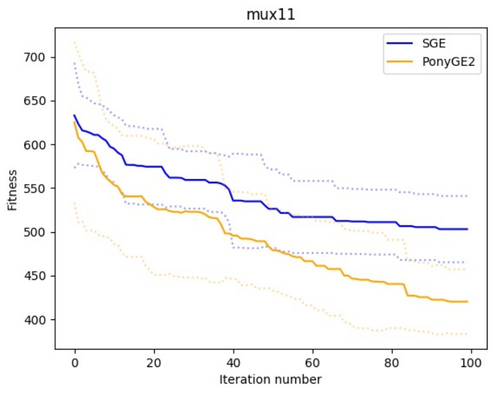


Figure 5.8: Multiplexer 11 Problem

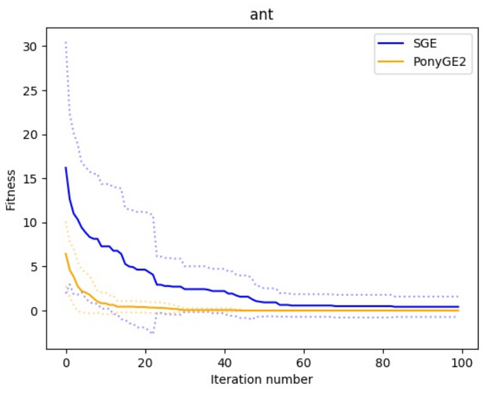


Figure 5.9: Artificial Ant Problem

reason is that PonyGE2 has a better initialization technique. As introduced in Chapter 3, PonyGE2 is not a basic version of GE, and many advanced techniques (includes initialization techniques) are optional in this system to enhance the performance of GE. In the case of the initialization process of hyper-parameter is just finished, and no proper configuration is available for both systems, the evolution of the system could be at relatively low efficiency. Under such a setting, the initialization technique would be determent for the performance of tested systems.

Name of Problem	Final Fitness (standard deviation)		Difference(+/-)
	PonyGE2	SGE	
StringMatch	15.0(±0)	16.3(±0.7)	+8.7%
Keijzer6	9.85(±30)e-5	1.21(±0.780)e-3	+1128.4%
Vladislavleva4	0.015(±0.001)	0.168(±0.005)	+1020.0%
Pagie	0.131(±0.010)	0.116(±0.003)	−11.5%
Banknote	0.525(±0.002)	0.570(±0.002)	+8.6%
Housing	4.879(±0.119)	4.555(±0.066)	−6.6%
5-parity	7.136(±0.472)	6.474(±0.696)	−11.3%
Artificial Ant	0.00(±0)	0.43(±1.15)	+∞
Multiplexer 11	420.3(±36.8)	503.1(±37.8)	+19.7%

Table 5.1: Final Fitness value of tested systems (PonyGE2 and SGE) on benchmark problems and their Difference[2]. A Positive difference represents PonyGE2 performs better and verse visa.

Table 5.1 summarizes the final result for system SGE and PonyGE2 on all tested benchmark problems. The final result for tested systems are recorded and the difference between them are represented by relative difference RD, which is calculated by formula RD $= (F_{\text{SGE}} - F_{\text{PonyGE2}})/F_{\text{PonyGE2}}$. Generally saying PonyGE2 performs better than SGE in the test we have finished: on more than 60% tested benchmark problems, PonyGE2 system acquired better(smaller) fitness values than SGE system did. However, it is unfair to say that PonyGE2 system is better than SGE system. What must be taken into consideration is, this result is only for the given configurations in this test, and some hyper-parameters may influence the result are remain default value due to some reasons mentioned in chapter 4. Hence, a more detailed discussion is necessary for the problem 'which system is better,' and this part will be left in chapter 6.

Actually, if we compare the performance of systems vertically, we can compare the performance of these systems vertically and see the results differently. In table 5.2, we compared these two systems' performance between the start of the hyper-parameter tuning and the end of it. Here we use absolute relative error to express the improvement these system has achieved, who follows the same idea of previous relative difference and calculated by $|F_{\text{Start}} - F_{\text{End}}|/F_{\text{End}}$.. Generally, both systems achieved better performance by hyper-parameter tuning. The banknote is the least improved problem for both systems, which has only 4.8% and 6.1% improvement respectively.

Problem	PonyGE2			SGE		
	First	Last	Imp.[3]	First	Last	Imp.[3]
StringMatch	16.1(±0.8)	15(±0)	7.4%	20.9±1.7	16.3(±0.7)	**28.5%**
Keijzer6	3.27(±9.8)e-4	9.85(±30)e-5	232%	6.53(±8.98)e-3	1.21(±0.780)e-3	**422%**
Vladislavleva4	0.028(±0.005)	0.015(±0.001)	**92.3%**	0.183(±0.001)	0.168(±0.005)	8.9%
Pagie	0.178(±0.024)	0.131(±0.010)	35.3%	0.172(±0.045)	0.116(±0.003)	**49.0%**
Banknote	0.550(±0.009)	0.525(±0.002)	4.8%	0.606(±0.002)	0.570(±0.002)	**6.1%**
Housing	5.542(±0.494)	4.879(±0.119)	13.6%	5.786(±0.619)	4.555(±0.066)	**27.0%**
5-Parity	9.27(±0.581)	7.136(±0.472)	29.9%	9.578(±1.677)	6.474(±0.696)	**48.0%**
Artificial Ant	6.5(±3.67)	0.0(±0)	∞	16.2(±14.3)	0.43(1.2)	3683%
Multiplexer 11	625.1(±92.0)	420.3(±36.8)	**48.7%**	633.0(±60)	503.1(±37.9)	25.8%

Table 5.2: The improvement of tested system by hyper-parameters tuning of tested systems, the number in parentheses is the standard deviation of corresponding data.

[2]A positive difference value means PonyGE2 acquires better result on that problem and verse visa.

Here, we assumed that the result of a GE system with finite configuration over a specific problem follows the normal distribution, and did a hypothesis testing to verify whether these differences caused by hyper-parameter tuning is the improvement rather than any exceptional cases. Considering the fact that the test number is smaller than 30, we use the t-test to estimate that these two groups of fitness value (before and after the hyper-parameter tuning) is different, which also means that such a process improves the performance of GE system. For every system on every tested problem, we test the difference independently in the following ways. We made the hypothesis on the fitness value F:

$$H_o : F_{before} = F_{after} \tag{5.1}$$

$$H_1 : F_{before} > F_{after} \tag{5.2}$$

Here, since the H_1 is representing GE system can perform better with tuned configuration, we shall use the threshold for one-tailed testing. And the formula used to calculate t-statistic is:

$$t = \frac{\bar{X}_1 - \bar{X}_2}{\sqrt{\frac{(n_1-1)S_1^2+(n_1-1)S_2^2}{n_1+n_2-2}(\frac{1}{n_1} - \frac{1}{n_2})}} \tag{5.3}$$

By calculating the t-statistical, we can get the p-values for both system in all tested problem in Table 5.3.

Name of Problem	PonyGE2		SGE	
	t-stat	**p-value**	**t-stat**	**p-value**
StringMatch	6.077	7.628e-6	9.466	2.441e-8
Keijzer6	0.973	0.341	2.472	0.023
Vladislavleva4	11.967	7.601e-11	10.595	3.998e-8
Pagie	8.015	2.425e-8	5.648	1.838e-5
Banknote	11.029	4.556e-10	9.045	3.556e-8
Housing	8.837	3.077e-8	5.828	8.457e-6
5-Parity	12.495	1.750e-14	7.453	1.070e-7
Artificial Ant	7.865	2.152e-7	4.120	0.0011
Multiplexer 11	6.697	4.274e-6	9.044	3.557e-8

Table 5.3: Result of t-statistical and p-value in hypothesis testing

In statistics, the Significance Level commonly used is usually $a = 0.05$. And for our tested problems, the p-values are mostly much smaller than this value except for the Keijzer6 problem. For these cases of $p < a = 0.05$, we can directly refuse the H_0 and saying that the fitness value before and after the hyper-parameter tuning has a significant difference in statistics. As for the only exception, the Keijzer6 problem, due to the fact that the fitness value is approaching the optimum (0), we can say that the fitness value does not show any significant improvement by hyper-parameter tuning, but the result is already good enough for this problem.
Besides, the tuned hyper-parameters shows some patterns at the end of the experiment. Underlying figures 5.2 and 5.2 shows the distribution of tuned hyper-parameters for PonyGE2 and SGE system in two parallel coordinates respectively. For PonyGE2 system, some of the parameters' name is categorized into integers for helping computer draw the graph. The mapping relations are listed in Table 5.4.
Even though not all hyper-parameters are selected to show in the Figure 5.2 due to its complex structure of configuration, it is still apparent to find out some configuration bias exists in the distribution of tuned hyper-parameters for system PonyGE2. For example, Type 1 (variable_onepoint) of crossover dominates in the selection of crossover, and a high crossover rate seems more attractive than a low crossover rate. The situation in mutation part is different. All settings are concentrated on Type 2 (subtree), Type 3(int_flip_per_ind), Type 1 (int_flip_per_codon) of mutation had not even been chosen at one time. Meanwhile, some color-based (which means they are problem-based) patterns are not clearly seen from this graph but can be found in the problem-independent graph for distribution of hyper-parameters in Auxiliary results in section 5.2.1.

[3]Imp. represent for Improvement.

Categories	Origianl Name	Representing Code (Integer)
Initialization	rhh	1
	PI_grow	2
	uniform_tree	3
Crossover	variable_onepoint	1
	variable_twopoint	2
	fixed_twopoint	3
	fixed_onepoint	4
Mutation	int_flip_per_codon	1
	subtree	2
	int_flip_per_ind	3
Selection	tournament	1
	truncation	2

Table 5.4: The Mapping relation between tuned hyper-parameters and their representing code in PonyGE2.

Categories	Origianl Name	Representing Code (Integer)
Problem	Ant	1
	Banknote	2
	Housing	3
	Keijzer6	4
	mux11	5
	Pagie	6
	Parity5	7
	String_Match	8
	Vladislavleva4	9

Table 5.5: The Mapping relation of Problems for Figure 5.2 and Figure 5.2.

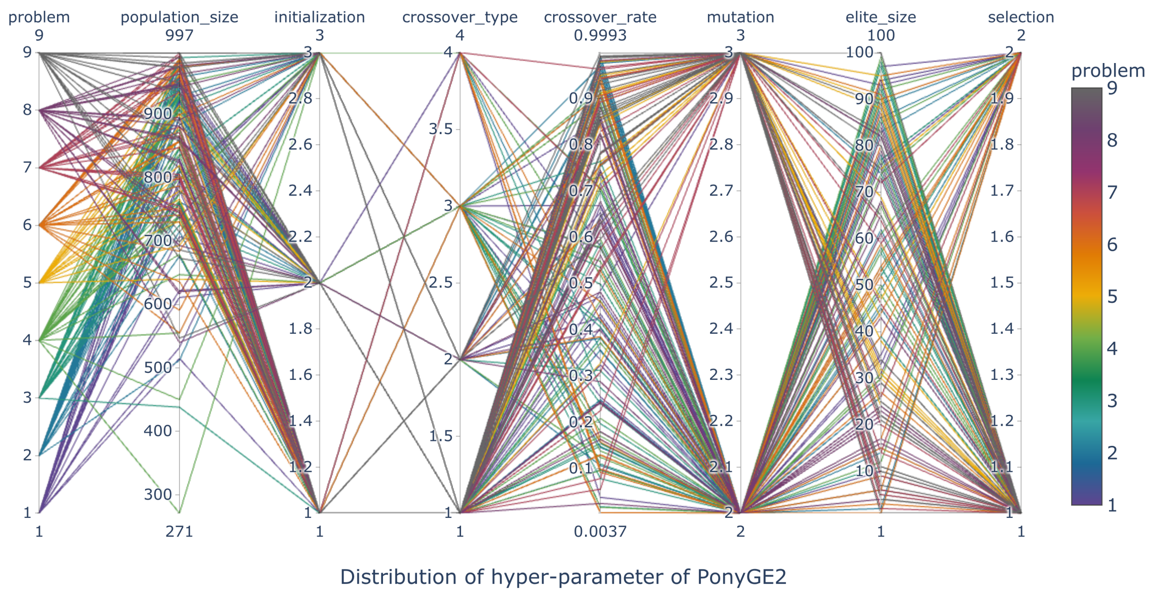


Figure 5.10: The distribution of tuned Hyper-parameter for PonyGE2 system. Names of these parameters in orders are 'Probelm','Population Size','Initialization (Method)', 'Crossover','Crossover Rate','Mutation','Elite Size','Selection' and . All non-numerical type of parameters have been categorized. Mapping between colors and problems can be found in Table 5.5.

Similar, some regular pattern can also be found in the case of the SGE system. Matched with our guess, the hyper-parameter of population size converged to a large number, and we believe it is because a large population is a benefit for the performance. It also seems many samples are located at a relatively low range of Elitism and a relatively high range of tournament size. One penitential reason for this is that a relatively small elitism size combines with a large tournament size may help to improve the efficiency of selection. As for the crossover rate, it seems a mid-rage crossover rate (around 0.5) is not that popular, although this trend is not very clear. However, in the last parameters, most configurations tuned to a low mutation rate expect for several orange lines, which are representing the cases of the multiplexer 11 problem. To look at the relationship between hyper-parameters pattern and problems more precise, the parallel coordinates of hyper-parameters for every problem and systems are separately showed in section 5.2.1.

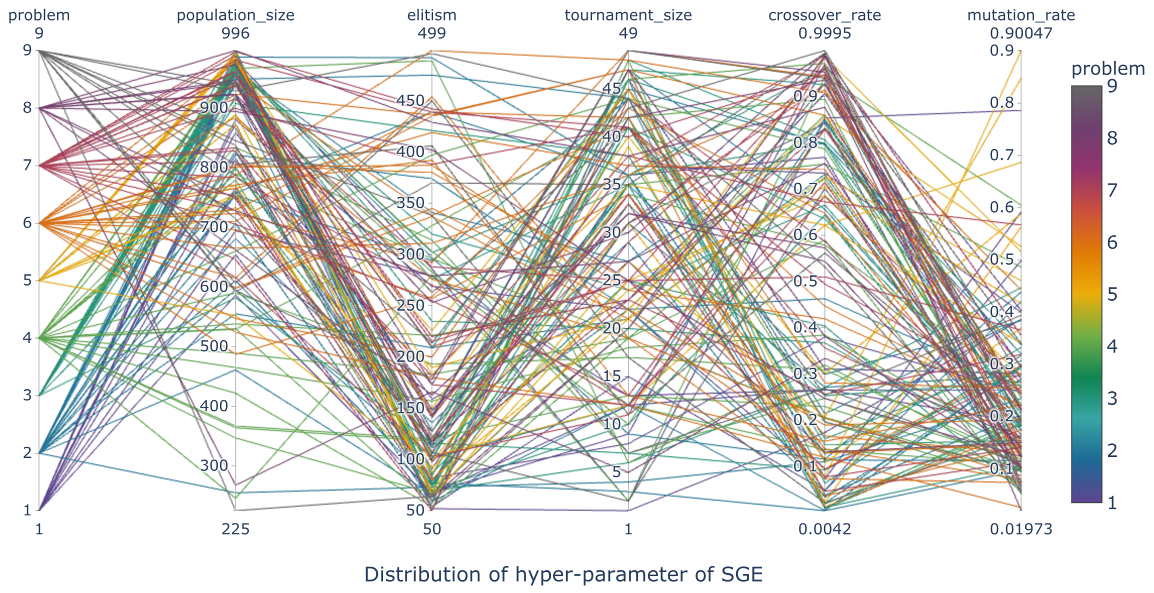


Figure 5.11: The distribution of tuned Hyper-parameter for PonyGE2 system.Names of these parameters in orders are 'Population Size', 'Elitism','Tournament Size','Crossover Rate' and 'Mutation Rate'. All non-numerical type of parameters have been categorized. Mapping between colors and problems can be found in Table 5.5.

5.2.1 Auxiliary results

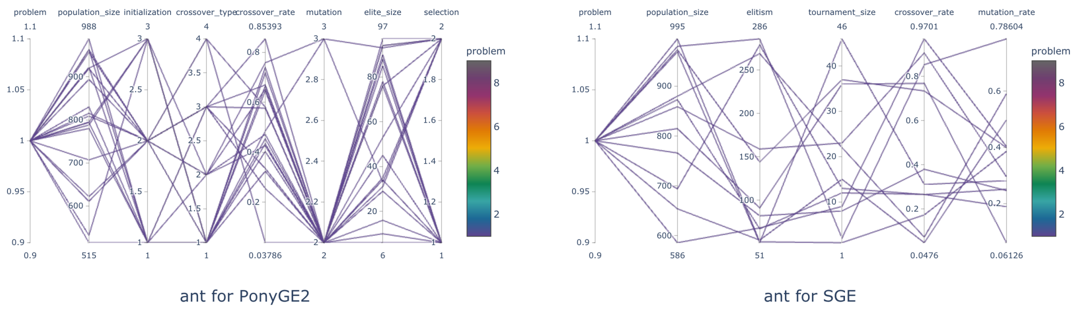


Figure 5.12: The distribution of tuned Hyper-parameter for Ant Problem.

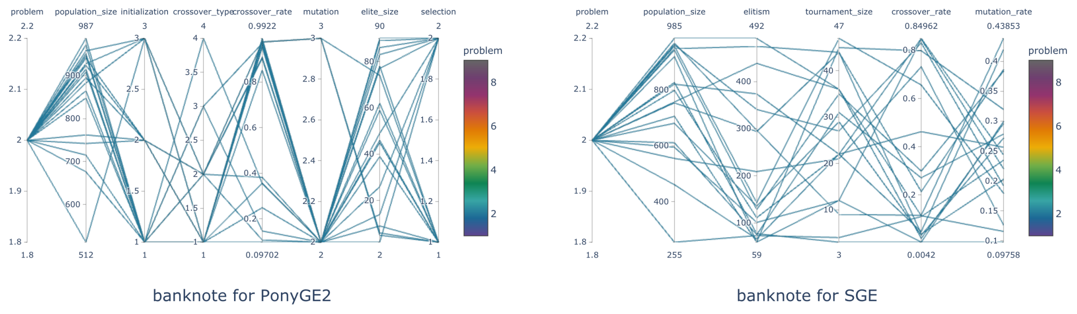


Figure 5.13: The distribution of tuned Hyper-parameter for Banknote Problem.

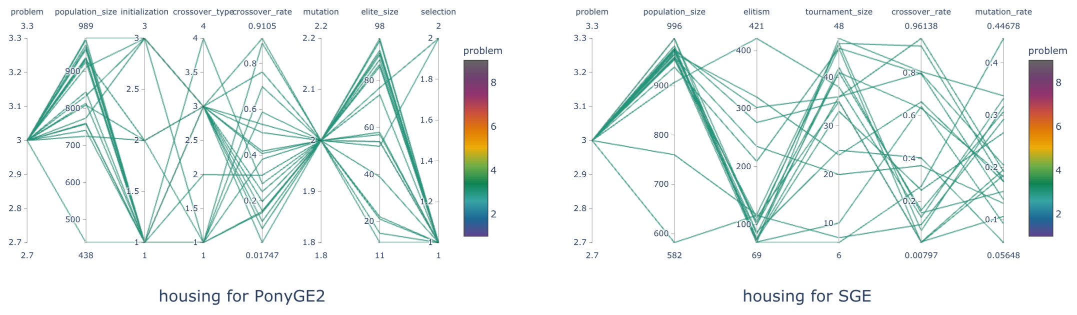


Figure 5.14: The distribution of tuned Hyper-parameter for Housing Problem.

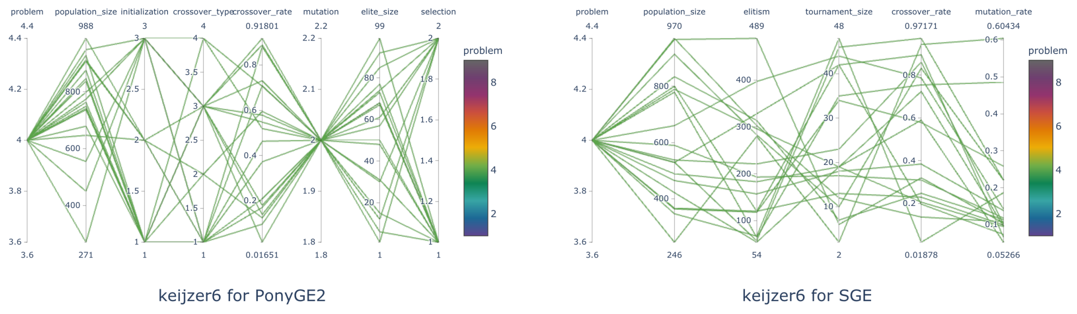


Figure 5.15: The distribution of tuned Hyper-parameter for Keijzer6 Problem.

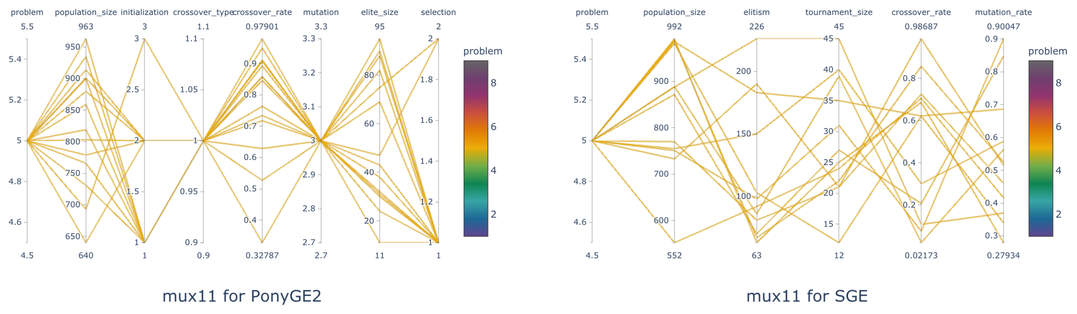


Figure 5.16: The distribution of tuned Hyper-parameter for Multiplexer-11 Problem.

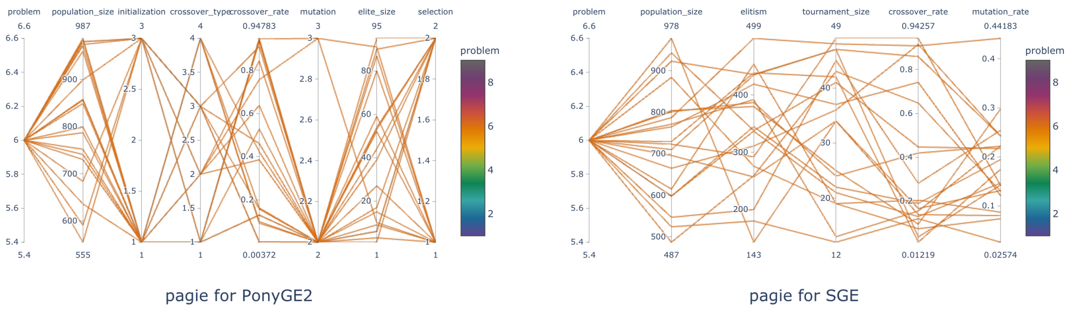


Figure 5.17: The distribution of tuned Hyper-parameter for Pagie Problem.

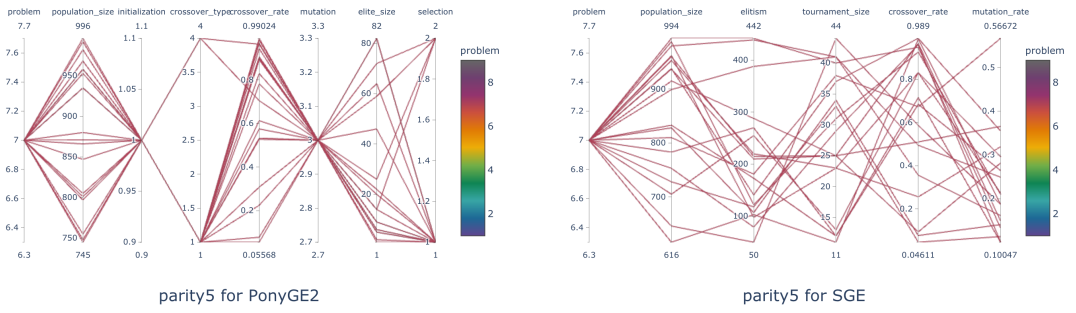


Figure 5.18: The distribution of tuned Hyper-parameter for 5Parity Problem.

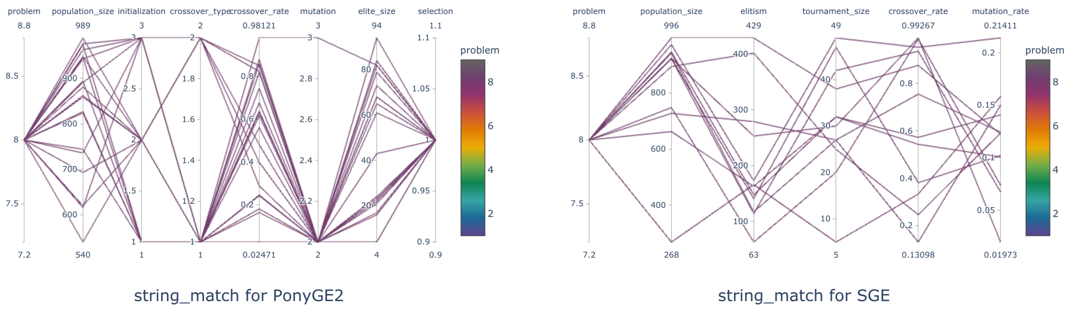


Figure 5.19: The distribution of tuned Hyper-parameter for String Match Problem.

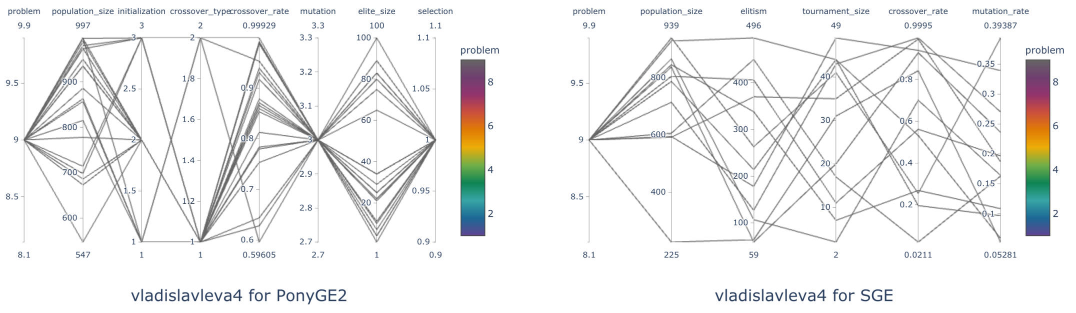


Figure 5.20: The distribution of tuned Hyper-parameter for Vladislavleva4 Problem.

Chapter 6

Conclusion and Discussion

6.1 Concusion for tested system

In this section, the result of comparison on two GE systems will be analyzed. Two tested systems, PonyGE2 and SGE both expressed the ability of grammatical evolution in dealing with the complex problem from different fields. If we compare these two systems horizontally, the conclusion comes to that the PonyGE2 system performs better than the SGE system to some extent, since PonyGE2 got better final results on more benchmark problems. However, this conclusion is under the limitation of our experimental settings, that all configurable parameters and their tuning range are shown at the end of chapter 4.

Even if we said that the PonyGE2 system performs better than the SGE system, it is still very hard to directly define which system is a better choice in a practical application. The main reason of that is, the lists of configurable hyper-parameters for two tested systems has a huge difference and some of them remains as the default value in our test. It makes the comparison between the two tested systems is not that precisely 'fair' to some extent. Meanwhile, the performance they can achieve is restricted not only by these hyper-parameters, but also dozens of factors, such like the grammar of the problem, the benchmark structure (which problems are selected to test) or even the time cost limit for a specific application.
However, considering the characters these two systems showed in this test, we conclude that PonyGE2 is currently a more powerful system, and it is suitable for most researcher and expert users. This conclusion is based on the following reasons:

- PonyGE2 performs better in more problems, no matter it is in the sense of final result or stability of fitness values. In some problem, the result of PonyGE2 in the first iteration of hyper-parameter tuning is even better than SGE got at the end of the tuning process.
- The number of configurable parameters in this system is considerable, which brought great extendibility but also difficult to get started with. Fortunately, detailed and easy-to-read documentation compensate for it.
- PonyGE2 is a well designed, modulated system. Users may modify the composition of this system easily and convert it into other GE variant. Furthermore, PonyGE2 is the second version of PonyGE series and is still in maintenance, whereas SGE seems not.

As for the SGE systems, we see it as a simplified GE system which is suitable for non-expert users.

- SGE has a concise configurable hyper-parameters list, and the ranges for most of the configurations are not necessary for further discussion since they are just a probability and can be easily set to $[0, 1]$.
- The performance of SGE is acceptable for most of the problems. Especially, SGE shows a better ability in dealing with complex problems than in simple symbolic regression problems, which may be a benefit of its high locality design. Considering that most non-expert usage is complex heuristic problems, SGE may fit these complex problem from real applications and non-expert users.

6.2 Discussion and Future work

Even though it seems that PonyGE2 performs better than SGE in the aspect of the final result, both of them have got an excellent result if we look at their performance on these tested problems. However, what makes the difference between the PonyGE2 system and the SGE system is waiting for discussion. As we know, canonical GE is mainly criticized for its insufficient random genome initialization, and the problem of low locality and high redundancy. Furthermore, SGE was firstly proposed to relieve the problem of low locality and high redundancy by designing a new mapping mechanism. Nevertheless, the result may imply that the mapping mechanism from SGE has not that good influences on the performance of these specific problems.

We dived into the mechanism of SGE and found that the limitation of SGE grammar that a maximum depth must be specified beforehand may make those individuals with higher complexity is unreachable for the SGE system under default configuration. In other words, the complexity of evolved individuals for SGE is linearly related to the recursion level in SGE, whereas they are not related in canonical GE with wrapping. The limit of recursion level may lead to the result that some local or global optimum points with the high complexity of construction are unreachable for SGE. However, the canonical GE still has the chance to find it, although the search may be inefficient as a result of its high redundancy and low locality. Although the limitation of maximal tree depth also exists in PonyGE2, the default value of it is set to a much higher value for the consideration of python $eval()$ stack limit and has a much broader space for the system to search in.

Meanwhile, PonyGE2 itself also makes some progress in dealing with the problem of insufficient initialization. As it stated in their work [52], it provided some tree-based initialization technique apart from the basic random genome initialization method, which can build all valid individuals. Also, the PonyGE2 system has a built-in valid individual monitoring mechanism. All invalid individuals generated will be discarded directly. Both ways can relieve the previously mentioned problems of GE and reduce some criticism from the community. Therefore, it is fairer to say that PonyGE2 is an advanced version of GE.

One another problem we found in this project is the benchmark design. In this project, several widely used problem is collected in the benchmark for testing GE systems, and the criteria for selecting is, to choose the most widely used problems in the community. In the process of reviewing works in the field of grammatical evolution, we found that many testing problems are prevalent purely because of historical reasons. It has never been detailed discussed how a scientific benchmark designed for GE should look like, despite an improper benchmark that may provide misleading information on performance. In this work, we got some inspiration for designing the benchmark from work [5], which gives some advice in benchmark design for Genetic Programming (GP). However, it is still disputable that grammatical evolution should be viewed as just a variant of GP or an independent stream of evolutionary algorithms, that probably leads to the fact that advises in work [5] does not suit GE well. Therefore, some work in researching the design of benchmark exclusive for GE is necessary for the community, which can also be our future work. In this work, a pipeline for testing GE systems has been constructed, and we believe this can largely relieve the workload of this future work.

Appendix A

BNF Definition

BNF files regulates the mapping process of Grammatical Evolution systems. Every Benchmakr problem owns their own BNF definition since they are describing different process to map the search space into their solution space. In this section, BNF definition for every benchmark prblems are listed.

StringMatch

```
<start>             ::=  <string>

<string>            ::=  <letter> | <letter><string>
<letter>            ::=  <vowel> | <consonant> | <char>
<char>              ::=  " " | ! | ? | , | .
<vowel>             ::=  <lower_vowel> | <upper_vowel>
<lower_vowel>       ::=  a|e|o|i|u
<upper_vowel>       ::=  A|E|I|O|U

<consonant>         ::=  <lower_consonant> | <upper_consonant>
<lower_consonant>   ::=  b|c|d|f|g|h|j|k|l|m|n|p|q|r|s|t|v|w|x|y|z
<upper_consonant>   ::=  B|C|D|F|G|H|J|K|L|M|N|P|Q|R|S|T|V|W|X|Y|Z
```

Keilzer6

```
<start>   ::=  <e>

<e>       ::=  <e>+<e> | <e>-<e> | <e>*<e> | div(<e>,<e>)
          |    psqrt(<e>) | sin(<e>) | tanh(<e>) | exp(<e>) | plog(<e>)
          |    x[0] | x[0] | x[0] | x[0] | x[0]
          |    <c><c>.<c><c>| <c><c>.<c><c>| <c><c>.<c><c>| <c><c>.<c><c>

<c>       ::=  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Vladislavleva4

```
<start>   ::=  <e>

<e>       ::=  <e>+<e> | <e>-<e> | <e>*<e> | div(<e>,<e>)
          |    psqrt(<e>) | sin(<e>) | tanh(<e>) | exp(<e>) | plog(<e>)
          |    x[0] | x[1] | x[2] | x[3] | x[4] | <c><c>.<c><c>

<c>       ::=  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Pagie

```
<start>    ::=  <expr>
<expr>     ::=  <expr><op><expr> | (<expr>) | <pre-op>(<expr>) | <var>
<op>       ::=  + | - | * | /
<pre-op>   ::=  sin | cos | exp | plog
<var>      ::=  x[<idx>]
<idx>      ::=  0 | 1
```

Banknote

```
<start>   ::=  <e>
<e>       ::=  (<e> <op> <e>) | <f1>(<e>) | <f2>(<e>, <e>) | <v> | <c>
<op>      ::=  + | * | -
<f1>      ::=  psqrt | plog
<f2>      ::=  pdiv
<v>       ::=  x[<idx>]
<idx>     ::   = 0 | 1 | 2 | 3
<c>       ::=  -1.0 | -0.1 | -0.01 | -0.001 | 0.001 | 0.01 | 0.1 | 1.0
<c>       ::=  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Boston Housing

```
<start>   ::=  <e>
<e>       ::=  (<e> <op> <e>) | <f1>(<e>) | <f2>(<e>, <e>) | <v> | <c>
<op>      ::=  + | * | -
<f1>      ::=  psqrt | plog
<f2>      ::=  pdiv
<v>       ::=  x[<idx>]
<idx>     ::=  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12
 <c>      ::=  -1.0 | -0.1 | -0.01 | -0.001 | 0.001 | 0.01 | 0.1 | 1.0
```

5-Parity

```
<start>   ::=  <B>
<B>       ::=  <B> and <B> | <B> or <B> | not ( <B> and <B> )
               | not ( <B> or <B> ) | <var>
<var>     ::   = b0|b1|b2|b3|b4
```

Max(py)

```
<start>     ::=  <defp>::<callp>
<defp>      ::=  def p()::x = 0.0::<code>::return x:
<callp>     ::=  XXX_output_XXX = p()

<code>      ::=  x = <expr> | for i in <seq>::<code>:
                 | x = <expr> ::<code>| for i in <seq>::<code>:::<code>
<expr>      ::=  <const> | x | (x + <const>) | (x * <const>)
<const>     ::=  0.5
<seq>       ::=  [<csitems>] | range(<i>+1)
<i>         ::=  0 | 1 | 2 | 3 | 4 | 5 | 6
<csitems>   ::=  <item> | <item>, <csitems>
<item>      ::=  <i>
```

Santa Fe Trail

```
<start>       ::=  begin <code> end
<code>        ::=  <line> | <code> <line>
<line>        ::=  <condition> | <op>
<condition>   ::=  ifa begin <opcode> end begin <opcode> end
<opcode>      ::=  <op> | <opcode> <op>
<op>          ::=  tl | tr | mv
```

Multiplexer

```
<start>  ::=  <B>

<B>      ::=  ( <B> ) and ( <B> )
         |    ( <B> ) or ( <B> )
         |    not ( <B> )
         |    if ( <B> ) ( <B> ) ( <B> )
         |    <var>

<var>    ::=  i0 | i1 | i2 | i3 | i4 | i5 | i6 | i7 | i8 | i9 | i10
```

Appendix B

Hyper-parameter List

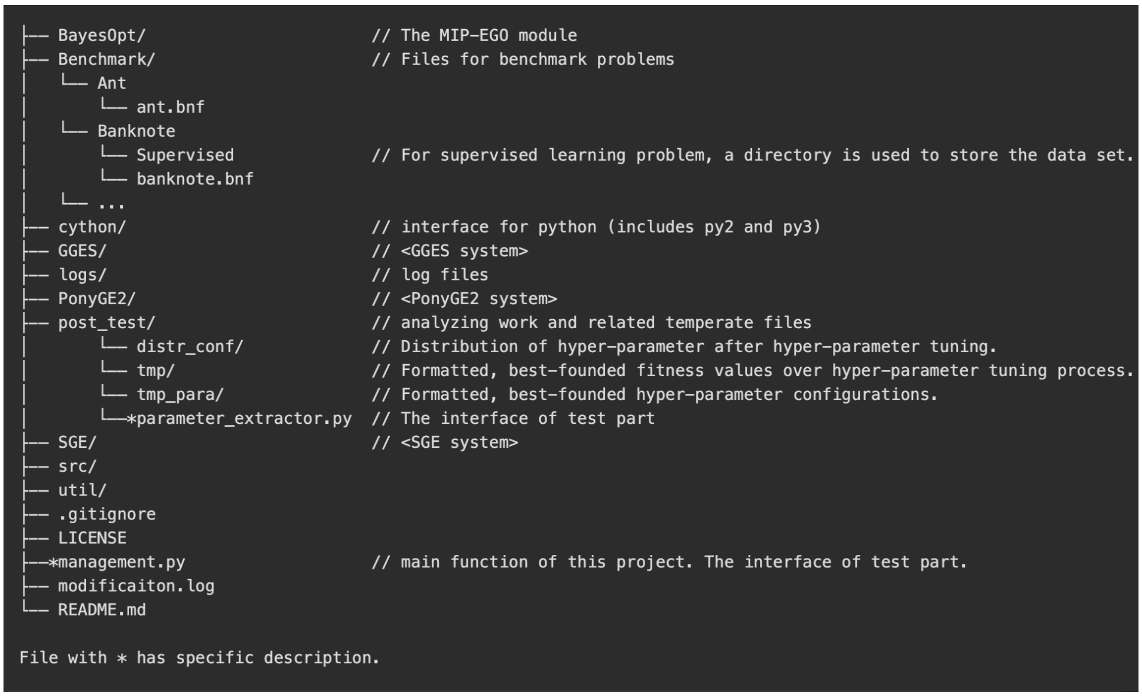
Listing B.1: example hyper-parameter list for PonyGE2 system

```
[{
  " name ":  "  INITIALISATION  ",
  " type ":  "  NominalSpace  ",
  " options ":  [" PI_grow ",  " rhh ",  "  uniform_tree  "]

},{
  " name ":  "  CROSSOVER  ",
  " type ":  "  NominalSpace  ",
  " options ":  ["   variable_onepoint   ",  "   variable_twopoint   ",  **
      **"  fixed_twopoint   ",  "  fixed_onepoint  "]

},{
  " name ":  "   CROSSOVER_PROBABILITY   ",
  " type ":  "  ContinuousSpace  ",
  " range ":  [0,1]

},{
  " name ":  "  MUTATION  ",
  " type ":  "  NominalSpace  ",
  " options ":  ["   int_flip_per_codon   ",  " subtree ",  "   int_flip_per_ind   "]
},{
  " name ":  "   MUTATION_PROBABILITY   ",
  " type ":  "  ContinuousSpace  ",
  " range ":  [0,1]
},{
  " name ":  "   MUTATION_EVENT_SUBTREE   ",
  " type ":  "  OrdinalSpace  ",
  " range ":  [1,5]
},{
  " name ":  "   MUTATION_EVENT_FlIP   ",
  " type ":  "  OrdinalSpace  ",
  " range ":  [1,100]
},{
  " name ":  "   SELECTION_PROPORTION   ",
  " type ":  "  ContinuousSpace  ",
  " range ":  [0,1]
},{
  " name ":  "  SELECTION  ",
  " type ":  "  NominalSpace  ",
  " options ":  [" tournament  ",  "  truncation  "]
},{
  " name ":  "  TOURNAMENT_SIZE  ",
  " type ":  "  OrdinalSpace  ",
  " range ":  [1,50]
},{
  " name ":  "  CODON_SIZE  ",
  " type ":  "  OrdinalSpace  ",
  " range ":   [200,1000]
},{
  " name ":  "   MAX_GENOME_LENGTH   ",
  " type ":  "  OrdinalSpace  ",
  " range ":   [100,500]
},{
  " name ":  "   MAX_INIT_TREE_DEPTH   ",
  " type ":  "  OrdinalSpace  ",
  " range ":  [5,20]
},{
  " name ":  "  MAX_TREE_DEPTH  ",
  " type ":  "  OrdinalSpace  ",
  " range ":   [10,50]
},{
  " name ":  "  POPULATION_SIZE  ",
  " type ":  "  OrdinalSpace  ",
  " range ":   [100,500]
}
]
```

Appendix C

Manual of System

C.1 File Structure



```
├── BayesOpt/                    // The MIP-EGO module
├── Benchmark/                   // Files for benchmark problems
│   └── Ant
│       └── ant.bnf
│   └── Banknote
│       └── Supervised           // For supervised learning problem, a directory is used to store the data set.
│       └── banknote.bnf
│   └── ...
├── cython/                      // interface for python (includes py2 and py3)
├── GGES/                        // <GGES system>
├── logs/                        // log files
├── PonyGE2/                     // <PonyGE2 system>
├── post_test/                   // analyzing work and related temperate files
│       └── distr_conf/          // Distribution of hyper-parameter after hyper-parameter tuning.
│       └── tmp/                 // Formatted, best-founded fitness values over hyper-parameter tuning process.
│       └── tmp_para/            // Formatted, best-founded hyper-parameter configurations.
│       └──*parameter_extractor.py  // The interface of test part
├── SGE/                         // <SGE system>
├── src/
├── util/
├── .gitignore
├── LICENSE
├──*management.py                // main function of this project. The interface of test part.
├── modificaiton.log
└── README.md

File with * has specific description.
```

Figure C.1: File Structure

C.2 Extend your benchmark

1. Write your own fitness function and interface

1. Write your fitness function `int evaluate_[problem_name](argv[])` or `#include` your `’problem.c’` in `’cython/fitness.h’`.
2. Add the declaration of your evaluate function under `cdef extern from "fitness.h"` : in `’interface.pyc’`.
3. Add the interface function for your test system language, following the scheme of `eval_[problem](argv[])`

Example(cython for python2 and python3):

```
cdef extern from "fitness.h" :
    double rmse(double *prediction_value, double *actual_value,int length);

def fitness_rmse(np.ndarray[double, ndim=1, mode="c"] y not None, np.ndarray[double,
     ndim=1, mode="c"] yhat not None):
    result = rmse(<double*> np.PyArray_DATA(y),<double*> np.PyArray_DATA(yhat),y.shape[0])
    return result
‘‘‘
```

2. Add new problem to test system

For PonyGE2 system:

To run your own problem in PonyGE2 system, you still need to 1. add a fitness class under `PonyGE2/src/fitness`, following the schema of

```
from fitness.base_ff_classes.base_ff import base_ff
from fitness.cython.interface import your_fitness_function
class problem_name(base_ff):

    def __init__(self):
        super().__init__()

    @eval_counter
    def evaluate(self, ind, **kwargs):
        return your_fitness_function(**kwargs)
```

2. add a parameter text file under `PonyGE2/src/parameters` 3. copy the BNF file of your problem into `PonyGE2/src/grammars`

For SGE system:

To run your own problem in SGE system,you need to write a scirpt python file for each problem you are going to test. Following code is a good template to start with.

```
import sys,os
sys.path.append(os.path.dirname(os.path.abspath(__file__)) + '/../')
from util.cython.interface import your_fitness_function

class PROBLEM_NAME:
    def evaluate(self, individual):
        error=your_fitness_function(**kwargs)
        return (error, {})


if __name__ == "__main__":
    import core.grammar as grammar
    import core.sge
    experience_name = "WHATEVER/"
    grammar = grammar.Grammar("../grammars/BNF_NAME", 5)
    evaluation_function = PROBLEM_NAME()
    core.sge.evolutionary_algorithm(grammar = grammar, eval_func=evaluation_function,
        exp_name=experience_name)
```

For SGE system:

1. For the test in GGES system, you need to write a test program for your selected problem (in `/demo`),copy your bnf into `/bnf` directory and assign your fitness function’s return value to `eval` function respectively.

2. For the reference of these work, please mimic the `/demo/template.c` file.
3. After finishing this part, you need to modify the `Makefile` by adding some declaration of your problem like following examples.

```
 BIN:
++  $(BINDIR)/YOUR_PROBLEM
++  $(BINDIR)/YOUR_PROBLEM: $(DEMO_OBJS) $(OBJDIR)/YOUR_PROBLEM.o $(LIB)
++ @echo linking $@ from $^
++ @$(CC) $(CFLAGS) $^ -o $@ $(LFLAGS)
‘‘‘
```

Since the GGES system has some problem to pass its standard output to python3, this part is not used in the thesis project.

C.3 Description of important files

Testing Part

Here, the details of the test module will be introduced from bottom to up. At the bottom of the system, the GE system needs to execute different basic problem and constantly tune its hyper-parameter. The way system got parameters is different by systems, for instance, PonyGE2 gets parameters by command and SGE takes configuration from the configuration file. We read in all hyper-parameters and feed them into system. The MIP-EGO module is running on the top of the testing system and asking for a 'Fitness value' as an indicator for hyper-parameter tuning, this value is extracted from the standard output of test systems. The iteration of "MIP-EGO gives configuration to test system" and "test system give objective value" will last an iteration number of times, which is 100 in this project and can be modified by users. Following codes shows how these works:

```
#FILE:Management.py
def run_XXX(cmd):
    '''
    :param cmd: command to run system
    :return: extracted final fitness value from standard output.
    '''
    ...
    return fitness

def obj_func(x):
    '''
    This fucntion feed hyper-parameters into system and execute it.
    :param x: hyper-paramater setting.
    :return: objective value for hyper-parameter tuning.
    '''
    ...

    # same configuration will be executed 5 times and took the average value as objective
        value
    pool = Pool(processes=5)
    for i in range(5):
        result.append(pool.apply_async(run_sge, args=(cmd,)))
    for i in result:
        fitness = i.get()
        if fitness == np.nan:
            err += 1
        else:
            valid_result.append(float(fitness))
    pool.close()
    f = np.average(valid_result)
    dev=np.std(valid_result)
    ...

    return f


def hyper_parameter_tuning_sge(n_step,n_init_sample,eval_type,
     max_eval_each,problem_set,para_list):
    '''

    :param n_step:
    :param n_init_sample:
    :param eval_type:
    :param max_eval_each:
    :param problem_set:
    :param para_list:
    :return:
    '''
    ...
    for problem in problem_set:
        ...
        opt = BO(search_space, obj_func, model, max_iter=n_step,
                 n_init_sample=n_init_sample,
                 n_point=1,
                 n_job=1,
                 minimize=minimize_problem,
                 eval_type=eval_type,
                 verbose=True,
                 optimizer='MIES')
        xopt, fitness, stop_dict = opt.run()
        ...
```

On a higher level of test module, we implemented test classes for every independent test systems, who works as the controller of the automatic test for specific system. The name of each test class follows the way of **Tester_*SystemName***, such like **Tester_PONYGE2**. For every test class, they are generally composed by several same methods:

```
 FILE:Management.py
class Tester_XXX:
    def __init__(self,n_step,n_init_sample,eval_type,max_eval_each,para_list=...):
        '''
        :param n_step: Iteration number of hyper-parameter tuning.
        :param n_init_sample: Initialization point number for hyper-parameter tuning.
        :param eval_type: Evaluation type.
        :param max_eval_each: Maximal allowed evaluation function calling number.
        :param para_list: Location of tuned hyper-parameter list in json.
        '''
        ...
    def make_interface(self):
        # copy the interface into src/fitness and build
        ...
    def refresh_interface(self):
        # refresh the interface for system
        ...
    def clear_log(self):
        # clear previous result and log files
        ...
    def give_problem(self,problem_set):
        # assign problem to this test, para 'problem_set' in list.
        ...
    def run_PonyGE2(self):
        # run the test(including hyper-parameter tuning)
        hyper_para_tuning_XXX.hyper_parameter_tuning_GGES(self.n_step, self.n_init_sample,
            self.eval_type, self.max_eval_each, self.problem_set,
                                                 self.para_list)
    def make_problem(self):
        # copy necessary files for test problem
        ...
```

On the top of test module, a 'Test manager' is in charge of the automatic comparison between systems. It ensures all tester share same configuration to run the test and also provides a simple interfaces to users.

```
#FILE:src/hyper_parameter_tuning.py
class TesterManager:
    def
         __init__(self,test_systems,test_problems,n_step,n_init_sample,eval_type,max_eval_each,para_dict):
        ...
    def run(self):
        base = os.getcwd()

        # clean previous test result
        global_log_cleaner()
        if 'PonyGE2' in self.test_systems:
            # ****Test PonyGE2*****
            tester = Tester_PONYGE2(n_step = n_step,
                                    n_init_sample = n_init_sample,
                                    eval_type=eval_type,
                                    max_eval_each=max_eval_each,
                                    para_list=self.para_dict+'/hyper_para_list_PonyGE2.json'
                                    )
            tester.give_problem(self.test_problems)
            tester.make_interface()
            tester.refresh_interface()
            tester.run_PonyGE2()
            os.chdir(base)

        if 'SGE' in self.test_systems:
            # ****Test SGE*****
            ...

        if 'GGES' in self.test_systems:
            # *****Test GGES*****
            ...

        print("All test finished, now quitting...")
```

To run a complete test, it is necessary to define all requested parameters in *"management.py"*. Following codes is a good example for running a test. In order to extract data conveniently, it is recommend to run the test by using command like *"nohup python3 management.py >logs/output_[MachineName].txt 2>&1 &"*

```
if __name__ == "__main__":
    global_log_cleaner() # clean previous test result

    #here to define the problem for the comparison
    full_problem_set=['ant','string_match','vladislavleva4','mux11']

    #shared parameters
    n_step=10
    n_init_sample=5
    eval_type='dict'
    max_eval_each=50000
    test_sys=['SGE','PonyGE2']

    test=TesterManager(test_sys,part_problem_set,n_step,n_init_sample,eval_type,max_eval_each,'/util')
    test.run()
\label{testing}
```

On the other side, the analyzing module is responsible for data extraction from generated log files and analyzing work. One point need to be mentioned is that, the usage of this module is only effective with specific naming method for generated log files, which is produced by previously mentioned command. In compare with the great variability of the analyzing work, the need of using parameter extraction is stable. Following codes specified how parameter extractor is implemented, and leaves analyzing part to users as it may be different from person to person according to their demands. Meanwhile, the two methods used for generating statistical graph in this project is also included, they are used for generating the distribution of hyper-parameter after hyper-parameter tuning and the comparison graph between different systems.

```
class PARAMETERS_EXTRACTOR:
    """
    This class is used to extract data from log files generated by running management.py.
    """

    def __init__(self, dir, problem_set):
        '''
        :param dir: location of log
        :param problem_set: tested problem
        '''
        ...

    def getpara(self, system_name):
        '''
        This funciton is used to extract the name of tuned HYPER_PARAMETER in the test, by
            reading the json file which
        control hyper-parameters in.
        :param system_name: Name of tested system
        :return: A list of tuned parameters
        '''
        ...

    def output_analyzer(self, f):
        """
        This funciton extract data from std_out of test, which start with 'output'.
        std_out are stored in those files with the name of 'output_MACHINE_NAME_IDNEX.txt'.
        :param f: files
        :return: A list includes fitness value in each iteration of tuning process for
            every systems and problems
           A List with following structure:[level1,level1,[...]]
           Level1:(SYSTEM_NAME,[level2],[level2],[...])
           Level2:[PROBLEM_NAME,[level3],[level3],[...]]
           Level3:[[Iteration_number,Fitness_value],[Iteration_number,Fitness_value],[...]]
        """
        ...

    def out_analyzer(self, system_name, f,type):
        '''
        In the benchmark test, for every problem the system will store its best-founded
            hyper-parameters settings, under
        the name of out_SYSTEMNAME_PROBLEM_TIME_MACHINENAME.txt. They are generated by
            class Tester.
        This funciton is used to extract the best founded hyper-parameter setting stored
```

```
            in one file f.
        A dict includes the parameter_name and it’s value will be returned.
        :param system_name:
        :param f:
        :param type: returned type, can be ’list’ or ’dict’
        :return: A list/dict of Tuned configuration
        '''
        ...


    def csv_writer(self, data):
        '''
        This function is used to transfer data into .csv file for further usage.
        :param data:
        :return:
        '''
        ...

    def run(self):
        '''
        Run though all files in target_dir and extract data according to the content it
            have.
        This produces two type of files:
            1. For each problem and system, a .csv file records all tuned
                hyper-parameters."Configuration_[SYSTEM]_[PROBLEM].csv"
            2. For each problem and system, a .csv file records all fitness value in each
                iteration of hyper-parameter tuning.
        '''
        ...
```

Analyzing Part

In compare with the great variability of the analyzing work, the need for using parameter extraction is relatively stable. Following codes specified several main methods in this part.

```
#FILE:post_test/parameter_extractor.py
class PARAMETERS_EXTRACTOR:
    """
    This class is used to extract data from log files generated by running management.py.
    """

    def __init__(self, dir, problem_set):
        '''
        :param dir: location of log
        :param problem_set: tested problem
        '''
        ...

    def getpara(self, system_name):
        '''
        This funciton is used to extract the name of tuned HYPER_PARAMETER in the test, by
            reading the json file which
        control hyper-parameters in.
        :param system_name: Name of tested system
        :return: A list of tuned parameters
        '''
        ...

    def output_analyzer(self, f):
        """
        This funciton extract data from std_out of test, which start with ’output’.
        std_out are stored in those files with the name of 'output_MACHINE_NAME_IDNEX.txt'.
        :param f: files
        :return: A list includes fitness value in each iteration of tuning process for
            every systems and problems
           A List with following structure:[level1,level1,[...]]
           Level1:(SYSTEM_NAME,[level2],[level2],[...])
           Level2:[PROBLEM_NAME,[level3],[level3],[...]]
           Level3:[[Iteration_number,Fitness_value],[Iteration_number,Fitness_value],[...]]
        """
        ...

    def out_analyzer(self, system_name, f,type):
        '''
        In the benchmark test, for every problem the system will store its best-founded
            hyper-parameters settings, under
        the name of out_SYSTEMNAME_PROBLEM_TIME_MACHINENAME.txt. They are generated by
            class Tester.
        This funciton is used to extract the best founded hyper-parameter setting stored
            in one file f.
        A dict includes the parameter_name and it's value will be returned.
        :param system_name:
        :param f:
        :param type: returned type, can be 'list' or 'dict'
        :return: A list/dict of Tuned configuration
        '''
        ...


    def csv_writer(self, data):
        '''
        This function is used to transfer data into .csv file for further usage.
        :param data:
        :return:
        '''
        ...

    def run(self):
        '''
        Run though all files in target_dir and extract data according to the content it
            have.
        This produces two type of files:
           1. For each problem and system, a .csv file records all tuned
               hyper-parameters."Configuration_[SYSTEM]_[PROBLEM].csv"
           2. For each problem and system, a .csv file records all fitness value in each
               iteration of hyper-parameter tuning.
        '''
        ...

def system_analyzer(target_dir='tmp/', show=False):
    '''
    Draw comparison graph for each problem.
    :param target_dir: target directory stores formatted fitness data over the
        configuration tuning.
    :return:
    '''
    ...

def conf_analyzer(target_dir='tmp_para/', show=False):
    '''
    Draw the distribution of hyper-parameter after hyper-parameter tuning.
    Used to find some patterns on hyper-parameter tuning.
    :param target_dir: The target directory stores formatted best-found configurations
```

```
        over the configuration tuning.
    :param show:
    :return:
    '''
    ...
```

Here, several information extractors which for log files generated by testing system are implemented, which can extract information from them and, this information will be converted to a formatted data file and stored. For example, for every system and every problem, a *.csv* file will be generated to record the change of fitness value over alliteration and every independent test. By doing this prepossessing work, generated data is much easier to users for analysis work.

Bibliography

[1] Hao Wang, Bas van Stein, Michael Emmerich, and Thomas Back. A new acquisition function for bayesian optimization based on the moment-generating function. In *2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 507–512. IEEE, 2017.

[2] Agoston E Eiben, James E Smith, et al. *Introduction to evolutionary computing*, volume 53. Springer, 2003.

[3] Michael O'Neill and Conor Ryan. Grammatical evolution: Evolutionary automatic programming in a arbitrary language, volume 4 of genetic programming, 2003.

[4] Conor Ryan, Michael O'Neill, and JJ Collins. Introduction to 20 years of grammatical evolution. In *Handbook of Grammatical Evolution*, pages 1–21. Springer, 2018.

[5] James McDermott, David R White, Sean Luke, Luca Manzoni, Mauro Castelli, Leonardo Vanneschi, Wojciech Jaskowski, Krzysztof Krawiec, Robin Harper, Kenneth De Jong, et al. Genetic programming needs better benchmarks. In *Proceedings of the 14th annual conference on Genetic and evolutionary computation*, pages 791–798. ACM, 2012.

[6] Hao Wang, Bas van Stein, Michael Emmerich, and Thomas Bäck. A New Acquisition Function for Bayesian Optimization Based on the Moment-Generating Function. In *Systems, Man, and Cybernetics (SMC), 2017 IEEE International Conference on*, pages 507–512. IEEE, 2017.

[7] Gregor Mendel. Versuche über pflanzenhybriden. In *Versuche über Pflanzenhybriden*, pages 21–64. Springer, 1970.

[8] Hugo De Vries. *Intracellular pangenesis.* Open Court Chicago, 1910.

[9] Wilhelm Johannsen. *Arvelighedslærens elementer.* 1905.

[10] Oswald T Avery, Colin M MacLeod, and Maclyn McCarty. Studies on the chemical nature of the substance inducing transformation of pneumococcal types: induction of transformation by a desoxyribonucleic acid fraction isolated from pneumococcus type iii. *Journal of experimental medicine*, 79(2):137–158, 1944.

[11] Alfred D Hershey and Martha Chase. Independent functions of viral protein and nucleic acid in growth of bacteriophage. *The Journal of general physiology*, 36(1):39–56, 1952.

[12] Seymour Benzer. Fine structure of a genetic region in bacteriophage. *Proceedings of the National Academy of Sciences of the United States of America*, 41(6):344, 1955.

[13] Francis HC Crick. On protein synthesis. In *Symp Soc Exp Biol*, volume 12, page 8, 1958.

[14] Sarah Leavitt. *Deciphering the genetic code: Marshall Nirenberg.* Office of NIH History, 2004.

[15] Chris A Kaiser, Monty Krieger, Harvey Lodish, and Arnold Berk. *Molecular cell biology.* WH Freeman, 2007.

[16] Jürgen Brosius. The fragmented gene. *Annals of the New York Academy of Sciences*, 1178(1):186–193, 2009.

[17] Matt Ridley and Paul Matthews. *Genome.* Howes, 2000.

[18] Wolfram Saenger. *Principles of nucleic acid structure.* Springer Science & Business Media, 2013.

[19] Jeremy M Berg, John L Tymoczko, and Lubert Stryer. Biochemistry, ; w. h, 2002.

[20] Finn Werner and Dina Grohmann. Evolution of multisubunit rna polymerases in the three domains of life. *Nature Reviews Microbiology*, 9(2):85, 2011.

[21] Joshua B Plotkin, Jonathan Dushoff, Michael M Desai, and Hunter B Fraser. Codon usage and selection on proteins. *Journal of molecular evolution*, 63(5):635, 2006.

[22] OpenStax. Ribosomes and protein synthesis, Oct 2016.

[23] M Nirenberg, P Leder, M Bernfield, R Brimacombe, J Trupin, F Rottman, and C O'neal. Rna codewords and protein synthesis, vii. on the general nature of the rna code. *Proceedings of the National Academy of Sciences of the United States of America*, 53(5):1161, 1965.

[24] What is genetic variation?, Sep 2017.

[25] Alfred G Knudson. Mutation and cancer: statistical study of retinoblastoma. *Proceedings of the National Academy of Sciences*, 68(4):820–823, 1971.

[26] Yael T Aminetzach, J Michael Macpherson, and Dmitri A Petrov. Pesticide resistance via transposition-mediated adaptive gene truncation in drosophila. *Science*, 309(5735):764–767, 2005.

[27] Scott W Doniger, Hyun Seok Kim, Devjanee Swain, Daniella Corcuera, Morgan Williams, Shiaw-Pyng Yang, and Justin C Fay. A catalog of neutral and deleterious polymorphism in yeast. *PLoS genetics*, 4(8):e1000183, 2008.

[28] Thomas Hunt Morgan. Sex limited inheritance in drosophila. *Science*, 32(812):120–122, 1910.

[29] Federica Turriziani Colonna. Barbara mcclintock's transposon experiments in maize (1931–1951). *Embryo Project Encyclopedia*, 2017.

[30] James D Watson. *Molecular biology of the gene.* Pearson Education India, 2004.

[31] W-H Li, Dan Graur, et al. *Fundamentals of molecular evolution.* Sinauer Associates, 1991.

[32] Ronald Aylmer Fisher. *The genetical theory of natural selection: a complete variorum edition.* Oxford University Press, 1999.

[33] Susumu Ohno. *Evolution by gene duplication.* Springer Science & Business Media, 2013.

[34] University of Michigan. Evolution and natural selection. `https://globalchange.umich.edu/globalchange1/current/lectures/selection/selection.html`. Accessed May 21, 2019.

[35] David B Fogel. *Evolutionary computation: the fossil record.* Wiley-IEEE Press, 1998.

[36] Lawrence J Fogel. Artificial intelligence through a simulation of evolution. In *Proc. of the 2nd Cybernetics Science Symp., 1965*, 1965.

[37] Lawrence J Fogel, Peter J Angeline, and David B Fogel. Approach to self-adaptation on finite state machines. In *Evolutionary Programming IV: Proceedings of the Fourth Annual Conference on Evolutionary Programming*, volume 355. MIT Press, 1995.

[38] John H Holland. Genetic algorithms and the optimal allocation of trials. *SIAM Journal on Computing*, 2(2):88–105, 1973.

[39] John Henry Holland et al. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.* MIT press, 1992.

[40] Wolfgang Banzhaf, Peter Nordin, Robert E Keller, and Frank D Francone. *Genetic programming: an introduction*, volume 1. Morgan Kaufmann San Francisco, 1998.

[41] John R Koza and John R Koza. *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press, 1992.

[42] Noam Chomsky. Formal properties of grammars. *Handbook of Math. Psychology*, 2:328–418, 1963.

[43] Alfonso Ortega, Marina De La Cruz, and Manuel Alfonseca. Christiansen grammar evolution: grammatical evolution with semantics. *IEEE Transactions on Evolutionary Computation*, 11(1):77–90, 2007.

[44] Conor Ryan. Grammatical evolution tutorial. In *Proceedings of the 12th annual conference companion on Genetic and evolutionary computation*, pages 2385–2412. ACM, 2010.

[45] Michael O'Neill and Conor Ryan. Grammatical evolution. *IEEE Transactions on Evolutionary Computation*, 5(4):349–358, 2001.

[46] Conor Ryan, John James Collins, and Michael O Neill. Grammatical evolution: Evolving programs for an arbitrary language. In *European Conference on Genetic Programming*, pages 83–96. Springer, 1998.

[47] Lewin Benjamin. Genes vii, 2000.

[48] Franco Mascia, Manuel López-Ibáñez, Jérémie Dubois-Lacoste, and Thomas Stützle. Grammar-based generation of stochastic local search heuristics through automatic algorithm configuration tools. *Computers & operations research*, 51:190–199, 2014.

[49] Conor Ryan and R Muhammad Atif Azad. Sensible initialisation in grammatical evolution. In *GECCO*, pages 142–145. AAAI, 2003.

[50] David Fagan, Michael Fenton, and Michael O'Neill. Exploring position independent initialisation in grammatical evolution. In *2016 IEEE Congress on Evolutionary Computation (CEC)*, pages 5060–5067. IEEE, 2016.

[51] Dirk Schweim, Ann Thorhauer, and Franz Rothlauf. On the non-uniform redundancy of representations for grammatical evolution: The influence of grammars. In *Handbook of Grammatical Evolution*, pages 55–78. Springer, 2018.

[52] Michael Fenton, James McDermott, David Fagan, Stefan Forstenlechner, Erik Hemberg, and Michael O'Neill. Ponyge2: Grammatical evolution in python. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 1194–1201. ACM, 2017.

[53] James McDermott Erik Hemberg. Ponyge.

[54] M Nicolau and D Slattery. libge-grammatical evolution library (2006).

[55] Michael O'Neill, Erik Hemberg, Conor Gilligan, Eliott Bartley, James McDermott, and Anthony Brabazon. Geva: grammatical evolution in java. *SIGEVOlution*, 3(2):17–22, 2008.

[56] Farzad Noorian, Anthony Mihirana de Silva, Philip HW Leong, et al. gramevol: Grammatical evolution in r. *Journal of Statistical Software*, 71(1):1–26, 2016.

[57] Pavel Suchmann. Grammatical evolution ruby exploratory toolkit.

[58] Nuno Lourenço, Francisco B Pereira, and Ernesto Costa. Unveiling the properties of structured grammatical evolution. *Genetic Programming and Evolvable Machines*, 17(3):251–289, 2016.

[59] Franz Rothlauf and Marie Oetzel. On the locality of grammatical evolution. In *European Conference on Genetic Programming*, pages 320–330. Springer, 2006.

[60] Nuno Lourenço, Filipe Assunção, Francisco B Pereira, Ernesto Costa, and Penousal Machado. Structured grammatical evolution: a dynamic approach. In *Handbook of Grammatical Evolution*, pages 137–161. Springer, 2018.

[61] Peter A Whigham, Grant Dick, James Maclaurin, and Caitlin A Owen. Examining the best of both worlds of grammatical evolution. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 1111–1118. ACM, 2015.

[62] Peter A Whigham et al. Grammatically-based genetic programming. In *Proceedings of the workshop on genetic programming: from theory to real-world applications*, volume 16, pages 33–41, 1995.

[63] M. D. McKay, R. J. Beckman, and W. J. Conover. A comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics*, 21(2):239–245, 1979.

[64] Hao Wang, Michael Emmerich, and Thomas Bäck. Cooling strategies for the moment-generating function in bayesian global optimization. In *2018 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8, July 2018.

[65] Maarten Keijzer. Improving symbolic regression with interval arithmetic and linear scaling. In *European Conference on Genetic Programming*, pages 70–82. Springer, 2003.

[66] Robin Harper. Spatial co-evolution: quicker, fitter and less bloated. In *Proceedings of the 14th annual conference on Genetic and evolutionary computation*, pages 759–766. ACM, 2012.

[67] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017.