Towards Identifying Salient Patterns in Genetic Programming Individuals

András Mátyás Joó

A thesis submitted for the degree of Doctor of Philosophy

Aston University

June 2010

Aston University

**Towards Identifying Salient Patterns
in Genetic Programming Individuals**

András Mátyás Joó

A thesis submitted for the degree of Doctor of Philosophy

2010

Thesis summary

This thesis addresses the problem of offline identification of salient patterns in genetic programming individuals. It discusses the main issues related to automatic pattern identification systems, namely that these (a) should help in understanding the final solutions of the evolutionary run, (b) should give insight into the course of evolution and (c) should be helpful in optimizing future runs.
Moreover, it proposes an algorithm, Extended Pattern Growing Algorithm ([E]PGA) to extract, filter and sort the identified patterns so that these fulfill as many as possible of the following criteria: (a) they are representative for the evolutionary run and/or search space, (b) they are human-friendly and (c) their numbers are within reasonable limits. The results are demonstrated on six problems from different domains.

Keywords

Genetic programming, tree mining, data mining.

Acknowledgments

Contents

List of Algorithms

List of Tables

List of Figures

Chapter 1

Introduction

The main objective of this thesis is the identification of salient patterns in offline genetic programming (GP) individuals. To set the stage in this section we discuss briefly the following key aspects:

- What are GP patterns?

- Why would one identify GP patterns?

- What are main the challenges?

- What is the essence of our approach?

- What results did we get?

- What could be the impact of our results?

- How could this work be extended?

After this short introduction we take a more systematic approach in introducing the reader into the key issues of evolutionary computation.

This review is more biased towards GP and the issues related to GP: codification, search operators and schema theorems. The second part of this chapter focuses on identification, mining and evaluation of patterns. We are going to discuss the most important techniques used by different researchers. The chapter ends with a case study involving pattern identification from the domain of the evolution of learning rules.

1.1 Goals of the thesis

Genetic programming [1] is a family of optimisation techniques. Based on the idea of automatic generation of computer programs by means of simulated evolution, GP is capable of solving complex problems [2]. In this thesis we focus on the identification of salient patterns which emerge during GP runs.

Unfortunately there is no consensus among different researchers on what GP patterns are. The notion is used with different meanings depending on the context and very often even the name is different. The terms schema, fragment, building-block etc. denote similar, more or less overlapping concepts.

There are two common aspects of the existing definitions:

1. GP patterns resemble GP individuals in terms of structure

2. GP patterns represent sets of GP individuals or sets of fragments of GP individuals, consequently patterns represent subsets of the search space.

It is reasonable to assume that some patterns are more important than others. The presence and proper usage of important patterns should result in an increased fitness of the container individuals.

This thesis focuses on the offline identification of important (salient) patterns. This means that we analyse the whole evolutionary run after the evolutionary run has finished and in doing so we consider all individuals from each generation.

There are several reasons why one would want to identify patterns in an offline manner. In our view the most important are:

1. GP rarely produces beautiful and / or elegant solutions but rather complicated ones which are hard to understand. A tool which identified the very essence of the final solution(s) would be valuable.

2. An offline pattern identification system would give a deep insight into how GP works and may reveal details that are hidden if we consider only the final solution(s).

3. Some of the identified salient patterns could serve as seed or modules for future runs.

The main challenge in analysing patterns lies in the sheer number of patterns that are present in a GP run. There are two basic approaches to deal with this: theoretical and empirical. Theoretical methods focus mainly on modeling the propagation of patterns. Though the field has shown significant progress during the last decade, the mathematical machinery involved is still cumbersome and hard to apply (see Section 1.3.1 for an overview).

Empirical methods use different heuristics to generate and filter the patterns. Several techniques have been described in the past and used for different purposes. (See Section 1.3.3 for a review.)

Our method, called collectively Extended Pattern Growing Algorithm ([E]PGA) is an empirical method. It uses a specialized deterministic tree-mining algorithm for mining for patterns which satisfy frequency, order ratio, saliency and other criteria. The resulting set of patterns are filtered through successive steps. In contrast to several algorithms described in the literature [E]PGA does not need a human expert to judge the quality of the patterns.

We tested [E]PGA on six problems of different difficulty taken from different domains: two constructional problems of different difficulty, a symbolic regression and a classification problem, a boolean function reconstruction problem and one problem from the domain of evolution of learning rules for perceptrons. The results we got for these problems are very encouraging. In the majority of the cases the patterns identified by [E]PGA statisfy the three criteria we proposed to meet: to be representative for the evolutionary run, to be human friendly and their number to be between reasonable limits.

They also indicate the limitation of [E]PGA: it does not address the problem of semantically equivalent patterns. One possible future direction this project could go would is the mining of the semantic space — as opposed to mining the syntactic space as it is done now.

1.2 Overview of evolutionary computation

Evolutionary algorithms (EA) are a set of population based, stochastic search techniques inspired by the fundamental ideas of Darwinian[1]evolution: sexual reproduction and survival of the fittest. EAs perform blind search, in the sense that they do not need problem specific information (e.g. gradient) to find a problem's optimum. This lets EAs to perform relatively well even in those cases where other optimization techniques may fail. Before presenting the basic EA let us introduce some terms related to EAs.

- Genotype. The representation of a possible solution of the problem is referred to as a *genotype*. Depending on the nature of the problem the space of genotypes can vary widely: $\{0, 1\}^n$, $\mathbb{R}^n$, the space of possible parsing trees, etc. The search is done in the genotype space. In the following we will denote the genotype space by $\mathbb{G}$.

- Phenotype. The materialization of a genotype, the collection of the observable properties is called a *phenotype*. We will use $\mathbb{P}$ to denote the phenotype space.

- Encoding. A function $C : \mathbb{P} \to \mathbb{G}$ that maps the phenotype space into the genotype space is called *encoding function*. Depending on $\mathbb{G}$ we talk about binary, real, tree, etc. encoding. Sometimes the genotype and the phenotype spaces are identical, in which case C degenerates to the identity function.

- Chromosome, gene, locus, allele. If a genotype has a linear structure then it is usually referred as a *chromosome*. An atomic element in the chromosome is called a *gene*, while its position is its *locus*. The allowed values for a particular gene are its *alleles*.

- Individual, population. The term *individual* is used ambiguously. Usually it refers to the two sides of the same entity, the genotype – phenotype pair, but it could also mean only one of these, depending on the context. A set of individuals is called a *population*. The population at a given time t will be noted by P_t.

- Fitness measure. A function $F : \mathbb{G} \to \mathbb{R}^p, p \in \mathbb{N}^*$ is called the *fitness measure* or simply the *fitness* and reflects the quality of an individual with respect to the problem to be solved. The goal of the EA is to optimize this function.

- Selection. A function $S : P_t \to P_t'$ that selects a subset of the population according to some criterion (in most cases the fitness measure) is called a *selection operator*. Selection can be positive or negative. Genetic material from individuals

---

[1]There is another popular evolutionary theory, Lamarckian evolution, which is sometimes used for optimization purposes as well [3–7].

selected by a positive selection operator will be propagated to their descendents. Individuals selected by a negative selection will be eliminated from the population.

- Replacement strategy. Selection is strongly related to the concept of *replacement strategy*. In a *generational model* offspring completely replace the actual population P_t and selection is always done among individuals of the same age. In a *steady state* model however, only a part of the population (usually the worse individuals) are replaced. In this scenario the population contains competing individuals of different ages. Generational models sometimes explicitly save the best individuals for the next generation. This latter technique is referred to as *elitism*.

- Recombination. The genetic material of positively selected individuals is used to generate new individuals by means of *recombination* (in some cases *crossover*). Different encodings feature different recombination operators. More formally, crossover can be considered as a function $X : \mathbb{G}^k \to \mathbb{G}^m$ where k parents give birth to m offspring ($k, m \geq 1$).

- Mutation. A function $M : \mathbb{G} \to \mathbb{G}$ that introduces small perturbations in the individual on which operates is called *mutation operator*.

- Evolutionary operators or search operators. Selection, recombination and mutation are known as the *search* or *evolutionary operators*. Sometimes *reproduction*, which makes a copy of an individual is regarded as an evolutionary operator as well.

The basic EA works in the following way. First it generates a population of randomly initialized individuals. Then, until a given stopping criterion is met it iterates through the phases of fitness assignment, selection and crossover. Fitness assignment evaluates the individuals and assigns them values that reflect the quality of the individuals. These are used by the selection operators to decide on the set of parents and on the individuals that will be eliminated. The set of parents is used to generate the offspring by means of recombination and mutation. Finally the offspring (partially) replace the old population. Usually the best individual from the last generation is returned as the solution for the problem. This basic scheme, which is used with more or less modifications in the different EA paradigms is given in Algorithm 1.1.

1.2.1 Classification of evolutionary algorithms

From the early works of Friedberg, [8] EAs have been growing different branches and can be classified by different criteria. The most common is classification by the encoding type, though fitness measure and population model are sometimes used as well.

Algorithm 1.1 Basic EA

1: Initialize(Population)
2: **while** (some termination criterion not satisfied) **do**
3:     Parents = Select(Population)
4:     Children = Crossover(Parents)
5:     Mutate(Children)
6:     Replace(Population, Children)
7: **end while**
8: **return** Best(Population)

Encoding

Binary encoding is one of the earliest EA approaches. This type of EA was described by Holland in [9] and is called *genetic algorithm* (GA). GA features fixed or variable length strings of bits as genotypes where one or more (successive) bits encode a property.

Genetic programming (GP) is the second main paradigm in EA. It was introduced by Koza in [1] and its aim is automatic computer program generation by means of evolution. Originally Koza used LISP expressions to represent computer programs but later a variety of representations were used: imperative programming languages like C, assembly or machine code, trees, graphs, etc. [10].

Evolution strategies (ES) were developed by Rechenberg and Schwefel in [11, 12] and focus on solving real-valued optimization problems. The main characteristics of ES are the use of linear genomes consisting of real values, the intensive usage of the mutation operator and the self adaptation of different strategy parameters.

Less popular encoding schemes include *learning classifier systems* [13], which use learning rules of form `if <rule> then <action>` and *evolutionary programming* ([14, 15]), which features finite state machines for individual representations.

Fitness measure

From the point of view of fitness measure EAs come in two flavours. As mentioned earlier, the fitness function is of the form $F : \mathbb{G} \to \mathbb{R}^p$. If $p = 1$ the problem is a single-objective optimization problem, i.e. we are searching the individual whose fitness minimizes (maximizes) F. If $p > 1$ we talk about a multi-objective optimization problem. Because there is no natural ordering in $\mathbb{R}^p, p > 1$, techniques based on Pareto optimality are usually used. For a discussion on the subject one should consult [16].

Problems that need the detection of multiple optima (multi-modal optimization) on the fitness landscape can use various population models. The most common ap-

proaches are niching and fitness sharing [17], crowding [18], competitive [19] and cooperative models [20] and parallel populations [21].

Parallel population models

From the point of view of parallelism there are three approaches. *Panmictic populations* [22] do not impose any restriction on the potential partners of an individual. In contrast *local* or *diffusion models* limit the potential partners to the neighbouring individuals. Finally, *island models* [21] feature subpopulations which are organized according to some topology and granularity and exchange a given number of individuals at certain time intervals using a given migration strategy.

1.2.2 Genetic programming

In the following we will focus on issues related to genetic programming. We will give a short taxonomy of the main GP flavors and discuss briefly how the main components of an EA (problem representation, population model, fitness assignment, parent and replacement selections, mutation and crossover) work in the case of GP.

Tree-based representation

Tree-based representation was proposed by Koza and described in detail in [1]. Koza used LISP expressions to represent individuals which undergo the evolutionary process.

In designing a tree-based GP an important step is deciding on the set of allowed terminals ($\mathbb{T}$) and functions ($\mathbb{F}$). The set $\mathcal{T}$ of all trees that can be built from $\mathbb{T}$ and $\mathbb{F}$ should be large enough to contain the solutions. This constraint is called *sufficiency*. However, $\mathbb{T}$ and $\mathbb{F}$ should not contain unnecessary elements, otherwise the search space will be too large to explore in reasonable time. [2]

Care should be taken for functions that could generate exceptions (e.g. square root, division, exponential function, etc.). If the functions from $\mathbb{F}$ are protected from generating exceptions they are said to satisfy the *closure property*.

Another issue that should be considered whether functions should accept inputs of any type, or just of a given type. In the former case we talk about *untyped* GP, in the latter about *strongly typed* GP [24].

---

[2]Recently Woodward [23] has proven that if modularity is allowed then the minimum number of primitives needed to express the target function is independent of the primitive set.

Linear representation

While the tree-based representation corresponds to the functional programming paradigm, the linear encoding stands for the imperative programming paradigm. In the former case a genotype is actually a function, which is evaluated by a (virtual) CPU in a recursive manner, in the latter case the program is built up by successive instructions which are executed or interpreted sequentially.

Early linear GP approaches closely resembled GAs [25, 26], but later more elaborate representations evolved. A particularly interesting branch is the evolution of machine code [27–30]. Because machine code is executed directly (no interpretation step is involved) machine GP brings several orders of magnitude speed gain compared to tree-based GP.

Graph-based representation

Another notable representation for GP is the graph-based representation [31–34]. In this scenario nodes are functions or terminals and links control the program flow. There are three important properties of the graph-based representation: (a) it is a more compact representation than tree-based GP; (b) it can be more efficient in terms of CPU and memory utilization since partial results are implicitly cached; and (c) special recombination and mutation operators are necessary that explore/exploit the search space, yet preserve the syntactic correctness of the individuals.

Hybrid approaches

There are many variations of these three basic models. Two notable hybrid approaches among these are *grammatical evolution* [35] and *cartesian genetic programming* [36]. Grammatical evolution features linear genomes with integer genes which are transcribed to programs using a given grammar making relatively easy the generation of programs in an arbitrary programming language. Cartesian GP also features linear genomes, each genome representing a program as a graph. The genes themselves are integers representing addresses in data, addresses of functions in a look-up table or additional parameters.

Fitness assignment

Fitness assignment is a three stage process. It involves the evaluation of individuals, assignment of the raw fitness values and the (optional) scaling of the raw fitness according to some criteria. Evaluation depends strongly on the representation. The higher the representation level, the more expensive the evaluation is in terms of CPU

Table 1.1: Main evaluation techniques for GP

Representation	Evaluation mode
Machine code	Direct execution
Higher level imperative programming language	Interpretation and / or compilation and execution
LISP expression	Interpretation by the LISP `eval` function
User defined representation	User defined interpreter

cycles. This means that zero overhead is achievable only with machine code representation. All other representations require interpretation and / or compilation step(s). This can be done with an off-the-shelf interpreter or compiler if the representation is an existing programming language or a custom interpreter in the case of a user defined representation.

The most frequently used evaluation techniques are summarized in Table 1.1.

Selection methods

Selection, whether positive or negative relies on the fitness of the individuals. Some methods use the fitness values directly to select the individuals. In these cases the selection probability is usually proportional to the individuals' fitness (e.g. roulette wheel selection, tournament selection). Other schemes use the fitness values indirectly, only to sort the individuals, and the selection is done based on the position of the individuals in the sorted list (e.g. rank selection).

General EA selections usually work well with GP without any modification. For a thorough treatment one should see [37].

Crossover types

There are a plethora of crossover methods in GP, regarded as the main search operators. In the case of linear representation crossover works by selecting randomly one or more crossover points in the parents' chromosomes and swapping the code sections defined by these. Usually the crossovers used in GA can be applied with little or no modification (e.g. one point, two points, uniform, etc) [37].

In the case of tree-based GP, crossover selects randomly two nodes in each parent and swaps the subtrees defined by these nodes. Sometimes there are restrictions on the minimum and maximum allowed heights of the subtrees as well as on the context of crossover points (e.g. homologous, context-preserving, etc.) [10].

Mutation types

Mutation can be implemented with different granularity from macro to micro mutation, where granularity means the amount of affected code. For example in the case of linear encoding mutation could replace several lines of code, a single line of code, an instruction with a compatible one or an operand of an instruction with a compatible one.

In the case of tree encoding usually there are only two levels: macro mutation would replace a whole subtree with another (randomly generated) one. Micro mutation on the other hand would replace only a single node with a compatible node (function with function of same arity and terminal with terminal of the same type). Mutations with different granularity levels usually coexist in a single implementation. For details see [10, 37].

1.3 Analysis of genetic programming

Understanding why evolutionary algorithms work is not a trivial task. Since the inception of evolutionary computation several theories and empirical methods have emerged trying to explain the different aspects of the evolutionary runs. This section gives a short review of the major theoretical and practical results in the field with a special emphasis on tree-based GP.

1.3.1 Theoretical aspects

Schema theorems

Schemata are similarity templates in the search space, i.e. subspaces of the search space. Schema theorems give an insight into how EAs work by describing how schemata propagate during the evolutionary run. There are two main aspects of schema theorems.

1. Encoding. With some encoding types (e.g. genetic algorithms) the definition of schema is almost self-evident. In the case of other encoding types (e.g. genetic programming) the definition is not straightforward and multiple alternatives exist.

2. The type of the schema theorem. *Pessimistic* schema theorems give only a lower bound on the expected number of individuals that will match a schema in the following generation. *Exact* schema theorems on the other hand give a precise prediction by the use of *transmission probability*, a factor that reflects the total effect of crossover, reproduction, selection and mutation.

Holland's schema theorem

The first schema theorem is due to Holland [9] and it applies to genetic algorithms. In GAs a schema is a string containing 3 symbols: 0, 1 and #. The # is the "don't care" symbol and can stand for either 0 or 1. For example a schema ##01 represents the following chromosomes: 0001, 0101, 1001, 1101. There are a few notions related to schemata. A *well defined position* in a schema H is a position containing either 0 or 1. The *order* of H, $\mathcal{O}(H)$ is the number of well defined positions. The *defining length* $\mathcal{L}(H)$ is the maximum distance between two well defined positions in H.

Holland's schema theorem gives a lower bound on the expected number of chromosomes that match H at timestep $t+1$ and is stated as follows:

$$E[m(H, t+1)] \geq$$
$$m(H,t) \cdot \frac{f(H,t)}{\bar{f}(t)} \cdot (1 - p_m)^{\mathcal{O}(H)} \cdot \left[1 - p_{xo} \frac{\mathcal{L}(H)}{N-1} \left(1 - \frac{m(H,t)f(H,t)}{M\bar{f}(t)} \right) \right], \quad (1.1)$$

where

- $m(H,t)$ it the number of chromosomes matching schema H at generation t

- $f(H,t)$ is the mean fitness of the chromosomes matching H

- $\bar{f}(t)$ is the mean fitness of the chromosomes in the population

- p_{xo} is the probability of crossover

- p_m is the probability of mutation

- N is the chromosome length

- M is the population size

- $E[m(H, t+1)]$ is the expected number of chromosomes matching the schema H at generation $t+1$.

Holland's schema theorem has been the target of criticism in the past (e.g. [38, 39]). This is due to the overinterpretation of the schema theorem, i.e. the conclusion that the growth of schemata is related to the quality of search performed by the GA. As Altenberg [38] points out: "The common interpretation of the schema theorem implicitly assumes that any member of an above-average schema is likely to produce offspring of above-average fitness, i.e. that there is a correlation between membership in an above-average schema and production of fitter offspring. But the existence of such correlation is logically independent of the validity of the schema theorem." What
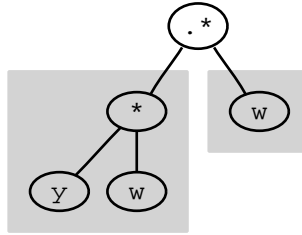
Figure 1.1: An element of Koza's schema {w (* y w)}. Shaded subtrees represent parts that match the schema.

Holland's schema theorem states is that the frequency of schemata that have above-average fitness will grow exponentially in the following generations. In [38] Altenberg shows how Holland's schema theorem can be derived from an earlier theorem from population genetics, Price's theorem [40].

Koza's GP schemata

The first definition of schema in GP is due to Koza. In [1] he defined a schema H as a set of LISP expressions (S-expressions). An S-expression is an element of a schema H if it contains *all* the elements of H as sub-expressions at least once. An example is given in Figure 1.1. Koza did not develop any formalized theory related to the propagation of schemata. His work [1] includes only an informal argument why Holland's schema theorem should be applicable for genetic programming.

O'Reilly's GP schemata

Koza's schema definition is extended by O'Reilly [41], who defined a schema as a multi-set of subtrees and *tree fragments*. A tree fragment is a tree with at least one leaf containing the "don't care" #. A "don't care" symbol can stand for any sub-tree. For example the schema [(- x y) (* # x) (- x y)] represents all programs that contain at least *twice* the subtree (- x y) and at least once a tree fragment (* # x). The tree fragment (* # x) stands for * rooted subtrees with the first right branch containing a single leaf node x. A pessimistic schema theorem specific for this kind of definition is also presented in [41].

Rosca's schemata

In Rosca's [42] definition a schema is a *rooted* tree fragment. The "don't care" symbol #, just like in the O'Reilly case, can stand for any sub-tree. For example a tree belongs to a schema (* # x) if its root node's key is * and the first right branch is x. The constraint enforced by Rosca means that a schema can be matched at most once by a tree. This makes possible the analysis of the propagation of the schema components by studying the change of the number of programs belonging to the schema.

Altenberg's model

The first non-pessimistic schema theorem in GP is the work of Altenberg [43]. He considered GP schemata as subtrees and gave an exact formula on the propagation of schemata under standard crossover and infinite population. The theory is based on a special selection operator, *soft brood selection*, meant to increase the *evolvability*, i.e. the probability that alterations in a program can produce fitness increase.

Langdon and Poli's schemata and hyperschemata

Another possible definition for GP schema is given by Langdon et al. in [44]. In their definition a schema is a rooted tree where the "don't care" symbol # stands for a single node that has the same arity as the node containing #. For example if we have a schema $H =$ (+ (# x) #) then any tree that belongs to this schema will have unary function in the place of the left-side # and a terminal in the place of the right-side leaf #. The authors argue that this schema definition is the closest to the original GA schema concept and present a pessimistic schema theorem related to the propagation of this type of schema.

A new, extended schema definition is given in [44] as well. A *hyperschema* can contain another "don't care" symbols, =, which stands for any subtree. The authors present an exact schema theorem featuring hyperschemata.

1.3.2 Building block hypothesis

In [45] Goldberg formulated a controversial hypothesis for GA, called the building block hypothesis (BBH). According to this, the low defining length, above-average fitness schemata, called building blocks tend to be combined together into larger schemata of potentially higher fitness.

BBH was criticized because of its precarious theoretical basis (e.g. [46–49]). These works emphasize the assumption of interaction of schemata, something which is missing from schema theory; the reliance of BBH on the stable fitness of schemata, which can not be guaranteed; the fact that BBH predicts many generations ahead the progress of GA etc.

Whether BBH is true or not in genetic programming is still the subject of intense debate (e.g. [49–52]). The major concerns related to the applicability of the BBH in GP are similar to ones for GAs along with the following. GP schemata are radically different in both form and behaviour from GA schemata because of their non-linear structure. GP building blocks should be resistant to the destructive effects of crossover through the consecutive generations, something which is hard to guarantee, especially with small tree sizes and / or without specialized crossover operators. Furthermore the

assumption that lower order schemata can be combined together implies the relative independence of subcomponents, which usually does not hold.

An alternative BBH for GP, dubbed as competitive building block hypothesis (CBBH) is proposed by Ryan et al. in [51]. According to this there are three areas of discovery in GP trees: (a) the *common area*, i.e. the root node and the surrounding nodes, where little or no useful exploration happens, (b) the *area of discovery*, i.e. the part below the common area, where the most useful discoveries are made and (c) the *tails*, where useful discoveries are unlikely are to be maintained. CBBH conjectures that (a) next to the co-operative, traditional non-rooted BBs there are rooted competitive building blocks and the evolution is driven by the combination of these two building blocks, and (b) the major part of the GP search concentrates on the extension of the common area.

1.3.3 Empirical analysis of genetic programming

Though schema theorems can predict the expected frequency of schemata for the next generation, they have several limitations. They are available for relatively simple GP models and their derivaton is tedious. Furthermore, the slightest change in the GP model requires complete rewriting of the corresponding schema theorem. Therefore the empirical methods are still valuable tools in understanding GP.

Relatively few studies deal with the empirical analysis of schemata. The inherent problem lies in the huge number of schemata. Different authors handle this with different techniques, the most frequent being: (a) only restricted schemata types are considered, (b) schemata are examined up to a given depth, (c) the test problems feature a restricted number of functions and / or terminals, (d) focusing on schemata that represent sets of schemata and (e) schemata are generated stochastically.

There are two main reasons behind the schemata / building blocks analysis in GP and there are two main approaches corresponding to these. Online identification methods, which are usually cheaper in terms of computational resources are used to explicitly generate functions / building blocks / subroutines that help GP to perform faster and / or generalize better. Offline methods, on the other hand less expensive in terms of CPU cycles and usually focus on understanding the GP runs and / or extract meta-information.

Online methods

The first technique falling in the first category is a modularization method described by Koza in [53]. He proposed an *encapsulation* operator that attempted to identify useful subtrees by selecting individuals using the standard selection scheme followed by picking arbitrary subtrees within the selected individuals. These subtrees were

assigned identifiers so that they could be referenced and reused later. The idea was extended in [54] with *automatically defined functions* (ADFs). The method consists of trees having two separate types of branches: function branches and the main program branch and a set of rules that govern the initial structure and the genetic operations allowed on the branches.

A method that contains both offline and online elements and is in certain ways the extension of ADFs was described by Brock in [55]. The idea was to evolve ADFs that were going to be reused in future runs. The method was severely limited by the fact that it needed a human expert to estimate the usefulness of ADFs.

Another notable usage of online pattern identification is due to Angeline [56]. Dubbed as *module acquisition* it consists of generalizing randomly selected subtrees by cutting down their branches and storing these so called *modules* in a genetic library so that they can be reused. Modules are treated like atomic units, unless they are removed from the genetic library.

In [57] Rosca presented an online adaptive method, *adaptive representation through learning* (ARL), which extracted blocks of code and if promising incorporated them as subroutines. Blocks were selected from individuals which showed the highest *differential fitness*, i.e. individuals for which the difference in fitness from their parents was maximal. The evaluation of a block of code was done based on the *activation* of the block, i.e. the number of times the root node of the block was executed during successive evaluations of the containing individual. ARL was demonstrated on a problem consisting of controlling an agent in a dynamic environment.

An interesting evaluation of schemata is done by Li et al. in [58]. The authors describe a method which evolves schemata directly by the means of an *instruction matrix*. This represents the genetic material of the population along with past measurements of the individuals that were sampled probabilistically from it. The fitness of the sampled individuals influence the instruction matrix after successful crossover and / or mutation. Li et al. report on competitive performance of the method when compared to standard GP on the symbolic regression, even-5-parity, artificial ant and the 11-multiplexer problems.

A more recent work is due to Kameya et al. The online method proposed in [59] is based on the protection of frequent subtrees mined from highly fit individuals. The protection is *soft* in terms that is performed by modifying the probability that these subtrees are destroyed. The method is demonstrated on three problems (symbolic regression, Santa-Fe trail and the royal tree problem). The authors report better results than with standard GP.

Offline methods

An early empirical analysis of genetic programming is due to Tackett, who was interested in extracting the "emergent knowledge from genetically induced programs" [60]. He presented a tree mining algorithm called GeneBanking which extracted all subtrees up to a given depth from offline GP runs. These subtrees (called traits) were then evaluated according to their maximal fitness (or maximal frequency on tie). The fitness of a subtree was defined as the average fitness of individuals that contained the subtree. The method, demonstrated on a constructional problem and a classification problem still needed human expertise to decide whether certain traits were truly salient or hitchhikers (i.e. inert patterns that attached themselves to salient patterns).

An alternative way to tackle the huge number of schemata processed by GP proposed by Smart et al. is to consider so called *maximal fragments* [61]. In the authors notation a fragment closely resembles Langdon-style [44] i.e. a subtree where the "don't care" symbol # stands for a single node that has the same arity as the node containing #. A maximal fragment is "a fragment, occurring in some subset P' of a given set of programs P, which can not be more specialized while retaining the same root, without ceasing to occur in all the programs in P'" [62]. The authors presented an algorithm, Subtree Set Splitter Algorithm (TRIPS) capable of extracting the maximal fragments from populations with 300 individuals, each up to 150 nodes. The algorithm was refined in [62], allowing more general schemata. No attempt was made to assess the usefulness of the identified maximal schemata.

Another notable attempt is the work of Majeed [63]. In his approach Langdon-type hyperschemata are probabilistically generated based on subtrees whose frequency in the last generation of the GP run is larger than 50%. The article focuses on evaluating these schemata by replacing them in container trees by neutral nodes and measuring the relative difference between the two fitnesses. Demonstrated on Koza's symbolic regression problem, the article emphasizes the different behavior of the same schemata in the different GP runs. One of the practical utilization of the method is the schema encapsulation of those patterns that prove to be salient in different runs.

A similar approach of evaluating schema, albeit with different purpose is presented by Langdon et al. in [64] in the analysis of repeated patterns in GP runs. Sensitivity analysis, as the authors named it, replaced subtrees in the container trees by their median values.

Other approaches

Other notable related works include an experimental analysis of schema creation, propagation and disruption using very simple settings and exhaustive search [65]; a hypothesis on how program expressions are evolved through rooted tree schemata

[66] and following the distribution of building blocks and the tendency of sharing components using simplification and compression [67].

1.3.4 Case study: evolution of learning rules of neural networks

An interesting application of EAs is the evolution of learning rules for neural networks. Neural networks (for a thorough treatment see [68]) in the majority of the cases are taught with algorithms which are either based on the Hebbian learning or on some sort of gradient search. The question is whether competitive learning rules can be generated using EAs.

The evolution of learning rules for neural networks usually is a two layered process. The outer layer is the evolutionary one, which produces different learning rules by means of evolutionary operators. The inner layer is a learning process, which evaluates the learning rule. The fitness of a learning rule is usually given by the (scaled) learning error. As the fitness can only be measured indirectly the fitness values are inherently noisy [69].

Though there had been trials to evolve learning rules for neural networks, (e.g. [70, 71]) the first description of the usage of genetic programming in evolving learning rules is reported by Bengio et al. [72]. They tried to find learning rules for a neural network consisting of two input, one hidden and one output unit. The neural network was tested on two dimensional classification problems. The set of terminals $\mathbb{T}$ consisted of the inputs of the network, the error of the network and the first derivative of the output neuron. The set of functions was $\mathbb{F} = \{+, -, *, /\}$. They found that the best evolved learning rule for a neural network closely resembled the backpropagation rule. They have also shown that the rule found by GP is better than rules found by genetic algorithm and simulated annealing with respect to the generalization capability of the neural network.

The work of Bengio et al. was extended by Radi et al. in several steps [73–77]. Their contribution is twofold. On the one hand they extended the search space by considering update rules of form

$$\Delta w_{ij}^l = f(x_{jp}^l, w_{ij}^l, d_{ip}, y_{ip}^l), \tag{1.2}$$

i.e. a function of inputs, weights, target values and outputs. On the other hand they applied a two stage evolution. In the first stage GP was applied to evolve learning rules for the output layer, while the hidden layers were trained with standard backpropagation. In the second stage the best rule from the first stage was used to train the output layer, while GP worked on evolving rules for the hidden layers. The test problems considered were: the XOR problem, N-M-N encoder, a character recognition problem and a display problem [77].

Radi et al. report on rediscovering the delta rule in the first stage for the output layer. The second stage resulted in rules of form $\eta SBP + \beta HB$, i.e. linear combinations of

the standard backpropagation and the Hebbian rule. These rules were found to be more robust in terms of speed, stability and generalization capability than the standard backpropagation with or without speed-up techniques (momentum method, resilient backpropagation).

While the previously mentioned articles [69–75, 77] focus on obtaining performant learning rules for different neural network architectures, relatively little work has been done on understanding and describing the dynamics of the evolution of learning rules.

An effort in this direction is the work of Neirotti et al. In their work [78] they tried to characterize good learning rules by identifying structures which are responsible for the increased fitness. Another aim was to investigate whether the emergence of such structures is ordered in time or not.

As target for the learning rules the authors considered perceptrons, which tried to learn linearly separable data produced by a teacher perceptron. Tree-based, strongly-typed GP was used with terminal set

$$\mathbb{F} = \{\boldsymbol{X}_\mu, \boldsymbol{W}_\mu, y_\mu, d_\mu, h\},\tag{1.3}$$

where

- $\boldsymbol{X}_\mu$ is the input vector

- $\boldsymbol{W}_\mu$ is the weight vector

- y_μ is the perceptron's output

- d_μ is the desired output and

- h is defined as $h = \dfrac{\boldsymbol{X}_\mu \cdot \boldsymbol{W}_\mu}{||\boldsymbol{W}_\mu||}$.

The function set consists of protected versions of square root, exponential, logarithm and division; arithmetic functions; addition, substraction, inner product and normalized inner product of vectors. Inside the fitness assignment phase perceptrons perform online, modified Hebbian learning with the following update rule:

$$\Delta \boldsymbol{W}_\mu = \frac{f d_\mu \boldsymbol{X}_\mu}{\sqrt{N}},\tag{1.4}$$

where f, called the *modulation function*, is the entity that undergoes evolution. The fitness function is a weighted average of the generalization error, i.e.

$$F = \sum_{\mu=1}^{p} \mu e_g(\mu),\tag{1.5}$$

where the generalization error for a training case μ is calculated as

$$e_g(\mu) = \left\langle \Theta(-y_\mu d_\mu) \right\rangle,\tag{1.6}$$

the average discrepancy between the desired and the actual output after presenting μ training examples ($\Theta(\cdot)$ is used to denote the Heaviside function).

The authors define two entropy terms. *Phenotypic* entropy is defined as the entropy of the normalized fitness values, that is

$$S = -\sum_k n^{(k)} \ln(n^{(k)}). \tag{1.7}$$

The *genotypic* entropy is defined as

$$H = -\sum_{s_q} \sum_i \omega(s_q|i) \ln \omega(s_q|i), \tag{1.8}$$

where $\omega(s_q|i)$ is the probability that allele s_q appears at locus i and reflects the microstructure of the individuals [79].

To characterize the internal structure of the programs the densities of two products were measured. *Surprise,* defined as $d_\mu h$ shows whether the classification made by the perceptron is correct or not: it is positive if the classification is correct and it is negative otherwise. *Performance,* defined as $W_\mu \cdot W_\mu$ is strongly correlated to the generalization error in the sense that the higher $||W_\mu||$, the lower e_g.

The authors report on three stages of the evolutionary process.

1. The early stage is characterized by relatively low fluctuations of the genotypic entropy and no occurrence of the performance term. Neither surprise nor performance is used.

2. The second is the transition phase that occurs around the middle of the evolutionary run. It brings bigger fluctuations in the phenotypic entropy, a sudden drop in the mean length of the individuals and in the fitness, a decrease in the density of surprise and shortly after that a sharp increase in the density of performance.

3. The late stage is characterized by decreasing trend in the phenotypic entropy and an increased usage of performance. Both surprise and performance are used correctly.

1.4 Structure of the thesis

The main contribution of this thesis is a set of algorithms, called collectively Extended Pattern Growing Algorithm ([E]PGA) to identify salient patterns in offline GP individuals. These algorithms, the test problems, the experiments performed and the results are organized as follows.

In Chapter 2 we discuss six problems which will serve as test beds for experiments. In Chapter 3 we describe [E]PGA and we analyse it from different perspectives. Chapter 4 contains the outcome of the performed experiments and identifies the forte and weak points of the algorithm in the light of these experiments. The following chapters contain a discussion on the method, we draw conclusions and list some alternative directions for future work. The Appendix highlights some of the practical issues we have encountered and gives details about the implementation.

Chapter 2

Test problems

Six test problems were considered during the analysis of [E]PGA. The idea was to choose problems that are known and / or representative for certain class of problems: two constructional problems of different difficulty, a symbolic regression and a classification problem, a boolean function reconstruction problem and one problem from the domain of evolution of learning rules for perceptrons.

All these problems featured generational, elitist, tree-based GP with tournament selection. The representations, fitness functions, the number of generations and other settings are detailed in the description of each problem individually.

2.1   TCP

This problem is equivalent to Tackett's constructional problem [60]. Let $\{P_1, P_2, \ldots, P_k\}$ be a set of patterns and let $\rho : \{P_1, P_2, \ldots, P_k\} \to \mathbb{R}$ be a reward function. The fitness of an individual χ is given by $f(\chi) = \max\{\rho(P_i)|P_i \text{ a subtree of } \chi\}$, i.e. the maximum reward among all subtrees of χ. The reward function is defined as:

$$\rho(P) = \begin{cases} 0.125 & \text{if } P = (z + x) \text{ or } P = (x - y) \\ 0.25 & \text{if } P = ((z + x) * \mathbf{S}) \text{ or } P = (\mathbf{S} * (x - y)) \\ 0.5 & \text{if } P = ((z + x) * (x - y)) \\ 0 & \text{otherwise,} \end{cases} \tag{2.1}$$

where $\{x, y, z\}$ and $\{+, -, *, .*, /, \exp()\}$ are the sets of allowed GP terminals and functions and $\mathbf{S}$ stands for any subtree.

The GP process consisted of 300 individuals evolving through 30 generations. The allowed tree heights were between 3 and 6.

(a) Level–*a* tree   (b) Level–*b* tree   (c) Level–*c* tree
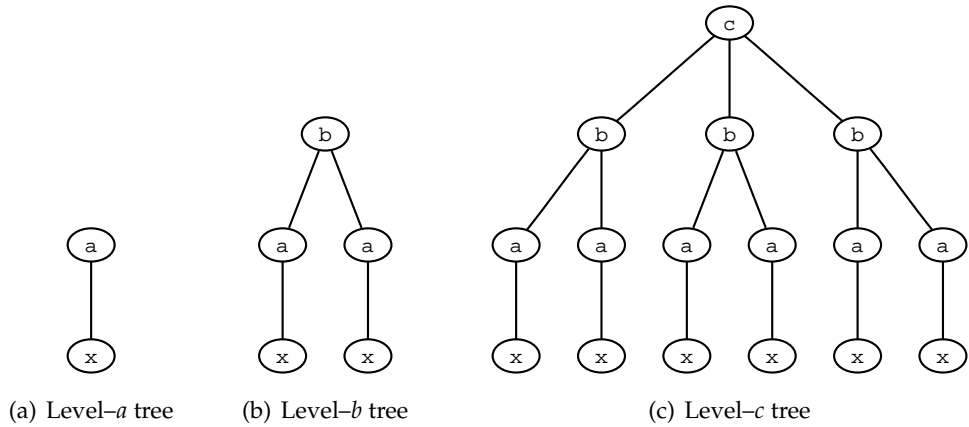
Figure 2.1: RTP. Original representations for perfect trees of level *a*, *b* and *c*.

2.2 RTP

RTP is a modified version of the royal tree problem from [80]. The objective is to build *perfect trees* that are subject to certain syntactical constraints. Recursively, the shape of a perfect tree can be defined as follows. A tree with a single node has a perfect shape. A tree of height $n > 2$ has perfect shape iff its root is the parent of $n - 1$ perfect trees. A tree is called perfect iff it has a perfect shape and the keys in the nodes on the different levels are the following: x for the leaves, a for the nodes on the first level, b for the nodes on the second level, etc. Figure 2.1 depicts three perfect trees successive levels.

GP was run with the following settings: 300 individuals evolved through 70 generations.

2.2.1 Implementation

The implementation of RTP is slightly different from the one presented in [80] in the sense that n-ary trees are represented as binary trees using the first-child-next-sibling technique. This is because of limitations of the implementation of the evolutionary module in [E]PGA, which can handle only binary trees.

Nevertheless, the change in representation introduced more functions and broke the symmetry, thus complicating the original problem. Individuals were built using the $\{a(), b(), c(), A(), B(), C(), x\}$ set of primitives, where a, b and c are unary functions and A, B and C are binary functions.

Examples on how level–*a*, level–*b* and level-*c* trees are transformed to binary trees are given in Figure 2.2.

The fitness of an individual is calculated according to [80]: "...the raw fitness of a subtree is the score of its root. Each function calculates its score by summing the weighted scores of its direct children. If the child is a perfect tree of the appropriate

31

(a) Level–*a* tree   (b) Level–*b* tree   (c) Level–*c* tree
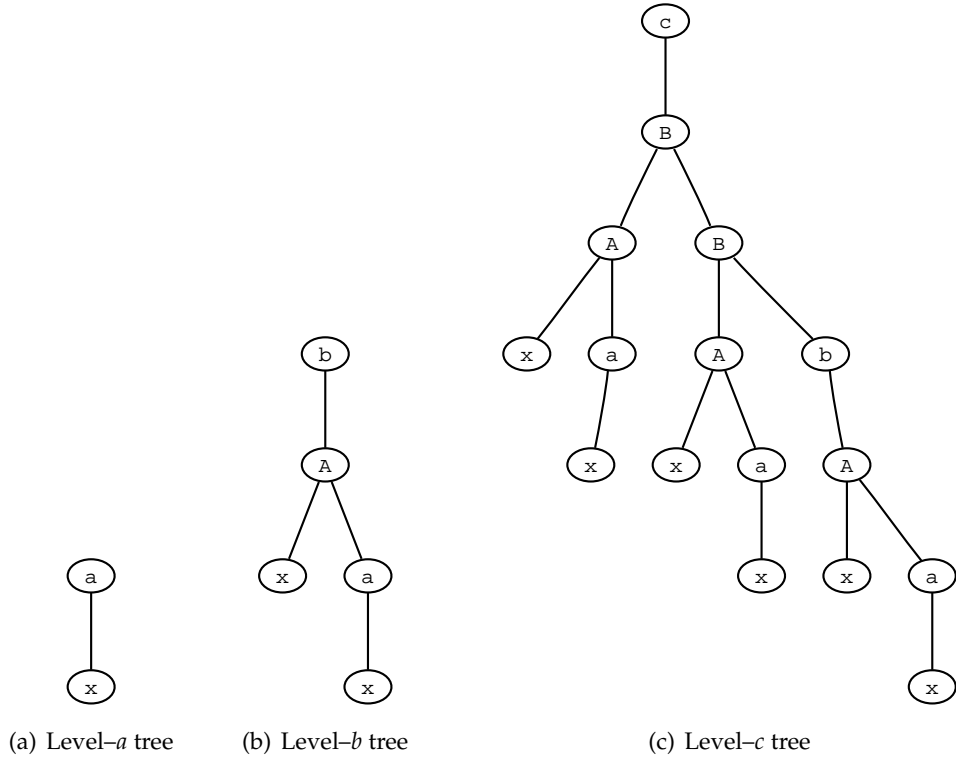
Figure 2.2: RTP. Binary representations for perfect trees of level *a*, *b* and *c*.

Table 2.1: RTP. Fitness values of the first 4 perfect trees

Pattern	Fitness
x	1
a(x)	4
b((xAa(x)))	32
c(((xAa(x))B((xAa(x))Bb((xAa(x))))))	384

level (for instance, a complete level-c tree beneath a d node), then the score of that subtree, times a FULL_BONUS weight, is added to the score of the root. If the child's root is incorrect, then the weight is PENALTY. After scoring the root, if the function is itself the root of a perfect tree, the final sum is multiplied by COMPLETE_BONUS. Typical values used are: FULL_BONUS=2, PARTIAL_BONUS=1, PENALTY=1/3, and COMPLETE_BONUS=2. The score base case is a level-a tree which has a score of 4 (the a–x connection is worth 1 times the FULL_BONUS, times the COMPLETE_BONUS)".

The fitness values of the first 4 perfect trees are given in Table 2.1.

Because of the changed representation, the routine that calculates the fitness has to be changed as well. The steps performed are described in Algorithm 2.1.

Algorithm Evaluate(*r*) first checks whether *r* is leaf and if so, it returns the corresponding score 1 or 0 depending on whether it is a perfect tree (steps $1 - 7$). If *r* is not a leaf it initializes three variables: *c* to the first child of *r*, the score of *r*, *s* to 0 and the number of evaluated children of *r*, *nr_child* to 0 (steps $8 - 10$).

The loop that follows progressively calculates the score of the root. First it calculates the weight associated to the child c (steps 12 − 21), then updates the root's score by the weighted score of the child (step 22) and jumps to the sibling of c. Because of the first-child-next-sibling codification this will be the right child of c. Finally it updates the number of evaluated children of r. The loop continues until all the children of r are evaluated (step 11). In the last step, the algorithm checks whether r itself is a perfect tree. If so, its score is multiplied by COMPLETE_BONUS.

Algorithm 2.1 Evaluate(r)

1: **if** $r.size = 1$ **then**
2:    **if** IsPerfectTree($r, 0$) **then**
3:       **return** 1
4:    **else**
5:       **return** 0
6:    **end if**
7: **end if**
8: $c = r.left$
9: $s = 0$
10: $nr\_child = 0$
11: **while** $c \neq null$ **and** $nr\_child < r.key.arity$ **do**
12:    $w = 0$
13:    **if** IsPerfectTree(c) **and** ProperSon(c, r) **then**
14:       $w = $ FULL_BONUS
15:    **else**
16:       **if** ProperSon(c, r) **then**
17:          $w = $ PARTIAL_BONUS
18:       **else**
19:          $w = $ PENALTY
20:       **end if**
21:    **end if**
22:    $s = s + w \times$ Evaluate(c)
23:    $c = c.right$
24:    $nr\_child = nr\_child + 1$
25: **end while**
26: **if** IsPerfectTree(r) **then**
27:    $s = s \times$ COMPLETE_BONUS
28: **end if**
29: **return** s

Proper_Son(), formalized in Algorithm 2.2 checks the correctness of a child node with respect to its father. (For example if the root is a C node then all of its children should be B nodes). In doing so it uses two arrays which contain the unary and binary functions. In both arrays the functions are sorted alphabetically.

If *child* consists of a single node then Proper_Son returns true iff *child* is equal to x and its father is $a()$ or $A()$ (steps 1 − 11). If *child* consists of more than a node then Proper_Son calculates two indices: $fidx$ and $cidx$. These denote the positions of

father.key and *child.key* in the array of the unary or binary functions. *child* is a proper son of *father* iff $fidx - 1 = cidx$.

Algorithm 2.2 ProperSon(*child*, *father*)

1: **if** child.size = 1 **then**
2:    **if** IsPerfectTree(*child*) **then**
3:       **if** *father.key* = UNARY_FUNCTIONS[0] **or**
      father.key = BINARY_FUNCTIONS[0] **then**
4:          **return** true
5:       **else**
6:          **return** false
7:       **end if**
8:    **else**
9:       **return** false
10:    **end if**
11: **end if**
12: $fidx = \text{idx\_of}(father.key, \text{UNARY\_FUNCTIONS}, \text{BINARY\_FUNCTIONS})$
13: $cidx = \text{idx\_of}(child.key, \text{UNARY\_FUNCTIONS}, \text{BINARY\_FUNCTIONS})$
14: **return** $fidx - 1 = cidx$

2.3 XSRP

XSRP is the classical symbolic regression problem $(x^4 + x^3 + x^2 + x + 1)$ taken from [53]. Individuals were built using the $\{+, -, /, *, x\}$ set of primitives. 100 points sampled uniformly from the $[-10, 10]$ interval served as training data. Maximum tree height was set to 9. The fitness of an individual χ was given by

$$f(\chi) = -\sum_i |\chi(x_i) - y_i|,$$

where (x_i, y_i) is the *i*th pair from the training set and $\chi(x_i)$ the evaluation of χ in x_i. GP evolved 200 individuals through 100 generations.

2.4 Donut

This classification problem was described in [60]. The task is to classify points from two interlinked chain links with radii $r = 1$ and $R = 5$ as shown in Figure 2.3. The training data consisted of 1000 points sampled uniformly from the two chain links. The fitness of an individual χ is given by

$$f(\chi) = -\sum_i |\operatorname{sgn}(\chi(x_i, y_i, z_i)) - t_i|,$$
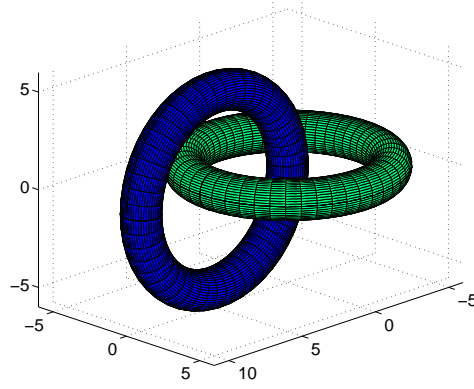
Figure 2.3: Donut problem. Described in [60], the task is to evolve a classifier that separates the two chain links.

where $(x_i, y_i, z_i, t_i) \in \mathbb{R}^3 \otimes \{\pm 1\}$ is the ith quadruple from the training set, representing a 3D point and its classification, and $\chi(x_i, y_i, z_i)$ is the evaluation of χ for (x_i, y_i, z_i). Individuals were built using the $\{+, -, /, *, \text{sqrt}, x_0, x_1, x_2, \varepsilon\}$ set of primitives, where $\varepsilon \in \{1, 2, 3, 4, 5\}$ is the ephemeral random constant (ERC) [53]. Maximum tree height was set to 10. GP evolved 1000 individuals through 50 generations.

2.5 MP6

MP6 is the 6-multiplexer problem as described by Koza in [53]. The fitness of an individual χ is given by

$$f(\chi) = \frac{1}{64} \sum_{i=1}^{64} |M(x_i) - \chi(x_i)|,$$

where $M(x_i)$ is the 6-multiplexer output for input x_i and $\chi(x_i)$ is the evaluation of χ for input x_i. Individuals were built using the $\{\text{xor, and, not, or, }A0, A1, D0, D1, D2, D3\}$ set of primitives[1]. Maximum tree height was set to 8. GP evolved 500 individuals through 50 generations.

2.6 ELRA

The evolution of learning rules (ELRA) for binary perceptrons is the most computationally intensive test problem. The settings are similar to [78]. Untyped, tree based, parallel and coarse grained GP is used for the evolution of learning rules. The set of terminals is $\mathbb{T} = \{x_\mu, w_\mu, y_\mu, d_\mu, \varepsilon\}$, where x_μ denotes the input vector, w_μ the weight vector, y_μ the perceptron's output, d_μ the desired output and $\varepsilon \in \{1, \ldots 10\}$ is the ERC.

---

[1]In actual expressions we use the C syntax for the operators: ^, &, ! and |.

The set of functions is $\mathbb{F} = \{-, +, *\}$. All these functions are defined both for vectors and scalars and work like the operators with the same name from MATLAB.

The fitness of an individual is given by weighted average of the generalization error taken with negative sign, i.e.

$$F = -\sum_{\mu=1}^{p} \mu e_g(\mu). \tag{2.2}$$

The generalization error for a training case μ is calculated as the average discrepancy over the training set between the desired and the actual output after presenting μ training examples to the learning algorithm, i.e.

$$e_g(\mu) = \langle \Theta(-y_\mu d_\mu) \rangle, \tag{2.3}$$

where $\Theta(\cdot)$ is the Heaviside step function.

The following settings were used in ELRA. The number of inputs of the student perceptron was set to 16. Prior teaching the weight vector from the student perceptron was initialized to $(111\ldots1)$. Allowed tree depth ranged from 2 to 15. The number of teaching epochs (equal to p from Equation 2.2) was 512. The number of training subsets considered in Equation 2.3 over which the average was taken was 8. The training examples were generated using a spherical perceptron. GP evolved 600 individuals through 50 generations.

Chapter 3

[E]PGA

We present the Extended Pattern Growing Algorithm ([E]PGA), a set of methods used to extract, sort and filter patterns from offline tree-based GP data. The goal is to produce a set of patterns that meet as many as possible of the following three criteria: (a) they are representative for the evolutionary run and/or search space, (b) they are human-friendly and (c) their number are within reasonable limits.

[E]PGA works on offline individuals that have been serialized during the evolutionary run. It consists of three parts: preprocessing the serialized GP individuals, mining for patterns with an algorithm called Pattern Growing Algorithm (PGA), and filtering the resultant set of patterns using a set of routines which will commonly be referred to as PostPGA (see Figure 3.1).
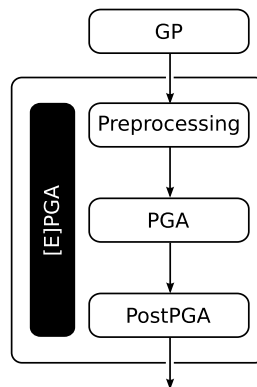

Figure 3.1: Schematic view of [E]PGA.

Notation

We define a *pattern* as a GP subtree that may contain *wild character*s in any of its nodes. A wild character matches any single node of the same arity. We consider three types of wild (joker) characters: @ for terminals, # for unary functions and ## for binary functions.
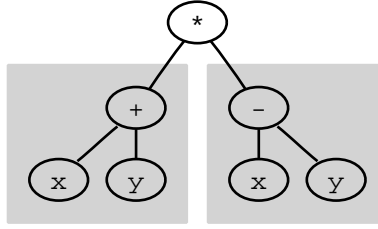
Figure 3.2: Tree containing twice the $(x\#\#y)$ pattern. The pattern's order ratio is $\frac{2}{3}$.

A pattern P is said to be *contained* in a tree T, if T contains at least one subtree T' such that every node of T' matches the corresponding node in P. A pattern can be contained multiple times in a tree. For example the tree $((x + y) * (x - y))$ contains twice the pattern $(x\#\#y)$.

Our pattern definition is similar to the schema definition used by Poli and Langdon in [81] with the exception that our patterns are positionless thus can be present multiple times in a GP individual. We have chosen this kind of definition for two reasons: (1) the Langdon and Poli type schemas are too big (because of the rootedness constraint) and thus too numerous to be analyzed within reasonable time; (2) it is not obvious how to generate and mine for hyperschemas if we allowed them [44].

We define the *order ratio* of a pattern as the proportion of non-wild character nodes in the total number of nodes of a pattern. This measure is more suitable than the *order* (which is equal to the number of non-wild characters in a pattern) because it provides more flexibility when working with patterns of different size.

We define the *frequency* of a pattern in a population of GP trees as the number of occurrences of the pattern in the population divided by the size of the population. Because a pattern can be present multiple times in an individual, the frequency of a pattern can be larger than 1.[1] Frequency measures the degree of proliferation of a given pattern in a population.

The *fitness* of a pattern is the average fitness of all individuals from the population which contain the given pattern at least once.

3.1 Preparation

Originally, as a preparatory step, we proposed the aggressive simplification of individuals [82], with a modified version of Equivalent Decision Simplification (EDS) [83] to clear them of bloat code and to process only the expressed parts. EDS detects equiv-

---

[1] A way of having a normalized frequency, i.e. in the $[0, 1]$ interval is to divide the number of occurrences by the total number of subtrees in the population. However, as this later is a large number, it would render the frequency almost infinitesimal.

alent subtrees non-deterministically and usually gives a good compression ratio. The modified version is described in Appendix, p.85.

However, as anonymous reviewers have pointed out, simplification may destroy important evolved patterns and the chosen simplification algorithm may introduce certain biases that would influence the set of resulting patterns. Furthermore, for some representations the simplification is not straightforward (e.g. artificial ant problem, lawnmower problem [53]). Because this question needs a thorough investigation that is out of scope of this thesis we let the preparatory simplification remain optional.

3.2   Pattern Growing Algorithm

After the simplification step, using an algorithm derived from tree-mining techniques [84], we identify those patterns from each generation whose frequency, height and order ratio are within some user defined ranges. The rationale behind these criteria are the following.

- We know from schema theorems that the number of important patterns increases exponentially (at least at the beginning). Therefore it seems reasonable to assume that there is a certain frequency threshold under which patterns are not important, and can be safely filtered out. This frequency threshold will be denoted by ε_f.

- Big patterns are hard to understand and very small patterns (e.g. of height 1) are uninteresting. Moreover, interpreting the results is easier if the resulting patterns are divided into different height classes. The set of allowed heights will be denoted by $\{\varepsilon_h, \ldots, \varepsilon_H\}$.

- Filtering by order ratio is necessary because low order ratio patterns carry low amounts of usable information. For example the pattern $(@\#\#(@\#\#d))$ is less informative than $(@ * (x * d))$ though both have the same structure and height. The interval of allowed order ratio values will be denoted by $[\omega, \Omega]$. [2]

The basic principle behind PGA is the same as that behind most tree-mining algorithms [84]. PGA generates candidate patterns and tests whether these patterns adhere to given criteria. Patterns fulfilling the criteria are collected and processed further with PostPGA.

A feature that distinguishes our algorithm from conventional tree mining techniques is the support of joker characters, which stand for single nodes of the appropriate arity.

Another property that makes our algorithm different is the way in which candidate patterns are built. Due to syntactic constraints imposed by GP trees, namely that GP

---

[2]Ω is for the sake of generality only, in all experiments it was set to 1.0.

subtrees can not be extended by adding children nodes to leaves, traditional tree mining algorithms can not be used. Instead patterns are grown incrementally from above, by adding roots and inserting existing patterns underneath.

3.2.1 The algorithm of PGA

PGA takes six parameters: the minimum and maximum allowed heights for a pattern (ε_h and ε_H), the minimum frequency threshold for a pattern (ε_f), the boundaries for the order ratio (ω and Ω) and the population which is to be mined ($\mathbf{P}_t$). $\mathbf{P}_t$ contains the individuals from generation t of the GP run and it uses the same internal representation as the GP code does.

PGA returns the set of patterns $\mathbb{P}_t$ from $\mathbf{P}_t$ whose height, frequency and order ratio are between the specified limits. The steps performed are formalized in Algorithm 3.1.

Algorithm 3.1 PGA(ε_h, ε_H, ε_f, ω, Ω, $\mathbf{P}_t$) returns $\mathbb{P}_t$

1: $h = 1$
2: Let $\mathbb{U}$, $\mathbb{B}$ and $\mathbb{P}_h$ be the set of unary, binary functions and terminals whose frequencies in $\mathbf{P}_t$ are above ε_f. Append the corresponding joker characters to each of these sets.
3: If $h = \varepsilon_H$ filter $\mathbb{P}_h$ by ω, Ω and ε_h and return $\mathbb{P}_h$.
4: $h = h + 1$
5: Denote by $\mathbb{P}_h^U$ the set of patterns of height h grown by placing each element of $\mathbb{P}_{h-1}$ under each element of $\mathbb{U}$.
6: Denote by $\mathbb{P}_h^B$ the set of patterns of height h grown by making root nodes each element of $\mathbb{B}$, picking left subtrees from $\mathbb{P}_{h-1}$ and right subtrees from $\bigcup\limits_{i=1}^{h-1} \mathbb{P}_i$.
7: **if** $h > 2$ **then**
8:     take each element of $\mathbb{P}_h^B$, make a copy of it, swap left and right subtrees of the root, and put back into $\mathbb{P}_h^B$.
9: **end if**
10: $\mathbb{P}_h = \mathbb{P}_h^B \bigcup \mathbb{P}_h^U$. Filter $\mathbb{P}_h$ by ε_f and ω.
11: Jump to 3.

PGA starts building candidate patterns from terminals, whose height is $h = 1$ and continues by gathering those unary and binary functions and terminals whose frequencies in $\mathbf{P}_t$ are above ε_f. These sets are denoted by $\mathbb{U}$, $\mathbb{B}$ and $\mathbb{P}_1$ respectively. Joker character # is inserted into $\mathbb{U}$, ## is inserted into $\mathbb{B}$ and @ is inserted into $\mathbb{P}_1$ (step 2).

If h has reached its maximum value, we delete all those patterns from $\mathbb{P}_h$ whose order ratio are not within the allowed limits (ω and Ω). Patterns whose height is less than ε_h are deleted as well (step 3).

If h has not reached the maximum limit ε_H, it is incremented and PGA continues by building candidate patterns of height h (step 4).

Patterns of depth h can be built in two ways: either putting the elements of $\mathbb{P}_{h-1}$ under a unary function node or by putting these elements under a binary node. The first case

is straightforward: each element from $\mathbb{P}_{h-1}$ is paired with every element of $\mathbb{U}$, and the resultant patterns are put in $\mathbb{P}_h^U$ (step 5).

When the roots are picked from the set of binary functions, care must be taken not to regenerate candidate patterns that have been generated already. To avoid this, PGA uses the following approach. For each binary function from $\mathbb{B}$, the left subtrees are taken from $\mathbb{P}_{h-1}$ and the right subtrees from *all* the patterns generated so far, $\bigcup_{i=1}^{h-1} \mathbb{P}_i$. The procedure is repeated by taking the left subtrees from $\bigcup_{i=1}^{h-1} \mathbb{P}_i$ and the right subtrees from $\mathbb{P}_{h-1}$. The resultant set of patterns is put in $\mathbb{P}_h^B$ (steps $6-8$).

All patterns from the union of $\mathbb{P}_h^U$ and $\mathbb{P}_h^B$ are filtered for frequency and minimum order ratio in step 10. This means that patterns whose frequencies are below ε_f and whose minimum order ratio will not reach ω are deleted. These filtering steps are detailed in Sections 3.2.3 and 3.2.5.

The last step of the algorithm closes the loop and jumps back to step 3.

Example. Suppose that the population we want to analyze, $\mathbf{P}_t$, contains the following individuals

$$\mathbf{P}_t = \{\exp(w), (d-y), (d-w), ((d-y)*x)\},$$

where $\{d, y, x, w\}$ is the set of terminals and $\{\exp(), -, *\}$ is the set of functions. Furthermore, suppose that we are interested in patterns of height 2 and 3, with minimum frequency $\varepsilon_f = 0.6$ and order ratio in $[0.6, 1]$.

PGA starts by building the patterns of height 1, i.e. the set of terminals. Because only d is frequent enough, $\mathbb{P}_1$ will contain only d and @. $\mathbb{U}$ will be empty since the only unary function $\exp()$ is not frequent enough. Note that in this case we do not add the unary joker character # to $\mathbb{U}$ because there is no unary function to generalize. The set of binary functions $\mathbb{B}$ will contain two elements: $-$ and ##.

The algorithm continues with building patterns of height 2, $\mathbb{P}_2$. Among the possible ways to build patterns of height 2, only 3 will make their way into $\mathbb{P}_2$: $(d-@)$, $(@-@)$ and $(d\#\#@)$. The set of patterns of height 3 will be empty because no pattern of height 3 meets the frequency and order ratio criteria.

The final step of the algorithm (3) filters $\mathbb{P}_1 \bigcup \mathbb{P}_2 \bigcup \mathbb{P}_3$ by ε_h (deleting d and @) and by ω (deleting $(@-@)$ and $(d\#\#@)$). Thus PGA will return a single pattern, $(d-@)$.

3.2.2 Computational complexity of PGA

Without any filtering in step 10 of Algorithm 3.1, PGA generates a prohibitive number of candidate patterns. Even for a few functions, terminals and relatively small h, the number of candidate patterns, $c(h)$ reaches astronomical values. Consider for example Figure 3.3, showing the evolution of $c(h)$ when $|\mathbb{U}| = |\mathbb{B}| = |\mathbb{P}_1| = n$, $n = 2, 3, 4$.

Figure 3.3: Evolution of the total number of candidate patterns on logarithmic scale when PGA does not use any filtering. The number of unary and binary functions and terminals are set to $n = 2, 3, 4$.

The total number of candidate $c(h)$ patterns generated by PGA without the filtering in step 10 can be calculated using the following recursive formula:

$$
c(h) = \begin{cases}
|\mathbb{P}_1| & \text{if } h = 1, \\
|\mathbb{U}||\mathbb{P}_1| + |\mathbb{B}||\mathbb{P}_1|^2 & \text{if } h = 2, \\
|\mathbb{P}_{h-1}| \left(|\mathbb{U}| + 2|\mathbb{B}| \sum_{i=1}^{h-1} |\mathbb{P}_i| \right) & \text{otherwise.}
\end{cases}
\tag{3.1}
$$

Profiling showed that the most costly operation in PGA is the frequency calculation. Thus from now on we will consider the frequency calculation as the atomic operation that determines the computational complexity of PGA.

The total number of candidate patterns generated can be approximated by expanding Equation 3.1. The following list shows the terms with the highest exponents of the expansion of Equation 3.1, for $h = 2, \ldots, 6$.

- $c(2) = |\mathbb{B}||\mathbb{P}_1|^2 + \ldots$

- $c(3) = 2 \, |\mathbb{B}|^3 |\mathbb{P}_1|^4 + \ldots$

- $c(4) = 2^3 \, |\mathbb{B}|^7 |\mathbb{P}_1|^8 + \ldots$

- $c(5) = 2^7 \, |\mathbb{B}|^{15} |\mathbb{P}_1|^{16} + \ldots$

- $c(6) = 2^{15} \, |\mathbb{B}|^{31} |\mathbb{P}_1|^{32} + \ldots$

It is easy to see that $c(h)$ grows super-exponentially. In the following we present three techniques (one already described in the literature and two novel) which speed up PGA.
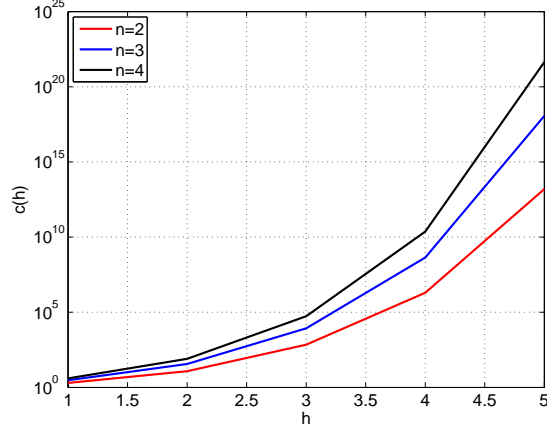
Figure 3.4: Evolution of the total number of candidate patterns on logarithmic scale when PGA uses the *a priori* filtering. The number of unary and binary functions and terminals are set to $n = 2, 3, 4$.

3.2.3 The a priori property

The *a priori* property of pattern inclusion [85] means the following. If a pattern P has frequency ε_f in a population then no pattern containing P can have larger frequency than ε_f. This means that filtering by ε_f has to be done each time a set of candidate patterns of height h is generated (step 10).

To see how the a priori property modifies PGA let us suppose that the proportion of candidate patterns that fulfill the frequency criterion decreases exponentially with the height of the patterns. The total number of candidate patterns $c_{ap}(h)$ in this case will be

$$c_{ap}(h) = \alpha e^{-h} c(h), \tag{3.2}$$

where α is a real constant. Figure 3.4 depicts this behavior with $\alpha = 2$. It can be seen that on this hypothetical case a priori decreases the computational complexity by several order of magnitudes.

A priori on real data

To check how the implementation of the apriori property affects real data, we calculated the total number of candidate patterns per generation. The averages for 50 runs for different problems shown in Figure 3.5 reveal that real data behaves significantly better than the hypothetical data we have used before, showing a much better increase rate.

3.2.4 Faster lookup and frequency calculation

For each candidate pattern P we have to decide whether its frequency is above ε_f or not. The naïve approach of parsing all trees from the population for every single
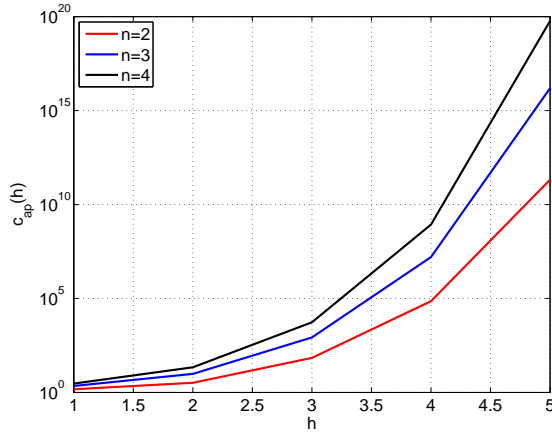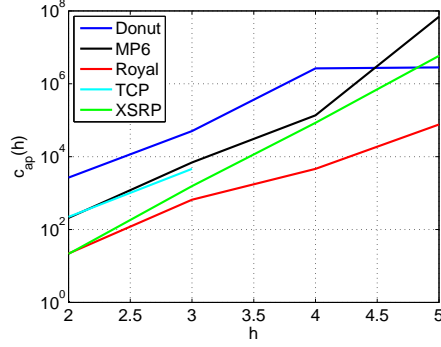
Figure 3.5: Evolution of the total number of candidate patterns per generation (on logarithmic scale) when PGA uses the *a priori* filtering for real data.

pattern is infeasible because of the sheer number of candidate patterns. To decrease the number of times when we have to check whether a (sub)tree contains a pattern or not we designed a hashing scheme. Though this does not reduce the computational complexity of PGA, it speeds up the atomic operation, i.e. the frequency calculation of a candidate pattern. First we introduce the hashing function itself then we explain its usage.

Skeleton hash

We introduced the skeleton hash [82] as a hash function which encodes the structure of a subtree or pattern. It is calculated using a preorder traversal without the node content information, as shown in Algorithm 3.2. SkeletonHash takes a pattern or a subtree X and returns a binary string σ_X, which is used in later algorithms as an unsigned integer.

Algorithm 3.2 SkeletonHash(X) returns σ_X

1: $\sigma_X = \emptyset$
2: Traverse X in a preorder way. On entering a node append σ_X 1, on exiting a node append σ_X 0.
3: **if** height(X) = 1 **then**
4:     $\sigma_X = 0$
5: **end if**
6: **return** σ_X.

An interesting property of the skeleton hashing is that if a pattern is extended (from above as it is done in PGA) then its skeleton hash can be readily extended as well without recalculating the parts we already know. For example if we extended pattern $((x + @) * y)$ with skeleton hash 11010010_2 from above, let us say $\exp(((x + @) * y))$, then the skeleton hash would be $\mathbf{1}110100100_2$. An example is given in Figure 3.6.
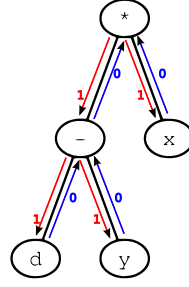
Figure 3.6: Skeleton hashing of $((d - y) * x)$. Whenever the preorder traversal enters a node the resulting binary string is appended a one, and whenever it exits a node, it is a appended a 0. The skeleton hash of $((d - y) * x)$ is $11010010_2 = 210_{10}$.

Using the skeleton hash

The hash function is used to speed up the frequency calculation of candidate patterns.[3] The first step consists of distributing all subtrees of a population P into a hashtable H. The second step uses H for the frequency calculation.

Each element of H contains a list of pairs (S, n_S), where S is a subtree and n_S is the number of occurrences of S in the population. The position where a subtree S is chained in H is given by $\sigma_S \mod |H|$, where σ_S is the hash value of S and $|H|$ is the size of the hashtable. These steps are formalized in Algorithm 3.3.

Algorithm 3.3 BuildHashtable(P) returns H

1: $H = [\text{nil}, \text{nil}, \cdots, \text{nil}]$
2: **for all** individuals T in population P **do**
3:     **for all** subtrees S of T **do**
4:         $\sigma_S = \text{SkeletonHash}(S)$
5:         $i_S = \sigma_S \mod |H|$
6:         $L = H[i_S]$
7:         Let $(S', n_{S'})$ be the pair in L so that $S = S'$.
8:         **if** $(S', n_{S'})$ does not exist **then**
9:             Append $(S, 1)$ to L.
10:        **else**
11:            $n_{S'} = n_{S'} + 1$
12:        **end if**
13:     **end for**
14: **end for**
15: **return** H

The algorithm works as follows. First it initializes all positions of the hashtable with null pointers (step 1). Afterwards it loops through the individuals of population P (step 2) and for each individual it considers every contained subtree S (step 3). If S is not yet present in the hashtable it should be inserted into it. If it is present then

---
[3] Because the skeleton hash was implemented at a very early stage of the project we do not have a rigorous comparison of the performance of PGA with and without it, except on the ELRA problem. This showed a 30–40-fold increase in speed compared to PGA without skeleton hash.

Index	Hash	Elements
0	0	w:3, x:2, d:2, y:1
1	210	$((d - y) * x)$:1
2	2	$\exp(w)$:1
...		
10	10	$(w * w)$:1, $(x * d)$:1, $(d - y)$:1
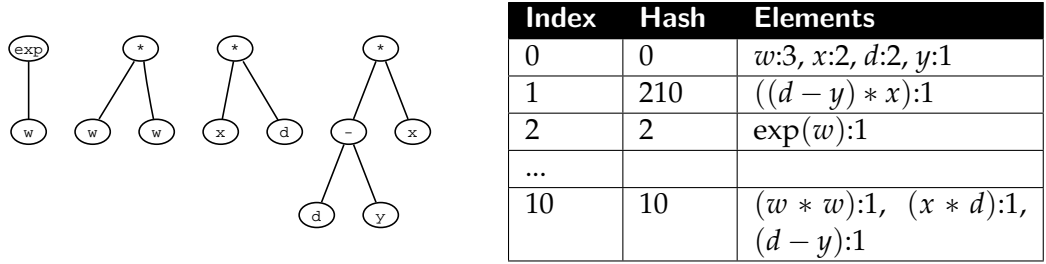
Figure 3.7: Fast frequency calculation with skeleton hashing. The figure presents a population of 4 GP individuals and the corresponding hash table (indices and skeleton hashes are shown for the sake of intelligibility only). Subtrees are stored along with their number of occurrences in the population. If one wants to find the frequency of a pattern, let's say $(@ * @)$, one calculates the skeleton hash of this pattern (10), computes the index (10 mod 11 = 10) and sums the occurrence of those subtrees in row 10 that match the pattern (2). The sum is divided by the number of individuals, which gives 0.5.

the associated counter should be incremented by one. In both cases the first step is to calculate the slot where S is or should be linked (steps 4–7). The loop ends with linking S into H with occurrence 1 (step 9) or incrementing the counter associated with S (step 11).

After H has been built, it is used for the fast lookup of those subtrees from H that match a pattern P and for the calculation of the frequency of P. The actual steps are given in Algorithm 3.4.

Algorithm 3.4 FastFrequency($P, H, |P|$) returns f_P

1: $\sigma_P = \text{SkeletonHash}(P)$
2: $i_P = \sigma_P \mod |H|$
3: $L = H[i_P]$
4: Parse the list of pairs (S, n_S) from L, and sum the occurrence numbers whenever P matches S. Denote this number by Σ_P.
5: **return** $\dfrac{\Sigma_P}{|P|}$.

FastFrequency takes three parameters: P, the pattern whose frequency has to be calculated, H, the hash table and $|P|$, the size of the population P. The algorithm first calculates the hash of P and jumps to the slot in H where the subtrees of P with the same hash are linked (steps $1 - 3$). Afterwards it calculates the sum of occurrences for those subtrees in list L which match P. The frequency returned is given by this sum divided by the population size. An example is shown in Figure 3.7.

3.2.5 Filtering by minimum order ratio

Consider the pattern P shown in Figure 3.8. It has an order ratio $O_P = 0$. Let us suppose that the maximum pattern height allowed is 3 and the minimum order ratio
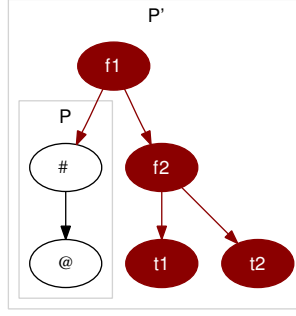
Figure 3.8: Filtering by minimum order ratio. The growth of pattern P can be stopped if it does not reach the minimum order ratio irrespective of the content of the new nodes.

$\omega = 0.8$. This means that even if all the remaining nodes of an extended pattern P' (containing P) are non-jokers, the order ratio of P' will not be larger than $0.\dot{6}$.

This implies that during the candidate pattern generation phase, the growth of a pattern P can be stopped if it does not reach the minimum order ratio ω. The stopping criterion can be formalized as

$$\omega > 1 + \frac{O_P - S_P}{S_M - S_{M_P} + S_P},\tag{3.3}$$

where O_P is the order of P, S_P is the size of P, S_M is the size of a complete binary tree of maximum allowed height for a pattern and S_{M_P} is the size of a complete binary tree of the same height as P. This filter is applied at step 10 of the PGA algorithm.

We compared PGA with and without the minimum order ratio filtering (MORF) by looking at 50 independent runs. Tables 3.1 and 3.2 shows the average number of generated candidate patterns without and with MORF. It can be seen that apart from patterns of height 2 all cases show improvements.

Table 3.1: MORF. Average number of candidate patterns per generation without/with MORF, heights 2, 3.

	Height 2	Height 3
XSRP	21.41 / 21.41	1548.99 / 692.87
RTP	21.89 / 21.89	656.74 / 320.68
Donut	2669.42 / 2669.42	50300.69 / 29360.89
MP6	208.93 / 208.93	6921.40 / 3237.47
TCP	222.13 / 222.13	4629.56 / 1586.79
ELRA	97.68 / 97.68	2069.30 / 751.92

3.2.6 Late patterns

Good patterns emerging in late generations may not have the time to spread fast enough to reach the minimum frequency threshold ε_f. Lowering this threshold close

Table 3.2: MORF. Average number of candidate patterns per generation without/with MORF, heights 4, 5.

	Height 4	Height 5
XSRP	85055.41 / 24628.13	5829241.61 / 1303021.06
RTP	4609.70 / 2621.24	76399.28 / 41774.97
Donut	2642695.52 / 883459.88	2820138.73 / 678485.62
MP6	135932.99 / 41628.26	69214313.75 / 7660187.13
ELRA	6264.05 / 2782.38	6347.12 / 6091.71

Table 3.3: Average number of unique patterns detected with fixed/decreasing frequency thresholds.

	Height 2	Height 3	Height 4	Height 5
XSRP	3.6/3.6	1.4/33.3	0.4/88.2	0/416.3
Donut	17.9/28.3	4.7/12.3	1.3/5.6	0.4/3.9
RTP	3.5/4.0	4.0/12.3	8.9/33.8	53.3/291.2
MP6	5.3/11.8	6.0/32.0	6.9/147.6	3.4/1370.0

to 0 on the other hand is computationally expensive since it nullifies the *a priori* property on which PGA relies.

A possible solution to this problem is to decrease continuously the frequency threshold such that at the end PGA considers all patterns that have appeared at least once. We propose the following formula to calculate ε_f^t at generation t:

$$\varepsilon_f^t \equiv \frac{\varepsilon_F - \varepsilon_f}{G} \cdot t + \varepsilon_f, \tag{3.4}$$

where ε_f and ε_F denote the initial and final frequency thresholds and G is the number of generations. The method is only effective if the GP run does not have a considerable number of late generations with very similar individuals.

We compared the original PGA using fixed frequency threshold with a modified PGA featuring dynamic frequency threshold. We looked at the average number of unique patterns of different heights in both cases. Data was gathered during 50 individual GP runs. These were stopped after 10 generations for XSRP, 20 generations for Donut, 30 generations for the RTP and 20 generations for the MP6 problem. The results are summarized in Table 3.3. It can be seen that apart from patterns of height 2 for XSRP all other cases present improvements.

Because it is computationally expensive, for the time being the usage of dynamic frequency threshold is optional.

3.3 PostPGA

PostPGA consists of a series of filters to reduce the multi-set of patterns $\{\mathbb{P}_1, \mathbb{P}_2, \dots\}$ generated by PGA. First it gathers all the patterns generated by PGA into a set $\mathbb{P} = \bigcup_t \mathbb{P}_t$ and assigns a saliency value to each element of $\mathbb{P}$. Secondly, based on the saliency values it deletes certain elements of $\mathbb{P}$.

3.3.1 Measuring saliency

Measuring the saliency of a pattern is not unequivocal. There are several aspects of a pattern which can be taken into account and the different authors consider these with different weights. The most frequent definitions (explicitely stated or implied from the context) of a *good/salient* pattern are the following: (a) a subtree that solves a subproblem i.e. a subroutine [53, 54, 56, 57], (b) a high fitness pattern [60], (c) a high frequency pattern [59, 60] (d) a subtree that solves the whole problem to a certain degree and (e) a pattern whose replacement is detrimental [63, 86].

Our reservations about the first four definitions are the following: definition (a) is problematic because automatic identification of subproblems is generally not straightforward, using (b) and (c) might be misleading because of hitchhiker patterns and finally (d) might favor bigger patterns over the smaller ones. Though (e) has its own peculiarities, throughout this work we will adopt this definition.

The rationale is the following. Deleting a salient pattern P from an individual should have destructive effects and should cause significant drop in the individual's fitness. On the other hand, the deletion of an inert pattern should be less destructive, the drop of fitness should not be that dramatic (the fitness might stay the same, or might even increase).

Though a pattern can not be deleted, each subtree it matches can be replaced with a constant expression. Originally we used an ad-hoc value 1 for this purpose. However, as anonymous reviewers have pointed out, a context dependent neutral value would be more adequate.

Context dependent neutral values

Though a universal context dependent value does not exist, whenever it was possible we used the neutral value for the given operation as the replacement value. Otherwise we fell back to 1 or $\mathbf{1} \equiv (1, 1, \dots, 1)$, depending on the problem and the context. Table 3.4 summarizes the used neutral values in different scenarios.

Note that the functions listed in the second column refer to the parent node of the pattern which is to be substituted and not to the root node of the pattern. For example if the pattern P is $(x + 1)$ and the container elite is $(x * (x + 1))$ then the neutral value

used is 1. However if the container elite is $(x + (x + 1))$ then the neutral value used is 0.

Table 3.4: Neutral values used in the different test problems.

Problem(s)	Function(s)	Neutral value(s)
TCP, RTP	all	1
XSRP, Donut	$+, -$	0
	$/, *$	1
ELRA	$+, -$	0 or **0**
	$./, *, .*$	1 or **1**
	$\log(), \exp()$	1
MP6	and, not, xor	1
	or	0

Saliency assignment

The destructive effect of replacing a pattern P can be quantified by the following formula:

$$\Delta_P \equiv \frac{\langle f(E_i - P) \rangle - \langle f(E_i) \rangle}{|\langle f(E_i) \rangle|}, \tag{3.5}$$

where $\langle f(E_i) \rangle$ is the average fitness of individuals containing P, and $\langle f(E_i - P) \rangle$ is the average fitness of individuals containing P after P has been replaced by a context dependent neutral constant.

Instead of using the whole population to measure the destructive effects, for efficiency reasons we use only the set of elites $\mathbb{E}$ gathered from all generations. This has multiple benefits, since (a) $\mathbb{E}$ contains each pattern to be tested, (b) each element of $\mathbb{E}$ has good fitness so that the potential destructive effects are visible and (c) $\mathbb{E}$ has relatively few elements.

This saliency formula was proposed in [82].[4]However, it turned out that it defines a kind of relative error thus it may not differentiate between patterns of different importance. Consider the following, not necessarily pathological example. Let E_1, E_2 be two elites, such that $f(E_1) = 1$ and $f(E_2) = 100$. Let X and Y be two patterns from E_1 and E_2 respectively such that $\Delta_X = \frac{f(E_1 - X) - f(E_1)}{|f(E_1)|} = -1$ and $\Delta_Y = \frac{f(E_2 - Y) - f(E_2)}{|f(E_2)|} = -1$. Though common sense tells that X is less important than Y both are associated the same saliency value.

An alternative for the saliency measure that solves this problem is the following:

$$\overline{\Delta}_P \equiv h(P) \left\langle \frac{f(E_i - P) - f(E_i)}{\mu(E_i, P)} \right\rangle, \tag{3.6}$$

---

[4]After [82] had been accepted we found out that a similar measure was proposed by Majeed in [63].

where $h(P)$ is the height of P and $\mu(E_i, P)$ is the number of appearances of P in elite E_i.

As $\overline{\Delta}_P$ is an average of scaled absolute errors, it does a better job in differentiating between patterns of different importance than the previous measure. We included scaling terms $h(P)$ and $\mu(E_i, P)$ to prevent overrating smaller patterns.

To understand why is this necessary consider the RTP problem. Without the scaling terms the only possible terminal node x would be marked as the most important pattern. This is hardly what the end user would expect.

Context dependency

The saliency of a pattern is dependent on the context in which appears. Considering again XSRP, let a pattern $A = P \cdot (x^4 + x^3 + x^2 + x + 1)$ where P is an arbitrary pattern evaluating to non-unity. The saliency measure defined with Eq. 3.6 will mark P non-salient, even though P might consist of fragments that in other contexts would be useful (e.g. x, $x + 1$, $x^4 + x$, etc.).

Therefore only those elites E_i are considered during the averaging in Eq. 3.6 in which the presence of P is benefical, i.e. $f(E_i - P) < f(E_i)$.

3.3.2 Filtering redundant, overgeneral patterns

In the following we will refer to patterns without joker characters as *simple* patterns, while to ones containing joker characters as *general* patterns. We say that a pattern P_1 *generalizes* another pattern P_2 iff

1. P_1 and P_2 have the same skeleton,

2. for each node n_i^1 from P_1 and corresponding node n_i^2 form P_2, n_i^1 is more general than n_i^2 or $n_i^2 = n_i^1$ and

3. there is at least one node in P_1, n_i^1 that is more general than the corresponding node n_i^2 from P_2.

If joker characters are allowed (i.e. $\omega < 1$) then a relevant fraction of the patterns generated by PGA may be redundant in the following sense. If P_g is a general pattern and P_s is a simple pattern from $\mathbb{P}$, such that P_g generalizes no simple pattern apart from P_s, then P_g does not hold any extra valuable information and can be deleted.

It might be tempting to check this unique generalization of P_g only within $\mathbb{P}$, but there are cases when P_g generalizes more simple patterns among which only one is above the minimum frequency threshold ε_f. Deleting the P_g in this case would be a mistake. Consider the case depicted in Figure 3.9. P_g generalizes two simple patterns P_s and
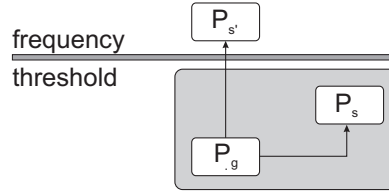
Figure 3.9: Redundant, overgeneral patterns where P_g should not be filtered out.
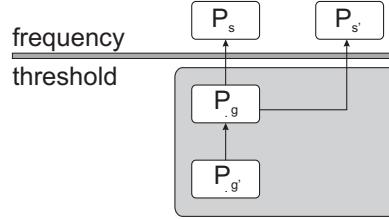

Figure 3.10: Redundant, overgeneral patterns where $P_{g'}$ should be filtered out.

$P_{s'}$, but only P_s is above the minimum frequency threshold. In this case P_g should not be filtered out.

Because all patterns below ε_f are lost during the tree mining phase, in most of the cases it is not possible to tell for sure whether P_g generalizes or not simple patterns outside $\mathbb{P}$. Fortunately the saliency values are good indicators in this case, and it is very likely that if P_g and P_s have the same saliency then P_g generalizes no simple pattern apart from P_s.

There is a second case which has to be considered. Let $\mathbb{S}$ be a set of simple patterns below ε_f, the minimum frequency threshold. Let $\mathbb{G} \subseteq \mathbb{P}$ be the set of general patterns above ε_f which generalize all the patterns from $\mathbb{S}$ and generalize no other simple patterns. Then any pattern $P_{g'} \in \mathbb{G}$ that generalizes a pattern $P_g \in \mathbb{G}$ can be filtered out. An example is shown on Figure 3.10. P_g is the simplest general pattern (in terms that contains the fewest joker characters) that generalizes P_s and $P_{s'}$. In this case $P_{g'}$ can be filtered out.

The steps of FilterOvergeneralPatterns, which performs the previously described filtering are given in Algorithm 3.5. The algorithm is explained through the following example.

Example. Suppose that we have 5 patterns $P_0 = f(y)$, $P_1 = f(x)$, $P_2 = f(@)$, $P_3 = \#(x)$ and $P_4 = \#(@)$. Among these P_0 is below the frequency threshold ε_f (and has been discarded by PGA), the rest of the patterns are above ε_f. Furthermore suppose that these patterns have been associated the following saliency values: $\overline{\Delta}_{P_1} = \overline{\Delta}_1, \overline{\Delta}_{P_2} = \overline{\Delta}_2, \overline{\Delta}_{P_3} = \overline{\Delta}_1, \overline{\Delta}_{P_4} = \overline{\Delta}_2$.

Let us simulate Algorithm 3.5 to find which of P_2, P_3, P_4 should be deleted. The first loop (steps $2-6$) updates the originally zero array occ to $[0,0,1,0]$. This means that P_3 generalizes a simple pattern above ε_f.

The second loop (steps 8 – 12) gathers the indices of patterns to be deleted. The condition inside the loop corresponds to the two overgenerality cases discussed above. At the end of the second loop, D will have two elements: 3 and 4. P_3 is overgeneral because it generalizes only P_1. P_4 falls in the second case, it is overgeneral because it does not bring any new information to P_2.

Consequently, the third loop (steps 13 – 15) deletes the marked patterns, in our case P_3 and P_4.

Algorithm 3.5 FilterOvergeneralPatterns($\mathbb{P}$) returns $\mathbb{P}$

1:   $occ = [0, 0, \dots, 0]$
2:   **for all** $i, j = 1 \dots |\mathbb{P}|, i \neq j$ **do**
3:     **if** $\mathbb{P}_i$ is simple **and** $\mathbb{P}_j$ generalizes $\mathbb{P}_i$ **and** $\overline{\Delta}_{\mathbb{P}_j} = \overline{\Delta}_{\mathbb{P}_i}$ **then**
4:       $occ[j] = occ[j] + 1$
5:     **end if**
6:   **end for**
7:   $\mathbf{D} = \oslash$
8:   **for all** $i, j = 1 \dots |\mathbb{P}|, i \neq j$ **do**
9:     **if** $occ[j] = 1$
      or $(occ[i] = 0$ **and** $\mathbb{P}_j$ generalizes $\mathbb{P}_i$ **and** $\overline{\Delta}_{\mathbb{P}_j} = \overline{\Delta}_{\mathbb{P}_i})$ **then**
10:       $\mathbf{D} = \mathbf{D} \cup \{j\}$
11:     **end if**
12:   **end for**
13:   **for all** $j \in \mathbf{D}$ **do**
14:     delete $\mathbb{P}_j$
15:   **end for**
16:   **return** $\mathbb{P}$

3.3.3   Filtering out invalidators

Some patterns, called invalidators [87] survive only because they suppress inviable code. Consider the example shown in Figure 3.11. The tree is a perfect solution for XSRP. The coloured part evaluates to 1 but contains an inert subtree $(D * (x - x))$, where D is an arbitrary expression. When evaluated, $(x - x)$ turns out to be salient because it suppresses the possible detrimental effects the inviable D might have.

Though invalidators have a considerable role in GP, their indirect manifestation suggest that their inclusion in the final list presented to the user should be optional.

Filtering out invalidators is problem dependent. In the case of XSRP, Donut and ELRA it consists of simplifying subtrees that evaluate to division or multiplication by zero or infinity. In the MP6 problem invalidators are in expressions equivalent to $(x \text{ and } 0)$ or $(x \text{ or } 1)$. Note that constructional problems like RTP are immune to this issue.

Currently PostPGA filters out only the simple invalidators, i.e. those not containing any joker characters. The simplification algorithm used is described in the Appendix.
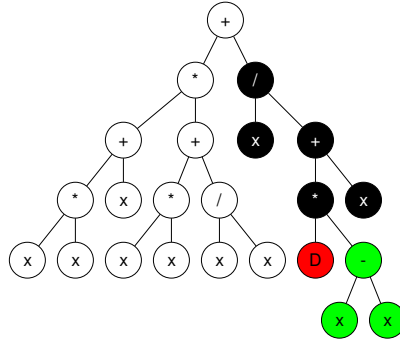
Figure 3.11: Invalidators. $(x - x)$ suppresses the possible detrimental effects of D thus it is detected as an important pattern.

3.3.4 Outliers

There is an inherent human expectation, especially for problems with known solutions, of what the building blocks should look like. For example, for XSRP there is a subtle expectation that the important patterns should contain the five successive powers of x. However, this is not necessarily the case and counterintuitive patterns may emerge. For example $1/(x - 1)$ is an important pattern because it plays a crucial role in the expression $(x^5 - 1)/(x - 1)$, which in its turn is a perfect solution. Practice has shown that these patterns often exhibit extremely good saliency values that lie outside the overall saliency distribution. Since there is no universally accepted way of detecting outliers, we drop all values falling more than 1.5 times the interquartile range below the first quartile as suggested in [88].

3.3.5 Modeling saliency classes

The patterns identified by [E]PGA along with the assigned saliency values are valid only for GP runs which had been examined. Patterns behave differently in independent GP runs and it is hard if not impossible to infer the behaviour of a pattern in a new run. Nevertheless, if there are highly salient patterns that occur in multiple runs this suggests that those patterns are candidates to be included as modules/subroutines in future runs. In the following we propose a method to assess the saliency profile of patterns belonging to the same height class originating from different independent runs.

To be able to (roughly) compare the saliencies from the different runs, at the end of PostPGA the saliency values are normalized to ensure that they are in the $[-1, 0]$ interval using the following formula

$$\overline{\Delta}_{P_i} = \frac{\overline{\Delta}_{P_i}}{\max_k |\overline{\Delta}_{P_k}|}. \tag{3.7}$$

54

Suppose that we have the saliency values of patterns of the same height S_h from multiple runs. S_h is a multiset [89], each element being from $[-1, 0]$. We wish to model this data by assuming that S_h has been obtained by sampling a random variable x. Without the loss of generality we can suppose that the probability density function of x, $p(x)$ is given by the linear combination of M component densities:

$$p(x) = \sum_{j=1}^{M} p(x|j)P(j), \tag{3.8}$$

where $p(x|j)$ is the jth component density function and $P(j)$ are the mixing parameters. In probabilistic terms $p(x|j)$ is the jth class-conditional density and $P(j)$ is the prior probability that the data point has been generated from component j of the mixture [90]).

The mixing parameters satisfy

$$\sum_{j=1}^{M} P(j) = 1. \tag{3.9}$$

Furthermore each component density is normalized, i.e.

$$\int p(x|j)dx = 1. \tag{3.10}$$

For the sake of simplicity we have chosen these component densities to be 1D Gaussians, i.e.

$$p(x|j) = \frac{1}{\sqrt{2\pi\sigma_j^2}} \exp\left\{ \frac{(x - \mu_j)^2}{2\sigma_j^2} \right\}. \tag{3.11}$$

This is not a limitation since any function can be approximated to an arbitrary precision with a linear combination of Gaussians [90]. The task of modeling reduces to finding the optimal (a) number of component densities M_{opt}, (b) mixing parameters P (c) cluster centers μ_j and (d) variances σ_j^2.

Let us introduce the notation $\Theta \equiv (\mu, \sigma^2, P)$, where Θ is a function of M, i.e. $\Theta = \Theta(M)$.

Calculating $\Theta(M)$ is relatively straightforward using the expectation maximization algorithm (EM) as described in [90]. Finding the optimal number of clusters however is less straightforward, especially with overlapping clusters. (For an overview of model selection see [91]). We have chosen the Bayesian information criterion (BIC) [92] because it is theoretically well founded, easy to implement and — as our results in Chapter 4 indicate — it produces very appealing results.

BIC is defined as

$$\text{BIC}(M, \Theta(M), S_h)) = -2\ln(\text{L}(\Theta(M))) + M\ln(|S_h|), \tag{3.12}$$

where L is the likelihood function. L is calculated as

$$L(\Theta(M)) = \prod_{n=1}^{|S_h|} p(x_n) \qquad (3.13)$$

$$= \prod_{n=1}^{|S_h|} \sum_{j=1}^{M} p(x_n|j)P(j). \qquad (3.14)$$

The optimal number of clusters is reached when BIC is minimized.

The minimization of BIC is formalized in Algorithm 3.6. SaliencyModeling() takes as input the data to be modelled and returns the optimal number of Gaussians together with the parameters of these Gaussians. It considers a series of possible values for M, $M \in \{M_{min} = 1, \ldots, M_{max} = \ln(|S_h|)\}$ and for each M it calculates $\Theta(M)$ using the expectation maximization algorithm. Then it calculates the Bayesian information criterion for model $\Theta(M)$ and stores it in b_M (step 3). After exiting the loop it chooses M_{opt} at which b is minimal and returns the corresponding $\Theta(M_{opt})$ and M_{opt}.

Algorithm 3.6 SaliencyModeling(S_h) returns $\Theta(M)$, M

1: **for all** $M \in \{M_{min}, \ldots, M_{max}\}$ **do**
2:    $\Theta(M) = EM(S_h)$
3:    $b_M = \text{BIC}(M, \Theta(M), S_h))$
4: **end for**
5: $M_{opt} = \underset{M \in \{M_{min}, \ldots, M_{max}\}}{\text{argmin}} b_M$
6: **return** $\Theta(M_{opt}), M_{opt}$

Chapter 4

Experiments

In the following we show how [E]PGA works on the test problems described in Chapter 2. We consider two scenarios for each problem: a close look on a single, randomly chosen run and a more general view on multiple runs.

If otherwise not specified, in all of the problems we are interested in patterns of height 2–5 and order ratios in $[0.8, 1.0]$.

4.1 Tackett's constructional problem

For Tackett's original problem, considering an arbitrary run, the GP quickly finds the perfect solution (Figure 4.1). The identified patterns for this particular run are shown in Table 4.1. The second column shows the generation when the frequency threshold is reached. Observe that although the perfect solution is found in generation 10 it takes 18 another generation until $((z + x) * (x - y))$ proliferates enough so that its frequency in the population is larger than $\varepsilon_f = 0.3$. The third column shows the saliency values. Since TCP is an easy constructional problem, these values are in good correlation with the reward function $\rho(\cdot)$ from Equation 2.1.

Table 4.1: TCP. Identified patterns with the generation in which the pattern reached the frequency threshold and the assigned saliency values

Pattern	t	$\overline{\Delta}_P$
(z+x)	22	-0.326
(x-y)	8	-0.337
(@*(x-y))	13	-0.565
((z+x)*(x-y))	28	-1.000

Because of the extreme similarity of the further runs to the one presented above (both in terms of patterns and saliency values), these have been omitted.
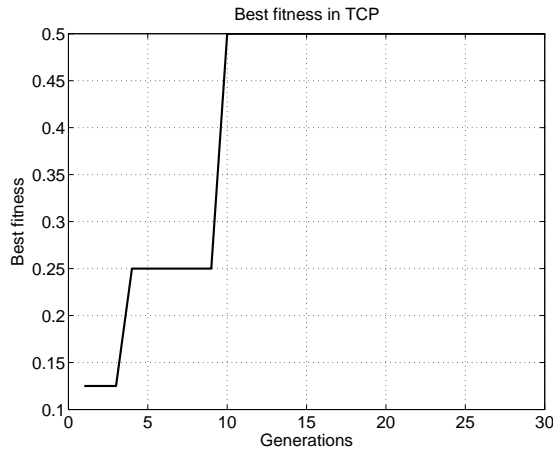
Figure 4.1: TCP. Best fitness evolution

4.2   The Royal Tree problem

Though harder than TCP, this constructional problem is solved in reasonable time by GP in the arbitrary run we are going to analyze in the following. The solution, a perfect tree of level–c tree is found around the generation 30. Figure 4.2 depicts the evolution of best fitness in this run.
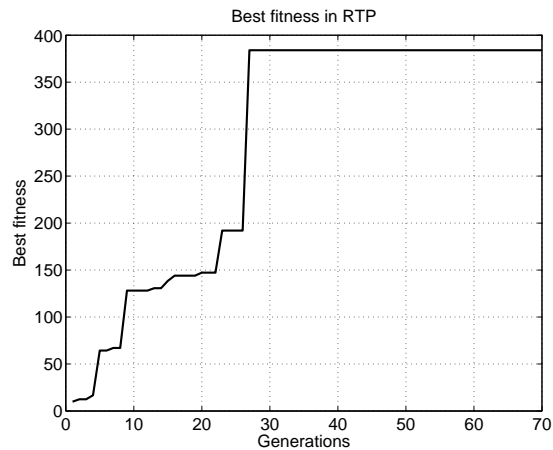
Figure 4.2: RTP. Best fitness evolution of an arbitrary run providing a perfect level–c tree

We ran [E]PGA with maximum pattern height in $\{2, \ldots, 5\}$, minimum order ratio set to 0.8 and minimum frequency threshold set to 0.3. The identified patterns with the first occurrence and the associated saliency values are given in Table 4.2. This captures nicely the emergence of bigger and bigger "building blocks". A lag between finding the best solution and the proliferation of bigger building blocks can be seen here as well.

Table 4.2: RTP. Identified patterns.

Pattern	t	$\overline{\Delta}_P$
a(x)	0	-0.178
(xAx)	0	-0.015
(xBx)	0	-0.000
b(x)	0	-0.000
c(x)	0	0.000
(xAa(x))	11	-0.359
(xA(x##x))	11	-0.093
#((xAa(x)))	44	-1.000

4.2.1 Multiple runs

We ran [E]PGA 100 times with the same settings. It turned out that on average 35.7% of the patterns identified by PGA were overgeneral, meaning that they fell into one of the two cases described in Section 3.3.2. After filtering the overgeneral patterns the total number of patterns left was 931, among which 41 were unique. The number of patterns occurring at least twice was 26. The top ten most frequent patterns are given in Table 4.3.

Table 4.3: RTP. Most frequent patterns considering 100 independent runs

Pattern	Frequency
a(a(x))	36
#((xAa(x)))	40
(xCx)	75
(xBx)	77
b((xAa(x)))	78
c(x)	87
b(x)	92
(xAx)	99
a(x)	100
(xAa(x))	100

It is interesting to see that some of these patterns, e.g. $b(x)$, $c(x)$, (xCx) occur with high frequencies in independent runs, although, as can be seen from their saliency profiles in Figure 4.3, their saliency is marginal. On the other hand certain patterns like $\#((xAa(x)))$ or $b((xAa(x)))$ "behave" well consistently. Somewhat between the two are smaller patterns like $a(x)$ or $(xAa(x))$ that build up larger, good patterns.

There is a tendency that can be observed if we consider all the patterns (not just the most frequent ten) from all the individual runs and draw the saliency profile of the patterns broken into different height classes (Figure 4.4). There is a shift in the saliency profiles from the right side of the figure to the left side as the heights of the patterns grow. This means that small patterns can proliferate with high probability in the GP
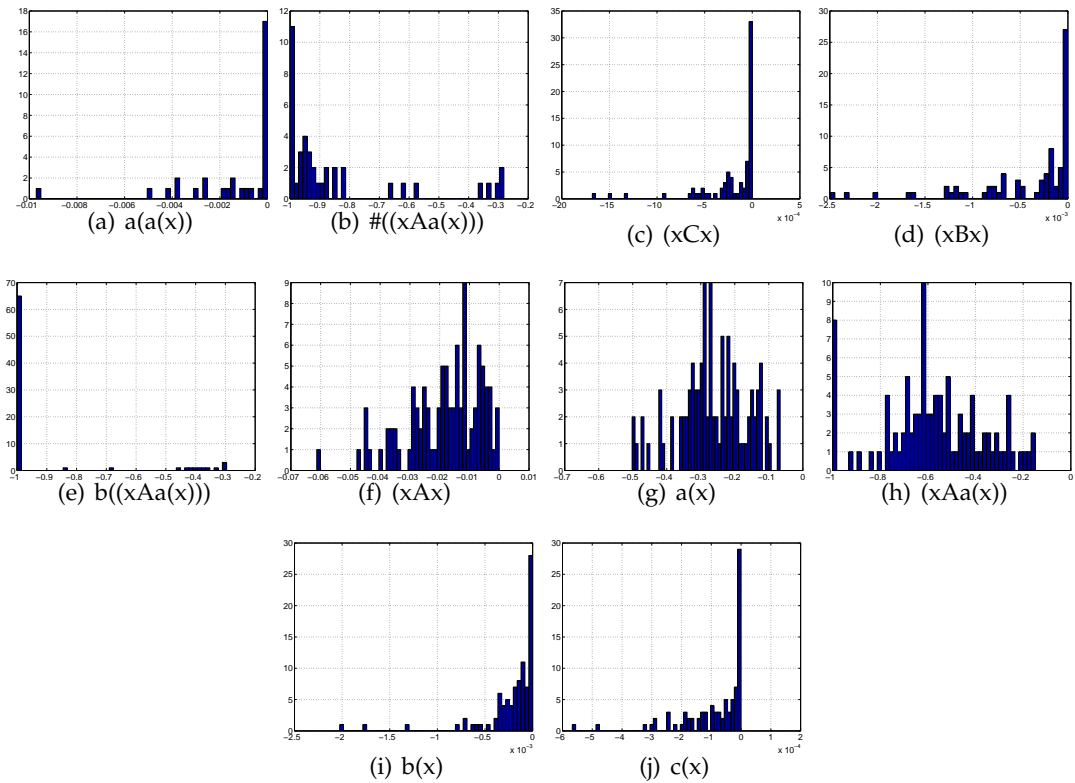
Figure 4.3: RTP. Saliency histograms for the most frequent patterns from 100 individual runs

individuals even if their saliencies are poor. However as the pattern grows larger this probability gets lower and lower.

The Gaussian mixture approximations with the number of saliency classes identified by Algorithm 3.6 are shown in Figure 4.5. Observe that the BIC+EM pair does a decent job in modeling the saliency histograms.
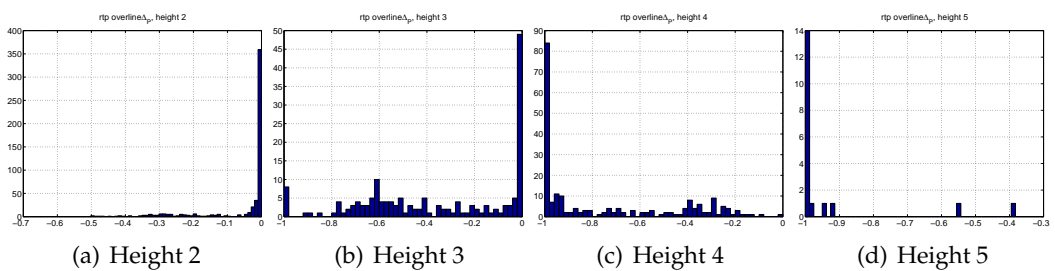
Figure 4.4: RTP. Saliency histograms of patterns of height 2–5. Notice the shift in the saliency profiles from the right side of the figure to the left side as the height of the patterns grow.

| (a) Height 2 | (b) Height 3 | (c) Height 4 | (d) Height 5 |
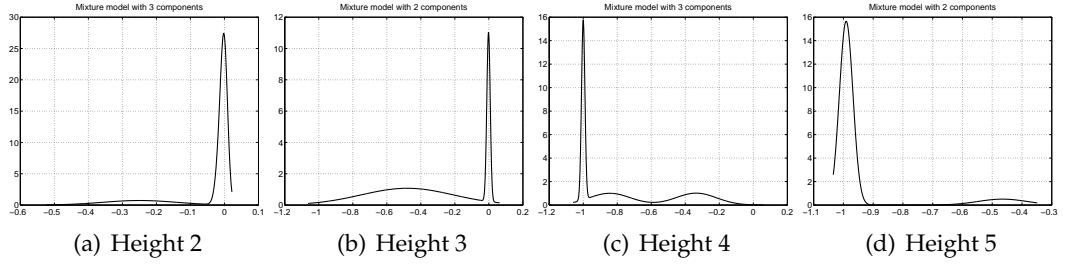
Figure 4.5: RTP. Modeling the saliency distribution of patterns of height 2–5.

4.3 Classic symbolic regression

Let us have a look at an arbitrary run providing a perfect solution for the XSRP problem. The evolution of the best fitness is depicted in Figure 4.6.
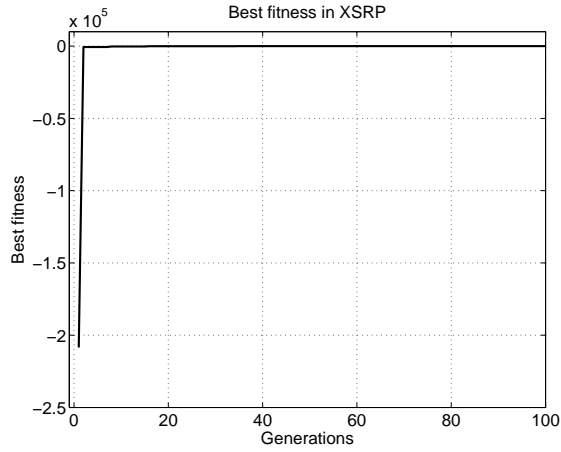
Figure 4.6: XSRP. Evolution of best fitness

The patterns of height between 2–5, identified by [E]PGA for $\varepsilon_f = 0.5$ and $\omega = 0.8$ are shown in Table 4.4. A lag between finding the best solution and the proliferation of bigger building blocks can be seen.

Table 4.4: XSRP. Identified patterns with the generation in which the pattern reached the frequency threshold and the assigned saliency values.

Pattern	t	$\overline{\Delta}_P$
(x.*x)	1	-0.327
(x./x)	1	-0.148
(x+x)	1	0.000
(x+(x.*x))	55	-1.000
(x+(x##x))	43	-0.800
(x##(x.*x))	23	-0.783
((x.*x)##x)	18	-0.020

61

4.3.1 Multiple runs

We ran [E]PGA 100 times with the same settings. On average 21.9% of the patterns mined by PGA were filtered out by FilterOvergeneralPatterns. The total number of patterns left was 707, among which 29 were unique. The number of patterns occurring in at least two runs was 20. The ten most frequent patterns are given n Table 4.5.

Table 4.5: XSRP. Most frequent patterns considering 100 independent runs

Pattern	Frequency
(x.*(x##x))	22
((x.*x)+x)	24
(x+(x.*x))	24
(x+(x##x))	52
((x##x)+x)	54
(x##(x.*x))	74
((x.*x)##x)	75
(x+x)	95
(x./x)	97
(x.*x)	99

The saliency profiles of these patterns show a blurry picture (see Figure 4.7). From the patterns of height 2 only $(x.*x)$ has a considerable saliency, the relevance of both $(x+x)$ and $(x./x)$ being marginal. The profiles of patterns of height 3 clearly shows that the these patterns behave significantly different in the different runs. However, the prevalence of these patterns and the fact that their saliency reached -1 at least 8 times (apart from $(x.*(x##x))$) suggest that they are truly important patterns.

The saliency histograms of the different height classes are shown in Figure 4.8. Interestingly no pattern of height 5 passed the frequency threshold. A similar shift to the one in RTP in the saliency profiles from the low saliency regions to the high ones can be observed as the patterns' heights grow from 2 to 4. The approximation of the saliency classes is shown in Figure 4.9.

(a) (x.*(x##x))　　(b) ((x.*x)+x)　　(c) (x+(x.*x))　　(d) (x+(x##x))

(e) ((x##x)+x)　　(f) (x##(x.*x))　　(g) ((x.*x)##x)　　(h) (x.*x)

(i) (x+x)　　(j) (x./x)
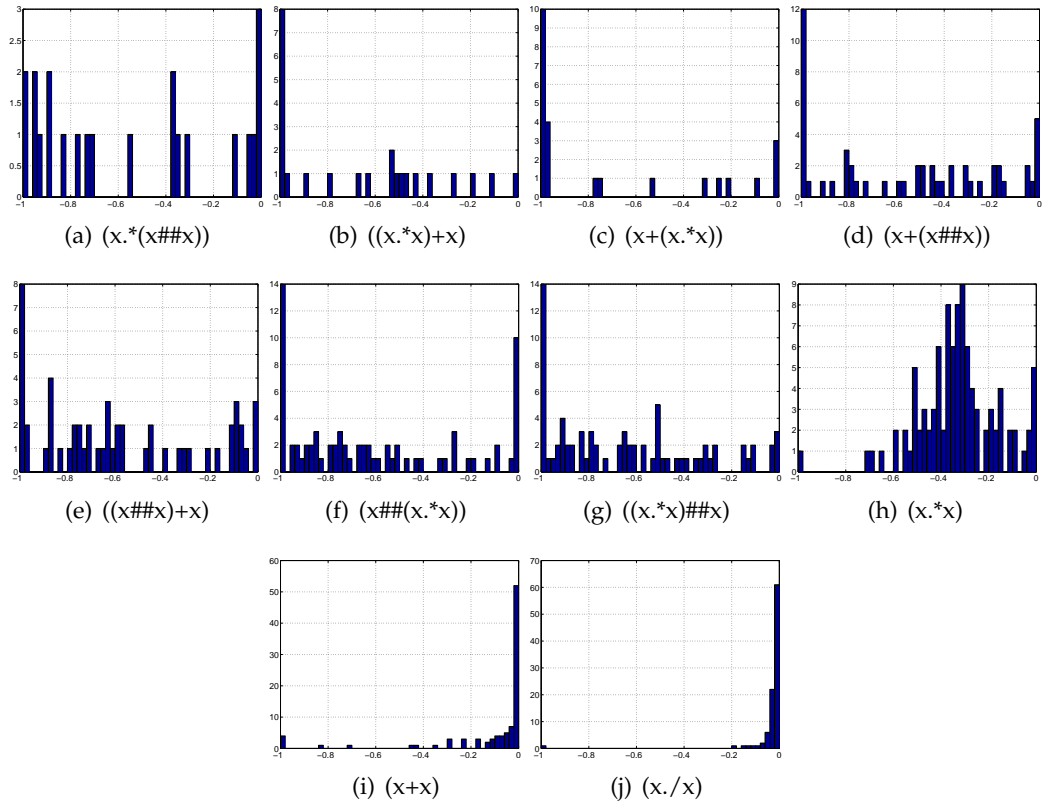
Figure 4.7: XSRP. Saliency histograms for the most frequent patterns from 100 individual runs.

(a) Height 2　　(b) Height 3　　(c) Height 4
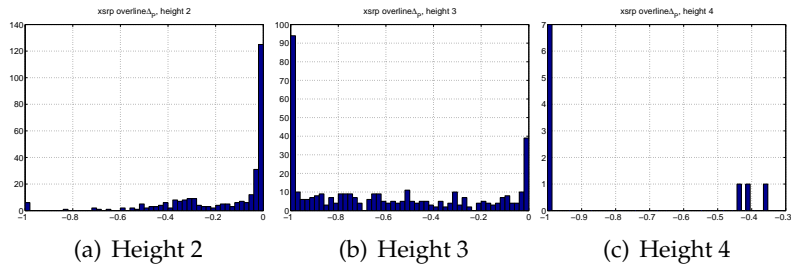
Figure 4.8: XSRP. Saliency histograms of patterns of height 2–4

(a) Height 2　　(b) Height 3　　(c) Height 4
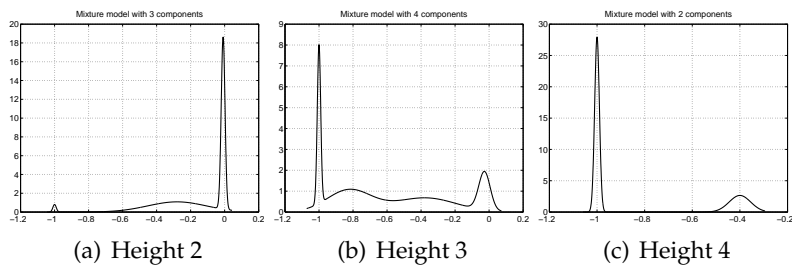
Figure 4.9: XSRP. Modeling the saliency distribution of patterns of height 2–4

63

4.4 Donut problem

The Donut problem is the first of the test problems that features the ephemeral random constant. This has two consequences. On the one hand the GP runs faster because it does not have to invent the real constants it may need. On the other hand the resulting patterns contain many ERCs which are hard to understand.

Let us consider an arbitrary, 80 generation run. The evolution of the best fitness values is shown in Figure 4.10.
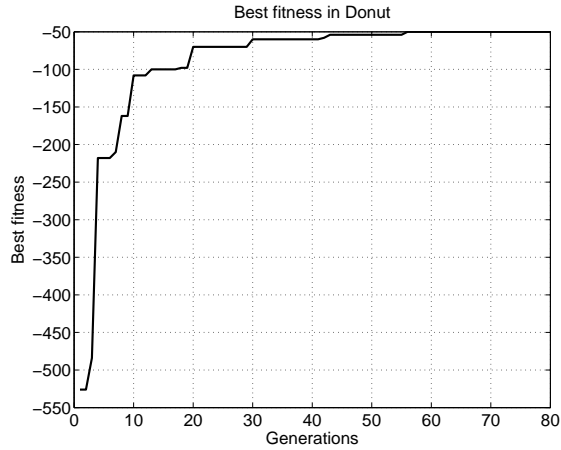
Figure 4.10: Donut. Evolution of best fitness

The best-of-run individual classified correctly 95% of the examples was $(((((x1.*3.000)-(2.000+x0))./(1.000+x1))+((((x1-x0)./1.000)+((2.000+4.000)-(x2+1.000)))./2.000))+((4.000+x0)./(x1-1.000)))$, which simplifies to

$$\frac{x_1 - x_0 - x_2 + 11}{2} + \frac{2x_0 - x_1 + 9}{x_1^2 - 1}.$$

The patterns identified by [E]PGA (height between 2–5, $\varepsilon_f = 0.5$ and $\omega = 0.8$) are listed in Table 4.6. Interestingly, though the number of generations was 80, no patterns of height larger than 3 managed to reach the frequency threshold. Furthermore it is hard to understand why certain patterns are marked better than others by PostPGA.

Table 4.6: Donut. Patterns identified in an arbitrary run.

Pattern	t	$\overline{\Delta}_P$
(1.000+x1)	19	-1.000000
(x1-1.000)	20	-0.453957
(2.000+x0)	23	-0.124491
(x1.*x1)	61	-0.007313

Table 4.7: Donut. Most frequent patterns considering 100 independent runs

Pattern	Frequency
(3.000+x0)	6
(x0+4.000)	6
(x1+2.000)	6
(x1+x1)	6
(x1./x1)	7
(x2./x2)	7
(x1-1.000)	9
(x1+1.000)	10
(x1.*x1)	11
(x2.*x2)	20

4.4.1 Multiple runs

Considering 100 independent runs the percentage of overgeneral patterns was 28.03%. The total number of remaining patterns was 589 among which 402 were unique. The number of patterns occurring at least twice was 82. The ten most frequent patterns among runs are shown in Table 4.7. The reason why only patterns of height 3 are among the most frequent patterns and the causes of the relatively low frequencies of these patterns needs further investigations. Unfortunately the low frequency of the patterns renders the saliency histograms of these patterns (see Figure 4.11) unusable.

The saliency histograms of patterns of height 2–5 from 100 independent runs are shown in Figure 4.12. Observe again that the overwhelming majority of the patterns are of height 2 or 3. These patterns tend to accumulate mainly on the right side of the histograms, suggesting neutral behaviour. Little can be said about the larger patterns because of their low occurrence.

The Gaussian mixture models of the patterns of height 2–5 are shown in Figure 4.13. While for patterns of height 2 and 3 the distributions resemble closely the corresponding histograms, for patterns of height 4 and 5 this does not happen because the low number of samples forces BIC to use simple models (see Equation 3.12).

(a) (3.000+x0) (b) (x0+4.000) (c) (x1+2.000) (d) (x1+x1)

(e) (x1./x1) (f) (x2./x2) (g) (x1-1.000) (h) (x1+1.000)

(i) (x1.*x1) (j) (x2.*x2)
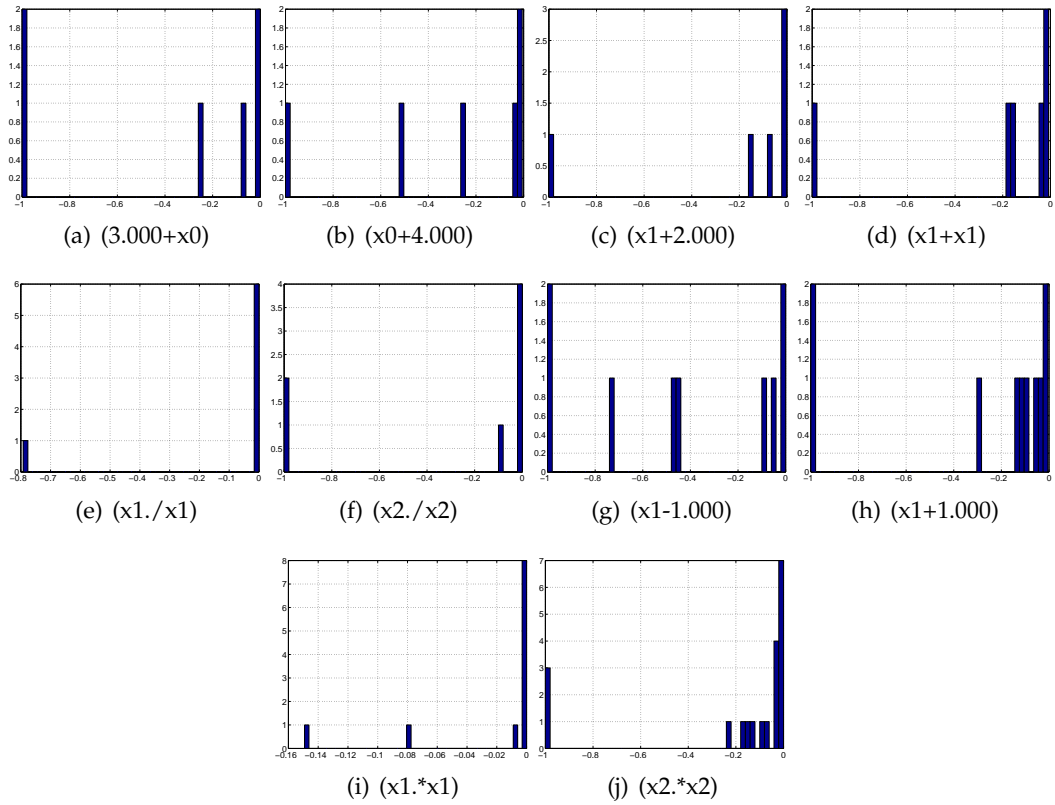
Figure 4.11: Donut. Saliency histograms for the most frequent patterns from 100 individual runs.

(a) Height 2 (b) Height 3 (c) Height 4 (d) Height 5
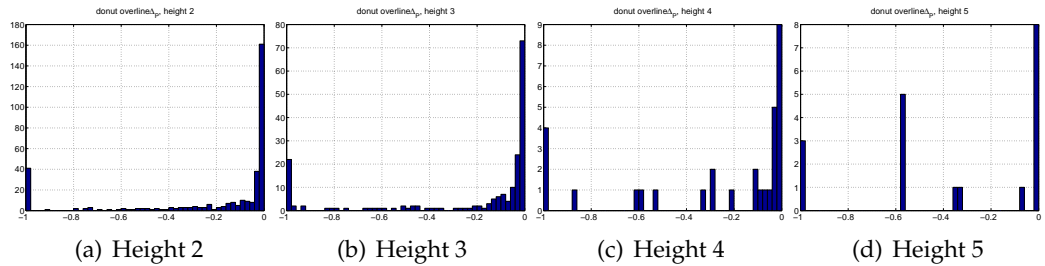
Figure 4.12: Donut. Saliency histograms of patterns of height 2–5

(a) Height 2 (b) Height 3 (c) Height 4 (d) Height 5
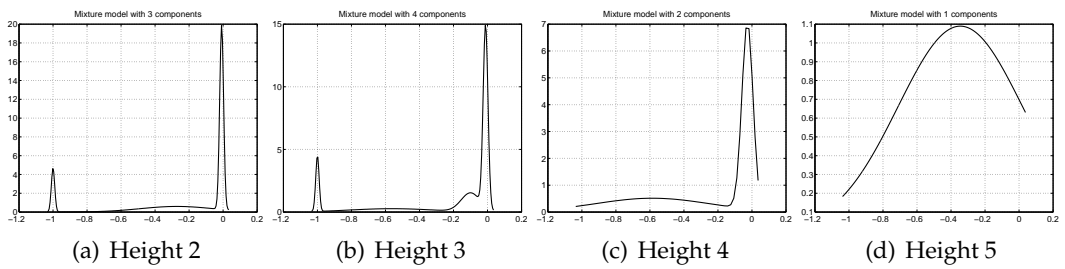
Figure 4.13: Donut. Modeling the saliency distribution of patterns of height 2–5

4.5 Evolution of learning rules

As in the previous cases let us consider first an arbitrary run. The evolution of the best fitness values is depicted in Figure 4.14.
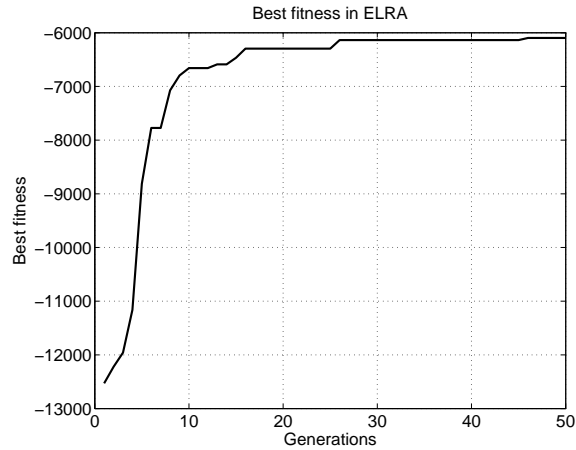
Figure 4.14: ELRA. Evolution of best fitness

The best of run individual was $((((2.000 * (x * d)) - (x * y)) * 9.000) - (((y + y) + (x + (x * y))) + ((x + (x * d)) * y)))$, which can be rewritten as $11x(d - y) + 7dx - 2y - x - dxy$. One might ask wether the "extra" term $-2x - x - dxy$, which is not part of the linear combination of the delta rule and Hebb's rule has any purpose. It turns out that it makes a difference. Without it the best individual's fitness is -7656.2, while with it is -6097.33. The corresponding generalization errors are shown in Figure 4.15.
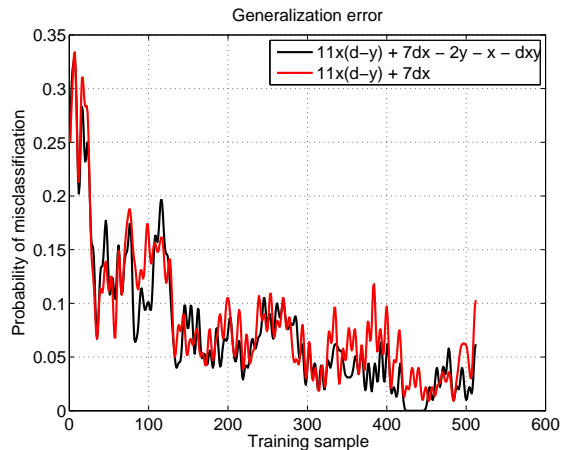
Figure 4.15: ELRA. Generalization error corresponding to $11x(d - y) + 7dx$ and $11x(d - y) + 7dx - 2y - x - dxy$.

The patterns identified by [E]PGA in this run are shown in Table 4.8. It shows how quickly the Hebbian rule penetrates the population, i.e. in just 5 generations $(d * x)$ is present in half of the individuals. The delta rule is discovered later.

Table 4.8: ELRA. Identified patterns for an arbitrary run

Pattern	t	$\overline{\Delta}_P$
(x*d)	10	-0.378001
(d*x)	5	-0.244585
(x*y)	9	-0.035331
(@*(x*d))	12	-0.862367
(2.000*(x*d))	13	-0.860992
((@*(x*d))-(x*y))	14	-1.000000
((2.000*(x*d))-(x*y))	16	-0.989183

4.5.1 Multiple runs

Considering 257 independent runs the ratio of overgeneral patterns was 0.717. The number of remaining patterns was 5982 out of which 3740 were unique. The number of patterns occurring at least in two runs was 482. The top 20 most frequent patterns are listed in Table 4.9. Observe that the majority of these are parts from either the delta rule or Hebb's rule. Furthermore, as with the previous experiments the most frequent patterns among the individual runs are the shortest patterns, strongly dominated by $(x * d)$ and $(d * x)$. These patterns occur in more than 97% of the runs the best individuals of the very first generation.

Table 4.9: ELRA. Most frequent patterns considering 153 independent runs

Pattern	Frequency
(y*(@*x))	16
(8.000*(d*x))	17
(8.000*(x*d))	17
((@*(d*x))-x)	17
((x*y)*@)	17
((x*d)*9.000)	18
(@*(y*x))	18
(d-x)	19
((x*d)*y)	20
((x*@)*y)	20
((d*x)*9.000)	21
(d-y)	26
(@*(x*d))	60
((d*x)*@)	68
(@*(d*x))	77
((x*d)*@)	90
(y*x)	99
(x*y)	101
(d*x)	181
(x*d)	191

The corresponding saliency profiles are shown in Figure 4.16.

(a) (y*(@*x))   (b) (8.000*(d*x))   (c) (8.000*(x*d))   (d) ((@*(d*x))-x)

(e) ((x*y)*@)   (f) ((x*d)*9.000)   (g) (@*(y*x))   (h) (d-x)

(i) ((x*d)*y)   (j) ((x*@)*y)   (k) ((d*x)*9.000)   (l) (d-y)

(m) (@*(x*d))   (n) ((d*x)*@)   (o) (@*(d*x))   (p) ((x*d)*@)

(q) (y*x)   (r) (x*y)   (s) (d*x)   (t) (x*d)
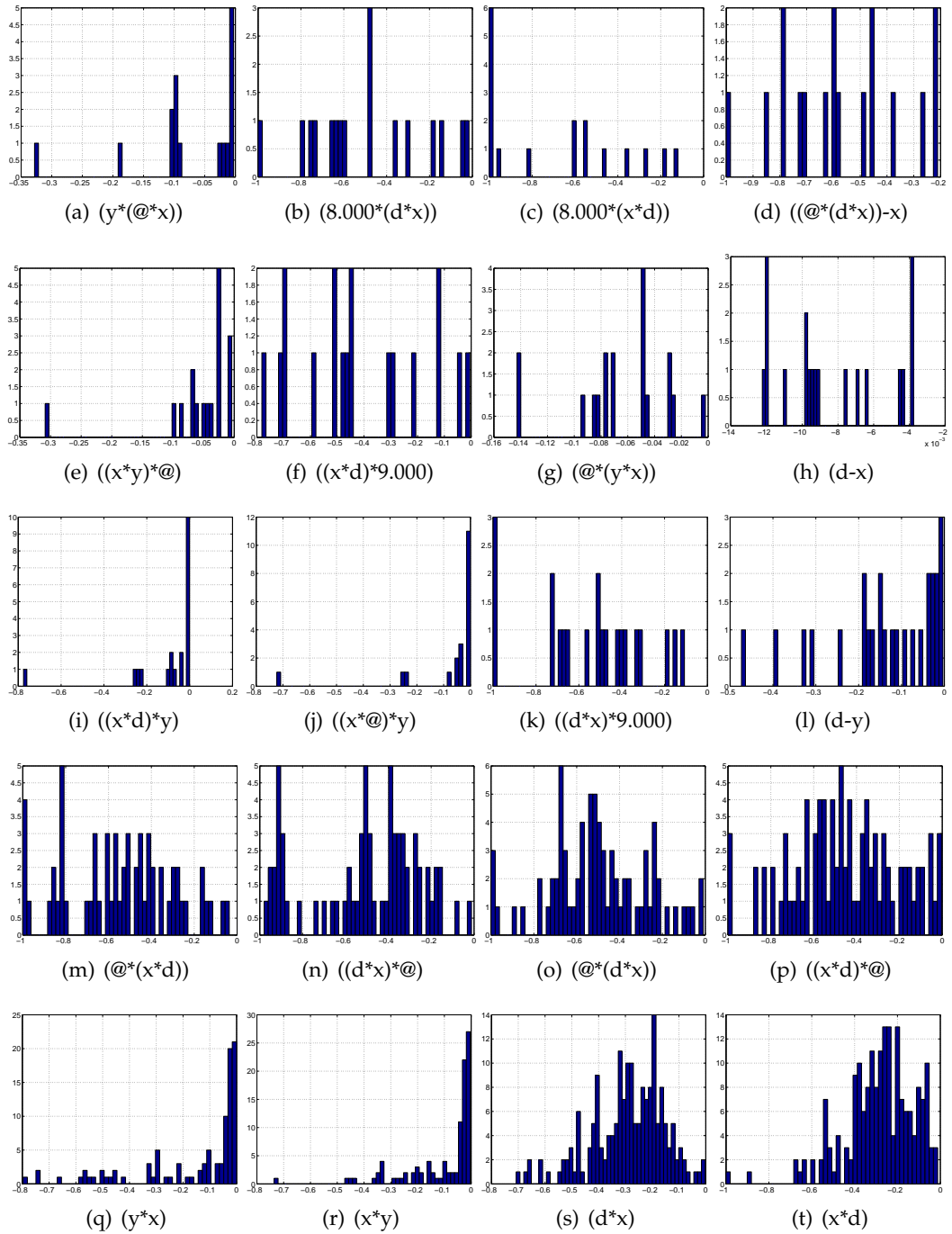
Figure 4.16: ELRA. Saliency histograms for the most frequent patterns from 257 individual runs

69

As with the other problems, all the frequent patterns among runs are short patterns of height 2 and 3. Another similarity is the variable behaviour of the majority of these patterns. However, it is worthwile to note that equivalent patterns show similar behavior among runs (e.g. $(x * d)$ with $(d * x)$; $(x * y)$ with $(y * x)$, etc.). The saliency histogram of patterns of height 2–5 and their approximation with Gaussian mixtures are shown in Figures 4.17 and 4.18. Observe a similar shift from the right side of the graphs to the left side as the height of the patterns grow. Interestingly, as opposed to the previous problems a great proportion of patterns of height 5 are given 0 saliency.

We hoped that another simple learning rule, Oja's rule [93] of form

$$\Delta w = \eta y(x - yw),$$

(where η is the learning rate) would appear as well in at least one of the runs. For unknown reasons this has not happened. Moreover, none of the best individuals from these runs used effectively the weight vector; w appeared in only expressions which were either inviable code or invalidators (e.g. $(w - w)$, $((y - y) * (w - x))$ etc.).

| (a) Height 2 | (b) Height 3 | (c) Height 4 | (d) Height 5 |
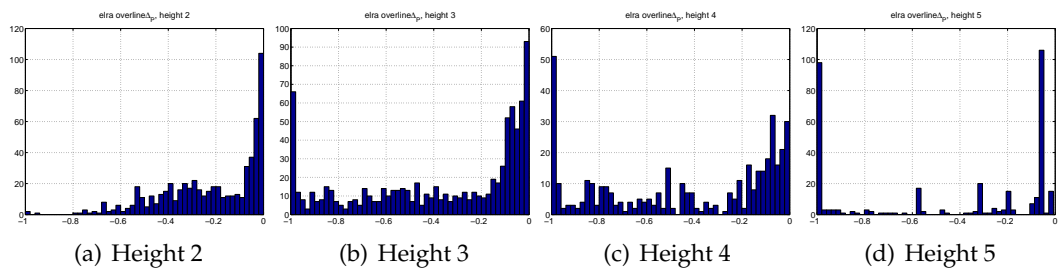
Figure 4.17: ELRA. Saliency histograms of patterns of height 2–5

| (a) Height 2 | (b) Height 3 | (c) Height 4 | (d) Height 5 |
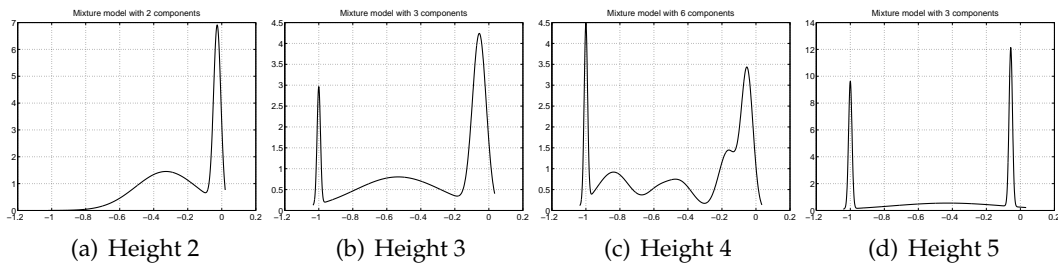
Figure 4.18: ELRA. Modeling the saliency distribution of patterns of height 2–5

4.6 6-multiplexer

The last problem we analyze with [E]PGA is the 6-multiplexer problem. The evolution of the best individuals' fitness is shown in Figure 4.19. The perfect score 1 is reached in the last generation by the following individual

```
((A1&(((A0&D3)|D2)&(D3|((D2^A0)&(!(D1)|D1)))))|(A1^((A0&D1)|
((((D0&(!(A1)&A0))^D0)|(A1&(D0&A0)))|A1)))).
```
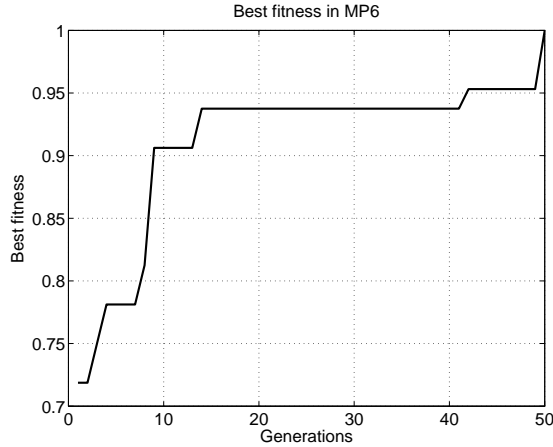

Figure 4.19: MP6. Evolution of best fitness

The patterns found by [E]PGA for this run are listed in Table 4.10.

Table 4.10: MP6. Identified patterns for an arbitrary run

Pattern	t	$\overline{\Delta}_P$
(A0^D1)	9	-0.319888
(A0&D3)	7	-0.234700
(A1|D0)	15	-0.070588
(D0&A0)	31	-0.018100
(A1&D1)	17	0.000000
!(A0)	15	0.000000
((D0&A0)^D0)	35	-0.582353
((A0&D3)|D2)	16	-0.535497
((D0&@)^D0)	34	-0.529412
((A0&@)|D2)	16	-0.518382
((@&A0)^D0)	34	-0.497326
(D0|!(A0))	18	-0.306723
((D0|!(A0))&A0)	18	-0.044818
(((D0|!(A0))&A0)^D0)	19	-1.000000

The perfect individual features less than 1/3 of of these patterns ((A0&D3), (D0&A0), ((A0&D3)|D2) and ((A0&@)|D2)) which is probably caused by the fact that it emerged in the last generation. This suggests that using decreasing frequency threshold would have been more adequate.

Table 4.11: MP6. Most frequent patterns considering 100 independent runs

Pattern	Frequency
(D3&A1)	29
(A0|D2)	30
(A1|D1)	31
(A1&D3)	32
(D1&A0)	33
(D3&A0)	33
(A1&D2)	35
(A0&D1)	41
!(A1)	59
!(A0)	70

4.6.1 Multiple runs

Considering 100 independent runs the ratio of overgeneral patterns was 0.698. The number of remaining patterns was 3185 out of which 2012 were unique. The number of patterns occurring at least in two runs was 250. The top ten most frequent patterns are listed in Table 4.11. No pattern of height greater than 2 managed to enter the top ten most frequent pattern. Actually the first pattern of height 3, (D3|!(A0)) was at position 40 with 7 appearances from the 100 runs. The saliency profile of these patterns are depicted in Figure 4.20. Most of them are concentrated on the right side of the histograms, suggesting hitchhiker-like behaviour ($\mathbf{E}[\overline{\Delta}_P] = 0.162$, $\text{var}(\overline{\Delta}_P) = 6.278e - 04$).

Similar shifting tendency observed with the previous problems can be seen if the identified patterns are broken into height classes. The saliency histograms and the corresponding Gaussian models are shown in Figures 4.21 and 4.22.

Summary

The performed experiments have shown the forte and the weak points of [E]PGA.

For all problems [E]PGA identified correctly the salient patterns and whith the exception of the Donut problem it recorded nicely the emergence of bigger and bigger building blocks. It reaffirmed that patterns behave differently depending on the context. This is was especially clear for small patterns frequent across multiple runs.

The experiments also showed that if ERCs are enabled then almost perfect solutions can be built of very primitive and small patterns, which are hard to interpret for humans. This was very accentuated in the case of the Donut problem. The difficulties we have encountered in Donut and MP6 highlighted the main weakness of [E]PGA, namely that it mines only the syntactic space and does not consider semantically equivalent patterns.

(a) (D3&A1) (b) (A0|D2) (c) (A1|D1) (d) (A1&D3)

(e) (D1&A0) (f) (D3&A0) (g) (A1&D2) (h) (A0&D1)

(i) !(A1) (j) !(A0)
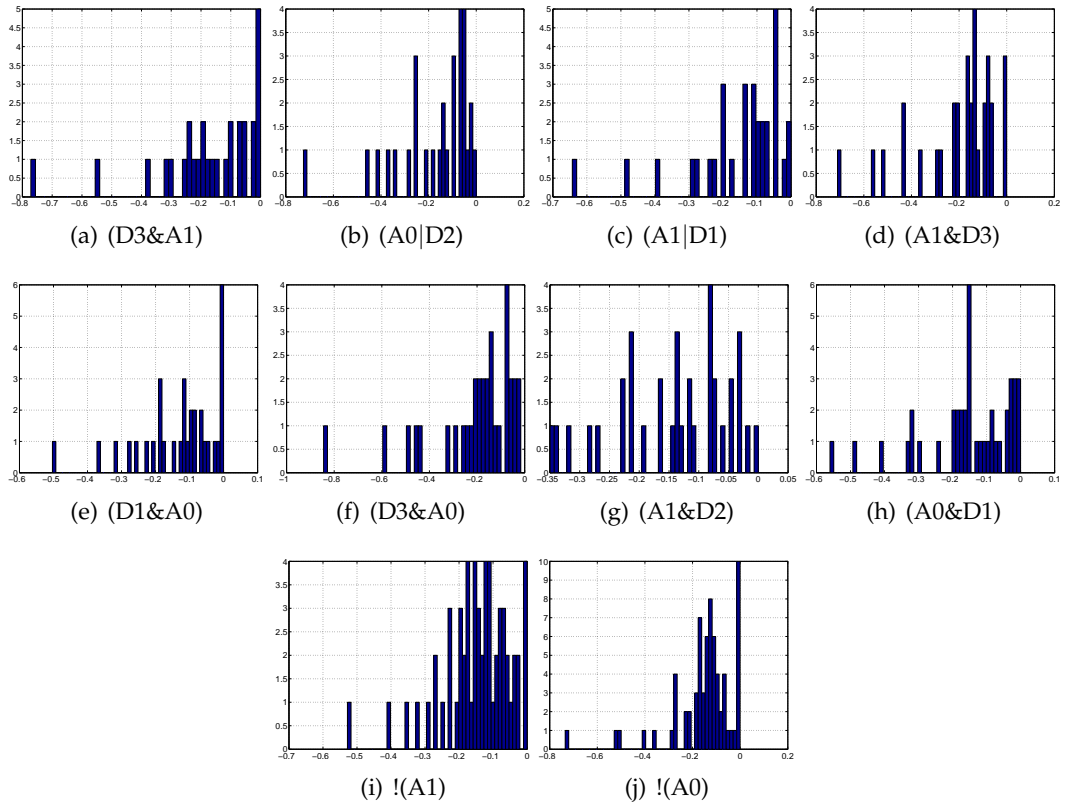
Figure 4.20: MP6. Saliency histograms for the most frequent patterns from 100 individual runs

(a) Height 2 (b) Height 3 (c) Height 4 (d) Height 5
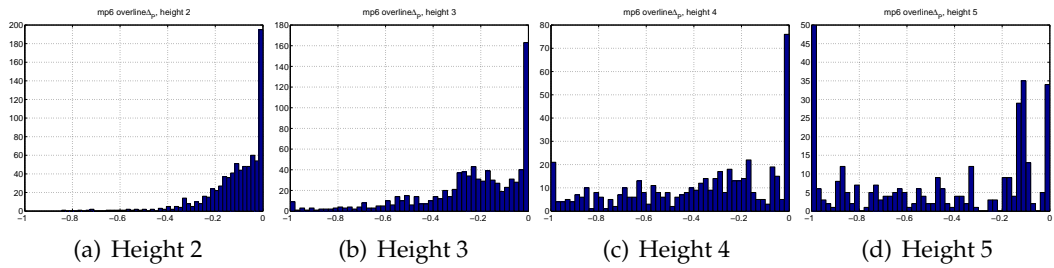
Figure 4.21: MP6. Saliency histograms of patterns of height 2–5

(a) Height 2 (b) Height 3 (c) Height 4 (d) Height 5
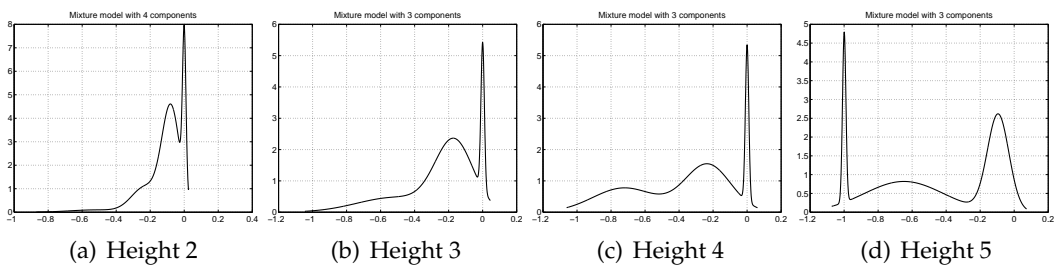
Figure 4.22: MP6. Modeling the saliency distribution of patterns of height 2–5

Chapter 5

Discussion

There are several observations we wish to make related to the results of the experiments presented in the previous sections.

[E]PGA mines the syntactic space and it uses properties from the semantic space, thus it is a hybrid method to identify salient patterns. During the tree mining phase [E]PGA does not take into consideration patterns that are functionally equivalent, nor does it when calculating the saliencies. Furthermore, as it was pointed out in [64] the active code segments may be diffused and not grouped together thus the identified patterns might fail to solve a clearly delimited part of the problem. At the moment we think that these are the reasons why the essence of the evolutionary runs could not be caught in the Donut and MP6 problems.

Mining in the semantic space and identifying functionally equivalent patterns is not straightforward. GP theory focuses on the propagation of syntactic properties and to our best knowledge at the time being there are no considerable works (if any) in this area.

The patterns identified by [E]PGA for a given run have meaning only for that particular run. Any extrapolation for other runs is risky since a given pattern can have a totally different behaviour in a different run.

If we consider all the patterns of the same height across the different runs and problems there is a tendency that can be observed. Patterns of height 2 tend to concentrate on the right side of the saliency histograms. As the pattern height grows larger the mean of the distribution moves to the left side. This means that small patterns can proliferate with high probability in the GP individuals even if their saliencies are poor. However as the pattern grows larger this probability gets lower and lower.

Although [E]PGA does identify important active patterns, it does not try to *explain* why those patterns are important, i.e. it answers the *what* but not the *why* question.

Chapter 6

Conclusions and future work

This thesis presented [E]PGA, a set of methods used to extract, sort and filter patterns from offline tree-based GP individuals. The goals of producing a set of patterns which are representative for the evolutionary run, are human-friendly and whose number are within reasonable limits were attained reasonably well for the considered problems: (a) Tackett's constructional problem, (b) a modifed Royal Tree problem, (c) Donut, a classification problem featuring ephemeral random constants, (d) Koza's classical symbolic regression problem, (e) evolution of learning rules and (f) 6-multiplexer. The performed experiments have shown the forte and the weak points of [E]PGA.

For Tackett's constructional problem [E]PGA identified correctly all the patterns which were defined in the reward function. Moreover, the saliency values were in correlation with the reward values.

In the case of the Royal Tree problem [E]PGA performed well again and recorded nicely the emergence of bigger and bigger "building blocks". Running [E]PGA on multiple instances of this problem also revealed the presence of hitchhiker patterns which occur with high frequency across the different runs but whose saliency is marginal.

Classic symbolic regression was the first problem to reaffirm the fact that patterns behave differently, depending on the context. This was captured nicely during multiple runs for patterns of height 2 and 3.

Donut, the classification problem featuring ERC showed that almost perfect solutions can be built of very primitive and small patterns. Unfortunately this also means that the patterns themselves are quite meaningless for humans.

The evolution of learning rules was the computationally most intensive experiment. [E]LRA identified correctly the emerging fragments of the Hebbian and the delta rule.

The multiplexer problem featured the most variables, six. We speculate that because of this and of the high degree of simmetricity of the problem the identified patterns are extremely hard to interpret.

The difficulties we have encountered in the Donut and multiplexer problems highlighted the main weakness of [E]PGA, namely that it mines only the syntactic space and does not consider semantically equivalent patterns.

The usage of Gaussian mixture models allowed us to model the saliency profiles of patterns of a given height across multiple runs. The performed experiments suggest that in most of the cases for a given pattern height the number of saliency classes is two or three.

The test problems we considered have in common an important property: the genome can be written as a single parse tree, which, apart from the constructional problems encodes a mathematical function. Extending / trying [E]PGA for other representations (e.g. nodes featuring arbitrary functions, or switching to graph or linear encoding) is a possible direction in which this project could move.

As stated before, the main limitation of [E]PGA is that it relies mainly on the syntactic space when it tries to identify the salient patterns. An alternative approach which focused more intensively on the expressed properties of the patterns would hopefully solve the issues mentioned above and would give at least partial answer to the *why* question as well.

Bibliography

[1] J. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection (Complex Adaptive Systems).* The MIT Press, Dec. 1992.

[2] "Human-competitive results produced by genetic and evolutionary computation." `http://www.genetic-programming.org/hc2005/main.html`.

[3] D. L. Whitley, V. S. Gordon, and K. E. Mathias, "Lamarckian evolution, the Baldwin effect and function optimization," in *Parallel Problem Solving from Nature – PPSN III* (Y. Davidor, H.-P. Schwefel, and R. Männer, eds.), (Berlin), pp. 6–15, Springer, 1994.

[4] S. Yoshi, K. Suzuki, and Y. Kakazu, "Lamarckian GA with genetic supervision," in *Evolutionary Computation, 1995., IEEE International Conference on*, vol. 1, p. 367, 1995.

[5] B. A. Julstrom, "Comparing Darwinian, Baldwinian, and Lamarckian search in a genetic algorithm for the 4-cycle problem," in *Late Breaking Papers at the 1999 Genetic and Evolutionary Computation Conference* (S. Brave and A. S. Wu, eds.), (Orlando, Florida, USA), pp. 134–138, 1999.

[6] B. J. Ross, "A lamarckian evolution strategy for genetic algorithms," in *Practical Handbook of Genetic Algorithms: Complex Coding Systems* (L. D. Chambers, ed.), vol. 3, pp. 1–16, Boca Raton, Florida: CRC Press, 1999.

[7] K.-H. Liang, X. Yao, and C. Newton, "Lamarckian evolution in global optimization," in *Industrial Electronics Society, 2000. IECON 2000. 26th Annual Conjference of the IEEE*, vol. 4, pp. 2975–2980 vol.4, 2000.

[8] R. M. Friedberg, "A learning machine: I," *IBM Journal of Research and Development*, vol. 2, pp. 2–13, 1958.

[9] J. Holland, *Adaptation in Natural and Artificial Systems.* The MIT Press, 1975.

[10] W. Banzhaf, P. Nordin, R. Keller, and F. Francone, *Genetic Programming : An Introduction : On the Automatic Evolution of Computer Programs and Its Applications.* Morgan Kaufmann, Nov. 1997.

[11] I. Rechenberg, *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Phd thesis, Technical University of Berlin, 1971.

[12] H. Schwefel, *Evolutionsstrategie und numerische Optimierung*. Phd thesis, Technical University of Berlin, 1975.

[13] J. Holland, "Escaping brittleness: The possibilities of general-purpose learning algorithms applied to parallel rule-based systems," *Machine learning: An artificial intelligence approach*, vol. 2, 1986.

[14] L. Fogel, "Autonomous automata," *Industrial Research*, vol. 4, pp. 14–19, 1962.

[15] L. J. Fogel, P. J. Angeline, and D. B. Fogel, "An evolutionary programming approach to self -adaptation on finite state machines," in *Evolutionary Programming*, pp. 355–365, 1995.

[16] E. Zitzler, K. Deb, and L. Thiele, "Comparison of multiobjective evolutionary algorithms: Empirical results," *Evolutionary Computation*, vol. 8, pp. 173–195, 2000.

[17] D. E. Goldberg and J. Richardson, "Genetic algorithms with sharing for multimodal function optimization," in *Proceedings of the Second International Conference on Genetic Algorithms on Genetic algorithms and their application*, (Cambridge, Massachusetts, United States), pp. 41–49, Lawrence Erlbaum Associates, Inc., 1987.

[18] K. A. D. Jong, *An analysis of the behavior of a class of genetic adaptive systems.* PhD thesis, University of Michigan, 1975.

[19] W. D. Hillis, "Co-evolving parasites improve simulated evolution as an optimization procedure," *Phys. D*, vol. 42, pp. 228–234, 1990.

[20] M. A. Potter and K. D. Jong, "A cooperative coevolutionary approach to function optimization," in *Parallel Problem Solving from Nature III*, pp. 249–257, Springer-Verlag, 1994.

[21] E. Cantu-Paz, "A summary of research on parallel genetic algorithms," tech. rep., University of Illinois at Urbana-Champaign, 1995.

[22] J. Gayon, *Darwinism's struggle for survival : heredity and the hypothesis of natural selection / Jean Gayon ; translated by Matthew Cobb.* Cambridge studies in philosophy and biology, Cambridge, U.K. ; New York :: Cambridge University Press, 1998.

[23] J. Woodward, "Modularity in genetic programming," in *Genetic Programming*, pp. 225–239, 2003.

[24] D. J. Montana, "Strongly typed genetic programming," *Evolutionary Computation*, vol. 3, pp. 199–230, 1995.

[25] M. Wineberg and F. Oppacher, "A representation scheme to perform program induction in a canonical genetic algorithm," in *Parallel Problem Solving from Nature III*, pp. 292–301, Springer-Verlag, 1994.

[26] W. Banzhaf, "Genotype-phenotype-mapping and neutral variation – a case study in genetic programming," in *Parallel Problem Solving from Nature III*, vol. 866, pp. 322–332, Springer-Verlag, 1994.

[27] P. Nordin, "A compiling genetic programming system that directly manipulates the machine code," in *Advances in Genetic Programming*, pp. 311–331, MIT Press, 1994.

[28] R. L. Crepeau, "Genetic evolution of machine language software," in *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications* (J. P. Rosca, ed.), (Tahoe City, California, USA), pp. 121–134, 1995.

[29] P. Nordin and W. Banzhaf, "Evolving Turing-complete programs for a register machine with self-modifying code," in *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)* (L. Eshelman, ed.), (Pittsburgh, PA, USA), pp. 318–325, Morgan Kaufmann, 1995.

[30] P. Nordin, W. Banzhaf, and F. D. Francone, "Efficient evolution of machine code for cisc architectures using instruction blocks and homologous crossover," in *Advances in genetic programming*, vol. 3, pp. 275–299, MIT Press, 1999.

[31] R. Poli, "Evolution of recursive transition networks for natural language recognition with parallel distributed genetic programming," Tech. Rep. CSRP-96-19, University of Birmingham UK, Dec. 1996. Presented at AISB-97 workshop on evolutionary computation.

[32] R. Poli, "Discovery of symbolic, neuro-symbolic and neural networks with parallel distributed genetic programming," in *Artificial Neural Nets and Genetic Algorithms: Proceedings of the International Conference, ICANNGA97* (G. D. Smith, N. C. Steele, and R. F. Albrecht, eds.), (University of East Anglia, Norwich, UK), Springer-Verlag, 1997.

[33] R. Poli, "Some steps towards a form of parallel distributed genetic programming," in *The 1st Online Workshop on Soft Computing (WSC1),* (Nagoya University, Japan), 1996.

[34] R. Poli, "Evolution of graph-like programs with parallel distributed genetic programming," in *Genetic Algorithms: Proceedings of the Seventh International Conference* (T. Back, ed.), (Michigan State University, East Lansing, MI, USA), pp. 346–353, Morgan Kaufmann, 1997.

[35] O. M. and R. C., "Grammatical evolution: A steady state approach," pp. 419–423, 1998.

[36] J. F. Miller and P. Thomson, "Cartesian genetic programming," vol. 1802, (Edinburgh), pp. 121–132, Springer-Verlag, 2000.

[37] D. Dumitrescu, *Evolutionary Computation*. CRC Press, 2000.

[38] L. Altenberg, "The schema theorem and price's theorem," *Foundations of Genetic Algorithms*, vol. 3, pp. 23–49, 1995.

[39] M. D. Vose, *The Simple Genetic Algorithm: Foundations and Theory*. The MIT Press, Aug. 1999.

[40] R. G. Price, "Selection and covariance," *Nature*, vol. 227, pp. 520–521, Aug. 1970.

[41] U.-M. O'Reilly, *An Analysis of Genetic Programmig*. Phd thesis, Carleton University, 1995.

[42] J. P. Rosca, "Analysis of complexity drift in genetic programming," in *Genetic Programming 1997: Proceedings of the Second Annual Conference* (J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo, eds.), (Stanford University, CA, USA), pp. 286–294, Morgan Kaufmann, 1997.

[43] L. Altenberg, "Emergent phenomena in genetic programming," in *Evolutionary Programming — Proceedings of the Third Annual Conference* (A. V. Sebald and L. J. Fogel, eds.), (San Diego, CA, USA), pp. 233–241, World Scientific Publishing, 1994.

[44] W. Langdon and R. Poli, *Foundations of Genetic Programming*. Springer, 2002.

[45] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., 1989.

[46] A. Wright, M. Vose, and J. Rowe, "Implicit parallelism," in *Genetic and Evolutionary Computation GECCO 2003*, p. 211, 2003.

[47] L. D. Whitley, "Fundamental principles of deception in genetic search," *Foundations Of Genetic Algorithms*, pp. 221—241, 1991.

[48] J. J. Grefenstette and J. E. Baker, "How genetic algorithms work: a critical look at implicit parallelism," in *Proceedings of the third international conference on Genetic algorithms*, (George Mason University, United States), pp. 20–27, Morgan Kaufmann Publishers Inc., 1989.

[49] U. O'Reilly and F. Oppacher, "The troubling aspects of a building block hypothesis for genetic programming," 1992.

[50] J. M. Daida, R. R. Bertram, J. A. Polito, and S. A. Stanhope, "Analysis of single-node (building) blocks in genetic programming," in *Advances in genetic programming: volume 3*, pp. 217–241, MIT Press, 1999.

[51] C. Ryan, H. Majeed, and A. Azad, "A competitive building block hypothesis," in *Genetic and Evolutionary Computation GECCO 2004*, pp. 654–665, 2004.

[52] K. Sastry, U. O'Reilly, K. Sastry, D. Hill, D. E. Goldberg, and D. E. Goldberg, "Building-Block supply in genetic programming," *In Proceedings Of The Genetic Programming Workshop On Theory & Practice (Chapter) 9*, p. pp., 2003.

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

[54] J. R. Koza, *Genetic Programming II: Automatic Discovery of Reusable Programs*. Cambridge Massachusetts: MIT Press, May 1994.

[55] O. Brock, "Evolving reusable subroutines for genetic programming," *In Artificial Life At Stanford*, pp. 11—19, 1994.

[56] P. J. Angeline, "Genetic programming and emergent intelligence," in *Advances in Genetic Programming* (K. E. Kinnear, Jr., ed.), ch. 4, pp. 75–98, MIT Press, 1994.

[57] J. P. Rosca, "Towards automatic discovery of building blocks in genetic programming," in *Working Notes for the AAAI Symposium on Genetic Programming*, pp. 78–85, AAAI, 1995.

[58] G. Li, K. H. Lee, and K. S. Leung, "Evolve schema directly using instruction matrix based genetic programming," in *Genetic Programming* (M. Keijzer, A. Tettamanzi, P. Collet, J. v. Hemert, and M. Tomassini, eds.), vol. 3447 of *Lecture Notes in Computer Science*, pp. 141–142, Springer Berlin / Heidelberg, 2005.

[59] Y. Kameya, J. Kumagai, and Y. Kurata, "Accelerating genetic programming by frequent subtree mining," in *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, (Atlanta, GA, USA), pp. 1203–1210, ACM, 2008.

[60] W. A. Tackett, "Mining the genetic program," *IEEE Expert*, vol. 10, pp. 28–38, June 1995.

[61] W. Smart, P. Andreae, and M. Zhang, "Empirical analysis of GP tree-fragments," in *Proceedings of the GECCO 2007*, vol. 4445 of *Lecture Notes in Computer Science*, (Valencia, Spain), pp. 55–67, Springer, 2007.

[62] W. Smart and M. Zhang, "Empirical analysis of schemata in genetic programming using maximal schemata and MSG," in *2008 IEEE World Congress on Computational Intelligence*, (Hong Kong), IEEE Press, 2008.

[63] H. Majeed, "A new approach to evaluate GP schema in context," in *GECCO '05: Proceedings of the 2005 Workshops on Genetic and Evolutionary Computation*, (New York, USA), ACM, 2005.

[64] W. B. Langdon and W. Banzhaf, "Repeated patterns in genetic programming," *Natural Computing*, vol. 7, no. 4, pp. 589–613, 2008.

[65] R. Poli and W. B. Langdon, "An experimental analysis of schema creation, propagation and disruption in genetic programming," 1997.

[66] J. P. Rosca and D. H. Ballard, "Rooted-tree schemata in genetic programming," pp. 243–271, 1999.

[67] R. McKay, J. Shin, T. H. Hoang, X. H. Nguyen, and N. Mori, "Using compression to understand the distribution of building blocks in genetic programming populations," *IEEE Congress on Evolutionary Computation*, pp. 2501–2508, 2007.

[68] S. Haykin, *Neural Networks: A Comprehensive Foundation*. New York: Macmillan, 1994.

[69] X. Yao, "Evolving artificial neural networks," *Proceedings of the IEEE*, vol. 87, pp. 1423–1447, 1999.

[70] D. J. Chalmers, "The evolution of learning: An experiment in genetic connectionism," in *Proceedings of the 1990 Connectionist Summer School* (D. S. Touretsky, J. L. Elman, T. J. Sejnowski, and G. E. Hinton, eds.), pp. 81–90, Morgan Kaufmann, 1990.

[71] J. Baxter, "The evolution of learning algorithms for artificial neural networks," *Complex Systems*, pp. 313–326, 1992.

[72] S. Bengio, Y. Bengio, and J. Cloutier, "Use of genetic programming for the search of a new learning rule for neutral networks," in *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, vol. 1, (Orlando, Florida, USA), pp. 324–327, IEEE Press, 1994.

[73] A. M. Radi and R. Poli, "Discovery of neural network learning rules using genetic programming," Tech. Rep. CSRP-97-21, University of Birmingham, School of Computer Science, Sept. 1997.

[74] A. Radi and R. Poli, "Genetic programming can discover fast and general learning rules for neural networks," (University of Wisconsin, Madison, Wisconsin, USA), pp. 314–323, Morgan Kaufmann, 1998.

[75] A. Radi and R. Poli, "Discovery of backpropagation learning rules using genetic programming," in *Proceedings of the 1998 IEEE World Congress on Computational Intelligence*, vol. 1, (Anchorage), pp. 371–375, IEEE Press, 1998.

[76] A. M. Radi and R. Poli, "Discovery of general learning rules for feedforward neural networks with step activation function using genetic programming," Tech. Rep. CSRP-99-1, University of Birmingham, School of Computer Science, Jan. 1999.

[77] A. Radi and R. Poli, "Discovering efficient learning rules for feedforward neural networks using genetic programming," Tech. Rep. CSM-360, Department of Computer Science, University of Essex, Colchester, UK, Jan. 2002.

[78] J. P. Neirotti and N. Caticha, "Dynamics of the evolution of learning algorithms by selection," *Physical Review E*, vol. 67, p. 041912, Apr. 2003.

[79] C. Adami, C. Ofria, and T. C. Collier, "Evolution of biological complexity," in *Proceedings of the National Academy of Sciences of the U.S.A.*, 2000.

[80] W. F. Punch, D. Zongker, and E. D. Goodman, "The royal tree problem, a benchmark for single and multiple population genetic programming," in *Advances in Genetic Programming 2*, ch. 15, pp. 299–316, Cambridge, MA, USA: MIT Press, 1996.

[81] R. Poli and W. B. Langdon, "A new schema theory for genetic programming with one-point crossover and point mutation," in *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pp. 278–285, Morgan Kaufmann, 1997.

[82] A. Joó, "Mining evolving learning algorithms," in *Proceedings of the 12th European Conference on Genetic Programming*, (Tuebingen), Springer, April 2009.

[83] N. Mori, R. McKay, X. H. Nguyen, and D. Essam, "Equivalent decision simplification: A new method for simplifying algebraic expressions in genetic programming," in *Proceedings of 11th Asia-Pacific Workshop on Intelligent and Evolutionary Systems*, 2007.

[84] Y. Chi, R. R. Muntz, S. Nijssen, and J. N. Kok, "Frequent subtree mining - an overview," *Fundam. Inf.*, vol. 66, no. 1-2, pp. 161–198, 2004.

[85] Y. Chi, R. R. Muntz, S. Nijssen, and J. N. Kok, "Frequent subtree mining - an overview," *Fundamenta Informaticae*, vol. 66, no. 1-2, pp. 161–198, 2004.

[86] A. Joó and J. P. Neirotti, "Towards identifying salient patterns in genetic programming individuals," in *Proceedings of GECCO 2009*, (Montreal, Canada), July 2009.

[87] S. Luke, "Code growth is not caused by introns," in *Late Breaking Papers at the 2000 Genetic and Evolutionary Computation Conference*, (Las Vegas, USA), Morgan Kaufmann, 2000.

[88] J. Renze, "Outlier." From MathWorld – A Wolfram Web Resource, created by Eric W. Weisstein. `http://mathworld.wolfram.com/Outlier.html`.

[89] E. W. Weisstein, "Multiset." From MathWorld – A Wolfram Web Resource, created by Eric W. Weisstein. `http://mathworld.wolfram.com/Multiset.html`.

[90] C. Bishop, *Pattern recognition and machine learning*. Springer, 2006.

[91] K. P. Burnham and D. R. Anderson, *Model selection and multimodel inference: a practical information-theoretic approach*. Springer, 2 ed., 2002.

[92] G. Schwarz, "Estimating the dimension of a model," *The annals of Statistics*, vol. 6, no. 2, pp. 461–464, 1978.

[93] E. Oja, "Simplified neuron model as a principal component analyzer," *Journal of Mathematical Biology*, vol. 15, pp. 267–273, Nov. 1982.

[94] P. Wong and M. Zhang, "Algebraic simplification of GP programs during evolution," in *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, (Seattle, USA), ACM, 2006.

[95] "GNU Multiple Precision Arithmetic Library." `http://gmplib.org/`.

[96] M. Zhang, Y. Zhang, and W. Smart, "Program simplification in genetic programming for object classification," in *Knowledge-Based Intelligent Information and Engineering Systems*, pp. 988–996, 2005.

[97] "STL." `http://www.sgi.com/tech/stl/`.

[98] "The open group base specifications issue 6, IEEE std 1003.1." `http://www.opengroup.org/onlinepubs/009695399/basedefs/` `/pthread.h.html`.

[99] "Graphviz, Graph Visualization Software." `http://www.graphviz.org/`.

[100] "BOOST, portable C++ source libraries." `http://www.boost.org/`.

Appendix A

The simplification algorithm

We used rule-based simplification applied in a bottom up fashion. At each level we iterated through the list of rules until no more simplifications were possible. If a function node had a constant child / children then it was evaluated and replaced with the corresponding value.

Due to the fact that the untyped GP was used some of the usual arithmetic simplification rules could not be applied and others had to be extended. Other issues arose because of the protectedness of some functions (e.g. `exp`). A snippet of the list of the primary rules is given below:

```
(T*0)-->zero
(T.*0[])-->zero
(W[]-0[])-->W[]
(T./(1./V))-->((V.*T).*one)
((U.*T)+(V.*T))-->((U+V).*T)
((a*T)+(a*V))-->(a.*(T+V))
```

The rules contain special variables that match different patterns in the trees to simplify. The meaning of these variables are summarized in Table A.1. No rule in the primary list contains its commutative counterparts (e.g. `(T*0)-->zero` is contained but `(0*T)-->zero` not). These extra rules are generated automatically with a simple backtracking-like algorithm.

Equal subtrees

Another issue that has to be addressed is to decide whether two expressions are equal or not. Consider for example `(exp((x+y))-exp((y+x)))`. The bottom-up algorithm can not handle this since it can not tell that `exp((x+y))` and `exp((y+x))` are equal.

Table A.1: Special variables in the simplification rules

Variable	Meaning
T,U,V	any subtree
W[]	any subtree that evaluates to a vector
z	any scalar variable
0[]	$\mathbf{0} = (0,0,\dots,0)$
1[]	$\mathbf{1} = (1,1,\dots,1)$
a,b	any scalar constant
zero	0 or $\mathbf{0}$, depending on the context
one	1 or $\mathbf{1}$, depending on the context

A solution for this problem was described by [94]. In their implementation a hash value is associated with every subtree. Two subtrees are said to be functionally equivalent if they are equal or they have the same hash value. The version described by [94] handles only arithmetic operators and scalars, but it can be extended for unary functions and vectors as well.

The hash of a subtree T, $h(T)$ is calculated as follows. If T is rooted by a unary expression of form $f(T')$ then the hash of T is $h(T) = h(f(h(T')))$. If T is rooted by a binary expression $f(T'_1, T'_2)$ then $h(T) = h(f(h(T'_1), h(T'_2)))$. Finally if T is a variable or constant then we calculate the hash value directly by an auxiliary function $h_p(T) = h(T)$.

The core of the hashing mechanism relies on h_p which maps every operand into $\mathbb{Z}_p^n$, where p is a relatively large prime number and n is the length of a vector. If applied to a variable, h_p returns (deterministically) an integer between 1 and p. For an integer vector $\boldsymbol{x}$, $h_p(\boldsymbol{x}) = (x_1 \mod p, x_2 \mod p, \dots, x_n \mod p)$. Finally for a floating point vector $\boldsymbol{x}$, $h_p(\boldsymbol{x}) = (h_p^f(x_1), h_p^f(x_2), \dots, h_p^f(x_n))$. The function h_p^f maps a floating point value to $\mathbb{Z}_p$ and is defined as $h_p^f(x) = (\lfloor x \cdot \varepsilon \rfloor \cdot \bar{\varepsilon}) \mod p$, where ε is a precision constant (of magnitude 10^3 in our case) and $\bar{\varepsilon}$ is its inverse in $\mathbb{Z}_p$.

Slightly problematic is the exponential function which quickly overflows. A remedy is to use an arbitrary precision floating point type instead the built-in float or double types (`mpf_t` from [95]).

Associativity

In order to simplify expressions like `((x-y)+(y-x))` we need associativity rules. However, their inclusion in the list of simplification rules is not straightforward. That is because associativity rules do not simplify directly an expression and infinite loops of transformations that lead nowhere can occur easily.

An alternative approach is derived from the ideas of DoubleDecker algorithm described in [96]. This effectively breaks down trees that are rooted by a binary operator in a positive and a negative set of subexpressions (respective to the root), then tries to

match pairs from opposite sets which can be simplified. The algorithm was extended to handle any unary and binary operator.

Probabilistically equivalent subtrees

Implementing a "perfect" simplifier was out of scope of this project. But since the above methods failed in certain circumstances we turned to an approximating algorithm, Equivalent Decision Simplification (EDS) [83]. The idea behind EDS is the following. If two sub-expressions behave the same for a significant number of inputs then they are probably equivalent, in which case the larger one can be replaced by the smaller one. EDS was integrated as a subcomponent of the rule-based simplifier.

Appendix B

Implementation

[E]PGA has a three-layered architecture, each layer implementing a different aspect of the system. A schematic view is shown in Figure B.1.

The lower layer contains two subsystems: `ExpTree` and `GP`. `ExpTree` implements concepts related to expression trees: `Node`, `Tree`, `Variable`, `VariablePool`, `Function`, `FunctionPool`, `Key` etc.

The `GP` subsystem contains the evolutionary module. It consists of classes like: `Individual`, `EvolutionaryAlgorithm`, `SearchOperatorPool`, `Population`, `Subpopulation`, etc.

The middle layer contains the implementation of PGA and PostPGA: `HashTable`, `PGA`, `PostPGA`, `Simplifier`, etc.

The upper layer contains the experiments. It is based on a flexible mechanism which allows running existing experiments with various parameters and / or adding new ones in a straightforward manner.

There are generic classes and functions grouped in the `Generics` module which serve all three layers. `Generics` includes for example the `Option` class which manages the parameters entered through the command line or a configuration file (see Figure B.2 for an example). Also in this category belongs `MPIJob` which allows the distribution of computational tasks to independent computing nodes.
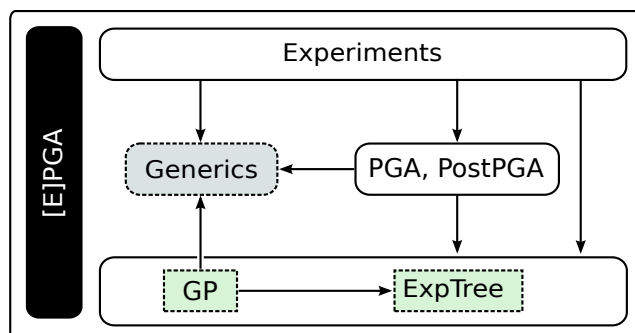

Figure B.1: Modules of [E]PGA

```
#---------------------------------------------------------------
#   -- SELECTING THE EXPERIMENT --
#---------------------------------------------------------------

experiment                        = experiment_pga3
ea-model                          = island
ea-target                         = learning-rule

#---------------------------------------------------------------
#   -- OPTIONS OF THE PARSING TREE --
#---------------------------------------------------------------

vector-size                       = 16
variables                         = x[] y d w[]
input-variable                    = x[]
output-variable                   = y
desired-output-variable           = d
weights-variable                  = w[]
training-set-file                 = spherical-16-65536

#---------------------------------------------------------------
#    -- ELRA SPECIFIC OPTIONS --
#---------------------------------------------------------------

ge-granularity                    = 512
ge-granules                       = 8
ge-integration-scaling            = linear
ge-smoothing-factor               = 0.33

#---------------------------------------------------------------
#    -- OPTIONS OF PGA AND POSTPGA --
#---------------------------------------------------------------

minimum-order-ratio               = 0.8
minimum-frequency                 = 0.5
maximum-frequency                 = 100
minimum-depth                     = 2
maximum-depth                     = 5
decreasing-frequency-threshold    = false
disable-pga-order-ratio-check     = false
saliency-formula                  = new
skip-postpga                      = false
consider-equivalent-patterns      = false
#track-pattern                    = (9.000*((@*@)+((x*d)-(y*x))))
normalize-saliencies              = true

#---------------------------------------------------------------
#   -- SIMPLIFICATION OPTIONS --
#---------------------------------------------------------------

allowed-values-x                  = -1 1
allowed-values-y                  = -1 1
allowed-values-d                  = -1 1
min-w                             = -10
max-w                             = 10
eds-simple-expressions            = 0 1 -1 x y d w
```

Figure B.2: Sample configuration file

Though it was omitted in their description for reasons of clarity, in practice whenever it was reasonable these algorithms were parallelized to make use of the underlying (parallel) hardware. For example the `evaluate()` method in `Population` detects the number of cores on the current CPU and creates the number of evaluation threads accordingly. For more coarse grained parallelism (needed for example in `PGA` or `PostPGA`) we used `MPIJob`.

[E]PGA was implemented in C++. The major external libraries that have been used include: C++ Standard Template Library (STL) [97], Pthreads [98], Graphviz [99], Boost::Serialization and Boost::MPI [100]. The scripting was done in Bash and MATLAB. The hardware platform consisted of a 32 node SGI cluster featuring 8-core, 2.5GHz Intel Xeon CPUs running SUSE Linux Enterprise Server 10.