Indirect Estimation of Distribution Algorithms for the Evolution of Tree-Shaped Structures

by

Elmira Ghoulbeigi

B.Sc. Azad University, 2007

A thesis

presented to Ryerson University

in partial fulfillment of the

requirements for the degree of

Master of Science

in the Program of

Computer Science

Toronto, Canada, 2010

Author's Declaration

Indirect Estimation of Distribution Algorithms for the Evolution of Tree-Shaped Structures

Elmira Ghoulbeigi

M. Sc. in Computer Science, 2010

Ryerson University, Toronto, Canada

Abstract

This thesis explores indirect estimation of distribution algorithms (IEDAs) for the evolution of tree structured expressions. Unlike conventional estimation of distribution algorithms, IEDAs maintain a distribution of the genotype space and indirectly search the solution space by performing a genotype-to-phenotype mapping.

In this work we introduce two IEDAs named PDPE and N-gram GEP. PDPE induces a population of programs, encoded as fixed-length gene expression programming (GEP) chromosomes, by iteratively refining and randomly sampling a probability distribution of program instructions. N-gram GEP attempts to capture regularities in GEP chromosomes by sampling the probability distribution of triplet of instructions (3-grams).

We tested the performance of these systems using a variety of non-trivial test problems, such as symbolic regression and the lawn-mower problem. We compared PDPE and N-gram GEP with their predecessors, probabilistic incremental program evolution (PIPE) and N-gram GP, and the canonical GEP algorithm. The results proved that our methodology is more efficient than PIPE and the canonical GEP algorithm.

Acknowledgements

I would like to offer my sincerest gratitude to the following people who have helped and inspired me throughout my master studies:

My supervisor, Dr. Marcus Vinicius dos Santos for his invaluable guidance and for being the role model of supervision. He spent countless hours editing this document and shaping my ideas. In addition, his enthusiasm, constructive criticism and, intellectual perfectionism contributed to my growth as a student and, a future researcher. Thank you Marcus, I am indebted to you more than you know.

My father and my mother, without whom this thesis would not have been possible. I am forever grateful to them for being a constant source of unconditional support and for empowering my goals at each turn of the road. Mom and dad I am honoured to have you as my parents.

The committee members, Dr. Eric Harley, Dr. Alireza Sadeghian and, Dr. Saeed Zolfaghari whose reviews helped improve this dissertation. I specially would like to thank Dr. Harley for his insightful comments on my writings and for patiently helping me whenever I barged into his office. I am also grateful to the department of Computer Science, chaired by Dr. Sadeghian, for generously supporting my trip to Switzerland where I attended ACM SAC'10 and presented "Probabilistic Developmental Program Evolution".

My aunt and colleague Shahnaz Sadeghian Rizi and my friend Delnavaz Mobedpour, for their precious assistance when it was most required. Shahnaz and Delnavaz I cannot thank you enough!

In conclusion, I deeply thank God for making this academic journey unforgettable by surrounding me with all these amazing people.

Dedication

For my father Abbas Gholbeigi, my mother Elahe Sadeghian Rizi, and my sister Elnaz.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The central hypothesis of this thesis is that an indirect estimation of distribution algorithm (IEDA) improves the evolvability of tree-structured expressions representing solutions for a certain class of non-trivial problems such as symbolic regression and the lawn-mower problem. The IEDA takes advantage of a search strategy that maintains a distribution of fixed-length linear strings encoding tree-shaped candidate solutions, and *indirectly* searches for the actual tree structures by performing a string-to-tree decoding step.

1.1 Motivation

Estimation of Distribution Algorithms (EDAs) are a class of evolutionary methods which approximate the probability distribution of good solutions found so far and sample that distribution to create new candidate solutions. These probabilistic algorithms can be broadly categorized into two classes: the first class, here called EDA genetic algorithms (EDA-GA), represents solution individuals as fixed-length binary strings, like in Genetic Algorithms (GAs). The other class, here called EDA genetic programming (EDA-GP), represents solutions as variable-sized tree structures, like in Genetic Programming (GP).

GAs use fixed-length binary strings to encode candidate individuals. More specifically, this methodology implicitly searches for solution individuals by using biologically inspired search operators, called *genetic operators*, that explicitly explore all the possible binary strings that encode the actual solutions. In GP, genetic operators *directly* explore the tree-shaped structures of the solution individuals. Moreover, in both GAs and GP there is a one-to-one mapping relationship between the encoding of a solution individual–an expression which is in turn evaluated in light of a fitness function–and the encoding of the structures on which the genetic operators operate. Similar to GA and GP, both EDA-GA and EDA-GP also rely on direct mappings for representations. It has been shown, however, that evolutionary systems which use a many-to-one mapping between representations offer more flexibility for finding better solutions [2]. Many-to-one mapping is also commonly found in the DNA/Protein translation. In effect, there are two major types of code sequences in the DNA: coding sequences which are transformed to proteins and non-coding sequences (introns) which are not transformed to proteins. Since DNA contains various combinations of coding and non-coding sequences, different sequences of codes might be translated to the same protein, yielding a many-to-one mapping between the DNA and proteins.

The one-to-one mapping used in both EDA-GA and EDA-GP imposes one of the following limitations on these types of systems: In EDA-GA, the fixed-length structure of individuals is easy to manipulate, but it limits the variety of functions that these entities are able to represent; in EDA-GP, variable-length trees incorporate the desired amount of functional complexity, but the hierarchical structure of these individuals is difficult to reproduce via genetic operators. In this work we aim to overcome the above limitation of traditional probabilistic algorithms by emulating the many-to-one representation mapping used in natural evolution. Therefore, we explore indirect EDA techniques which advance their search procedure by directly searching for fixed-length strings and indirectly exploring the tree structures through the application of a string-to-tree mapping step.

1.2 Methodology

To achieve this goal, in a pilot study we first explored the capabilities of a univariate EDA using an indirect solution representation. Specifically, we integrated the PIPE probabilistic algorithm [30] with the indirect encoding used in Gene Expression Programming (GEP). The new algorithm, called Probabilistic Developmental Program Evolution (PDPE) [9], induces a population of programs, encoded as GEP individuals, by iteratively refining and randomly sampling a probability distribution of program instructions stored in an array of probabilities, called PPC. As in PIPE, we refine the probabilities stored in PPC by increasing the probabilities of the best members of the population. In addition, to efficiently explore the search space we mutate single probabilities in the PPC.

We compared PDPE with PIPE and the canonical GEP algorithm on a function regression and on the 6-bit parity problem. We assessed the performance of the PDPE algorithm in terms of fitness variance and solution quality: PDPE outperformed PIPE in terms of fitness variance and solution quality. It also outperformed GEP in terms of solution quality but not fitness variance.

Our next step was to extend the above approach to a multivariate EDA that also takes advantage of an indirect solution representation. In this case, inspired by the N-gram GP algorithm introduced in [26], we generate GEP individuals using an *N-gram* [20, p. 192–195] as our probabilistic model. This prototype holds the probabilistic dependencies amongst triplets of elements and is adapted overtime according to the elite members of the current population as described in [26].

We compared the new methodology, called N-gram GEP, with the PDPE algorithm on the function regression and the 6-bit parity problems. The N-gram GEP methodology outperformed PDPE both in terms of variance and solution quality. We then compared the N-gram GEP algorithm against the canonical GEP methodology on the polynomial and the lawn-mower problems. In this case to, N-gram GEP showed a better performance than the GEP algorithm in terms of solution quality and scalability. Finally we measured the performance of N-gram GEP against the N-gram GP algorithm in terms of pattern preservation. Our results, in this case, did not show any clear evidence of substantial performance difference.

The rest of this thesis is organized as follows: In the next section we briefly introduce the terminology pertaining to this thesis. In Chapter 2 we review the literature pertinent to this work and describe our methodology in Chapter 3. We then present our experimental results and analyze them in Chapter 4. Finally, we conclude this document in Chapter 5, where we also present our future research directions.

1.3 A Few Words on Terminology

Broadly speaking, Evolutionary Computation (EC) emerges from the application of evolutionary biology in Computer Science. For this reason, the terminology used in EC owes to each of these fields.

In the above paragraphs, we have purposely used terms from the computer science jargon. We mentioned strings when referring to encoded problem solutions; we mentioned trees when referring to data structures that emulate a hierarchical tree structure denoting an expression; and we mentioned string-to-tree mapping when referring to the decoding from string to tree representation. In the following, we introduce the basic EC terminology used in this work.

EC is a machine learning methodology that searches a problem space consisting of a set of candidate problem solutions called a *population*. An element of this set is called an *individual*. In EC, *genotype* or *genome* refers to the encoding used to represent an individual. Genotype structure may vary depending on the EC methodology. For example, genetic algorithms (GAs) [14] represent individuals as binary strings, while GP [17] uses LISP [22] expressions, GEP [7] uses strings over a finite alphabet, and cartesian genetic programming [21] uses arrays of integer numbers. *Phenotype* or *phenome*, in evolutionary computation, is the structure that undergoes fitness evaluation. Some EC methodologies have distinguished genotypes and phenotypes: in GAs, for example, the genome is the fixed-length binary string which encodes the phenotype while the phenome is the structure which represents a candidate solution, *e.g.*, a mathematical expression, a multidimensional array, a graph, and the like. This is also the case in GEP where the genome is a fixed-length chromosome while the phenotype is a variable-size tree structure denoting an *expres-*

sion tree. Other EC methodologies, such as GP, use a unique structure as both the genotype and phenotype; in GP, LISP expressions represent both the genome and the phenome of an individual.

Similar to biological systems the genome, also called *chromosome*, maintains the genetic material of an individual and is directly subject to genetic variation. In addition, the transformation of the genotype to the phenotype is often obtained via a genotype-to-phenotype mapping also called *chromosome expression*.

The *search space* (genotypic space) in EC is define as the set of structures which encode the feasible solutions. The *solution space* (phenotypic space), on the other hand, is composed of the actual solution individuals. In EC methodologies such as GP the search space and the solution space are equivalent. In GAs and GEP, however, the search space is composed of all genotypes and the solution space contains the corresponding phenotypes.

Chapter 2

Background and Related Work

This work concerns estimation of distribution algorithms (EDA) for genetic programming (GP). The study presented here focuses on approaches that maintain a probability distribution of the search space and indirectly search the solution space via a genotype-to-phenotype mapping. This chapter reviews the following EC methodologies on which this thesis is built: Genetic Programming (GP), the forefather of EC methodologies that evolve tree-shaped solutions of problems; Gene Expression Programming (GEP), an evolutionary technique which manipulates linear genomes and indirectly searches for tree-shaped solutions via a decoding of the genotype; and EDAs for GP methodologies.

2.1 Evolutionary Computation

Since this work is about exploring evolutionary computation (EC) techniques for inducing tree-shaped, program-like structures, we deem appropriate to start this chapter with a brief historical overview of EC.

The idea of considering evolution as a computational process was initially formed in the 1930s [16]. Only in the 1960s, however, empowered by the availability of inexpensive digital computers as a modelling tool, work in the field actually flourished. The following are some prominent works of that period: Rechenberg and Schwefel [28], Fogel [8] and, Holland [15]. These initial studies on evolutionary algorithms (EAs), unearthed two main issues: how to effectively represent the characteristics of implementable systems, and how such systems might be used to solve problems.

As a result, much of the EA research in the upcoming years focused on gaining additional insight into these issues by implementing new techniques or extending the existing methodologies.

In 1966, Fogel et al. [8] introduced Evolutionary Programming (EP), a methodology for developing artificial intelligence using an evolutionary process. The individual representations used in evolutionary programming are usually tailored according to the problem domain. For example, in real-valued optimization problems, individuals consist of fixed-length real-valued vectors; whereas in the traveling salesman problem the individuals are composed of ordered lists. A distinguishing characteristic of EP from other evolutionary approaches such as Evolution Strategies (ES) and GP, is that in this methodology there is usually no exchange of genetic material between individuals and thus the main genetic operator used is mutation. The main purpose of ES, developed by Rechenberg [29] in the early 70's, was to solve hydrodynamic optimization problems. For this reason, individuals in this paradigm were represented as vectors of real numbers. Unlike EP, in ES both recombination and mutation operators are used.

In 1975, Holland proposed Genetic Algorithms (GA) as an alternative evolutionary search methodology [14]. The main individual representation used in GA is a fixed-length bit-string. Even though both mutation and recombination are used in GA, the recombination operator is often considered as the primary operator in this algorithm.

Up to early late 1980s the research and development of EA methodologies was done without much interaction among various groups. In early 1990s, however, communication among various EA groups during conferences improved the understanding of different paradigms, their similarities and differences. In addition, scientists from EA communities decided to unify their view by choosing the term "evolutionary computation" as the name of the field. Crossbreading of ideas was another effect of such interactions which contributed to the improvement of existing issues and the creation of new EA categories, such as the ones described in the next sections of this chapter [16].

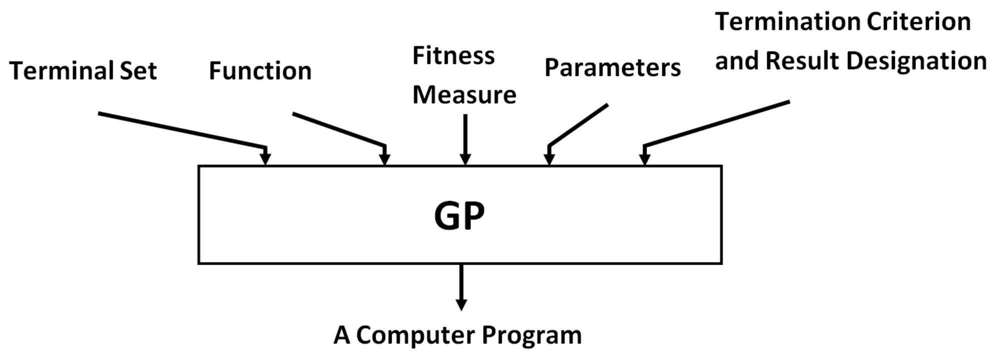


Figure 2.1: GP as a system

2.2 Genetic Programming

Genetic Programming is an evolutionary computation methodology which evolves tree-structured computer programs and mathematical expressions [17]. Seen as a system, GP gets a user-defined problem description and produces a tree-structured computer program or mathematical expression that solves the problem (Figure 2.1 [17]). The 5 inputs that a user is required to provide to a GP system are as follows: The *terminal set* includes the independent variables of a problem; the *function set* contains the primitive functions for each branch of the to-be-evolved program; the *fitness measure* reflects how well an individual is able to solve a given problem while the *control parameters*, such as the population size, control the runs; the *termination criterion* determines when the evolutionary algorithm ends; and in most cases the best individual found so far is also designated as the *result of the run*.

The main genetic operators used in GP are *reproduction*, *mutation* and *crossover*. The reproduction operator simply duplicates an individual and inserts it into the new population. Like in reproduction, the mutation operator is applied on one individual. Specifically, given an individual selected by some selection method, this operator changes a node or a subtree in the parent individual to another node or subtree. The crossover operator combines the genetic materials of two selected parents by exchanging certain parts from both parents.

In canonical GP the search space (genotypic space) and solution space (phenotypic space) are considered equivalent. Developmental approaches, on the other hand, distinguish between the search and solution space. Developmental Genetic Programming (DGP) was initially introduced as an EC methodology which mapped binary strings (genotypes) to more complex structures (phenotypes) [2]. Throughout the years DGP has expanded to include other techniques which use this concept in different EC methodologies.

In [2], Banzhaf states that Developmental Genetic Programming outperforms the canonical GP methodology. There he argues that the genotype-to-phenotype mapping in this approach provides the unrestricted application of the genetic operators and guarantees feasible solutions in the phenotypic space.

2.3 Gene Expression Programming

Gene Expression Programming (GEP) [7] is a developmental genetic programming methodology, *i.e.*, it uses an indirect representation for encoding tree expressions into fixed-length strings of symbols. In addition, the translation from the linear *genotype* (genome) to the hierarchical *phenotype* (expression trees) allows GEP to maintain the advantages of a modifiable, unconstrained genome while preserving the benefits of adaptable phenotype structures for more complex behaviours.

In GEP, each *chromosome* is composed of one or more fixed-length *genes*. Each gene is divided into two consecutive regions: *head* and a *tail* (see Figure 2.2). The head may contain symbols that represent both functions or terminals (variables and constants), whereas the tail contains only terminals. For each problem, the length of the head (H) is decided by the user, whereas the length of the tail (T) is a function of H and the number of arguments of the function with more arguments (N) and can be calculated as $T = H * (N - 1) + 1$. The head-tail structure of a gene ensures that every function has the required number of arguments available, thus ensuring the syntactical correctness of the expressed tree.

In GEP, the genome representation codes for an expression tree. Each GEP gene codes for

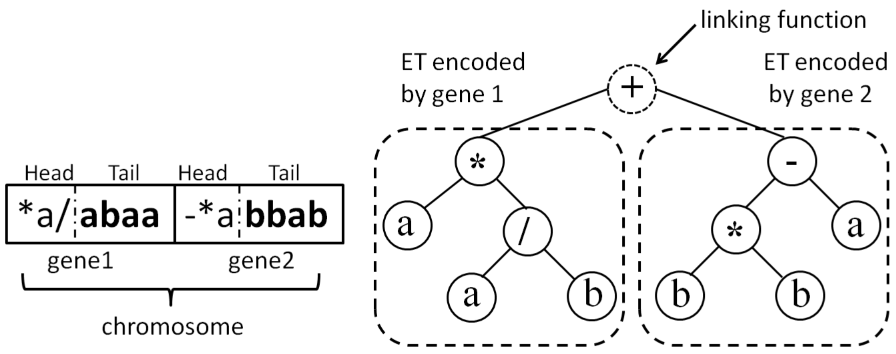


Figure 2.2: A sample chromosome and its expression tree

a tree, expressed by traversing the gene from left to right and building the tree in a breadth-first manner. In multigenic chromosomes, the expressed trees (for each gene) are connected by a linking function, which is also an input parameter of the algorithm. For example, Figure 2.2 shows a GEP multi-genic chromosome and the respective tree it codes for, representing the mathematical expression $a^2/b+b^2-a$.

The genetic operators mostly used by GEP are mutation, transposition and recombination. The mutation operator randomly changes an element of a chromosome into another element. In the head of the gene a function can be changed into another function or terminal and vice versa, while in the tail, a terminal can be only replaced by another terminal. The transposition operator (there are, in fact, three kinds of transposition operators in GEP [7]) randomly moves a segment of the chromosome to another location in the same chromosome. The recombination operator exchanges segments of genetic material between two parent chromosomes. There are three kinds of recombination operators in GEP: One-point recombination, two-point recombination and gene recombination [7].

2.3.1 Handling of Numerical Constants

Solutions to many problems may consist of expressions that contain numerical constants. GP solves the problem of constant creation using a special terminal called *ephemeral random constant*, as follows: For each random constant used in the trees of the initial population, a random number of a special data type in a specific range is generated. Further, these numerical constants are moved around from tree to tree by the application of the crossover operator.

In [7], Ferreira introduced two techniques for handling numerical constants. One uses an extra terminal “?” and an additional domain *Dc* composed of the symbols chosen to represent ephemeral random constants. The *Dc* domain comes after the tail and has a length equal to T. The random constants of each gene are created during the generation of the initial population and kept in an array. The value of each random constant is assigned during gene expression. The other technique directly manipulates the numerical constants and does not include the special facilities proposed in the first technique. In our approach we do not handle random constant as in GEP. Instead, as will be shown in Chapter 3, we draw on the approach introduced in [30] to instantiate the ephemeral random constants.

2.4 Estimation of Distribution Algorithms

Estimation of Distribution Algorithms (EDAs) are population based evolutionary methods which replace genetic operators, the means for exploring the search space, with probabilistic models of promising solutions, and sample that model to create new candidate solutions. The procedure of a typical EDA is as follows: At first, a random population of solutions individuals is generated. Then, better solutions are selected from the initial population and the distribution of this set of individuals is estimated. Finally, new solutions are generated according to this estimate and added to the population. This process is repeated until the termination criterion is met [24].

Early EDAs such as Population Based Incremental Learning (PBIL) [1] and Compact Genetic Algorithm (cGA) [12] assume no interactions among variables in a problem. PBIL uses binary strings of fixed length to encode the solution individuals and replaces the population of solutions with a so-called probability vector which is updated according to a simple incremental rule after performing selection on a set of candidate solutions. Like in PBIL, in cGA the solution population is represented by a single probability vector but it uses a different selection scheme and update rule. This group of EDAs which do not consider any interdependencies among variables work very well for linear problems, nevertheless they experience great difficulty solving problems with strong interactions [24]. The Mutual Information-Maximizing Input Clustering (MIMIC) [3] and the Bivariate Marginal Distribution Algorithm (BMDA) [25] algorithms are examples of early attempts to consider pairwise interactions among variables in a problem. To estimate the distribution of the selected solutions the MIMIC algorithm takes advantage of a simple chain distribution while BMDA uses a set of mutually independent dependency trees for the same purpose. Later approaches address problems that involve higher order interactions amongst variables. The Extended Compact Genetic Algorithm (ECGA) [11], for example, generates new solutions according to the marginal distribution of mutually independent variable sets, whereas the Bayesian Optimization Algorithm (BOA) [23] technique uses a bayesian network to describe the dependency relationship of variables.

From a representational point of view, the above methods work on problems defined on fixed-length strings over a finite alphabet. On the other hand, the class of estimation distribution techniques known as Probabilistic Model-Building Genetic programming (PMBGP) uses methods able to tackle problems where the solutions are computer programs [30]. The Probabilistic Incremental Program Evolution (PIPE) learning algorithm [30] is an earlier PMBGP technique that uses a probabilistic tree to evolve GP-style computer programs. The PIPE methodology accounts only for univariate interactions. Estimation of Distribution Programming (EDP) [37], on the other hand, takes into account interactions among the parent and child nodes in the explicit tree-like structure of GP chromosomes.

Similar to PIPE, the Extended Compact Genetic Programming (ECGP) [31] algorithm generates parse trees. To estimate the marginal distribution of disjoint tree-node clusters, it uses the marginal product models introduced in ECGA [11].

The BOA Programming (BOAP) [19] is an extension of BOA for the evolution of programs trees. To be able to model hierarchical programs through BOA, this algorithm represents program trees in curried form. Although BOAP is capable of capturing multivariate interactions among variables, this approach has two shortcomes: it may generate syntactically invalid programs, and produce a large conditional probability table (CPT) [13].

Since GP uses many types of nodes (e.g. functions and terminals), PMBGPs such as ECGP and EDP suffer from the problem of huge CPT size. A large CPT not only consumes a lot of memory but also requires many samples to construct a proper network model and CPT. In Program Optimization with Linkage Estimation (POLE) [13] bayesian networks were used to estimate the distribution of promising solutions. This approach uses a special type of hierarchical structure called expanded parse tree to overcome the problem of large CPT size of prior PMBGPs. An extended parse tree is a full tree, *i.e.*, all terminals (leaf nodes) are at the same depth (or level). This property of the expanded parse tree reduces the CPT size, as one does not have to take into account the possibility of both function and terminal sets at different depths of the tree.

Another novel EDA is the Probabilistically Guided Gene Expression Programming [6] which uses the differential evolution to evolve the numerical parameters of Hidden Markov Models (HMM). The evolved HMMs are used to generate the candidate solutions in form of Prefix Gene Expression Programming (PGEP) chromosomes. This approach indirectly introduces chromosomal variation by applying the genetic operators directly to the underlying probabilistic model. Therefore, the probabilistic model is considered the genotype while the PGEP chromosome and its associated tree structure play a phenotypic role.

N-gram GP [26] is a recently proposed EDA which evolves linear-GP-type [5] computer programs. This algorithm applies the concept of n-grams from natural language processing to capture regularities in the language necessary to solve a problem. This application yields a smaller model space in which good patterns can be identified with less sampling. Also a higher degree of regularity can be observed in the evolved solution population. Another distinct feature of N-gram GP is that it explicitly models the program length distribution to be used during the search space. This approach intuitively limits the search to programs of manageable size without using any predefined program length limitation.

Grammar model-based EDA-GP is an indirect EDA-GP approach based on Grammar Guided Genetic Programming (GGGP) [34]. The main search mechanism in GGGP is the conventional genetic search. In addition, this method uses a grammar as a formal model to limit the search space. Grammar model-based EDA-GP, on the other hand, uses grammar as a probabilistic model. Grammars were originally proposed to sample the hierarchical structure of natural or formal languages [34].

There are two different variants of the Grammar model-based EDA-GP. The first category of models, which includes algorithms such as Stochastic Grammar-based Genetic Programming (SG-GP) [27], only learns the probability associated with a grammar structure. The second category of models, which includes Bosman's work [4], Program Evolution with Explicit Learning (PEEL) [33], and Grammar Model-based Program Evolution (GMPE) [32], learns both the grammar structure and the probability associated with the grammar. SG-GP [27] is the earliest Grammar model-based EDA-GP. SG-GP uses a distribution model based on stochastic grammars to overcome *bloat*[1]. Since in this method the overall structure of the grammar is fixed and does not change with the progress of the search, this algorithm is either very slow or stops quickly. PEEL [33] addresses this shortcome by introducing a change of grammar structure during the search procedure. GMPE [32] replaces the conventional genetic operators with a probabilistic model called Stochastic Context Free Grammar (SCFG) [20]. The stochastic grammar model in GMPE is updated according to the best individuals in the current population; the new population is then generated by sampling this grammar. GMPE is a highly flexible model and is able to represent various forms of building blocks studied in GP. This algorithm, however, is computationally very expensive.

[1]Bloat, in GP, involves the rapid growth in size of individuals without any fitness improvement.

The work presented in this thesis introduces an indirect EDA approach for evolving tree-structured expressions. Our method combines the developmental representation of GEP with the PIPE and the N-gram GP paradigms and creates two new algorithms called repectively PDPE and N-gram GEP. The new algorithms generate GEP chromosomes using a probabilistic model. This model is an abstract structure which represents the statistical distribution of the population of solution and is updated according to the best solution individuals.

Chapter 3

Methodology and Implementation

As discussed in Chapter 1, the central hypothesis of this work is that the evolution of tree-structured expressions can be improved by an Estimation of Distribution Algorithm (EDA) that takes advantage of an indirect search strategy: the EDA maintains a distribution of the genotype space and *indirectly* searches the solution space via the genotype-to-phenotype decoding.

To verify this hypothesis, first we developed a univariate IEDA named PDPE [9]; this methodology integrates GEP's indirect solution representation with PIPE's learning algorithm [30]. Next we explored the capabilities of an IEDA that employs the same genome representation, but uses an n-gram model for sampling a multivariate probability distribution. The proposed approach, here named N-gram GEP, draws on the N-gram GP evolutionary algorithm introduced in [26] to maintain a probabilistic model of a population of GEP individuals. The next two sections provide a detailed description of these new systems.

3.1 Probabilistic Developmental Program Evolution

PDPE is a novel IEDA which follows PIPE's learning algorithm but uses a different encoding. More specifically, in PDPE population individuals are represented by GEP chromosomes which are transformed into expression trees for fitness calculations. In addition, instead of using the probabilistic tree structure (PPT) (Figure 3.1) presented in [30], PDPE stores the distribution of promising solutions in a fixed-length probabilistic prototype chromosome (PPC) (Figure 3.2). Each position in the PPC contains a probability distribution over the instruction set, as well as a random

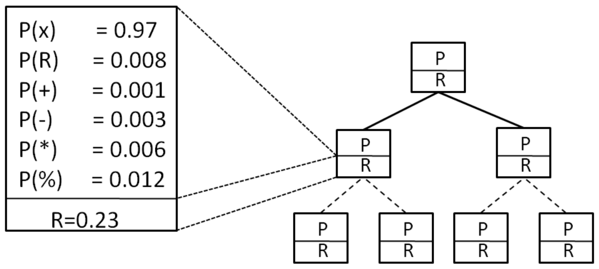


Figure 3.1: PPT

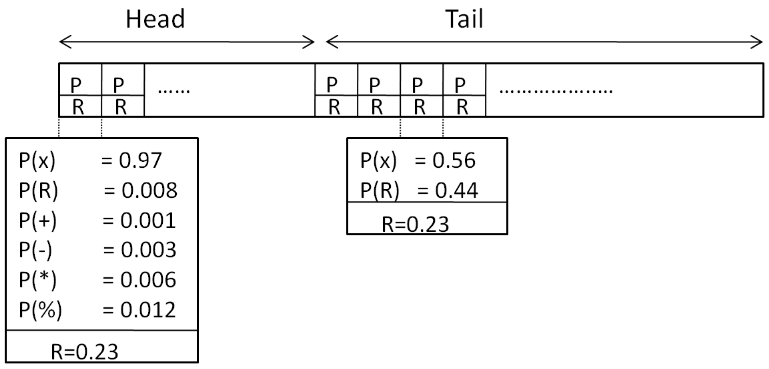


Figure 3.2: PPC with a single gene

constant.

Unlike PIPE, which grows and shrinks the PPT to deal with programs of varying sizes, PDPE does not need to prune the PPC, as it uses the same fixed-length structure of a GEP chromosome. Furthermore, because of its fixed-sized structure, the PPC elements are initialized all at once, rather than "on demand" as in PPT nodes.

3.1.1 Individual Representation & Generation

This section introduces PDPE's problem representation and describes how new individuals are created according to the PPC.

Individual Instructions. Chromosome instructions belong to two disjoint subsets, a function set $F = \{f_1, f_2, ..., f_k\}$ containing k functions of arity equal or greater than 1, and a terminal set $T = \{t_1, t_2, ..., t_l\}$ with l terminals of arity 0.

Generic Random Constants. A generic random constant is a terminal which accessed during program creation is instantiated to a random value from a set of predefined constants, or a value previously stored in PPC.

Individual Representation. Individuals are represented in fixed-sized GEP chromosomes of length n. The head section of an individual contains functions or terminals from FUT while the tail section contains terminals from T.

Individual Generation. To create a chromosome *Chr*, PPC elements are accessed from left to right starting from C_0. For each position C_n of the PPC, an instruction I is selected as follows: if C_n is in the head, then $I \in F \cup T$ is selected; if C_n is in the tail, then $I \in T$ is selected. In both cases, I is selected with probability $P_n(I)$. This instruction is denoted I_n. If $I_n = R$, then the random ephemeral is instantiated as described in [30]. This process is repeated for all genes of the PPC.

3.1.2 The PDPE Algorithm

PIPE's basic learning algorithm (shown in Algorithm 1) combines two forms of learning: Generation-Based Learning (GBL) and Elitist Learning (EL). GBL is PIPE's main learning algorithm and it is composed of 5 distinct phases: (1) creation of program population, (2) population evaluation, (3) learning from population, (4) mutation of PPT, and (5) PPT pruning. EL is used to make the best Individual found so far an attractor.

Algorithm 1: PIPE's learning algorithm

```
begin
    GBL
    repeat
        with probability $P_{el}$ do EL
        otherwise do GBL
    until termination criterion is reached;
```

Based on PIPE's GBL, we proposed the modified learning methodology shown in Algorithm 2, which integrates PIPE's learning strategy according to GEP's individual representation as follows:

It starts by generating a population of multigenic GEP genomes from the PPC as explained in Section 3.1.1. After that, it expresses each chromosome and evaluates the respective expression tree. In the next step, the algorithm updates the probabilistic structure; first it indexes the best chromosome of the current generation as Chr_b, then it modifies the probabilities of PPC such that the probability of creating Chr_b increases. The details of this adaptation is similar to the *adapt-ppt-towards* procedure presented in [30]. Further, to explore the solution area "around" Chr_b, Algorithm 2 mutates the prototype chromosome, as described in the original PIPE algorithm. In the final step, the algorithm indexes, the best chromosome found so far as Chr^{el}, and during elitist learning it adapts PPC towards Chr^{el}, as proposed in [30].

Algorithm 2: PDPE EL and GBL

```
begin
    GBL:
    Population_ Initialization
    Evaluate_ fitness_ of_ individuals
    $Chr_b$= best_ of_ generation_ chromosome
    $Chr^{el}$= best_ of_ all_ chromosome
    PPC_ adaption_ towards_ Chrb
    Mutate_ PPC
    EL:
    PPC_ adaption_ towards_ Chrel
```

3.2 N-gram GEP

N-gram GEP is a multivariate estimation of distribution algorithm with an indirect solution representation. This methodology borrows the N-gram GP algorithm [26] and draws on the idea of n-grams for representing the dependencies among elements of an individual. Unlike N-gram GP, which represents solutions as variable-length linear GP programs, N-gram GEP uses GEP's indirect encoding for representing candidate solutions. More specifically, this algorithm generates fixed-length GEP chromosomes according to a probabilistic model and transforms them to variable-length tree structures for fitness evaluations. Similar to the EDA proposed in [26], the probabilistic model used here is a 3-gram model. It is also worth to remark that, due to the fixed-length nature of GEP chromosomes, in our approach we do not use the length distribution as proposed in [26].

3.2.1 Individual Representation & Generation

This section provides a description of the structures used in N-gram GEP. Specifically, it covers the primitive elements of an individual and explains how the underlying probabilistic model is used to create a new offspring.

Individual Instructions. An individual in our algorithm is a GEP chromosome. The elements of this chromosome are chosen from the defined function and terminal sets. As shown in Section 3.1.1, the function set contains operators of arity equal or greater than 1 while the terminal set consists of independent variables which have an arity equal to 0.

$\mathbf{M}^{(3)}$. As in [26], our probabilistic model is a 3-gram model, represented as a 3-dimensional matrix $M^{(3)} = (m_{l,m,n})$ in which the indices l,m, and n range over the instruction set $\{I_1, ..., I_N\}$. Each matrix entry $m_{l,m,n}$ represents the probability of indexed instruction n appearing in a position, say i, in a chromosome, given that indexed instructions l and m appeared in positions $i-2$ and $i-1$, respectively. From $M^{(3)}$, two matrices $M^{(2)} = (m_{l,m})$ (2-dimensional) and $M^{(1)} = (m_l)$ (a vector) are obtained. The elements of these matrices are $m_{l,m} = \sum_n m_{l,m,n}$ and $m_l = \sum_m m_{l,m}$, respectively. Note that $M^{(1)}$ and $M^{(2)}$ denote the marginal probability mass functions of the joint probability function represented by M^3.

Individual generation. Inspired by the approach introduced in [26], the routine *genChromosome* (Algorithm 3) presents our method for sampling the probabilistic model $M^{(3)}$. To obtain the first leftmost element, say r, located in position $i-2$ of the head region of a GEP gene, we select an element from the instruction set based on the probabilities stored in $M^{(1)}$. To obtain the second

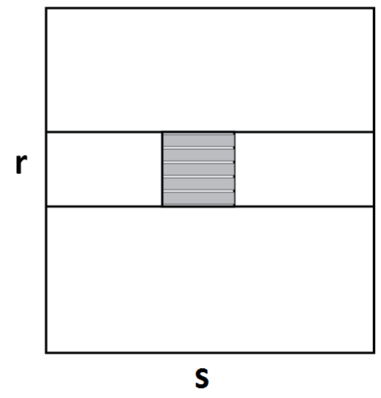


Figure 3.3: Schematic view of $M^{(2)}$

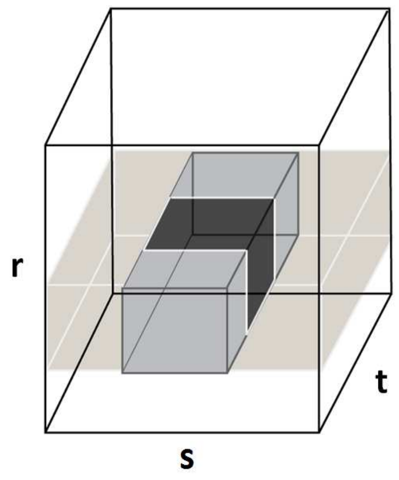


Figure 3.4: Schematic view of $M^{(3)}$

element, say s, located in position $i-1$ of a gene, we select an element of the instruction set according to the probabilities prescribed by the row of $M^{(2)}$ indexed by r, *i.e.*, the element located in position $i-2$ (see Figure 3.3). To obtain the third element, say t, (and all subsequent elements located in position i of the head of a gene), we select an element of the instruction set according to the probabilities stored in $M^{(3)}$'s r row (*i.e.*, the matrix row indexed by the element located in position $i-2$ of the gene), of $M^{(3)}$'s s page (*i.e.*, the matrix slice indexed by the element located in position $i-1$ of the gene). Figure 3.4 schematically shows the portions of $M^{(3)}$ used in the determination of t.

To obtain elements of the tail region, we use a procedure similar to the one described above.

The difference is that, since the tail domain only includes terminals, we cannot select the terminal based on all the probabilities stored in the respective row of $M^{(3)}$. Instead, for each element of the tail we normalize the probabilities corresponding to the terminals indexing that row based on the probabilities of the whole instruction set, then select the terminal using roulette wheel selection.

Algorithm 3: genChromosome pseudocode

```
genChromosome($M^{(1)}$,$M^{(2)}$,$M^{(3)}$)
begin
    /* Head starts                                                    */
    Select the first instruction x1, based on the probabilities stored in $M^{(1)}$ via roulette
    selection
    Select the second instruction x2, based on the probabilities stored in the x1-th row of
    $M^{(2)}$ via roulette selection
    for i=3 to Head length do
        Select xi based on $M^{(3)}{}_{xi-2,xi-1}$ /* via roulette selection      */
    /* Tail starts                                                    */
    for i=Head length+1 to Gene length do
        Select xi based on Norm($M^{(3)}{}_{xi-2,xi-1}$) /* via roulette selection */
```

3.2.2 The N-gram GEP Algorithm

N-gram GEP (Algorithm 4) draws on the N-gram GP algorithm proposed in [26]. The EDA in

Algorithm 4: N-gram GEP main loop

```
N-gram GEP
begin
    Initialise the distributions $M^{(3)}$
    repeat
        Compute marginals of $M^{(3)}$ to obtain $M^{(1)}$ and $M^{(2)}$
        for i = 1... popsize do do
            With probability 1/popsize, pop[i]= elitist
            With probability 1-1/popsize, pop[i]= mutate(genChromosome($M^{(3)}$, $M^{(1)}$, $M^{(2)}$)
        elite=truncationSelection (pop)
        updateProbabilities ($M^{(3)}$, elite)
    until Solution found or max number of iterations exhausted;
    return best individual found
```

N-gram GEP starts by initializing $M^{(3)}$ using a uniform distribution, *i.e.*, if we consider the total number of instruction as N, all the entries of $M^{(3)}$ are initialised to $1/N^3$. Then the algorithm proceeds to generate a new population mostly by sampling $M^{(3)}$. Occasionally, on average once per generation, the best individual found in the current run is re-inserted into the population. To make sure that diversity is maintained, a point mutation is applied on the individual returned by *genChromosome*. In the next step, similar to other EDAs, a truncation selection is used to select the best individuals of the current generation which are then stored in the elite set. This set is then used to update the entries of the probabilistic model. Following [26], we set the top 1/5 of the population as elite members.

We update $M^{(3)}$ using an additive update rule as shown in Algorithm 5. The arrays in this algorithm are not explicitly zeroed before they are updated. In this way the model used to create the upcoming generation will also depend on the good individuals created in previous generations. In addition, the learning rate η_M determines how much the current elite members influence the probabilistic model.

Algorithm 5: Learning in N-gram GEP

```
updateProbabilities ($M^{(3)}$, elite)
begin
    for all x in elite do
        for all genes g in chromosome do
            for j=3...geneLength do
                $M^{(3)}{}_{x_{j-2},x_{j-1},x_j} = M^{(3)}{}_{x_{j-2},x_{j-1},x_j} + \eta_M/N^3$
        $M^{(3)} = M^{(3)} / \sum_{l,m,n} M^{(3)}{}_{l,m,n}$
```

3.3 Experimental Setup

This section presents the experimental settings used to test the performance of PDPE and N-gram GEP.

3.3.1 Comparing PDPE with PIPE and GEP

To compare PDPE and GEP with PIPE, we used the *same* experimental setup proposed in [30]. Here, for all the experimental setups, $F = \{+, -, *, \%, sin, cos, exp, rlog\}$. For the function regression problem, $T = \{x, R\}$, and for the 6-bit parity problem, $T = \{x_0, x_1, x_2, x_3, x_4, x_5, R\}$, where x_i, $0 \le i \le 5$, and R denotes the input variables and the generic random constant in [0;1). To determine a suitable value for H, we ran a set of test experiments to examine which head/tail combination would yield acceptable results in terms of solution quality and fitness variance.

Function Regression: The function regression to be approximated is:

$$f(x) = x^3 * e^{-x} * cos(x) * sin(x) * (sin^2(x) * cos(x) - 1)$$

The training (test) data set D_{tr} (D_{te}) is composed of 101 equally distant points in the interval [0;10] ([0.05;10.05]). D_{tr} is used to calculate fitness during the creation of the population, and D_{te} is used to test how well the best evolved individuals generalize. The fitness value of each solution individual $\mathcal{S}$ is $\mathcal{F}(\mathcal{S}) = \sum_{\forall x \in D_{tr}} |f(x) - \mathcal{S}(x)|$. The generalization performance, $\mathcal{G}(\mathcal{S}) = \sum_{\forall x \in D_{te}} |f(x) - \mathcal{S}(x)|$, is the fitness measure of the individuals when considering the test set D_{te}. The settings used to run the experiments for PDPE and GEP are presented in Table 3.1; for PIPE, the same settings presented in [30] are applied. To statistically evaluate the results, we conducted 50 independent runs for each algorithm and time constrained each run to 100,000 chromosome evaluations (CE) as in [30].

6-Bit Parity Problem: The 6-bit parity function has six boolean input variables which are either 1 or 0. The output of this function is 1 (0) when the number of its non-zero inputs are odd (even). We used 64 training instances for this problem. The fitness of a solution is the number of instances it classifies incorrectly. As a result, the best (worst) fitness is 0 (64) for classifying all (no) instances correctly. Because of the Boolean nature of the problem the real-valued output of a solution is mapped to zero (negative) or 1 (positive). Table 3.1 presents the parameters used to run

Parameters	GEP		PDPE		N-gram GEP	
	S.R.	**6-B.P.**	**S.R.**	**6-B.P.**	**S.R.**	**6-B.P**
Population Size	150	30	150	25	50	20
Head Length	9	7	7	7	8	7
Number of genes	5	3	7	3	7	4
Linking function	+	*	+	*	+	*
Chromosome Length	95	45	105	45	119	60
Mutation rate	0.044	0.044	-	-	1	1
1-pnt. recomb. rate	0.3	0.3	-	-	-	-
2-pnt. recomb. rate	0.3	0.3	-	-	-	-
Gene recomb. rate	0.1	0.1	-	-	-	-
Gene transp. rate	0.1	0.1	-	-	-	-
IS transposition rate	0.1	0.1	-	-	-	-
IS elements length	1,2,3	1,2,3	-	-	-	-
RIS transp. rate	0.1	0.1	-	-	-	-
RIS elements length	1,2,3	1,2,3	-	-	-	-
Truncation selection ratio	-	-	-	-	5	5
η_M Learning rate	-	-	-	-	8	8

Table 3.1: Parameters for the function regression (SR) and 6-bit parity (6-BP) problems

the experiments for the PDPE and GEP algorithms. In regards to PIPE, we used the same settings described in [30]. We conducted 100 runs for each algorithm and set *CE*=500,000 for each run.

3.3.2 Comparing N-gram GEP with PDPE and GEP

We first compared N-gram GEP with the PDPE algorithm on the function regression and the 6-bit parity problems using the same setting introduced in Section 3.3.1. The parameters used to run this set of experiments are presented in Table 3.1. To compare N-gram GEP against the canonical GEP algorithm, we adopted the *same* experimental settings and problems used in [26], namely the Polynomial and the `Lawn-Mower` problems:

Polynomial: This problem consists of a set of symbolic regression functions where the goal is to evolve a mathematical expression which fits a polynomial of form $x+x^2+...+x^d$, where $5 \leq d \leq 12$, is the degree of the polynomial, and $x \in [-1,1]$ reflects the input variable. We specifically sampled the polynomial at 21 equidistant points $x \in \{-1.0,-0.9,...,0.9,1.0\}$. The goal is to minimize the fitness and the fitness measure was the sum of absolute differences between the target polynomial and the output produced by the candidate solution over the 21 fitness cases. Following [26], we implemented the polynomial problem using two function sets: F1 = $\{+,*\}$ and F2 =$\{+,-,*,/\}$. In both cases the terminal set was T = $\{x\}$.

Lawn-Mower: This problem was presented in [26] and is a variation of the Lawn Mower problem introduced in [18]. In the original Lawn Mower problem, the goal is to evolve a program which guides a robotic lawnmower to mow all the grass in a tiled square lawn. In this case, we considered lawns of size $d*d$ where $5 \leq d \leq 12$. In this version of the problem, the lawnmower performs one of the following three actions: moves forward and mow the tile it lands on (Mow), turns left by 90 degrees (Left), turns right by 90 degrees (Right). In the original problem, the fitness measure is equal to the number of non-mowed tiles at the end of the execution of the program. In [26], to make the problem more difficult, two more constraints are introduced: (1) the length of a candidate program is limited to $4*d^2$, and (2) the lawnmower must be energy efficient. To implement the second constraint, the fitness function shown in Equation 3.1 is corrected in a way that it promotes the evolution of programs which mow the whole lawn with least number of moves, and that stop immediately after having mowed the last tile.

$$fitness = \begin{cases} 0.0001*n, & \text{if all tiles are mowed} \\ 0.1*l+t, & \text{otherwise} \end{cases} \tag{3.1}$$

In Equation 3.1, n, l, and t respectively denote the number of extra moves, the program length, and the number of un-mowed tiles. Based on the above, we used F=$\{$Mow,Progn$\}$ and T=$\{$Left,Right$\}$ as the function and terminal sets, respectively. Note that Progn is a two argument function which evaluates its arguments, returning the value of the second (rightmost) argument; progn's structure allows for the creation of lawn-mowing programs of variable length. Since N-gram GP uses fixed-length programs, the Progn operator was not used in [26]. Tables 3.2 to 3.5 present the parameters used to run the experiments for the polynomial and the Lawn-Mower test problems. The results presented in [26] does not quantitatively show the performance of N-gram GP in the aforementioned experiments. Therefore, we were not able to compare N-gram GP and N-gram GEP in terms of performance. In the following, we compare these two approaches in terms of pattern preservation.

3.3.3 Pattern Learning Analysis

To explore the type of learning that is happening in N-gram GEP and to be able to compare this system with the N-gram GP algorithm, we used a set of experiments to find satisfactory answers to the following research questions : (1) Does N-gram GEP learn longer patterns based on the correlations acquired from the occurrence of triplet of instructions? For example if the triplets *abc* and *bcd* have both high probabilities, then the 4-tuple *abcd* might be the result of a sequence which starts with *ab* (2) How often does N-gram GEP take advantage of such correlations and how long are the patterns that it learns? (3) What is the role of repetition in the found patterns? (4) What is the relationship between length of the found solutions the number of genes and the chromosome length?

To answer these questions, we used the polynomial and `lawn-mower` problems mentioned earlier in this section; The settings used to run these experiments are listed in Tables 3.6 and 3.7. Note that in these experiments we have chosen longer head sizes and smaller number of genes; this technique allowed us to easily detect repeating patterns in the solution population that we were verifying. We took the following approach for analyzing pattern occurrence: after running one of the above experiments, we randomly chose one run in which an individual with the best fitness was created. We stored the $M^{(3)}$ matrix pertaining to that run and created 100 individuals based on the probabilities stored $M^{(3)}$. Finally, we analyzed the preservation of patterns based on the common frequencies seen in the generated individuals.

3.3.4 Parameter Guidelines

In our experiments for PDPE, GEP, and N-gram GEP we chose the direct parameters (head size and number of genes) by trying different chromosome combinations before running the actual experiments. For example, for the polynomial of degree 5, in case of N-gram GEP, we set the population size to 50, number of generations to 100, and number of runs to 10; we then tested this system using head sizes equal or greater than 3 and changed the number of genes. In each case we chose the chromosome setting which optimally yielded an individual with the desired fitness (0) and minimum solution size. For the `lawn-mower` problem we also had a maximum length constraint. In that case, we first found the approximate head size according to the imposed constraint ($4*d^2 = (H+T)*N$) by setting the number of genes (N) to 1. Then we tested the system using different combinations of head sizes and number of genes which met the maximum length and selected the setting which had the individual with the best fitness and minimum solution size to run our main experiments. Thus, the best method for selecting the proper chromosome setting while tackling other problems would be to test the system with a set H and N combinations. Specifically, trying different head sizes (small to large) with various number of genes helps the user to select a suitable (H, N) pair according to the best fitness measure and solution size.

Parameters	**GEP (Polynomial Problem)**							
Polynomial	**Pol.5**	**Pol.6**	**Pol.7**	**Pol.8**	**Pol.9**	**Pol.10**	**Pol.11**	**Pol.12**
Fitness Evaluations	20000	20000	20000	20000	20000	20000	20000	20000
Independent runs	1000	1000	1000	1000	1000	1000	1000	1000
Generations	2000	2000	2000	2000	2000	2000	2000	2000
Population Size	10	10	10	10	10	10	10	10
Head Length	4	5	6	8	7	10	6	25
Number of genes	6	5	5	5	7	6	11	7
Linking function	+	+	+	+	+	+	+	+
Chromosome Length	54	55	65	85	105	126	143	175
Mutation rate	0.044	0.044	0.044	0.044	0.044	0.044	0.044	0.044
1-pnt. recomb. rate	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3
2-pnt. recomb. rate	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3
Gene recomb. rate	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
Gene transp. rate	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
IS transposition rate	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
IS elements length	1,2,3	1,2,3	1,2,3	1,2,3	1,2,3	1,2,3	1,2,3	1,2,3
RIS transp. rate	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
RIS elements length	1,2,3	1,2,3	1,2,3	1,2,3	1,2,3	1,2,3	1,2,3	1,2,3

Table 3.2: Parameters for the Polynomial (Pol) problem

Parameters	**GEP (Lawn-Mower Problem)**							
Lawn Size	**Size.5**	**Size.6**	**Size.7**	**Size.8**	**Size.9**	**Size.10**	**Size.11**	**Size.12**
Fitness Evaluations	20000	20000	20000	20000	20000	20000	20000	20000
Independent runs	1000	1000	1000	1000	1000	1000	1000	1000
Generations	2000	2000	2000	2000	2000	2000	2000	2000
Population Size	10	10	10	10	10	10	10	10
Head Length	3	4	9	11	11	14	17	21
Number of genes	7	8	6	7	9	8	8	8
Linking function	progn	progn	progn	progn	progn	progn	progn	progn
Chromosome Length	49	72	114	161	207	232	280	344
Mutation rate	0.044	0.044	0.044	0.044	0.044	0.044	0.044	0.044
1-pnt. recomb. rate	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3
2-pnt. recomb. rate	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3
Gene recomb. rate	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
Gene transp. rate	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
IS transposition rate	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
IS elements length	1,2,3	1,2,3	1,2,3	1,2,3	1,2,3	1,2,3	1,2,3	1,2,3
RIS transp. rate	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
RIS elements length	1,2,3	1,2,3	1,2,3	1,2,3	1,2,3	1,2,3	1,2,3	1,2,3

Table 3.3: Parameters for the Lawn-Mower problem

Parameters	**N-gram GEP (Polynomial Problem)**							
Polynomial	**Pol.5**	**Pol.6**	**Pol.7**	**Pol.8**	**Pol.9**	**Pol.10**	**Pol.11**	**pol.12**
Fitness Evaluations	20000	20000	20000	20000	20000	20000	20000	20000
Independent runs	1000	1000	1000	1000	1000	1000	1000	1000
Generations	2000	2000	2000	2000	2000	2000	2000	2000
Population Size	10	10	10	10	10	10	10	10
Head Length	4	6	4	6	8	8	10	10
Number of genes	5	4	8	7	6	7	7	8
Linking function	+	+	+	+	+	+	+	+
Chromosome Length	45	52	72	91	102	119	147	168
Mutation rate	1	1	1	1	1	1	1	1
Truncation selection ratio	5	5	5	5	5	5	5	5
η_M Learning rate	8	8	8	8	8	8	8	8

Table 3.4: Parameters for the Polynomial (Pol) problem

Parameters	N-gram GEP (Lawn-Mower Problem)							
Lawn Size	**Size.5**	**Size.6**	**Size.7**	**Size.8**	**Size.9**	**Size.10**	**Size.11**	**Size.12**
Fitness Evaluations	20000	20000	20000	20000	20000	20000	20000	20000
Independent runs	1000	1000	1000	1000	1000	1000	1000	1000
Generations	2000	2000	2000	2000	2000	2000	2000	2000
Population Size	10	10	10	10	10	10	10	10
Head Length	3	4	9	10	11	15	16	22
Number of genes	6	7	5	7	8	7	8	7
Linking function	progn	progn	progn	progn	progn	progn	progn	progn
Chromosome Length	42	63	95	147	184	217	264	315
Mutation rate	1	1	1	1	1	1	1	1
Truncation selection ratio	5	5	5	5	5	5	5	5
η_M Learning rate	8	8	8	8	8	8	8	8

Table 3.5: Parameters for the Lawn-Mower (LM) problem

Parameters	N-gram GEP- Pattern Analysis (Polynomial Problem)							
Polynomial	**Pol.5**	**Pol.6**	**Pol.7**	**Pol.8**	**Pol.9**	**Pol.10**	**Pol.11**	**pol.12**
Fitness Evaluations	20000	20000	20000	20000	20000	20000	20000	20000
Independent runs	1000	1000	1000	1000	1000	1000	1000	1000
Generations	200	200	200	200	200	200	200	200
Population Size	100	100	100	100	100	100	100	100
Head Length	11	10	12	11	12	12	14	16
Number of genes	2	3	3	4	4	5	5	5
Linking function	+	+	+	+	+	+	+	+
Chromosome Length	46	63	75	92	100	125	145	165
Mutation rate	1	1	1	1	1	1	1	1
Truncation selection ratio	5	5	5	5	5	5	5	5
η_M Learning rate	8	8	8	8	8	8	8	8

Table 3.6: Parameters for the Polynomial (Pol) problem

Parameters	**N-gram GEP- Pattern Analysis (Lawn-Mower Problem)**							
Lawn Size	**Size.5**	**Size.6**	**Size.7**	**Size.8**	**Size.9**	**Size.10**	**Size.11**	**Size.12**
Fitness Evaluations	20000	20000	20000	20000	20000	20000	20000	20000
Independent runs	1000	1000	1000	1000	1000	1000	1000	1000
Generations	200	200	200	200	200	200	200	200
Population Size	100	100	100	100	100	100	100	100
Head Length	10	11	11	10	11	15	16	22
Number of genes	2	3	4	7	8	7	8	7
Linking function	progn	progn	progn	progn	progn	progn	progn	progn
Chromosome Length	42	63	92	147	184	217	264	315
Mutation rate	1	1	1	1	1	1	1	1
Truncation selection ratio	5	5	5	5	5	5	5	5
η_M Learning rate	8	8	8	8	8	8	8	8

Table 3.7: Parameters for the Lawn-Mower (LM) problem

Chapter 4

Results and Discussion

This chapter presents, analyzes, and discusses the results of the experiments introduced in Chapter 3.

4.1 PDPE versus PIPE and GEP

Function Regression. Figure 4.1 illustrates the performance of the PDPE, GEP, and PIPE algorithms on the test data set (D_{te}) defined in Chapter 3. The graph shows performance υ against number of solution individuals with $\mathcal{F}(\mathcal{S}) \leq \upsilon$ and $\mathcal{G}(\mathcal{S}) \leq \upsilon$ [30]. Figure 4.2 depicts PDPE's (true) variance for different chromosome sizes, and also shows how fitness of the best individual changes for different chromosome sizes.

In regards to GEP vs. PIPE, the best 19% (16%) of all GEP runs found solution individuals with a better performance than all the PIPE runs on the test (training) data sets. Furthermore, only the worst 26% (15%) of all GEP runs only found chromosome that perform worse on the test (training) data sets than the top program individuals found by all PIPE runs. GEP thus shows a better performance than PIPE on the function regression problem. Moreover, PIPE shows twice more variance than GEP (fitness values of GEP solutions range from 0.76 to 8.47, whereas PIPE's range from 1.18 to 16.64). This is expected because GEP's genetic operators search the genotype space more effectively than PIPE's random sampling of the probability of distribution stored in the PPT [38].

In regards to PDPE vs. PIPE, the top 26% (20%) of all PDPE runs found chromosomes that

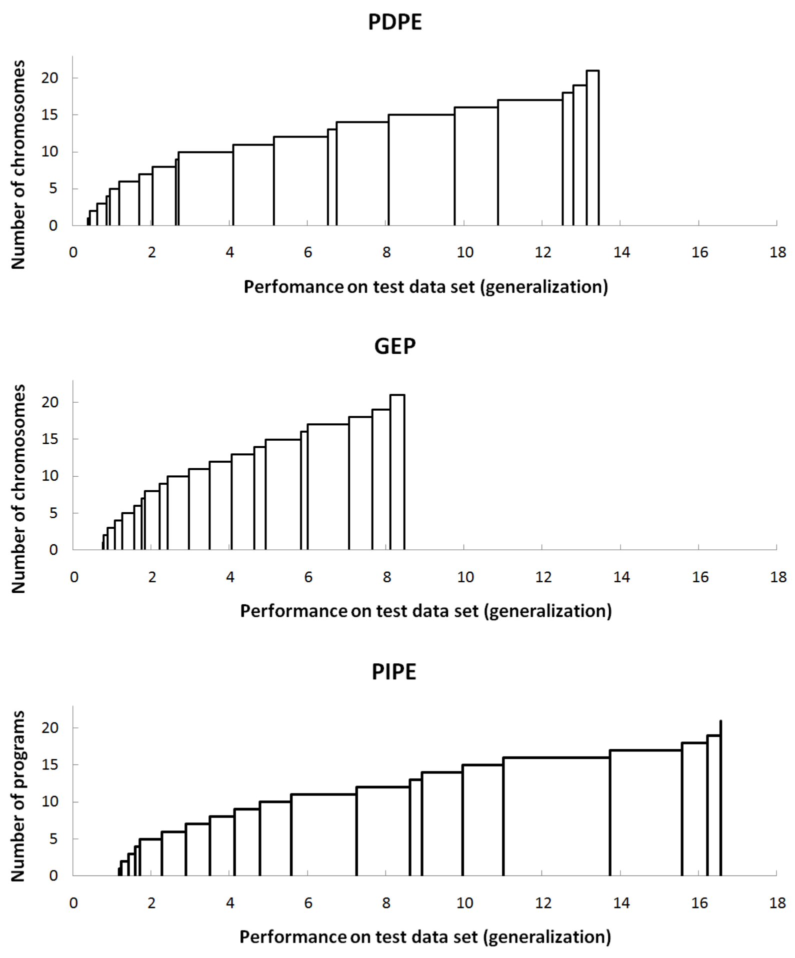


Figure 4.1: Results for the function regression problem

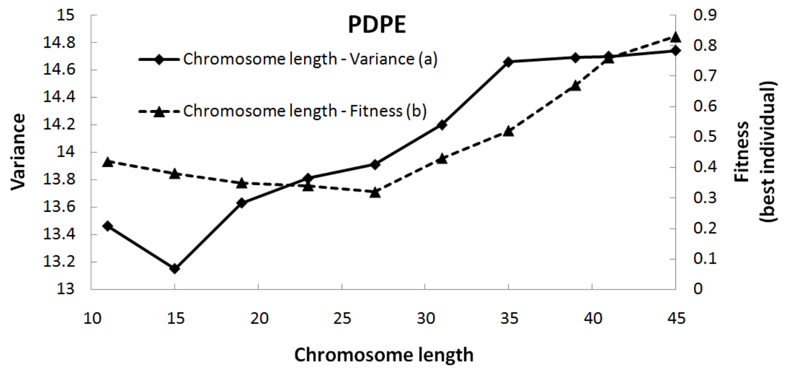


Figure 4.2: The correlation between chromosome length and variance (a), and chromosome length and fitness (b).

performed better on the test (training) data sets D_{tr} (D_{te}) than all PIPE runs. In addition, the worst 24% (26%) of all PDPE runs only found chromosomes that perform worse on the test (training) data sets than the best programs found by all PIPE runs. Thus, the best PDPE's solution individuals perform better than PIPE's best programs. Also, PDPE showed 15% less variance than PIPE.

The graphs in Figure 4.2 show the performance of our algorithm for different chromosome settings. The results show that for a wide range of chromosome sizes PDPE shows better fitness values with smaller variance than PIPE (15.46). However, as shown in Figure 4.2, increasing genome size in PDPE affects negatively both variance and fitness. We conjecture such bigger variance in PIPE stems from the way its tree shaping operators operate; growing trees that are too small in the early stages of evolution, and later pruning trees that are already too big. PDPE, on the other hand, represents the probability distribution in the fixed-length structure of the PPC, which seems to provide a more appropriate window into the search space, thus increasing the chances of the mutation operator finding promising solutions more often.

In regards to PDPE vs. GEP, the best 12% (11%) of all PDPE runs found chromosomes that outperformed all GEP runs on the test (training) data sets. On the other hand, the worst 27% (30%) of all PDPE runs found chromosomes that performed worse on the test (training) data sets than the top chromosomes found by all GEP runs. Therefore, we can conclude that the top chromosomes generated by PDPE outperform GEP's best solutions. In this case, PDPE's variance is 41% higher than GEP's.

6-Bit Parity Problem. Table 4.1 summarizes the results from the application of each algorithm (PIPE, GEP, and PDPE) for classifying the 64 training instances used in the experiment; the number of nodes in this table denotes the average number of nodes in the expression trees of the found solutions.

As the 6-bit parity results in Table 4.1 demonstrate the PDPE algorithm is the best among the three algorithms. In fact, it solves the problem more often (higher percentage) with less complex solutions (smaller number of nodes on average).

Algorithm	**Solved**	**Nodes (median)**
PDPE	94%	50
PIPE	70%	61
GEP	91%	55

Table 4.1: Results for the 6-bit parity problem

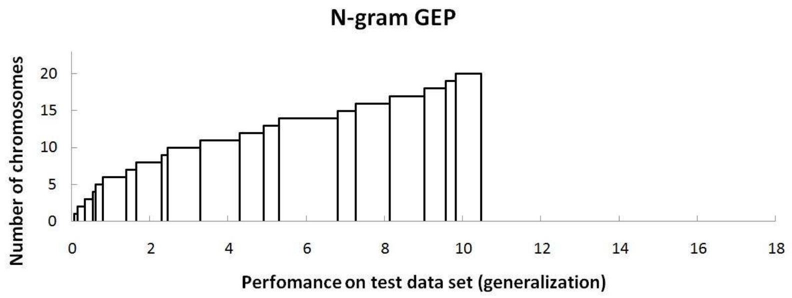


Figure 4.3: Performance of N-gram GEP on the function regression problem.

4.2 N-gram GEP versus PDPE and GEP

Function regression and 6-bit parity problem. We first compared the N-gram GEP algorithm with the PDPE learning methodology on the function regression and the 6-bit parity problem. Figure 4.3 illustrates the performance of N-gram GEP on the test data set (D_{te}). As far as performance is concerned the, top 9% (7%) of all N-gram GEP runs found chromosomes that performed better on the test (training) data sets D_{tr} (D_{te}) than all PDPE runs. Furthermore, the worst 17 % (13%) of all N-gram GEP runs found chromosomes that performed worse on the test (training) data sets than the top chromosomes found by all PDPE runs. Thus, the best individuals in N-gram GEP outperform the best individuals in PDPE. In terms of variance, N-gram GEP is 31.25% less variant than the PDPE algorithm. In the 6-bit parity problem N-gram GEP solved 97% of the cases with 52 nodes on average.

Polynomial. For the polynomial problem we compared the N-gram GEP algorithm with the canonical GEP methodology[1] based on their success rate. In evolutionary computation, success rate can be defined as the number of individuals having the desired fitness over all number of fitness

[1]The term *canonical GEP* refers to the original GEP algorithm introduced by Candida Ferreira in [7].

evaluations. For our experiments the acceptable error (desired fitness) was set to 1.05.

For the polynomial problem we first tested the unigenic version of N-gram GEP methodology against the multigenic version of this system. As explained in Chapter 3, our probabilistic model is a three-dimensional matrix ($M^{(3)}$) and holds the probability distributions pertaining to the elements of one gene. In the unigenic version of N-gram GEP individuals consist of one gene which has an $M^{(3)}$ associated with it. In the multigenic version of this system individuals are composed of multiple genes; each gene has its corresponding $M^{(3)}$. To make these two systems comparable the chromosome length in the unigenic N-gram GEP $(H+T)$ is set equal to $(H+T)*N$ in the mutigenic version of this system. Figure 4.4 shows the success rate of the multigenic N-gram GEP and the unigenic N-gram GEP on polynomials of degrees 5 to 12 for the function set F2 $=\{+,-,*,/\}$. Notice that the success rate for the multigenic N-gram GEP over all degrees of the polynomial problem outperforms the unigenic version of this system. Also, the multigenic N-gram GEP proves to be more scalable. In fact, as d (the degree of the polynomial) increases the performance of this version drops slower than the unigenic system. We conjecture that having a chromosome with multiple genes contributes in searching for more varied candidate solutions. In addition, because there is an $M^{(3)}$ associated with each gene, the pattern preserved in these blocks are represented more accurately by the individual $M^{(3)}$ matrices.

From this point onwards we used the multigenic N-gram GEP system to perform our experiments; thus, this algorithm is simply called N-gram GEP.

Figure 4.5 illustrates the success rate of N-gram GEP algorithm and the canonical GEP algorithm on the F2 $=\{+,-,*,/\}$ function set for the polynomials 5 to 12. Notice that for polynomials 5 and 6 GEP has a better performance. However, as d (the degree of the polynomial) increases the performance of this algorithm drops substantially whereas N-gram GEP shows more scalability.

In another experiment we compared the performance of N-gram GEP algorithm and the canonical GEP methodology on the function set F1 = $\{+,*\}$ for polynomials 5 to 12. As the result in Figure 4.6 demonstrates, a smaller function set makes this polynomial problem much easier to be

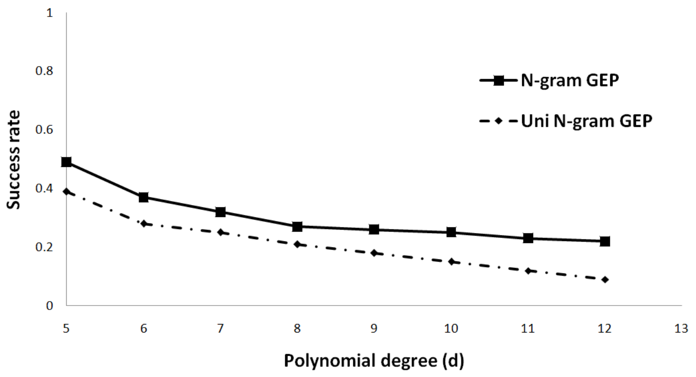


Figure 4.4: Success rate of unigenic N-gram GEP and multigenic N-gram GEP.

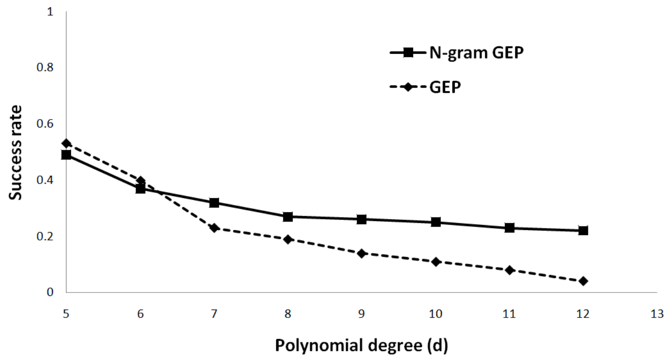


Figure 4.5: Success rate of N-gram GEP and GEP algorithm using F2 function set.

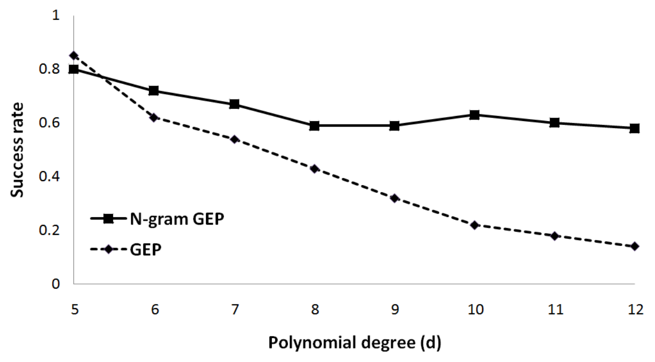


Figure 4.6: Success rate of N-gram GEP and GEP algorithm using F1 function set.

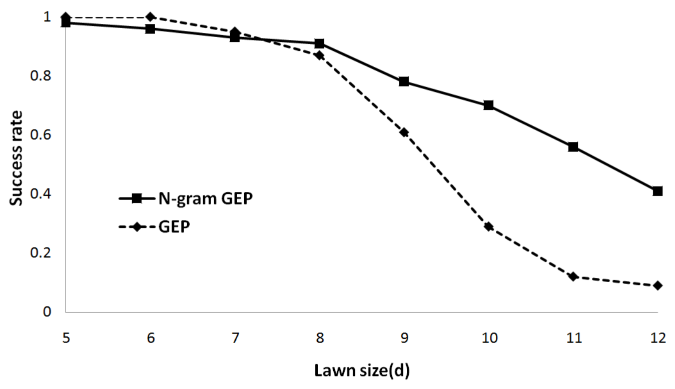


Figure 4.7: Success rate of N-gram GEP and GEP algorithm on Lawn-Mower problem.

solved. Also, similar to the results presented in Figure 4.5 for simpler polynomials of degree 5 and 6 the GEP algorithm outperforms N-gram GEP, but for polynomials of degree 7 and higher N-gram GEP shows to be more scalable and also exceeds canonical GEP in terms of performance.

`Lawn-Mower.` Figure 4.7 illustrates the success rate for the Lawn-Mower problem for lawn sizes $5*5$ to $12*12$ for the N-gram GEP algorithm and the canonical GEP methodology. In the diagram it can be seen that for smaller lawn sizes, *i.e.*, 5 to 7, GEP proves to be a better problem sover. Nevertheless, as the lawn sizes increases N-gram GEP demonstrates more scalability, and thus outperforms the GEP algorithm.

4.3 Pattern Learning Analysis

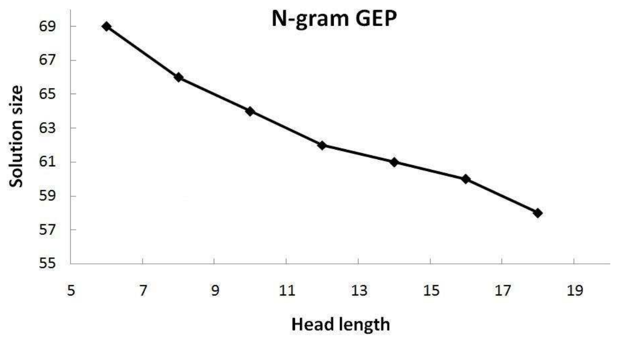
Polynomial. To better understand the importance of pattern regularities in the polynomial problem, we first focused on common patterns in the solution population. In the next step, we analyzed how these sequences contributed in creating a better offspring population. We also studied the M^3 matrix which was used to create the solution individuals, and assessed how the information stored in this probabilistic model guided the search for finding highly fit patterns.

Poli and McPhee reported in [26] that the N-gram GP algorithm was able to solve the polynomial problem in a highly patterned way when the function set F1 = $\{+,*\}$ was used. In N-gram GEP, however, we did not detect highly patterned sequences. Instead, there were certain sequences that proved to be effective in terms of fitness enhancement and could be found either in one gene or in several genes. We conjecture that the difference in the method of finding common sequences and repeating patterns in N-gram GP and N-gram GEP lies in their different individual representation and solution organization. For the case of N-gram GEP, let us analyze the patterns seen in the polynomial of degree 7 ($x^7+x^6+x^5+x^4+x^3+x^2+x$). In the solution population pertaining to this polynomial we found two occurring triplets in more than one gene of the 100 individuals that we verified: $***, **x$. The triplet $***$ is necessary for creating x^d; for example x^4 would be $***xx...$ and x^6 would $*****xx...$. The sequence $**x$ could be seen more frequently in two cases: (1) in the transition from the head to the tail section of genes, (2) at the beginning of those genes encoding the expressions of the kind x^d, where d was odd; for example x^5 would be encoded in such gene as $**x**xx...$. This triplet is also useful for creating different x^ds where d is even. Note that the two first elements of triplets $***$ and $**x$ are the same; if during the search the third element would be wrongly chosen then the fitness of the individual could substantially change. We looked at the probabilities of the associated M^3 of a gene which contained both triplets to see how the probabilities reflected this fact. We observed that the probability of $**$ being proceeded by a $*$ was 90.61% while the probability of this sequence being proceeded by an x was 3.01%. This shows that the system has learned over time that $**$ should be mostly followed by $*$.

As far as learning longer sequences is concerned, in our experiments N-gram GEP did not clearly show that it is able to capture longer sequences based on the information acquired from triplet of instructions. We believe that testing this system with more complex polynomials will demonstrate the true performance of N-gram GEP from this point of view.

We also tested the relationship among the chromosome length, number of genes and pattern repetition for polynomial of degree 7 in the same problem. As Figure 4.8 shows, in the first scenario we increased the head size and did not change the number of genes. The results proved that the longer the head size (chromosome length) of an individual, the shorter the solution size. In the second scenario, we tested the effect of increasing the number of genes while maintaining the same head size. In this case, increasing the number of genes negatively affected the solution size; in fact, the more the number of genes, the longer the solution size. We conjecture the reason behind these results is the GEP's individual representation: the head part contains both functions and terminals while the tail part contains only terminals and therefore there is less pattern variety in the tail. Thus, the longer the head part is the better repeating patterns will be preserved and the shorter the solution size will get.

`Lawn-Mower.` Similar to [26], in the lawn-mower problem we did not notice substantial matrix convergence for any specific triplet. This is because the function set of this problem contains only two function which are `progn` and `Mow`. Nevertheless, in this problem, we observed a different type of pattern preservation: The head part of the genes were almost all filled with the `Mow` function. In the M^3 related to a particular gene we observed that there was a high probably (85%) of starting the gene with the `Mow` function; also, the probability the first `Mow` function being proceeded with a second `Mow` function was equal (80%). These probabilities were reported respectively (74%) and (75%) in [26]. We also observed that function combinations such as `prognprognmow` appeared more likely to occur (30%-50%), whereas function/terminal sequences such as `prognleftright` had quite low probabilities (7%-15%). Thus, our observations show that the lawn-mower problem



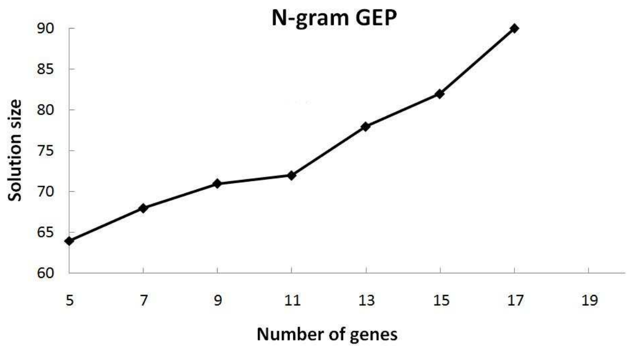


Figure 4.8: The relationship between head size, number of genes, and solution size

has not learned any specific pattern sequences. Instead, as in N-gram GP, it has acquired techniques to mow the square lawns in a more efficient manner.

Chapter 5

Conclusion and Future Work

This work introduced an indirect estimation of distribution algorithm for improving the induction of tree structured computer programs and mathematical expressions. Our methodology maintains a distribution of the genotypic space and searches for better and better solutions in the phenotypic space by performing a genotype-to-phenotype decoding step.

To verify the validity of our approach, we first developed a univariate IEDA named PDPE which combines GEP's indirect representation with PIPE's learning algorithm. This algorithm replaces the population of solution individuals with a probabilistic prototype GEP chromosome (PPC). Similar to PIPE's prototype tree (PPT), PPC probabilities are shifted towards the best found chromosome solutions in the current population.

We compared PDPE with the conventional GEP algorithm and the PIPE methodology on a function regression problem and the 6-bit parity problem. On the function regression problem, PDPE's best (worst) solutions were better (worse) than GEP's best (worst) solutions. Also, PDPE's best (worst) solutions were better (better) than PIPE's best (worst) solutions on the same problem. On the 6-bit parity problem, PDPE found more reliable and less complex solutions than both GEP and PIPE algorithms.

It is also worth mentioning that the PPC does not require operations, like in PIPE, for properly shaping the structure of the probability distribution. In PDPE a suitably chosen chromosome size ensures a parsimonious use of memory for the PPC. The choice of chromosome size is, however, problem dependent. Therefore, finding an appropriate value for this parameter would require other heuristics or expert knowledge on the problem domain. Alas, since there is no such a thing as a free lunch, that is the price one has to pay for better solution quality, smaller variance, and a more parsimonious use of memory.

In the next step, we developed a multivariate IEDA named N-gram GEP. This learning methodology borrows the idea of n-grams from the approach presented in [26] and uses a 3-gram as a probabilistic model for capturing the intra-dependencies amongst the elements of GEP chromosomes. Unlike N-gram GP, our system does not use an explicit length distribution; instead, this algorithm generates fixed length GEP chromosomes which translated to variable-length tree structures for fitness evaluation.

In our approach, first we compared N-gram GEP with the PDPE algorithm on the symbolic regression and the 6-bit parity problem. On the function regression problem N-gram GEP's best (worst) solutions were better (better) than PDPE's best (worst) solutions. On the 6-bit parity problem N-gram GEP solved the problem with a higher percentage and more nodes on average. Then we compared N-gram GEP with the canonical GEP algorithm on the Polynomial and the `lawn-mower` problems. On both problems our system outperformed the canonical GEP algorithm in terms of success rate and scalability. We then analyzed the pattern learning capabilities of N-gram GEP on the same problems and compared that with the N-gram GP methodology. On the Polynomial problem, our system did not find specific sequences. Instead, it found some major "building block" triplets for improving the overall fitness of individuals in different genes of the chromosome. On the `lawn-mower` problem, similar to [26], this algorithm did not find problem solving pattern sequences. Instead, N-gram GEP learned techniques for optimally mowing the squared lawns. A far as time efficiency is concerned, we did not notice any remarkable difference between PDPE, N-gram GEP, GEP and, the PIPE methodology.

Also, it is worth mentioning that in terms of features the work presented in this thesis has two main contributions: first, it extends univariate and multivariate EDAs to take advantage of an indirect search strategy by using Gene Expression Programming representations. Second, other EDAs such as PIPE [30] and N-gram GP [26] often need extra operations for controlling the solution sizes. In our approach, however, because of the fixed-length nature of GEP individuals we do not use any additional length-controlling strategy. This feature has its downside: for each experiment, the user chooses the optimum solution size, by testing different combinations of chromosome (head) sizes and number of genes.

5.1 Future Work

The experimental results presented in this work proved the effectiveness of our indirect probabilistic approach. Nevertheless, there are several points in this thesis which would be worthwhile of further investigations:

One limitation of the PDPE algorithm is its inability to capture multivariate dependencies amongst variables. Exploring ways for extending this methodology to overcome this limitation would be an interesting topic for future work. It would be also beneficial to test PDPE (and its extensions) on other problems, such as classification and neuroevolution.

This thesis presented a new EDA with an indirect search strategy by using the GEP's individual representation. It would be interesting to explore the capability of the N-gram GEP algorithm with other developmental approaches of evolutionary computation in which there is a mapping between the genotype and the phenotype; for example grammar guided genetic programming approaches such as the ones presented in [10, 35, 36].

In the N-gram GEP algorithm, we were successful in capturing intra-dependencies amongst triplets of instructions (3-grams) in each chromosome. A potential future investigation would be to extend the $M^{(3)}$ matrix to capture the dependencies of shorter and longer sequence of instructions (*i.e.* 2-grams and 4-grams) and test the extended version of this system with more complicated problems where a higher degree of dependency is needed amongst the solutions' components. Although capturing longer sequences is computationally expensive, it would allow us to observe how these sequences would affect the learning capabilities of the N-gram GEP algorithm. We also intend to test the influence of different parameter settings, *e.g.* population size, on the PDPE and N-gram GEP algorithms.

Also, its worth to remark that EDAs are successful evolutionary methods which use the global information about the search space to produce offspring. It is known, however, that these algorithms lack the ability to gather information about the locations of the solutions in the search space [38]. An interesting future work would be to investigate methods that explore the population probability distribution to guide the genetic operators used in GEP.

Glossary

EA	Evolutionary Algorithm
EC	Evolutionary Computation
EDA	Estimation of Distribution Algorithms
GA	Genetic Algorithm
GEP	Gene expression Programming
GP	Genetic Programming
IEDA	Indirect Estimation of Distribution Algorithms
PDPE	Probabilistic Developmental Program Evolution
PIPE	Probabilistic Incremental Program Evolution
PMBGP	Probabilistic Model Building Genetic Programming
PPC	Probabilistic Prototype Chromosome
PPT	Probabilistic Prototype Tree

Bibliography

[1] S. Baluja. Population based incremental learning: A method for integrating genetic search based function optimisation and competitive learning. *Tech.Rep.No.CMU-CS-94-163*, 1994.

[2] W. Banzhaf. Genotype-phenotype-mapping and neutral variation - a case study in genetic programming. In *Proceedings of PPSN III*, 1994.

[3] J. S. D. Bonet, C. L. Isbell, and P. Viola. Mimic: Finding optima by estimating probability densities. In *Advances in Neural Information Processing Systems*, 1996.

[4] d. J. E. Bosman, P.A.N. Grammar transformations in an EDA for genetic programming. *Special Session: OBUPM - Optimization By Building and Using Probabilistic Models, GECCO*, 2004.

[5] M. Brameier and W. Banzhaf. A comparison of linear genetic programming and neural networks in medical data mining. *IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION*, 5:17–26, 2000.

[6] B. M. Cerny, C. Zhou, W. Xiao, and P. C. Nelson. Probabilistically guided prefix gene expression programming. In *NICSO*, pages 15–26. 2007.

[7] C. Ferreira. *Gene Expression Programming: Mathematical Modeling by an Artificial Intelligence (Studies in Computational Intelligence)*. Springer-Verlag, 2006.

[8] L. J. Fogel, A. J. Owens, and M. J. Walsh. *Artificial Intelligence through Simulated Evolution*. John Wiley, 1966.

[9] E. Ghoulbeigi and M. V. dos Santos. Probabilistic developmental program evolution. In *Proceedings of ACM-SAC*, pages 1138–1142, 2010.

[10] F. Gruau. On using syntactic constraints with genetic programming. *Advances in Genetic Programming*, 2:377–394, 1996.

[11] G. Harik. Linkage learning via probabilistic modeling in the ECGA. *IlliGAL Report No. 99010*, 1999.

[12] G. Harik, F. Lobo, and D. Goldberg. The compact genetic algorithm. In *Proceedings of ICEC*, 1998.

[13] Y. Hasegawa and H. Iba. Optimizing programs with estimation of bayesian network. In *Proceedings of CEC*, pages 1378–1385, 2006.

[14] J. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.

[15] J. H. Holland. *J. ACM*, 9(3):297–314, 1962.

[16] D. J. Kenneth A. *Evolutionary Computation: a unified approach*. MIT press, 2006.

[17] J. Koza. Genetic programming: On the programming of computers by means of natural selection. *MIT Press*, 1992.

[18] J. R. Koza. Genetic programming II automatic discovery of reusable programs. *The MIT Press*, 1994.

[19] M. Looks, B. Goertzel, and C. Pennachin. Learning computer programs with the bayesian optimization algorithm. In *Proceedings of GECCO*, pages 747–748, 2005.

[20] C. D. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing*. MIT Press, 1999.

[21] J. F. Miller and P. Thomson. Aspects of digital evolution: Geometry and learning. In *Proceedings of ICES'98, Lecture Notes in Computer Science*, pages 25–35, 1998.

[22] A. Narayanan and N. Sharkey. *An introduction to LISP*. John Wiley & Sons Inc., 1986.

[23] G. D. Pelikan, M. and E. Cantú-Paz. BOA: The bayesian optimization algorithm. *Proceedings of GECCO*, pages 525–532, 1999.

[24] M. Pelikan, D. E. Goldberg, and F. G. Lobo. A survey of optimization by building and using probabilistic models. *Computational Optimization and Applications*, 21(1):5–20, 2002.

[25] M. Pelikan and H. Mühlenbein. The bivariate marginal distribution algorithm. In *Advances in Soft Computing - Engineering Design and Manufacturing*, 1999.

[26] R. Poli and N. F. McPhee. A linear estimation-of-distribution GP system. In *Proceedings of EuroGP, LNCS4971*, 2008.

[27] S. M. Ratle, A. Avoiding the bloat with probabilistic grammar-guided genetic programming. *Proceedings of EA*, 2310:255–266, 2001.

[28] I. Rechenberg. Cybernetic solution path of an experimental problem. Technical report, Royal Air Force Establishment, 1965.

[29] I. Rechenberg. *Evolutionsstrategie: optimierung technischer systeme nach prinzipien der biologischen evolution*. Frommann-Holzboog, 1973.

[30] R. Salustowicz and J. Schmidhuber. Probabilistic incremental program evolution. *Evolutionary computation*, pages 123–141, 1997.

[31] K. Sastry and D. E. Goldberg. Probabilistic model building and competent genetic programming. *Genetic Programming Theory and Practice*, pages 205–220, 2003.

[32] Y. Shan, R. McKay, R. Baxter, H. Abbass, D. Essam, and H. Nguyen. Grammar model-based program evolution. *Proceedings of CEC*, pages 478–485, 2004.

[33] Y. Shan, R. I. Mckay, H. A. Abbass, and D. Essam. Program evolution with explicit learning: A new framework for program automatic synthesis. *Proceedings of CEC*, pages 1639–1646, 2003.

[34] Y. Shan, R. I. McKay, D. Essam, and H. A. Abbass. A survey of probabilistic model building genetic programming. In *Scalable Optimization via Probabilistic Modeling*, pages 121–160. 2006.

[35] P. Whigham. Grammatically-based genetic programming. *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pages 33–41, 1995.

[36] L. K. S. Wong, Man Leung. Inducing logic programs with genetic algorithms: the genetic logic programming system. *IEEE expert*, 10(5):68–76, 1995.

[37] K. Yanai and H. Iba. Estimation of distribution programming based on bayesian network. In *Proceedings of CEC*, pages 1618–1625, 2003.

[38] Q. Zhang, J. Sun, and E. P. K. Tsang. An evolutionary algorithm with guided mutation for the maximum clique problem. *IEEE Trans. Evolutionary Computation*, 9(2):192–200, 2005.