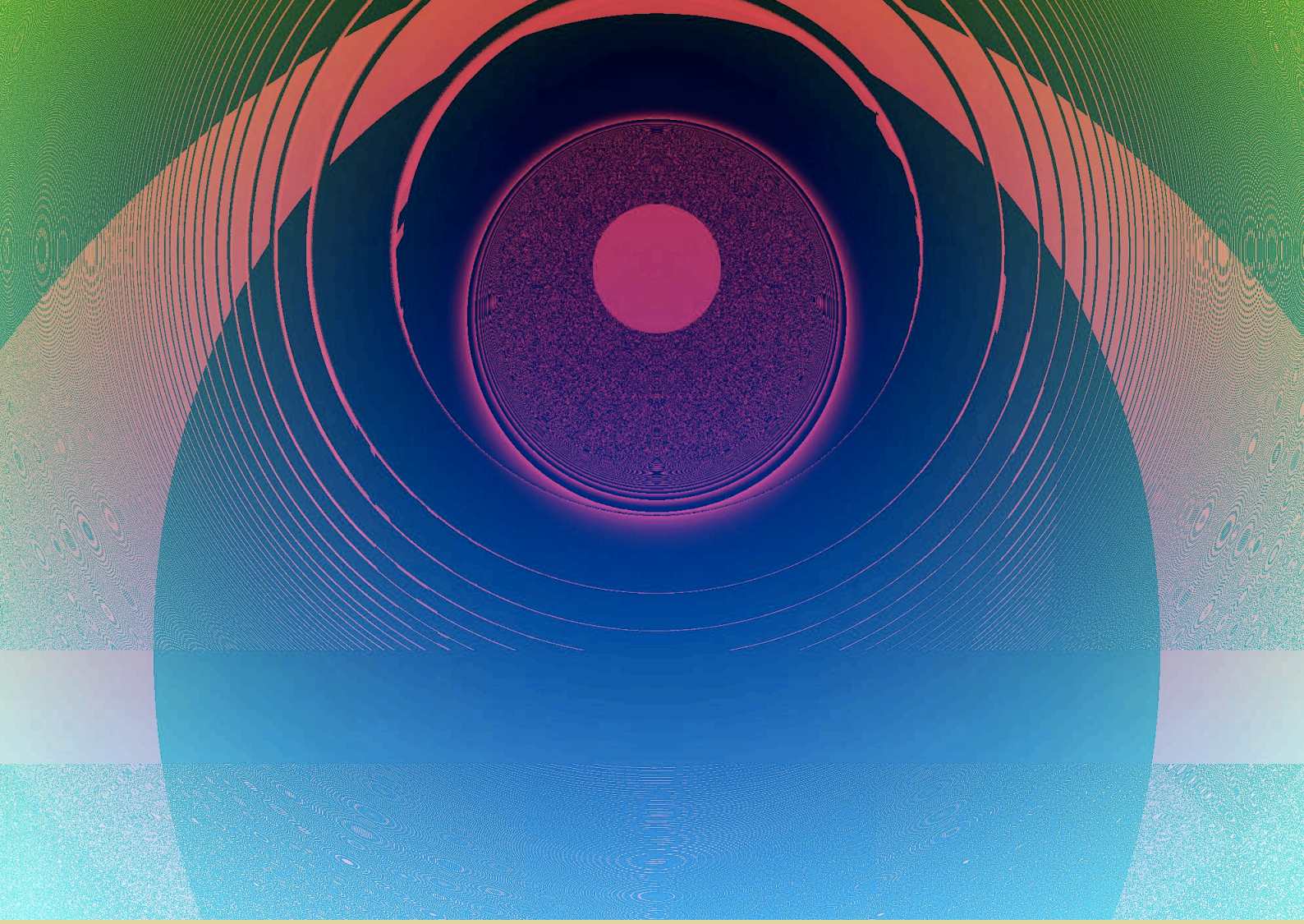


Nuno António Marques Lourenço

Enhancing Grammar-Based Approaches for the Automatic Design of Algorithms

Doctoral thesis submitted to the Doctoral Program in Information Science and Technology,
supervised by Full Professor Ernesto Jorge Fernandes Costa and Assistant Professor Francisco José Baptista Pereira, and presented
to the Department of Informatics Engineering
of the Faculty of Sciences and Technology of the University of Coimbra.

September 2015

Universidade de Coimbra

Enhancing Grammar-Based Approaches for the Automatic Design of Algorithms

A thesis submitted to the University of Coimbra
in partial fulfillment of the requirements for the
Doctoral Program in Information Science and Technology

by

Nuno António Marques Lourenço
naml@dei.uc.pt

Department of Informatics Engineering
Faculty of Sciences and Technology
University of Coimbra

Coimbra, September 2015

Financial support by Fundação para a Ciência e a Tecnologia,
SFRH/BD/79649/2011.

Enhancing Grammar-Based Approaches
for the Automatic Design of Algorithms

Cover Image: Evolved using the SGE algorithm
and the ideas behind NEvAr [Machado and Cardoso, 2002]

This dissertation was prepared under the supervision of

Ernesto Jorge Fernandes Costa
Full Professor
of the Department of Informatics Engineering
of the Faculty of Sciences and Technology
of the University of Coimbra

and
Francisco José Baptista Pereira
Assistant Professor
of the Department of Informatics and Systems
of the Polytechnic Institute of Coimbra

To the ones who always have been there, and will always be.

Acknowledgements/Agradecimentos[1]

Chegado o momento em que concluo este trabalho flui em mim um turbilhão de emoções que não consigo descrever. Os olhos começam a ficar húmidos, porque percebo que esta fase está a chegar ao fim. Sei que é um dos momentos mais marcantes da minha vida, e que irei recordar para sempre. Tudo isto só foi possível porque partilhei esta viagem com um conjunto de pessoas incríveis que me ajudaram nos momentos bons, e nos momentos menos bons. Este trabalho é partilhado com todos eles.

Primeiramente quero agradecer aos meus orientadores o professor Ernesto Costa e o professor Francisco Pereira, pois sem eles não teria chegado até aqui. Ao professor Ernesto, quero agradecer por ter aceite fazer parte desta aventura. Quero agradecer por tudo o que me ensinou, e por todas as nossas conversas, as mais formais e as mais informais. Obrigado por me mostrar, que independentemente da dificuldade de uma tarefa, desistir nunca é uma opção.

Os meus agradecimentos ao professor Francisco são tantos nem sei por onde começar. Talvez pelo inicio. Obrigado por me ter aceite como aluno numa bolsa de investigação em 2009. Obrigado por 2 anos mais tarde, ter aceite fazer parte desta aventura. Obrigado por todas as conversas que fomos tendo, e que tanto contribuíram para o meu crescimento, quer científico, quer pessoal. Obrigado pela paciência. Obrigado por ter contribuido para a pessoa que sou hoje.

Um agradecimento para as pessoas com quem partilhei a E5.8. Muito obrigado Alcides, Andreia, Filipe, Maryam e Pedro pelos momentos que passamos no laboratório e no DEI. Bem sei que em alguns desses momentos desconfiaram da minha sanidade mental, mas não se preocupem que os médicos dizem que está tudo bem.

Do grupo de pessoas da E5.8 há duas a quem quero agradecer em especial. Ao Rui Lopes quero agradecer pelas conversas e trocas de ideias, principalmente quando havia aquela água fantástica do Padrinho. Agradeço-te ainda o teres-me fornecido o template para poder escrever a tese. Um agradecimento especial também ao Ivo Gonçalves por ter feito esta caminhada comigo. Juntamente com a Joana, foram o meu porto de abrigo,

[1] For personal reasons, the acknowledgements are written in Portuguese. My sincere apologies to the non-Portuguese speakers.

quando as coisas estavam mais bravas. Recordo com alegria todas as discussões científicas, e as nossas conversas sobre O tudo.

Agradeço ao Diogo Laginha, ao Fábio Pedrosa e ao Marco Simões, e aos restantes *ninjas* pelas discussões que tivemos nos nossos jantares mensais.

Um agradecimento muito especial também para o Vitor, que mesmo estando longe, foi parte integrante do percurso. Um muito obrigado também à Teresa, e restante familia por me fazerem sentir em casa sempre que os visito. Um obrigado também à Jessica por se lembrar de dizer um olá sempre que vem a Coimbra.

Quero agradecer aos meus amigos de Foz de Arouce e arredores por me terem ajudado a levar este barco a bom porto. Eles sentiram muitas vezes as minhas frustrações, sem terem culpa nenhuma. Um agradecimento muito especial à Flávia por ser a minha *irmãzinha*. Agradeço também à Anabela pela curiosidade que sempre demonstrou sobre as coisas que eu andava a fazer, e por ter aturado muitos dos meus problemas. Obrigado Rita, pelos nossos cafés de domingo à tarde. Obrigado Tânia pelo teu sorriso, e pelos constantes encorajamentos, principalmente na parte final. Obrigado Inês, pelas discussões que me faziam chorar a rir. Obrigado João por sempre mostrares preocupação. Obrigado André Carvalhinho pela companhia durante os jogos do Glorioso. Obrigado Ana pela paciência que foste tendo. Obrigado André Rodrigues e Sérgio Simões pela amizade que partilhamos desde crianças.

Por último, agradeço àqueles que sempre estiveram comigo. Quero agradecer à minha familia por tudo o que têm feito por mim. Só graças a vocês é que todo este trabalho foi possível. Quero deixar o agradecimento especial aos meus pais, pelo enorme esforço que sempre fizeram para que eu pudesse chegar onde cheguei. Obrigado pelo apoio que sempre me deram. Um agradecimento especial à minha mãe por ter ouvido (e sentido) em primeira mão todos os meus problemas.

Sob pena de me ter esquecido de alguém, deixo aqui o agradecimento geral a todos os que contribuíram de alguma forma para eu chegar até aqui. Obrigado!

Nuno Lourenço
Coimbra, Setembro 2015

Resumo

Os Algoritmos Evolucionários (AE) são métodos computacionais de procura estocástica inspirados pelos conceitos da selecção natural e da genética. Este tipo de algoritmos tem sido usado com sucesso para resolver problemas em dominios da aprendizagem, do design e da optimização.

Para utilizar um AE é necessário definir as suas componentes principais, como por exemplo os operadores de variação, os operadores de selecção de pais, e os mecanismos de selecção de sobreviventes. O desempenho de um AE pode ser altamente melhorado se cada uma destas componentes for ajustada para o problema especifico que se pretende resolver. Normalmente estas modificações são feitas manualmente e requerem um grau de conhecimento elevado. Para tentar melhorar este processo, os investigadores têm vindo a propor algoritmos para automaticamente criar AE. Estes novos métodos usam um (meta-) algoritmo que combina as diversas componentes e parâmetros, de maneira a criar a estratégia que melhor se aplica ao problema em questão. Neste contexto surge a área das Híper-Heurísticas (HH), cujo principal objectivo é o desenvolvimento de meta-algoritmos que sejam eficientes.

A Programação Genética (PG), e em particular as variantes baseadas em representações gramaticais são habitualmente utilizadas como motor de pesquisa nas HH. Este trabalho prentende estudar e analisar em que condições a eficácia dos métodos de pesquisa pode ser melhorada, no contexto da evolução automática de AE.

As principais contribuições podem ser divididas em três aspectos. A primeira consiste na construção de uma framework de HH baseada em Evolução Gramatical (EG). A framework está dividida em duas fases complementares: Aprendizagem e Validação. Na aprendizagem, um motor de EG é usado para combinar as componentes de baixo nível que estão especificadas numa Gramática Livre de Contexto. Na validação, os melhores algoritmos encontrados são aplicados a cenários diferentes dos da aprendizagem, para analisar a sua capacidade de generalização.

A segunda contribuição está relacionada com a análise do impacto que as condições de aprendizagem têm na estrutura final dos algoritmos que estão a ser aprendidos e consequentemente na sua capacidade de optimização. Além disso é feita uma análise da relação que existe entre a qualidade dos algoritmos na fase de aprendizagem, e a qualidade dos algoritmos na fase de validação. Em concreto, analisa-se se os melhores algoritmos da fase de aprendizagem mantêm o seu bom desempenho na fase de validação.

Por fim, a última contribuição é uma proposta de uma nova representação para EG que permite resolver alguns problemas relacionados com a exploração do espaço de procura.

Abstract

Evolutionary Algorithms (EA) are stochastic computational methods loosely inspired by the principles of natural selection and genetics. They have been successfully used to solve complex problems in the domains of learning, design and optimization.

When using an EA practitioners have to define its main components such as the variation operators, the selection and replacement mechanisms. The performance of an EA can be greatly enhanced if the components are tailored to the specific situation being addressed. These modifications are usually done manually and require a reasonable degree of expertise. In order to ease the use of EAs some researchers have developed methods to automatically design this type of algorithms. Usually, these methods rely on an (meta-) algorithm that combine components and parameters, in order to learn the one that is most suited for the problem being addressed. The area of Hyper-Heuristics (HH) emerges in this context focusing on the development of efficient meta-algorithms.

Genetic Programming (GP), specifically the grammar based variants, are commonly used as HH. In this work, we study and analyze the conditions in which Grammatical Evolution (GE) can be enhanced to automatically design EAs.

The main contributions can be divided in three aspects. Firstly, we propose an HH framework that relies on GE as the search algorithm. The proposed framework is divided in two complementary phases: Learning and Validation. In Learning the GE engine is used to combine low level components that are specified in a Context Free Grammar. In the second phase, Validation, the best algorithms learned are selected to be applied to scenarios different from the learning, in order to evaluate their generalization capacity.

Secondly we study the impact that the learning conditions have in the final structure of the algorithms that are being learned. Moreover, we analyze the relationship between the quality exhibited by the algorithms during learning and their effective optimization ability when used in unseen scenarios. In concrete we analyze if the best strategies discover in learning still have the same good behavior in validation.

Our final contribution addresses some of the limitations exhibited by Grammatical Evolution. The result is a novel representation with an enhanced performance.

Keywords

Automatic Design of Algorithms, Evolutionary Algorithms, Evolutionary Computation, Genetic Programming, Grammatical Evolution, Hyper-Heuristics

Contents

List of Tables

List of Figures

List of Grammars

List of Algorithms

1

Introduction

"Computers are not the Thing. They are the thing that gets us to the Thing."

Joe, *Halt and Catch Fire*

Nature is a great designer and problem solver. Looking around us it is possible to see *endless forms most beautiful* that resulted from years and years of constant evolution. Diversity and complexity are the words we use most to describe what has unfolded before our eyes. Amazed by nature ingenious way of solving problems, man has always tried to understand the way it works. Thanks to the natural selection theory posit by Darwin, the discovery of the inheritance mechanisms by Mendel, and the comprehension of the molecular basis of biological phenomena, we now understand better why we see what we see and why we are what we are.

With the appearance of computers and the establishment of Computer Science as an academic discipline it is natural that researchers try to imitate nature, proposing computational models that mimic those mechanisms and processes responsible for promoting such diversity and complexity. The objectives are (at least) twofold: to understand even better the inspirational natural counterpart systems, and to use the models as computational devices to solve complex problems. Most computer scientists are interested in the latter aspect, that is, to develop computational models loosely inspired by the concepts of natural selection and genetics to solve hard problems in the domains of learning, design and optimization. By hard we mean those problems that either do not have an

analytical solution or, even if they do, are computationally intractable. As a result of this human endeavor a new scientific area called Evolutionary Computation (EC) was brought to light.

Evolutionary Computation deals with the design, study and application of Evolutionary Algorithms (EAs), a family of stochastic search procedures that iteratively improve a population of candidate solutions guided by an objective function. The improvement is obtained by selecting the most promising ones (according to the objective function), and promoting some stochastic variations using operators similar to mutations and recombinations that take place in biological entities.

Since the debut of EAs in the 1960s, it triggered a series of research, looking for new bio-inspired approaches. As a result, many alternative models were proposed and added to the family. The main differences between them were mostly focused on the representation used for the candidate solutions, on different mechanisms for selecting the parents and/or the survivors, and on the variation operators used.

In the early 1990s Genetic Programming (GP), another variant of EAs, was popularized by John Koza [Koza, 1992]. The aim of GP is to automatically evolve programs that solve problems, i.e., without having to explicitly program those solutions. The solutions are represented as tree expressions that can be executed to determine their quality. As happened with the others variants, over the years alternative implementations were proposed to GP. One remarkable example is Grammatical Evolution (GE), whose distinctive feature is the existence of a clear separation between genotype and phenotype. Variation operators are applied in the first level, whereas the final solution is evaluated in the phenotype level. This two-level architecture requires a mapping process to transform the genotype into the phenotype. In the case of GE this transformation is based on a grammar. More precisely, we start with a linear genotype that is used to guide a derivation of the grammar from which we extract the executable expression tree.

1.1 Motivation

Evolutionary Algorithms are now mature, and have been intensively studied to better understand their strengths and weaknesses. One of the most important results, known as the No Free Lunch Theorem [Wolpert and Macready, 1997], states that there is not an algorithm that, on average, performs better than any other, when all classes of problems

are considered. This apparently negative result explains in part why EC researchers spend so many efforts looking for good algorithms for a *concrete* problem. However, EAs have many components and parameters and the task of hand-designing the best combination of them (i.e., the one that gives the best performance) is far from trivial. It requires an high degree of expertise and experimental fine-tuning.

Since the early days of EC, there have been proposals for self-adaptive algorithms that are able to adjust online some of their parameters, at the same time they are trying to solve a given problem. A remarkable example is the adaptation of the mutation probability. *Meta-adaptation* is another perspective that is gaining relevance as a form of *offline adaptation*. Concretely meta-adaptation relies on a (meta-) algorithm to explore the space of possible strategies, thereby learning the best possible combination of componentes and parameters for a specific problem. This search process is akin to supervised learning: the meta-algorithm creates/learns an algorithmic strategy, which is then applied to an instance (or instances) of a problem. Meta-adaptation has attracted the attention of many researchers, specially in the form of Hyper-Heuristics (HH).

Hyper-Heuristics is an emergent area of research and one key issue is the choice of the meta-algorithms. In this disseration we advocate, as others researchers did, that EAs are a natural choice to search the space of EAs [Oltean and Groşan, 2003, Tavares and Pereira, 2012, Dioşan and Oltean, 2009, Poli et al., 2005b]. Our major goal in this work is to show that, under certain conditions, Grammatical Evolution is a suitable meta-heuristic to automate the design of complete EAs.

1.2 Contributions

This dissertation proposes novel contributions to the area of Hyper-Heuristics. More specifically: (1) it specifies and implements a framework to create and validate bio-inspired algorithms; (2) it analizes the impact of the learning conditions on the evolved strategies; (3) it proposes an enhancement to the meta-algorithm itself, by devising a new representation for the GE algorithm to address some of its known limitations.

1.2.1 Framework

The first contribution corresponds to the development of a framework to automatically design bio-inspired algorithms. We propose an architecture divided in two phases: *Learning* and *Validation*. In Learning a search methodology seeks for promising problem solving techniques. In this work we rely on GE since it can automatically evolve variable length algorithmic strategies (e.g., computer programs), and is thereby an appropriate search method to handle the components used to create novel optimization strategies. In the second phase, Validation, the evolved algorithmic strategies are applied to unseen scenarios to analyze their solving capabilities. We conducted a study to see whether the framework was able to evolve algorithms to tackle a specific problem. [Lourenço et al., 2012]

1.2.2 The Influence of the Learning Conditions

The second contribution addresses the conditions in which learning occurs. We rely on the framework previously proposed to empirically analyze how the learning conditions impact the structure of the evolved algorithms. Based on the insights obtained with these experiments we provide some guidelines to aid in the construction of better HH [Lourenço et al., 2013, Lourenço et al., 2014].

To keep the computational burden within a reasonable level, the evaluation of the strategies being learned relies on simplified fitness criteria. However, it is not clear if the limited evaluation step compromises the identification of the best algorithmic strategies. We analyse if there is any correlation between the quality exhibited by strategies during learning and their effective optimization ability when applied to unseen scenarios [Lourenço et al., 2015b].

1.2.3 Improving Grammatical Evolution

The last contribution is related to the improvement of the GE search engine itself. GE has some known issues related with locality and redundancy. We propose an alternative representation, called Structured Grammatical Evolution (SGE), where we address these issues. Specifically we analyze its effectiveness in a set of benchmark problems. Moreover, an analysis of SGE properties (locality and redundancy) is conducted to understand why

it has a good perfomance [Lourenço et al., 2015a].

1.3 Roadmap

The remainder of this document is organized as follows. In Chapter 2 we describe the background concepts necessary to better understand our work. Additionally we present the state of the art in automatic design of algorithms. In Chapter 3 we detail and validate the GE based Hyper-Heuristic that is at the center of this work. In Chapter 4 we analyse the impact that the learning conditions have in learning. In Chapter 5 we investigate if the fitness criteria used in learning provide enough information to identify the most effective and robust strategies. In Chapter 6 we address some of the GE's limitations, and propose a new genotypic representation. Finally Chapter 7 presents our main conclusions and future work.

2

Background

This Chapter presents some background concepts, required for the research described in the following chapters of this dissertation. It starts by presenting the main concepts of Evolutionary Algorithms (EAs) in Section (2.1). Then Section 2.2 reviews recent research related with the problem of automatic design of algorithms.

2.1 Evolutionary Algorithms

This Section introduces the core concepts of Evolutionary Algorithms (EAs). It does not aim to exhaustively enumerate all the available approaches. Rather we focus our attention on its main components and how they work. For a more in-depth review of all EA variants the reader may refer to [Eiben and Smith, 2003, Floreano and Mattiussi, 2008].

Historically there are several variants of EA. These include Evolution Strategies (ES) [Beyer and Schwefel, 2002], Evolutionary Programming (EP) [Fogel et al., 1966], Genetic Algorithms (GA) [Holland, 1975] and Genetic Programming(GP) [Koza, 1992]. However, they share the same common underlying idea: the simulation of evolution by natural selection (proposed by Darwin) of a population of artificial individuals via application of selection, variation operators, and reproduction. These components are guided by a fitness function that evaluates each individual, measuring the quality of the solution it represents. The application of these components is repeated for several iterations, and over time it is expected that the overall quality of the individuals in the population improves.

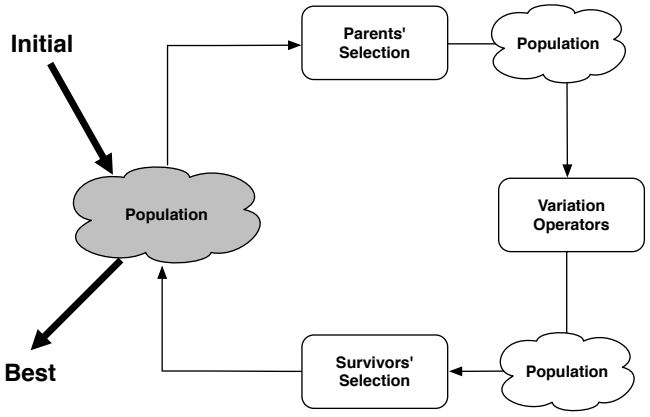


Figure 2.1: Evolutionary Algorithm

The process stops when a predetermined termination criterion is met (e.g., when a maximum number of iterations is achieved). Each artificial individual in EAs encodes a single candidate solution to a problem under consideration. The general structure of a simple EA is depicted in Fig. 2.1.

Despite these simple processing rules, EAs are robust problem solving methods able to quickly identify good quality solutions in hard problems. In the last years they have been applied with success in many fields of research, from engineering to machine learning. Amongst the examples of such applications are software optimization [Langdon and Harman, 2013], antenna design [Lohn et al., 2005], predictive modeling [Archetti et al., 2007], and algorithm design [de Sá and Pappa, 2013], [Tavares and Pereira, 2012], [Burke et al., 2010b].

2.1.1 Components

This section details the main components of EAs. The most important components are:

- Representation
- Evaluation
- Parent Selection
- Variation Operators

- Survivor Selection

Alongside with these components, one has also to define a set of numeric parameters. These parameters will be further detailed below.

Representation

While using an EA it is necessary to define how the solutions to the problem under consideration should be codified. There are several possible representations in the literature, such as: binary strings, real numbers, permutations, graphs, and trees.

Evaluation

Evaluation relies on a *fitness function* to estimate how good a solution is. This function usually is a mathematical expression and it plays a critical role in the evolution process, as it allows the comparison between problem solutions.

Parent Selection

Each individual in the population may be selected, probabilistically, to participate in the breeding of a new population.

There are various different parent selections methods that can be used, such as fitness-proportional selection, ranked-based selection or tournament-based selection. In fitness proportional selection the individuals are selected based on their relative fitness. In rank selection the individuals are selected with a probability in proportion to their ordinal ranking by fitness.

In the tournament selection, a number of individuals TS is randomly selected from the population. After they are compared with each other, and the best one is selected. Usually, this process is repeated N times, were N is the population size. The advantage of using tournament is that we can tune the selective pressure by changing the number of individuals that compete with each other in the tournament. Less individuals in the tournament implies a low selective pressure, whilst a large number of individuals results in a large selective pressure. Although fitness and rank based selection are commonly used, currently tournament is the most popular and most used form of selection.

Variation Operators

The role of these operators is to create new individuals (the offspring) using the parents. These operators are usually divided in two types: *1*) recombination, and *2*) mutation.

Recombination operators create variation in the population by taking two, or more individuals, as input, and rearrange their information to create new solutions. Mutation takes one individual as input and slightly modifies it.

Both of these operators are used in a stochastic manner.

Survivor Selection

Usually, the number of individuals in an EA is kept fixed. The survivor selection mechanism determine the solutions that proceed to the next iteration of the EA. *Generational* is the simplest survivor selection strategy. In this method the newly produced offspring replace the entire old population of individuals. However, using this strategy good individuals discovered at a certain generation might be lost, preventing them from further reproduction. The *elitism* strategy avoids this, by ensuring that the best individuals are preserved.

Alternative survivor selection strategies allow the offspring to compete with their parents for a place in the population of the next generation. The decision about which individual should survive can be based on several criteria (e.g., age, fitness values, diversity measures).

2.1.2 Parameters

When building an EA, one has to define a set of numeric parameters. These parameters impact the EA's behavior. Commonly used parameters are:

- Population size (N) - Represents the number of solutions in the EA;
- Number of Generations - Represents the duration of the evolutionary process (evaluation, reproduction, and survivor selection);
- Variation Operators Rate - Defines how often the variation operators will be applied. Usually recombination operators have an high probability of being used, whilst mutation has a small probability;

- Tournament Size (TS) - Defines the number of individuals that are selected to participate in tournament selection. Note that as the tournament size grows, the selective pressure (i.e., the probability of selecting the best individual) also grows.

2.1.3 Genetic Programming

Genetic Programming (GP) is an EA where each individual is an algorithmic strategy, i.e., a computer program [Koza, 1992], [Banzhaf et al., 1998], [Poli et al., 2008]. GP is a technique for getting computers to solve problems automatically.

The most common way to represent programs in GP are syntax-trees, where each node represents a component of the program. However, there are other forms to represent programs in GP. Linear GP [Brameier and Banzhaf, 2007], Cartesian GP [Miller and Thomson, 2000] and grammar-based GP [Whigham, 1995], [O'Neill and Ryan, 2003] are remarkable examples of alternative representations for GP. In concrete, the representations based on grammars are quite relevant for this dissertation.

The *primitive set* defines the components that can appear in a GP program. In turn the primitive set can be divided in two non-overlapping sets: *Terminals* (T) and *Functions* (F). The F set includes a number of functions with arity greater than 0, whereas the T set contains 0-arity functions, constants and the program's external inputs[1]. An example of a Terminal set is $T = \{1, x, 0\}$, where 0 and 1 are two arbitrary constants, and x is the input variable.

The function set is typically driven by the nature of the problem domain. For example in the domain of numeric problems, the function set may be composed by the 4 arithmetic functions: $F = \{*, +, -, /\}$. Fig. 2.2 shows an example of a GP program built with the described primitive set.

In order to understand how GP works, the next section shows a toy example.

An Example

Symbolic regression is perhaps the most well-known benchmark problems for GP. The goal is to approximate a function given a set of known target points (Fig. 2.3).

Firstly we need to decide which are the T and F sets. Since our goal is to evolve a polynomial with one variable, x, the terminal set must include it. We also define a set of

[1] They typically have the form of named variables such as x or y.

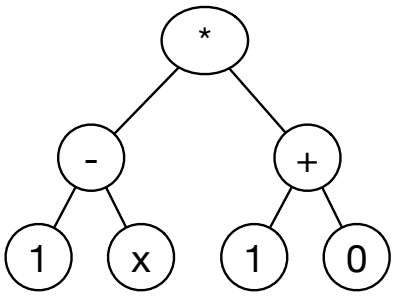


Figure 2.2: Example of Genetic Programming program.

arbitrary constants that can be used: 1, 0. Since we are dealing with a numeric problem, it is impossible to define the entire set of constants that exist. Hence, we provide a small set of constantes that can create the others that are missing. For example, the expression $\frac{1}{1+1}$ represents the 0.5 constant. In the end T is defined as:

$$T = \{x, 0, 1\} \tag{2.1}$$

Since no restrictions are given about which functions should be used to evolve the program, the function set is composed by the four arithmetic operations: addition (+), subtraction (-), multiplication (*) and division (/). Note that the division employed here is protected, that is, if the denominator is 0, it returns the value 1. Hence, F is defined as:

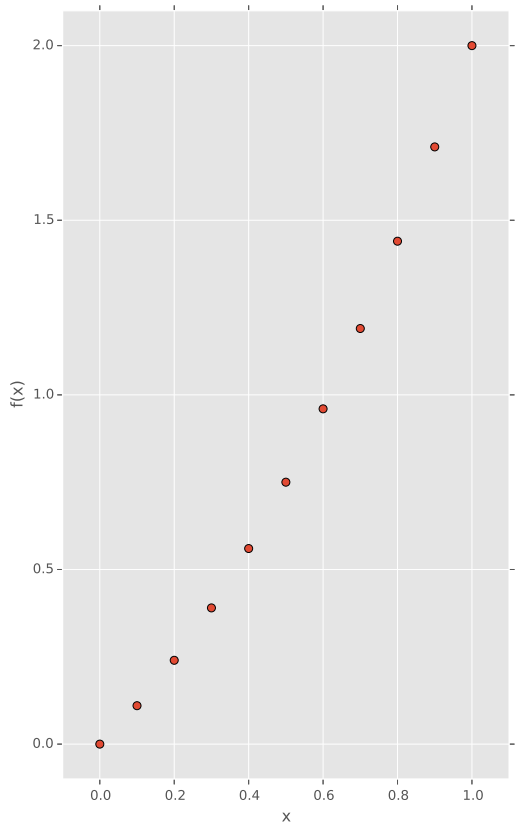
$$F = \{+, -, *, /\} \tag{2.2}$$

The next step is to define a fitness measure that assigns quality to each individual. In this type of problems the most commonly measure used is the sum of the absolute error between the output of an individual and the desired output:

$$Error = \sum_{i=1}^{n} |g(x_i) - f(x_i)| \tag{2.3}$$

where, n is the number of output cases, $g(x_i)$ is the output of the individual for the input x_i, and $f(x_i)$ is the known value for the input x_i.

Now that we have defined the function and terminals sets, and how to evaluate indi-



x	f(x)
0	0
0.1	0.11
0.2	0.24
0.3	0.39
0.4	0.56
0.5	0.75
0.6	0.96
0.7	1.19
0.8	1.44
0.9	1.71
1	2

Figure 2.3: Target function. The table on the right shows the known points, and the plot (on the left) shows a visual representation of the same points.

viduals we can start the GP run. For the sake of simplicity, a population of 6 individuals will be considered.

Initialization

The first step in a typical GP run is to create the population of possible solutions. There are three main methods to build an initial population: *full*, *grow*, and *ramped-half-and-half*. These methods generate individuals that do not exceed an user specified maximum *depth*. The depth of a tree is the number of edges that need to be transversed to reach a leaf, starting from the tree root. The tree root is assumed to be at depth 0.

The grow initialization select nodes from the whole primitive set (i.e., $F \cup T$) until a certain tree depth is reached. Once the depth limit is reached only terminals may be chosen.

In the full initialization, only nodes from the F set are selected, until a certain tree depth is reached. At the tree limit, only terminals may be chosen, just as in the grow method.

The ramped-half-and-half method is a combination of the two methods described above, in which half of the population is created by grow, and the other half is created by the full method. This is done for each depth level (from minimum to maximum), to ensure that trees of different sizes are generated.

Assuming that we define a maximum tree depth of 2, the six individuals that compose our population were randomly created using the ramped-half-and-half and are present in Fig. 2.4.

Evaluation

Each individual in the population is evaluated to measure its quality. Randomly created individuals, such as the ones from Fig. 2.4, typically have very poor fitness values. To calculate the fitness of an individual we use the Eq. 2.3. Table 2.1 shows how to compute the error of the individual depicted in Fig. 2.4a).

Parent Selection

The next step corresponds to the probabilistic selection of the individuals, based on their fitness. Better individuals should have higher chances of being selected to produce new

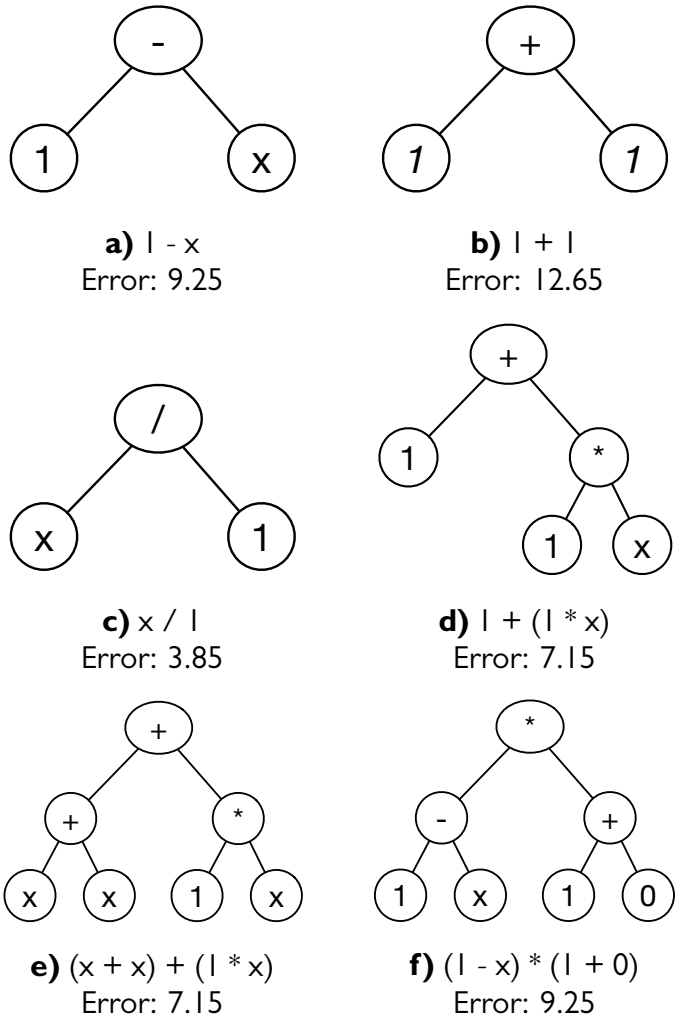


Figure 2.4: GP initial population of six randomly generated individuals.

Table 2.1: Example of how to evaluate the individual represent in Fig. 2.4a)

x	**f(x)**	**g(x)**	**|g(x)-f(x)|**
0	0	1.0	1.0
0.1	0.11	0.9	0.79
0.2	0.24	0.8	0.56
0.3	0.39	0.7	0.31
0.4	0.56	0.6	0.04
0.5	0.75	0.5	0.25
0.6	0.96	0.4	0.56
0.7	1.19	0.3	0.89
0.8	1.44	0.2	1.24
0.9	1.71	0.1	1.61
1	2	0	2.0
Error (Eq. 2.3)			9.25

Table 2.2: Example of the application of the tournament selection with size 2

Tournament Individual 1	**Tournament Individual 2**	**Winner**
1 + (1 * x)	1 - x	**1 + (1 * x)**
1 + 1	x / 1	**x / 1**
(1 - x) * (1 + 0)	1 + 1	**(1 - x) * (1 + 0)**
1 - x	x / 1	**1 + (1 * x)**
(1 - x) * (1 + 0)	x / 1	**x / 1**
1 + (1 * x)	1 + 1	**1 + (1 * x)**

solutions. In this example we will rely on the tournament selection mechanism, with $TS = 2$. However other methods can be used. Table 2.2 shows the application of the tournament selection. The first two columns of the table represent the individuals that were randomly selected from the population. The last column (*Winner*) represents the individual that won the tournament.

Variation operators: Recombination and Mutation

Using the set of parents, the next step is the creation of new individuals via the variation operators.

Recombination is usually a binary operator, that receives two parents, and combine their trees to generate a two new individuals. In the standard recombination operator one

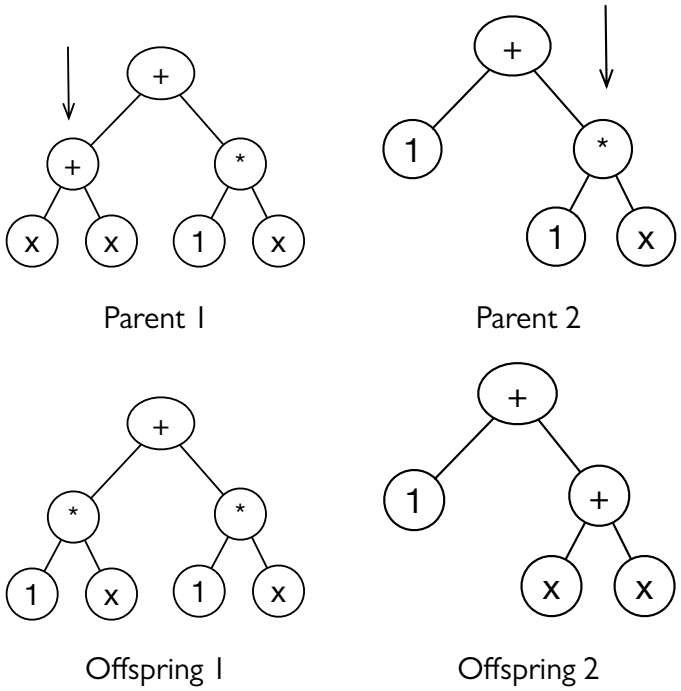


Figure 2.5: Example of the crossover application. The arrow in each parent indicates the crossover points, i.e., the root of the trees that will be exchanged.

subtree is randomly selected in each parent. If the subtrees are syntactically equivalent, they are swapped. Fig. 2.5 shows a possible example of the application of crossover.

Mutation is an unary operator, that receives one individual, and modifies it by randomly selecting a subtree and replacing it by a newly random generated tree within the initial depth limits. Fig. 2.6 shows an example of how mutation works.

Termination

The evolutionary process ends when a predefined number of generations is achieved.. When this happens, the best-so-far individual is returned as the result of the GP execution.

2.1.4 Grammatical Evolution

In recent years Grammar-Based Genetic Programming (GBGP) has been used to aid the automatic design of algorithms [Pappa and Freitas, 2009], [Burke et al., 2012], [Marshall

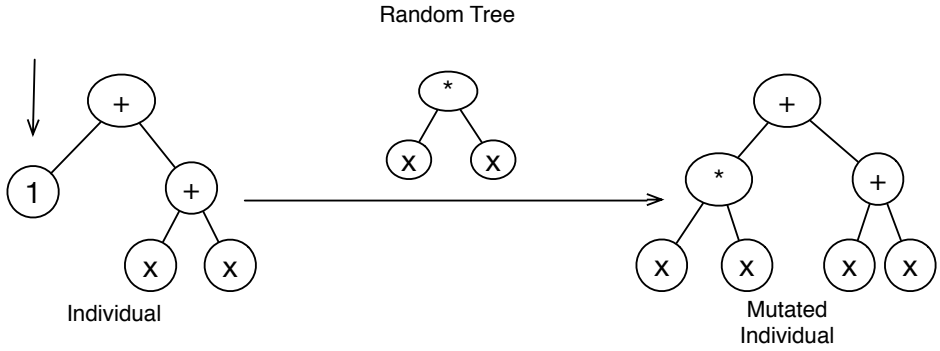


Figure 2.6: Example of the subtree mutation application. The arrow in the parent indicates the place were mutation will happen, i.e., which subtree will be replaced.

et al., 2014]. One remarkable example of GBGP is Grammatical Evolution. Other variants of GBGP exist and an in-depth review is presented in [McKay et al., 2010].

Grammatical Evolution (GE) was first proposed by Ryan et al. [Ryan et al., 1998]. As with standard GP, the goal of GE is to evolve executable algorithmic strategies. GE is different from other non grammar-based GP variants, for there is a clear separation between the genotype and the phenotype. The former is a linear sequence of integers, each one called a codon[2]. The phenotype is the final executable program, usually in the form of a tree expression, constructed using a grammar. The use of a grammar allow us to construct programs in any language. For instance, GE has been used to generate programs in Java, C, or Python.

Given the two distinct levels a mapping process is necessary to translate a genotype into a final program. The mapping relies on the productions rules of a context-free grammar (CFG). A CFG is a tuple $G = (N, T, S, P)$, where N is a non-empty set of non-terminal symbols, T is a non-empty set of terminal symbols, S is an element of N called axiom, and P is a set of production rules of the form $A ::= \alpha$, with $A \in N$ and $\alpha \in (N \cup T)^*$. N and T are disjoint. We say that $W \in (N \cup T)^*$ derives $Z \in (N \cup T)^*$, i.e. $W \Rightarrow Z$, if and only if $W = uxv$, $Z = uyv$ and there is a production in the grammar of the form $x ::= y$. The $\Rightarrow$ is a relation over the set of $\alpha \in (N \cup T)^*$. Its reflexive and transitive closure is denoted by $S \overset{*}{\Rightarrow}$. Each grammar G defines a language $L(G)$, *i.e.*, the set of all sequences

[2]The original proposal considered a binary sequence. The binary sequence would then be converted to an integer one, where each group of 8 bits corresponded to a codon. Nowadays the binary sequence is optional.

of terminal symbols that can be derived from the axiom, also called words, denoted by $L(G) = \{w : \overset{*}{\Rightarrow} w,\ w \in T^*\}$.

The translation of the genotype into the phenotype is done by simulating a leftmost derivation from the axiom of the grammar guided by the genotype. The genotype is scanned from left to right and each codon is used to determine the production rule that should be selected to expand the leftmost non-terminal symbol of the current partial derivation tree. Suppose that we have a genotype $(8, 15, 33, 10, 27, 6)$ and the following production rule:

$$
\begin{aligned}
< expr > ::= & < expr >< op >< expr > && (0)\\
& |(< expr >) && (1)\\
& | < pre-op > (< expr >) && (2)\\
& | < var > && (3)\\
< op > ::= & + && (0)\\
& | - && (1)\\
< var > ::= & x && (0)\\
& | y && (1)
\end{aligned}
$$

and we want to rewrite the symbol $< expr >$.

To determine which alternative will be used, we take the first codon and divide it by the number of options for $< expr >$. The remainder of that operation identifies the selected option. In our example, $8\%4 = 0$ and the symbol is rewritten as $< expr >< op >< expr >$. Then the second integer is read, and the same method is used for the leftmost non-terminal of the derivation. In Fig. 2.7 we show the complete derivation process. Sometimes the length of the genotype is insufficient to complete the translation process. If this happens, the sequence is repeatedly reused in a process known as wrapping. When a pre-determined maximum number of wrappings is exceeded, the process stops and the worst possible fitness is assigned to that solution. For a detailed description of the process, consult [O'Neill and Ryan, 2003].

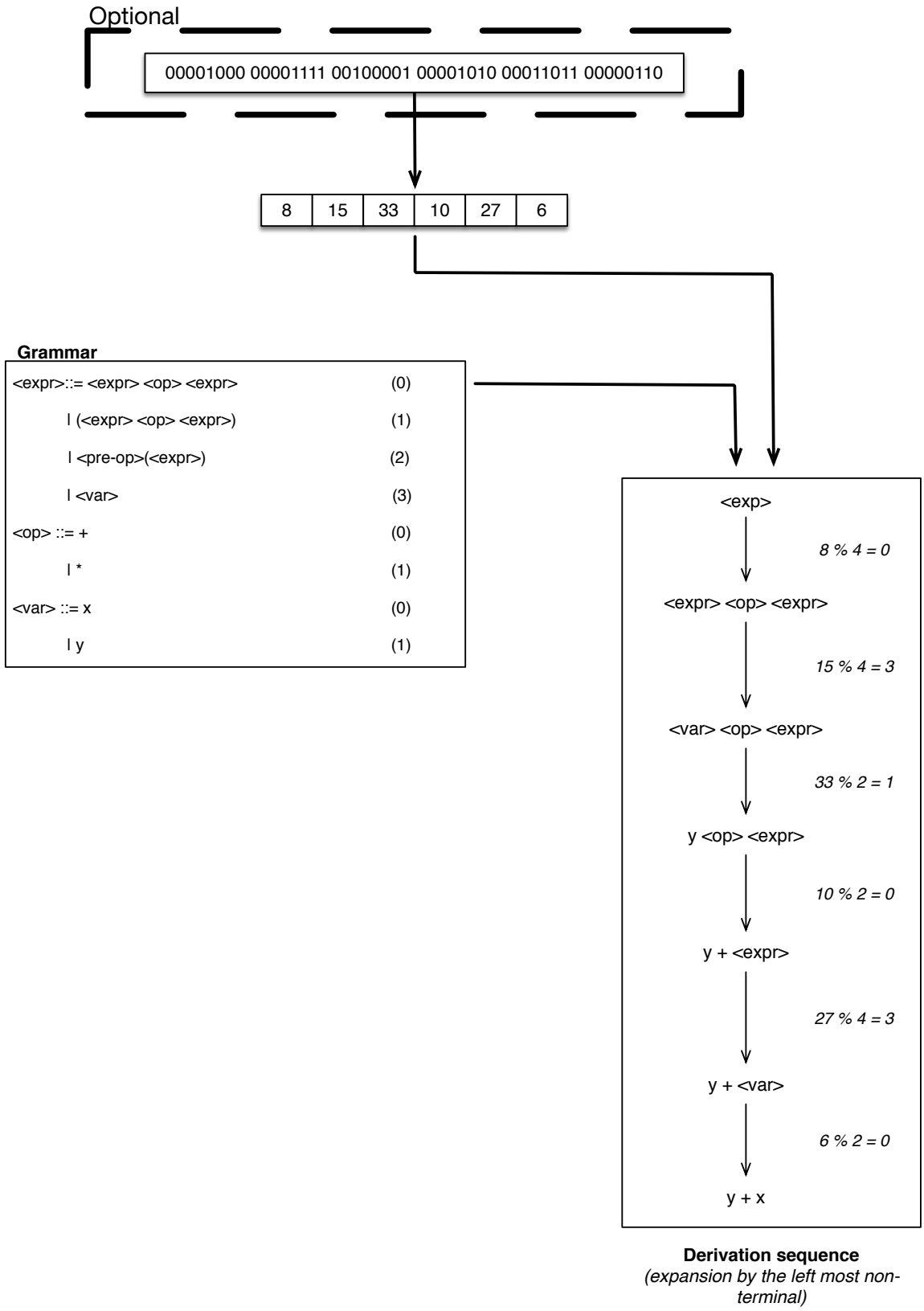


Figure 2.7: Adapted from [McKay et al., 2010]: An illustration of the Grammatical Evolution mapping from a binary chromosome. The integer values are used in the mapping function to decide which production rule from the grammar to apply to the current non-terminal symbol. This results in the generation of a derivation sequence.

2.2 Automatic Design of Algorithms

When using Evolutionary Algorithms (EAs) practitioners are faced with several challenges about their design. These challenges include decisions about devising the most adequate representation for solutions, finding the set of variation operators, choosing the selection mechanisms, or defining the numeric parameters for each of the algorithm components. One possible approach is to use a trial-and-error tuning, where practitioners experiment several combinations of the components before adopting the one that achieves reasonable good results. Although this might seem a reasonable way of configuring an EA, there are no clear design rules. Every time that one needs to use an EA in a new problem, the whole process of decision/selection has to be done again. Moreover the trial-and-error approach can prevent the reproducibility of results, since most of the times the choice of the components is not properly explained and justified.

The necessity of removing this trial-and-error process from the human side, and the necessity of creating more systematic methods of configuring EAs, motivated a growing interest in the automatic design of algorithms. The aim of the automatic design of algorithms is to delegate the task of searching for the best configuration to the algorithm itself, or to other algorithms that could select and combine the components in the way that seemed most beneficial. When the algorithm performs the modifications itself (i.e., changes its configuration) while running, it is said to have *online adaptation*. Evolution Strategies (ES) were the first EA dialect to introduce the concept of online adaptation. ES were equipped with mechanisms to modify the rate of the variation operators while solving a problem. Inspired by the concepts behind ES, researchers started to modify other EAs to include adaptation mechanisms, in order to improve the algorithm's performance (Section 2.2.1).

Offline adaptation happens if the search for the best configurations is performed *before* the algorithm execution. *Meta-adaptation* is an example of offline adaptation. In *meta-adaptation* two algorithms are used: one for problem solving, and another one (the meta-algorithm) to build a configuration for the first one, i.e., for the problem solver. The configurations of the first algorithm are built by selecting and combining various low level components.

The development of meta-adaptive frameworks has attracted the attention of many researchers in recent years, specially in the form of Hyper-Heuristics (HH) (Section

2.2.2). We focus our attention on meta-evolutionary algorithms, i.e., on approaches that use EAs to search for the best configurations. Nevertheless there are alternatives that rely on other algorithmic configurations methods like [Shan et al., 2006, Marmion et al., 2013]. The iterated racing techniques proposed by [López-Ibánez et al., 2011] have shown good results while configuring different types of algorithms.

2.2.1 Online Adaptation

Evolution Strategies (ES) were one of the first EAs to introduce the concept of online adaptation (or self-adaptation). The mechanism of adaptation proposed consisted on counting the successful and unsuccessful mutations for a certain number of generations. A mutation is said to be successful if the fitness of the individual improves. If more than 1/5th of the mutations are successful, then the mutation amplitude should be increased. Else, if less than 1/5th of the mutations are successful, the mutation amplitude should decrease. Finally, if exactly 1/5th of the mutations are successful, then mutation amplitude is kept unchanged. Later Schwefel [Schwefel, 1981, Schwefel, 1993] introduced an improved self-adaptation mechanism for changing the ES's parameters. This method is based on the adaptation of the covariance matrix of a normal distribution.

Following the same ideas behind ES, [Davis, 1989] applied them in the context of a steady-state EA with two operators: recombination and mutation. In this work each operator has a certain fitness, which is modulated by a function that makes a relation between the total number of individuals created and the ones that are highly fit (i.e., are promising solutions). This function works on generation windows, meaning that its value is updated every certain number of generations. The update is based on the following: an operator sees its fitness increased by either creating highly fit individuals, or by setting the conditions for the appearance of such individuals. In his experiments, recombination and mutation started with the same initial fitness. At each iteration of the algorithm, an operator is selected using a probability that is based on the operator's fitness. The individual that was created by the application of the operator goes into the population and replaces a low fitness individual. Each individual keeps a record of which operator created it. If the new created individual is an improvement over the parent(s) used to create it, the operator receives a credit for the creation of the individual. The credit function is cumulative in such a way that it goes back to parents, grandparents, and so

on back to a predefined level of ancestors. These dynamic adjustments of the operator fitness are used to select the operators based on their usefulness for the evolution stage: if the population is too similar the selection rate of mutation should be increased. On the other hand, if the population is very diverse, the selection rate of mutation should be decreased. The proposed work showed that the described method improved the performance of the EA on some problems that were used as benchmarks.

[Spears, 1995] presented another EA with online adaptation, which evolved the type of recombination operator that should be used. In this work each individual in the population had an extra bit. This bit is then used to choose between two types of recombination operators: 0 indicated that the recombination operator should be a uniform-crossover; and 1 indicated that the recombination operator should be a two-point crossover. Hence when two individuals were selected for reproduction, the algorithm examined the last bit. If the two bits were 0's, uniform-crossover is used. If the two bits were 1's, then two-point crossover is used. Finally, if the two bits were different one of the two recombination operators was randomly chosen. The approach was tested on a set of problems, and it appeared to obtain good performance results. However the author claims that the performance improvements do not come from the method itself, but rather from the fact the there were two recombination mechanisms at the EA's disposal. Moreover he says that the test problems used in the study could not benefit from the fact of having self-adaptive mechanisms.

[Angeline, 1996] studied two new self-adaptive crossover operators for Genetic Programming (GP) inspired by the self-adaptive mechanisms existing in ES. The first crossover is Selective Self-Adaptive Crossover (SSAC), and adapts values that determine where a crossover will occur in an individual. The second crossover, Self-Adaptive Multi-Crossover (SAMC) adapts how crossovers are applied to the individuals. Both of the self-adaptive crossovers were applied to three benchmark problems and compared to the standard GP crossover, and the experiments showed that the self-adaptive methods performed consistently as good or better than the standard one. Next [Edmonds, 2001] proposed a method that allows the adaptation of GP operators along with the candidate solutions to the problem being tackled. The operators are encoded using a tree representation, and can perform the following operations:

1. Return the left side tree of random node;

2. Return the right side tree of random node;

3. Return the root node of the left side tree;

4. Cut the tree that passes in a certain node.

Each operator receives two parent trees as arguments alongside with the points were the crossover will occur. The operators's population has its own fitness function, which reflects the improvement that the usage of that operator brings to the population. The approach is applied to the *even-parity bit* problem. The results achieved showed that this approach was more effective than standard GP. However the author claims that the technique tends to be sensitive to biases in the functions/rules that are used to build the operators, which makes the approach brittle.

PushGP, is a programming language that supports *auto constructive evolution* [Spector and Robinson, 2002]. Auto constructive evolution is a self-adaptive mechanism that allows programs to construct their variation operators, their offspring, and ultimately their evolutionary process. This means that each evolved program will contain code that is responsible for reproduction and diversification. PushGP was used in symbolic regression problems, showing promising results.

More recently, [Kramer and Koch, 2007] proposed an EA that self-adapted its own variation operators. In this work, the recombination operator has its parameters (e.g. crossover points) inside the population's solutions. This allows them to evolve together. In [Kruisselbrink et al., 2011] the authors explored the self-adaptation mechanisms in EA that were applied to problems that could be represented using a binary encoding.

2.2.2 Meta-Adaptation

This section reviews some of the approaches that use EA as meta-adaptation algorithms. The first part corresponds to the approaches that adapt the components of a traditional EA. The second part focuses on the approaches that adapt the components of Swarm Intelligence algorithms [Eberhart et al., 2001]. The last part presents other approaches that used EA as meta search strategies to construct novel problem solving techniques.

Evolutionary Algorithms

In [Grefenstette, 1986], Grefenstette proposes a meta-EA to optimize the parameters of another EA that is being used to solve a particular problem. The proposed meta-EA is limited to a subclass of EA, which is characterized by the following 6 parameters:

- Population size (N)
- Crossover Rate (C)
- Mutation Rate (M)
- Generation Gap (G)
- Scaling Window (W)
- Survivor Selection Strategy (S)

The Scaling Window (W) is the minimum value that the fitness function can assume, which is then used to perform some fitness scaling operations. The approach defines a set of discrete intervals for each one of the parameters. The Generation Gap (G) represents the percentage of individuals that are replaced in each generation.

The proposed method takes into account two types of performance: the online performance (averaged of all tested structures over the course of the search), and the offline performance (averaged of all tested structures using a simulation model). Two experiments were made: one to assess the online performance and another to assess the offline performance. These experiments resulted in two algorithms, that were compared, in a set of test functions and in a real problem, with a standard EA with the following parameters: (N=50, C=0.6, M=0.001, G=1.0, W=7, S=E), where E represents an elitist strategy. In both cases the optimized algorithms achieved a better performance than the standard EA. Nevertheless, the author recognizes that the method has some drawbacks, such as the fact it is necessary to choose a particular parameterized subclass of EA's to explore. Other components are neglected, such as the replacement strategy. Similar approaches were later proposed in [Bäck, 1994, Eiben et al., 1999, Smith and Eiben, 2009].

[Oltean and Groşan, 2003] propose an EA that is able to evolve an EA. The approach works at two levels: the first (the meta-level) consists of a steady-state EA with a

fixed population size, fixed operator probabilities, fixed selection and replacement mechanisms. The second level consists in the solutions encoded in the chromosome of the meta-EA. As such the operators' probabilities can vary, as well as the population size, replacement and selection mechanisms. The EA used in the meta-level is based on the Multi Expression Programming (MEP) model [Oltean, 2002]. The MEP model uses a fixed-encoding for the chromosomes. Each chromosome is composed by a set of genes, where each gene can encode a terminal (from the set T of terminal symbols) or a function (from the set F of functions). When a gene encodes a function, it has pointers towards its arguments. Thus, the first element in the chromosome must be a terminal symbol in order to obtain syntactic valid programs. Alg. 1 presents an example of a MEP chromosome. In this example consider $T = \{a, b, c, d\}$ and $F = \{+, -, *, /\}$. The numbers on the left positions are labels/pointers, which are used as the function arguments.

Algorithm 1 Example of a MEP chromosome [Oltean, 2002]

1: a
2: b
3: + 1, 2
4: c
5: d
6: + 4, 5
7: * 2, 6

The chromosome translation is performed in a top-down manner, starting in the first gene. The genes that correspond to terminal symbols are translated into simple expression, whilst the genes with functions correspond to more complex expressions. For instance the number 7 gene is translated into the expression $b * (c + d)$.

In order to evolve full-featured EA the authors had to define the primitive set. The mechanisms that typically appear on EA:

- Initialize - Unary operator that generates a random solution;
- Select - Binary operator that selects one from two solutions;
- Crossover - Binary operator that recombines two solutions;
- Mutation - Unary operator that changes a solution.

Taking these operators into account the only operator that is not dependent on already existing solutions is the *Initialize* operator. Therefore it belongs to the terminal set $T = \{Initialize\}$. All the others belong to the function set $F = \{Select, Crossover, Mutation\}$. These sets are then used to build MEP chromosomes, which correspond to an EA (Alg. 2).

Algorithm 2 Example of an evolved EA using MEP [Oltean, 2002]

1: Initialize
2: Initialize
3: Mutate 1
4: Select 1, 3
5: Mutate 4
6: Crossover 2,5

In the evolution process, the meta-EA has to know the quality of each EA that is encoded in the chromosomes. Therefore, each EA encoded in the meta-EA is executed in a particular problem. The quality of the best solution found by the evolved EA is returned as feedback to the meta-EA.

The proposed approach was tested in a function optimization scenario and the results showed that the approach is effective. Nevertheless there are no comparisons with an EA specially implemented for the problem, and the proposed approach only allows the evolution of non-generational EA. This approach was further explored in [Oltean, 2007]. They compared the results of the evolved algorithms with a standard EA specially designed for the problems being solved, and obtained equivalent results. In [Oltean, 2005], the author extended the previous work in order to evolve a generational EA.

As before the approach works at two levels. The meta-EA is based on Linear Genetic Programming (LGP). LGP uses a representation where programs are written in an imperative language, like C, instead of tree-based GP expressions of a functional programming language, like LISP. An individual in LGP is represented as a variable-length sequence of C language instructions. These instructions operate on one or two variables, called *registers*, or in constants, from a predefined set. The result is stored in a third register, called *destination register*. The inputs to a certain chromosome are given using the registers' initialization. Alg. 3 (adapted from [Brameier and Banzhaf, 2001]) shows an example of a LGP chromosome.

Algorithm 3 LGP Example

$r[0] = r[1] + 95$
$r[5] = r[3] - 10$
$r[2] = r[4] + r[5]$
$r[7] = r[6] * r[2]$
$r[0] = r[1] + 9$

To evolve an EA the meta-EA will apply a function, from the set of functions, to one or two registers, and the result will be stored in a third register. The set of functions is composed by three types of genetic operators that commonly appear in EA. Since the objective of the work was to evolve a generational EA, each individual of the meta-EA has a loop. Furthermore a mechanism to randomly initialize the population was also added to each individual. Taking into account all of these considerations an example of an evolved EA by the meta-EA using LGP is presented in Alg. 4.

Algorithm 4 EA evolved using LGP [Oltean, 2005]

```
Randomly_initialize_the_population();
for k = 1 → MaxGenerations do
    Pop[0] = Mutate(Pop[5]);
    Pop[7] = Select(Pop[3], Pop[6]);
    Pop[4] = Mutate(Pop[2]);
    Pop[2] = Crossover(Pop[0], Pop[2]);
    Pop[6] = Mutate(Pop[1]);
    Pop[2] = Select(Pop[4], Pop[3]);
    Pop[1] = Mutate(Pop[6]);
    Pop[3] = Crossover(Pop[5], Pop[1]);
end for
```

The assessment of the individual's quality is similar to the previous approach. When performing the experiments, the author used a training set and a test set. The former corresponds to the problem instances that the meta-EA uses to evolve the algorithms. The latter corresponds to the problem instances that are used to measure the algorithm's generalization ability. The experiments were conducted in three different scenarios: *1*) Function optimization, *2*) Traveling Salesman Problem, and *3*) Quadratic Assignment Problem. In each scenario a comparison with a standard EA was made. The results showed that for every scenario the meta-EA was able to evolve EAs that per-

formed similarly to the one used for comparison.

[Dioşan and Oltean, 2009] presented a new meta-EA to evolve EAs. The approach is similar to the previous ones, and it was applied to function optimization benchmarks, confirming the results previously obtained in [Oltean, 2002, Oltean, 2005, Oltean, 2007].

[Tavares et al., 2004] proposed the evolution of genotype-phenotype mapping functions. In their work the authors propose a GP algorithm that evolves a population of mapping functions, which are then used by an EA to solve a function optimization problem. Experimental results showed that the GP algorithm is able to find mapping functions that can obtain results as good as the ones that are designed by hand.

In [Woodward and Swan, 2011], the authors propose a framework to evolve selection mechanisms to EAs using Register Machines (RM). RM are a computational representation that consists of a list of instructions and a list of registers which act as memory and are updated according to the program's instructions. The results obtained showed that there is a margin to increase the performance of the commonly used selection mechanism, since the framework was able to evolve strategies that outperform the standard fitness proportional and rank selection. This work was later extended in [Woodward and Swan, 2012] to automatically design mutation operators for EAs. They rely again on RM to represent the list of instructions that compose the mutation operator. The results confirm the viability of the approach, as the evolved mutation strategies were able to statistically outperform the human-design approaches. In spite of the good results obtained, the use of RM as representation might be seen as a drawback. The programs created are a black-box, and it is very difficult to look at the strategies and understand the way they work.

[Smit and Eiben, 2010] introduced REVAC, which is a meta-EA to tune the parameters of another EA. In concrete they selected an EA that won the CEC-2005 Special Session on Real-Parameter Optimization, and tried to improve its performance. The results obtained showed that the REVAC was able to improve the results of the EA. The results obtained in the article reflect why automatic design of algorithms is important. By selecting an EA that had won a competition, which must have been carefully tuned and designed to achieve the best results possible, the room for further improvements were small. However they were able to find a set of parameters that increased the EA's performance. Now, if we consider EAs that are being developed by practitioners that have not been pushed to their limits, the margins for improvements are much bigger.

[Martin and Tauritz, 2013] proposes a framework based on GP to evolve Black-Box algorithms. In their work the GP is used to evolve parse trees that represent a single iteration of an algorithm. The parse tree is composed by a set of operators commonly used in EAs with a binary representation. They use a simple problem to validate the approach, and show that the evolved algorithms can outperform hand-design approaches. However, as the authors acknowledge, the evolved approaches have the tendency to become overspecialized, i.e., they only perform well on the instances where they were learned.

Swarm Intelligence

[Poli et al., 2005a, Poli et al., 2005b] developed a meta-EA based on GP that is able to automatically create new Particle Swarm Optimization (PSO) algorithms for a certain set of problems. The main objective of this work was to categorize PSO, and explore extensions to the traditional PSO. The meta-EA uses a set of simple elements considered essential to create a PSO algorithm. To determine the quality of the PSO that is being evolved, the authors selected two benchmark problems related to function optimization. Thus, at each iteration every individual, that corresponds to a PSO is executed, and the fitness of the meta-EA individual corresponds the fitness of the best solution found. Experimental results showed that the evolved PSO can effectively solve the addressed problems.

[Tavares and Pereira, 2010] proposed a meta-EA to evolve pheromone trail updates for Ant Colony Algorithms (ACO) [Dorigo and Stützle, 2004]. This approach relies on standard GP as meta-EA. As such, one key aspect is the definition of the terminal and function sets. The authors selected a terminal/function set as simple as possible, yet it allowed the construction of all known ACO algorithms. Similar to other approaches already described, the fitness of each meta-EA individual corresponds to the best encoded solution found by the strategy encoded by that individual, while solving a certain problem instance. Results obtained with the Traveling Salesman Problem (TSP) showed that the proposed approach was able to evolve strategies to solve the instance for which they were evolved. Moreover the strategies also showed good generalization abilities for they were able to solve other instances different from the ones used in the training process. [Tavares and Pereira, 2011a, Tavares and Pereira, 2011b] extended the previous

work. In order to ensure that only valid strategies were evolved, they adopt a meta-EA based on Strongly Typed GP (STGP) [Poli et al., 2008]. STGP is a GP variant that forces data types on terminals, on function parameters and on function return values. These restrictions provide some advantages over standard GP, for instance, when one is working with multiple data types. However it imposes extra caution while developing STGP operators since they need to ensure that the relations between types are still valid. [Tavares and Pereira, 2011a] uses the STGP approach to evolve strategies to the TSP problem, and confirmed the results previously obtained. In [Tavares and Pereira, 2011b], they used the same approach to evolve strategies to another problem: the Quadratic Assignment Problem (QAP). They studied both generalization and scalability capacities of the evolved strategies. Furthermore they also studied the inter-problem generalization, i.e. if the updated rules evolved in a certain problem (e.g., TSP), had the same behavior when applied to a different problem (e.g., QAP). The results showed that strategies evolved in the TSP worked well on the QAP but the opposite was not true.

[Tavares and Pereira, 2012] proposed a Grammatical Evolution (GE) framework to evolve complete ACO algorithms. The adopted grammar allows the creation of a full algorithmic structure and the selection of specific components for each optimization step. The approach's experimental analysis focuses on the capacity of the framework to evolve well-known structures, and if it could evolve novel ACO algorithm structures that could improve the algorithms effectiveness. This framework is particularly relevant because it relies on a grammar to guide the creation of strategies. Several advantages emerge when using a grammar-based system, the most important being related with understandability. This means that we can look and the strategies that are being generated and see what components are being used in each phase of the algorithm.

Hyper-Heuristics

The term Hyper-Heuristic (HH) was used for the first time by Dezinger et al. [Denzinger et al., 1997] to describe a framework that combined several artificial intelligence methods in the context of automatic theorem proving. Later, the term was used to describe a "heuristic to search heuristics", in the context of combinatorial optimization problems [Cowling et al., 2001]. Recently the definition of HH was extended by [Burke et al., 2010a] to "an automated methodology for selecting or generating heuristics to solve

hard computational search problems".

The foundation of the current body of HH can be traced back to the early 1960s, to a work in Operational Research by Fisher and Thompson [Fisher and Thompson, 1963]. In this work they showed that by combining scheduling rules in production scheduling, they could obtain superior results to using any of the rules separately.

[Fang et al., 1993] proposed an EA to automatically evolve heuristics for a job-scheduling problem. The approach consisted of combining simple heuristics usually used in the job-scheduling problems. The experimental results showed that the approach was effective and it could compete with more traditional methods used to solve the problem.

[Dorndorf and Pesch, 1995] tackled the problem of job shop scheduling. They proposed a EA to optimize the sequence of decisions that should be made to obtain good results in the problem. They used a set of simple priority heuristics that are commonly applied to solve conflict problems in the scheduling. The representation of each individual in the EA corresponds to the number of operations to schedule in the underlying problem instance. The algorithm presented some good results when compared to other algorithms used in the same problem.

[Wah et al., 1995] studied two important problems in designing efficient algorithms: *1*) automated design of problem solving heuristics; *2*) systematic search of heuristics that can be applied to unseen problems. To study such problems, the authors used an EA based learning system called TEACHER. This learning system has three phases of learning: classification, learning, and generalization. The first phase partitions the test cases into distinct subsets. In the learning phase, the goal is to find effective Heuristic Methods (HM) for each of a limited set of subdomains. In the last phase, called generalization, the goal is to find a HM from the set of learned HM and see if it has the same high level of performance improvement on unlearned subdomains. Furthermore they proposed a method to schedule the evaluation of the generated heuristics, since that can demand large quantities of computational resources.

[Drechsler et al., 1996] presented an EA that combine heuristics for Ordered Binary Decision Diagrams (OBDD) minimization, starting from a given set of basic operations. The difference to other previous approaches to OBDD minimization is that the EA does not solve the problem directly. Rather, it develops strategies for solving the problem. It is assumed that the problem to be solved has the following property: there is a defined a non empty set of optimization procedures that can be applied to a given (non-optimal)

solution in order to further improve its quality. These procedures are called Basic Optimization Modules (BOM). These BOM are the basic modules that will be used. Each heuristic is a sequence of BOM. The goal of the EA is to find a good (or even optimal) sequence of BOM such that the overall results obtained by the heuristics are improved. The fitness function defined by the authors takes into account two aspects of the heuristics: the cost and the quality. The cost is related with how efficient a heuristic can be, and the quality is related with how good a solution to the problem being solved is. Furthermore, the authors define two parameters, which can control the trade-off between efficiency and quality.

[Pappa and Freitas, 2009] presented an approach to automate the design of data mining algorithms. The main goal of data mining is to extract knowledge from data and transform it into a human-understandable knowledge for further use. The proposed approach uses GBGP to search for rule induction algorithms. A rule induction algorithm tries to extract patterns from data and build some rules to make assertions about the the data. The data discovered by the GBGP framework showed competitive results when compared with state of art human-designed algorithms.

[Burke et al., 2012] presented an approach to evolve heuristics for the bin packing problem. The approach uses GE to evolve the heuristics. The experimental results of the work focuses on the quality, efficiency and consistency of the evolved heuristics. Taking into account these aspects the proposed approach showed capacity to evolve heuristics that satisfied all of them. Nevertheless the authors suggest that there is room for improvements since the optimal solution was not obtained as often as they desired.

3

Automated Design by means of Grammatical Evolution

The construction of Hyper-Heuristic (HH) frameworks can alleviate the task of deciding the most appropriate EA's components for a given problem. Additionally, these frameworks may introduce some benefits in terms of performance improvement. The previous Chapter reviewed recent works that aim to automatically design Evolutionary Algorithms (EAs). Based on the works reviewed, we concluded that the frameworks that use on grammars tend to be more appellative when compared to others.

This Chapter presents and details a computational framework that is able to evolve EAs using Grammatical Evolution (GE) [Lourenço et al., 2012]. GE relies on grammar that defines the space of possible algorithmic strategies. The benefits of using this approach is an increased flexibility on defining the syntactic restrictions of the strategies being evolved. Another advantage of using a grammar based approach is that it is possible to describe the strategies in pseudo-code or any programming language ready to be compiled. Moreover it is possible to look at the evolved algorithms and obtain knowledge about the way they work. Given this, and its recent past of successful applications, we decided to build an HH based on GE.

The framework establishes a two phase architecture (Section 3.1). In the first phase GE searches for promising algorithmic strategies. The second phase analyses the best learned algorithms and measures their generalization abilities.

To validate the framework, the Royal Road functions (RR) [Mitchell et al., 1991] were selected as the benchmark problem to assess the ability of the proposed approach to

evolve EAs (Section 3.2).

3.1 Framework

The goal of this section is to systematize and detail the HH framework proposed in this dissertation. Fig. 3.1 provides a visual description of the framework. The first phase, *Learning*, corresponds to the design and construction of algorithms. The second phase, *Validation*, evaluates the best strategies in unseen scenarios.

Learning

This phase is divided in two levels. In the first level, *Search*, a GE-based HH is adopted to search the space of possible algorithmic strategies for a given problem. This space is defined by a grammar, where its production set specifies the components and the general structure of the algorithms. Using a grammar to specify what we want to learn allows us to build a generic framework, that can be used to learn algorithms with an arbitrary structure. Practitioners are left with the task of choosing the algorithmic components and the necessary rules to build valid strategies.

The second level, *Problem Domain*, evaluates the strategies that are being evolved by GE, i.e., measures the solving capacity of the algorithm in the problem at hand. Fig. 3.2 gives further details on the entire evaluation process. The first part of this process is to apply the algorithm to instances of the problem. Each algorithm (Alg_i) is evaluated in a certain number of instances (I), which are selected beforehand. Another aspect that should be taken into consideration is the type of instances used. On one hand they have to accurately reflect the properties of the problem. On the other hand, the process should not take to long, in order to avoid large computational overheads. These two forces are usually contradictory, i.e., easy instances usually do not reflect accurately the problem's properties, and have to be carefully balanced.

Another decision that has to be made concerns the definition of the number of runs (N), the number of iterations (K), and the maximum population size (M) of learning. These parameters have a direct influence in the running time of the algorithm in each instance.

Another important decision that has to be made is related to the initial set of solutions

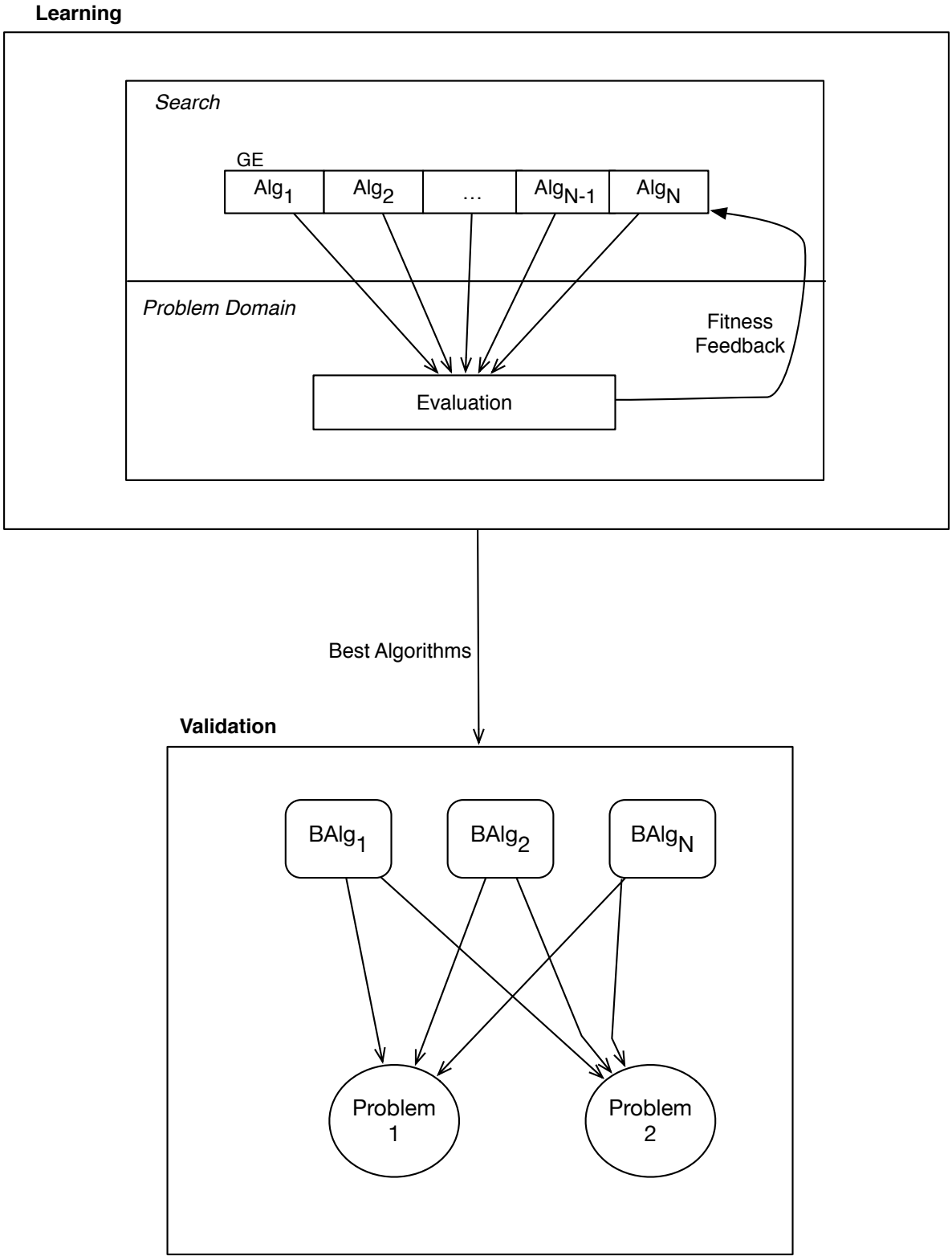


Figure 3.1: Architecture of the grammar-based Hyper-Heuristic developed

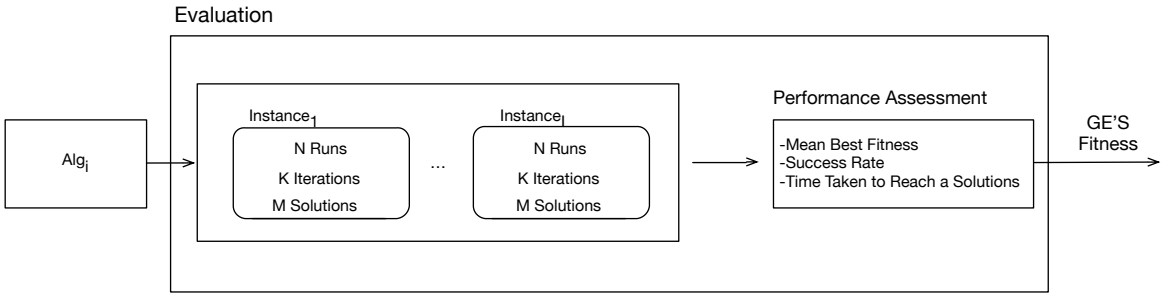


Figure 3.2: Details of the Evaluation Step

of the algorithm being evaluated. In our case all the algorithms are provided with the same set of initial solutions when they are evaluated. This means that all the algorithms start from the same point in the search space. This allows a fair comparison between different strategies and reduces the variance of the results. At every new iteration of GE the initial set of solutions is modified.

The evaluation module returns a numerical value to the GE, estimating the optimization capacity of that algorithm. The selection of an appropriate metric is important, because the information is used by the GE-HH to compare algorithms. Several metrics can be considered such as the Mean Best Fitness, the number of times that the algorithm successfully solves a given problem (Success Rate), or the time taken to reach a certain solution. Additionally it is possible to consider a combination of these metrics.

The parameters described above define the learning conditions. They are a crucial part of the overall learning process and might influence the structure and optimization ability of the algorithms. Currently, there are no silver bullets on how to accurately define the learning conditions, but the next few chapters provide some useful insights on these issues.

Validation

The aim of Learning is to identify promising solutions to a certain problem. However, we need to verify if the evolved strategies can solve tasks beyond the ones used in Learning. The aim of Validation is to verify the optimization capacities on the evolved strategies in different instances of the same problem, and/or instances of different problems. We

start by selecting the best strategies evolved ($BAlg_i$). The selection of the best strategies is based on two criteria. The first is the quality obtained while solving learning problems. When ties exist we use the second criterion: time taken to reach a solution.

Then each algorithm is executed to assess the degree to which they are accurate, robust and consistent in scenarios beyond learning. We rely on statistical inference tools [Field, 2009] to analyze and compare the results obtained. This analysis usually includes comparisons with hand design approaches to the problems under consideration.

In this work the analysis is supported by the non-parametric Friedman's ANOVA [1] test that checks for statistical differences in the means of the different algorithms being compared. This test is used because the samples do not follow a normal distribution. When differences are detected, the *post-hoc* Wilcoxon Signed Rank Test, with Bonferroni correction, is applied to perform the pairwise comparisons. In both tests we consider samples of size equal to 30.

After performing the pairwise comparisons, and when statistical differences were detected, we measured their effect size, which indicates the magnitude of the difference. The results are reported using the following graphical notation: A +++ sign indicates that the algorithm in the row is statistically better than the one in the column, and that the effect size is large ($r \geq 0.5$). A ++ sign indicates that there are statistical differences, and that the effect size is medium ($0.3 \leq r < 0.5$), whereas a + identifies a significant difference with a small effect size ($0.1 \leq r < 0.3$). A - signals scenarios where the algorithm in the row is worst than the one in the column. Finally, a $\sim$ indicates that no statistical differences between the algorithms were found.

3.2 Evolving Evolutionary Algorithms to the Royal Road Functions

The Royal Road functions (RR) functions [Mitchell et al., 1991] define optimization scenarios where population-based algorithms with crossover and mutation tend to outperform methods that do not rely on a combination of these variation operators. Moreover, the hardness of RR instances can be adjusted by changing the value of some parameters. It is therefore an appropriate environment to study the ability of a computational framework

[1] With a significance level of $\alpha = 0.05$

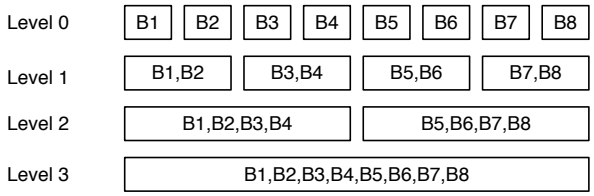


Figure 3.3: Royal Road Function with 3 levels and 8 blocks

to automatically discover EA structures.

RR were introduced by Holland, Mitchell and Forrest aiming to provide insight into the optimization behavior of EAs. These functions were designed in such a way that they can be solved by a simple population-based algorithm with crossover and mutation, but not by a hill-climber [Jones, 1994]. More precisely, the study was searching for answers to the following questions [Mitchell et al., 1991]:

1. Which problems are more suitable for EA's?

2. What is the effect of crossover on the EA's performance on different landscapes?

3. How does crossover helps to find good quality solutions?

A RR function takes a binary string as input, and produces a real value. The problem corresponds to a search task in which one wants to find strings with high fitness values. The RR can be described as a mapping: $F : \{0, 1\}^n \rightarrow \mathbb{R}$, where n is the size of the binary string. Binary strings encoding solutions are composed by a sequence of 2^k non-overlapping contiguous regions, where k is a parameter that defines the instance of the RR. Each region is divided in two sections: a section of b bits called block, followed by a section of g bits called gap. Thus, a region is composed by $(b + g)$ bits. *A complete block* is defined when all bits of the block are set to 1.

Furthermore, the RR functions are composed by *levels*. Levels correspond to contiguous sequences of 2^l complete blocks, where $0 \leq l \leq k$. Fig. 3.3 represents how the levels are defined for a RR instance with $k = 3$.

3.2.1 Evaluation of the Royal Roads

The evaluation of the RR functions is meant to capture one landscape feature of particular relevance to EAs: the presence of fit low-order building blocks that recombine to produce fitter, higher-order building blocks. The evaluation proceeds in 2 steps: the *PART* calculation and the *BONUS* calculation [Jones, 1994]. The parameters that are necessary for the fitness assessment are:

m^* Used in the PART fitness. It is the maximum number of 1's that a block may contain before being penalized. As an exception, if a block is complete it is not penalized;

v Used in the PART fitness. If the number of 1's in the block is m^* or less, it adds v, otherwise it adds -v. If the block is complete *v* does not contribute to the fitness.

u^* Value added to the BONUS part by the first completed blocks at each level;

u Value added to the BONUS part by the second or subsequent completed blocks.

PART

This step of the evaluation considers blocks individually. Each block receives a fitness score and in the end the individual block fitnesses are all summed up to produce the PART contribution to the overall fitness. The fitness of each block is based only on the number of bits 1 that it contains. Every 1 up to a limit m^* adds a value v to the block's fitness. However if a block contains more than m^* 1s, but less than b, it receives $-v$ for each 1 over the limit. Finally, if a block has all bits set to 1 it receives nothing from the PART calculation. Assuming $m^* = 4$ and $v = 0.02$ (Table 3.1) the PART fitness of Fig. 3.4 is: $PART = 0.00 + 0.08 + 0.00 + (-0.04) = 0.04$.

BONUS

In the BONUS step, complete blocks and some combinations of complete blocks are rewarded. In RR functions there are $k + 1$ distinct levels. At all levels, the first sequence of completed blocks receives fitness u^*, and additional sequences of completed blocks receive u. Assuming $u^* = 1.0$ and $u = 0.2$ the BONUS fitness of Fig. 3.4 is: $BONUS = 1.0 + 0.2 = 1.2$.

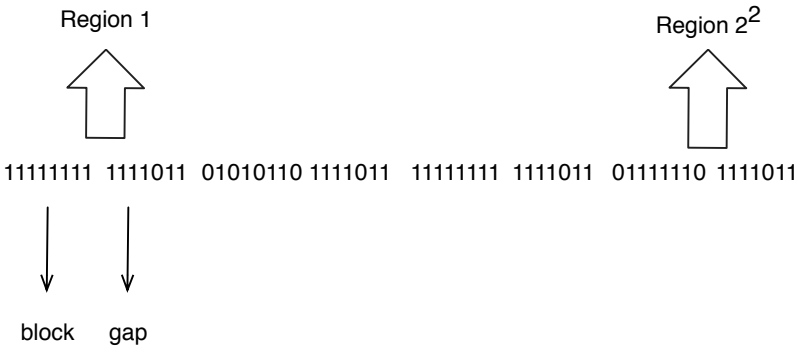


Figure 3.4: Royal Road Binary String with k = 2, b = 8, g = 7

1's in block	0	1	2	3	4	5	6	7	8
Block Fitness	0.00	0.02	0.04	0.06	0.08	-0.02	-0.04	-0.06	0.00

Table 3.1: PART block fitness for the RR default parameters with $m^* = 4$ and $v = 0.02$

The total fitness corresponds to the sum of both PART and BONUS. Using the results of the example used, the total fitness is: $PART + BONUS = 0.04 + 1.2 = 1.24$.

3.2.2 Grammar Specification

This section specifies the Context Free Grammar to evolve EAs. It is composed by set of components described in the literature for binary EAs (Grammar 3.1). All algorithms that can be evolved by this grammar are population-based. The $<$ *proportion* $>$ parameter defines the population size of the population in terms of the maximum number of individuals allowed in learning. The parent selection strategies are the ones commonly used by practitioners: fitness-proportional (Roulette Wheel, and Stochastic Universal Sampling (SUS)), rank-based, and tournament. The recombination operators are the commonly used Single Point Crossover, Multi Point Crossover and Uniform Crossover. The mutation operator is the bit flip. Note that the evolved strategies might involve zero or more combinations of these operators. In terms of survivor selection, it is possible to have generational strategies, and strategies with replacement by rank. Finally, the algorithms checks if the bests individuals of the previous generation should pass along to the new one. All the parameter settings needed by the components are also specified in the

grammar.

The grammar allows the evolution of strategies similar to the ones that shown in Alg. 5 and Alg. 6.

Algorithm 5 Example of Genetic Algorithm evolved with Grammar 3.1

```
1: (lambda,0.5)
2: while not termination condition do
3:     evaluate
4:     RouleteWheel()
5:     SinglePointCrossover(0.9)
6:     PointMutation(0.05)
7:     RankReplacement
8:     Elitism(0.01)
9: end while
```

Algorithm 6 Example of (20, 5) Evolution Strategy evolved with Grammar 3.1

```
1: (lambda,0.5)
2: while not termination condition do
3:     evaluate
4:     RouleteWheel()
5:     PointMutation(0.8)
6:     Generational
7: end while
```

3.2.3 Experiments

This section studies the ability of the HH framework to evolve EAs to solve the RR problem, using the grammar already described. The first part of the study focuses on the Learning phase. In the second part, we analyze the optimization ability by applying the obtained algorithms to several different RR instances. The RR instances used are described in Table 3.2.

3.2.4 Learning

The first phase of the experimental study was dedicated to analyze the capacity of the GE framework to evolve EAs. The settings used in the framework are presented in Table

$N = \{start, proportion, pop\_parameter, EA, selection, t\_size, integer\_const,$
$variation, operator, recombination, mutation, replacement, e\_size, prob, elitism, random\_per\}$
$T = \{0.01, 0.05, 0.1, 0.25, 0.5, 0.75, 0.9, 1.0, OnePoint, NPoint, Uniform,$
$PointMutation, Generational, RankReplacement, lambda, mu, evaluate,$
$random\_integer, random\_0\_1,$
$RouleteWheel, SUS, RankBased, Tournament, (,)\}$
$S = \{start\}$

And the production set P is:

$$< start > ::= (< pop\_parameter >, < proportion >)(< EA >)$$
$$< proportion > ::= 0.25|0.5|0.75|1.0$$
$$< pop\_parameter > ::= lambda|mu$$
$$< EA > ::= while(notterminationcondition) evaluate < selection >< variation >$$
$$< replacement >< elitism >$$
$$< selection > ::= RouleteWheel|SUS|RankBased|Tournament(< t\_size >)|\lambda$$
$$< t\_size > ::= < integer\_const >$$
$$< integer\_const > ::= random\_integer$$
$$< variation > ::= < operator >< variation > |\lambda$$
$$< operator > ::= < recombination > (< prob >)| < mutation > (< prob >)|\lambda$$
$$< recombination > ::= OnePoint|NPoint(< integer\_const >)|Uniform|\lambda$$
$$< mutation > ::= PointMutation|\lambda$$
$$< replacement > ::= Generational|RankReplacement|\lambda$$
$$< e\_size > ::= 0.01|0.05|0.1$$
$$< prob > ::= 0.01|0.05|0.1|0.5|0.9|1.0| < random\_per >$$
$$< elitism > ::= Elitism(< e\_size >)|\lambda$$
$$< random\_per > ::= random\_0\_1$$

Grammar 3.1: Grammar used to evolve EAs for the Royal Road Functions.

	Parameters							
Instance	k	b	g	m^*	v	u^*	u	Optimum
1	3	6	5	3	0.02	1.0	0.3	7.3
2	4	8	7	4	0.02	1.0	0.2	12.8
3	4	8	7	2	0.02	1.0	0.2	12.8
4	5	8	7	2	0.02	1.0	0.2	23.1
5	5	8	7	4	0.02	1.0	0.2	23.1

Table 3.2: Royal Road functions used

Parameter	**Value**
One Point Crossover Probability	0.9
Bit Flip Mutation	0.01
Codon Duplication Probability	0.01
Codon Pruning Probability	0.01
Population Size	100
Selection	Tournament with size equal 5
Replacement	Steady State
Codon Size	8
Number of Wraps	3
Generations	50
Number of Runs	30

Table 3.3: Grammatical Evolution Parameters

3.3.

The quality of each individual generated by the GE is assessed by solving instance 1 of Table 3.2. When solving one RR instance, the most effective algorithms are able to accomplish three main tasks: *1*) create complete blocks from scratch; *2*) complete nearly finished blocks; and *3*) join complete blocks. Therefore, to assign fitness to a solution generated by the GE, the GE framework performs three runs (one for each task), of the aforementioned instance. In each different run the initial population is seeded with solutions that allow an assessment of the ability of the EA to succeed in one of the previously identified tasks (e.g., the ability to join complete blocks is tested by starting the EA with an initial population containing solutions with some already completed blocks). The EA ran for 2000 evaluations and the fitness value provided as feedback to the GE corresponded to the mean of the best level achieved in each of the scenarios.

The algorithms evolved by the GE were able to find the best solution for the selected learning instance. The average fitness obtained by the 30 best algorithms (one from each run) in the training phase was 7.099356 ($\pm$ 0.530).

Another objective of this experiment is to analyze if the GE engine is able to discover innovative EA structures. Clearly, the defined grammar imposes syntactic limitations in the organization of evolved algorithms, thereby hindering the emergence of unusual structures. Moreover, it is well known that standard EA are effective in the optimization of RR functions.

An inspection of the 30 best-evolved algorithms (i.e, the best evolved strategy in each GE run) confirms that the computational framework tends to converge to solutions similar to those regularly used to solve the RR functions. Most of the differences appear in the selection and replacement methods. In table 3.4 we present the frequency of appearance of the components in these best solutions (values are in percentage). The operators' components are not exclusive, as they can appear together in the EA. In what concerns selection, there is not a clear winner, although roulette wheel is the least adopted method. As for replacement, the one based on rank clearly outperformed the generational approach. That is a more conservative strategy, maintaining in the population solutions that are not outperformed by descendants (in terms of fitness). Results show that keeping good solutions in the loop help to enhance the effectiveness of algorithms when seeking good RR solutions. In terms of operators it is possible to see that NPointCrossover is rarely adopted. On the other hand the Single Point Crossover and Point Mutation have the same percentage of appearance, which is an indication that they tend to appear together.

3.2.5 Validation

This section analyzes how the evolved algorithms behave in instances that are different from the one used in training. This will help verifying if the evolved EA are competitive with the standard algorithm in RR optimization.

Firstly, we analyze if there was any evolution occurring during the learning. Thus we selected the 30 best individuals of the initial population, 30 best individuals of the middle population (i.e. after 25 generations), and 30 best individuals of the last population and applied them to instance 2 of Table 3.2. Fig. 3.5 presents the results of this analysis. The

Components		
Replacement	Generational	16.7%
	RankReplacement	80.0%
Selection	RouleteWheel	13.3%
	Rank	30.0%
	Stochastic Universal	26.7%
	Tournament	26.7%
Operators	SinglePointCrossover	36.7%
	NPointCrossover	16.7%
	UniformCrossover	30.0%
	PointMutation	36.7%

Table 3.4: Frequency of components appearance on the EA evolution phase

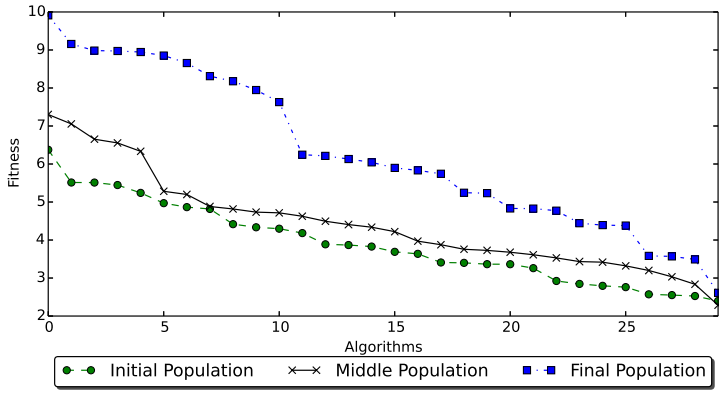


Figure 3.5: Evolution Analysis of 30 best individuals selected from the initial, middle and final populations

horizontal axis identifies the algorithms from best (algorithm with id 0) to worst (algorithm with id 29). The vertical axis shows the quality obtained by each algorithm. The empirical results show that evolution is occurring: looking at the fitness values of the individuals of the initial population we see that they are clearly the worst ones. The individuals in the middle population start to exhibit good capacities when good solutions are discovered, whilst the ones on the last population have the capacity of discovering better solutions than the other ones.

The next study is focus on the generalization capacity of the strategies. In this set of experiments we selected the seven best evolved algorithms. This selection was based on

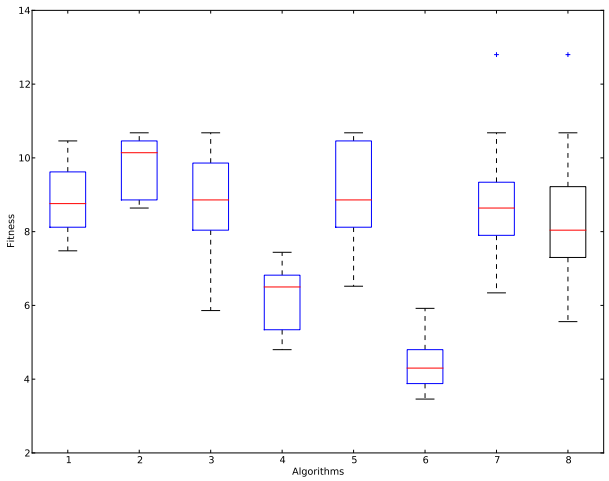


Figure 3.6: Validation results

Table 3.5: Friedman's ANOVA statistical test

$\chi^2(7)$	p-value
124.7	$< 2.2 * 10^{-16}$

fitness. Additionally we built a commonly used EA to optimize the RR: roulette wheel selection, single point crossover with **0.9** probability, point mutation with **0.01** probability and generational replacement without elitism. Each algorithm was applied to instance 2 of Table 3.2. Now, each algorithm is allowed to perform 256000 function evaluations [Mitchell et al., 1991]. Optimization results are presented in Fig. 3.6. The results show that the evolved algorithms have a similar behavior when compared to the standard one (algorithm number 8 presented in the figure).

The statistical results of Table 3.5 reveal that there are differences between the algorithms. However, the pairwise comparisons presented in Table 3.6 reveal that there are no statistical differences between the evolved algorithms and the standard EA. Although, two of the algorithms are worse (Alg4 and Alg6) than the standard, and there is one that is better (Alg2). An analysis of the components of the worst algorithms reveals that Alg6 relies on Uniform Crossover as the main variation operator, which is probably too disruptive for the RR optimization. The two typical variation operators, point mutation and single point crossover, are presented in Alg4. However the crossover has low probability

Pair	Result
Alg1 - StandardEA	$\sim$
Alg2 - StandardEA	$+$
Alg3 - StandardEA	$\sim$
Alg4 - StandardEA	$-$
Alg5 - StandardEA	$\sim$
Alg6 - StandardEA	$-$
Alg7 - StandardEA	$\sim$

Table 3.6: Results of the Wilcoxon Signed Rank test

rate, delaying the combination of blocks.

The structure of Alg2 is different from the usual (see below). It has 2 types of crossover (NPoint crossover with 16 cut-points and single point crossover, both with a high probability rate), separated by the application of point mutation with a low probability rate.

Alg2

```
1: (lambda,0.5)
2: while not termination condition do
3:     evaluate
4:     RankSelection()
5:     NPointCrossover(1.0, 16)
6:     PointMutation(0.01)
7:     SinglePointCrossover(1.0)
8:     RankReplacement
9: end while
```

Since the algorithm is different from the usual architectures it was applied to three additional instances of the RR (instances 3, 4, 5 of Table 3.2). Table 3.7 presents the results of the statistical comparison between Alg2 and the standard EA, using the Wilcoxon Signed Rank test. The presented results show that the standard EA performs better in the two harder instances ($m^* = 2$), and that the Alg2 perform better in the other.

These results seem to indicate that Alg2 delays the creation of complete blocks, due to the disruptive nature of a recombination operator with many points. Thus, and since we use a small m^*, we start to penalize incomplete blocks earlier, which leads to higher overall fitness penalizations. On the contrary, when we use an higher m^*, the behavior of Alg2 improves, as this defines a situation where incomplete blocks are less penalized.

Instance	Result
Instance 3	−
Instance 4	−
Instance 5	+

Table 3.7: Statistical Results of the comparison Alg2-Standard EA

3.3 Summary

This Chapter introduced a two phase HH framework. In the first phase, Learning, a Grammatical Evolution (GE) engine is used to combine low level components that are specified using a Context Free Grammar. In the second phase, the best algorithms learned are selected to be applied to unseen scenarios to evaluate their generalization capacity.

To demonstrate the viability of the approach, the framework was used to evolve Evolutionary Algorithms to tackle the Royal Road functions benchmark. The grammar used by GE is composed of a full set of commonly used EA components. The flexibility of the grammar is limited, hindering the appearance of unusual architectures, yet it allows some variations on number and order of operators that can appear.

To assess the capacity of the GE framework to evolve EAs, the RR functions were used as the benchmark problem since they have one good algorithmic solution.

The experimental results revealed that the framework is able to evolve well known algorithm architectures. Furthermore an architecture that is different from the traditional ones was evolved. Additional experiments were conducted to assess the effectiveness of this new architecture.

These initial experiments suggest that the automatic evolution of EA is possible. Furthermore, it raises several important research questions such as: *1*) How can we assess the quality of an EA, and retain the important information, while reducing the computational effort? *2*) How do the Learning conditions (e.g. number of generations conceded for learning) influence the structure of the algorithms? *3*) Which information should we take from the evaluation of the evolved algorithms?

4

Learning Conditions

Building on the preliminary results discussed on the last Chapter, we aim to address the impact of the learning conditions on the structure and effectiveness of the evolved algorithms. While learning, one must decide on a set of conditions related to the evaluation of the algorithms. The definition of such conditions might influence the structure of the algorithms being learned and the final quality of the strategies. Gaining insight into these situation is crucial to provide useful guidelines to define better HH.

In this Chapter we apply the framework previously described to learn EAs for a family of Knapsack Problems (Section 4.1). The first set of experiments analyses how learning is affected by the time that the algorithms have to run (Section 4.2). The length of the evaluation is defined by two parameters: the population size and the number of generations.

The second set of experiments is concerned with the influence of different algorithmic designs. The goal is to verify if the HH is able to build strategies for different types of architectures. Specifically, we focus our attention on how different replacement methods influence the creation of selection strategies (Section 4.3).

4.1 Benchmark Problems

4.1.1 The 0/1 Knapsack Problem

The *0-1 Knapsack Problem* (KP) is a combinatorial optimization problem often used as benchmark. It can be described as follows. Given a set of n items, each of which with some profit p_j and some weight w_j, how do we select a subset of items that maximizes the profit while keeping the sum of the weights bounded to the maximum capacity C of the knapsack. Formally this is stated as:

$$\max \sum_{j=1}^{n} x_j p_j \tag{4.1}$$

subject to

$$\sum_{j=1}^{n} x_j w_j \leq C \tag{4.2}$$

where

$$x_j = \begin{cases} 1 & \text{if the item } j \text{ is selected} \\ 0 & \text{otherwise} \end{cases} \tag{4.3}$$

If condition 4.2 is not satisfied, the solution is considered invalid. When this happens, the fitness is penalized. There are several penalty functions described in the literature [Michalewicz, 1996]. We adopted the linear penalty. Based on this, the fitness function is defined as:

$$\max \sum_{j=1}^{n} x_j p_j - Penalty(x) \tag{4.4}$$

where

$$Penalty(x) = \begin{cases} \rho(\sum_{j=1}^{n} x_j w_j - C) & \text{if } \sum_{j=1}^{n} x_j w_j > C \\ 0 & \text{otherwise} \end{cases} \tag{4.5}$$

with $\rho = max_{j=1\ldots n}(\frac{p_j}{w_j})$.

For the experiments conducted, 3 instances with different sizes were created, using the R script present in Appendix A.1. The general characteristics of these instances (number of items, optimal solution) are summarized in Table 4.1.

Table 4.1: Summary of Knapsack Instances used

	Parameters	
Instance	Items (n)	Best Solution
KP1	100	39087
KP2	1000	333460
KP3	3000	1011522

4.1.2 Multidimensional Knapsack Problem

The Multidimensional Knapsack Problem (MKP) is a combinatorial optimization problem, with a wide range of applications, such as cargo loading, cutting stock problems, resource allocation in computer systems, and economics. It can be described as follows: given two sets of n items and m constraints, where each item j has an associated profit $p_j, j = 1..n$, and a set of weights w_{ij} (each weight is linked to a specific constraint $i = 1...m$), the goal is to find a subset of items that maximizes the profit, without exceeding the given constraint capacities C_i. Formally:

$$\max \sum_{j=1}^{n} x_j p_j \tag{4.6}$$

subject to

$$\sum_{j=1}^{n} w_{ij} x_j \leq C_i, \quad i = 1, ..., m \tag{4.7}$$

where

$$x_j = \begin{cases} 1 & \text{if the item } j \text{ is selected} \\ 0 & \text{otherwise} \end{cases} \tag{4.8}$$

The problem described in Section 4.1.1 is a special case of the MKP, when $m = 1$. The issue of invalid solutions is also present. To deal with this we follow the approach of the previous section and penalize invalid solutions. Following the recommendations of [Gottlieb, 2001], the fitness function is:

$$\max \sum_{j=1}^{n} x_j p_j - Penalty(x) \tag{4.9}$$

where

$$Penalty(x) = \frac{p_{max} + 1}{w_{min}} * \max\{CV(x,i) \quad | \quad i = 1...m\} \tag{4.10}$$

$$p_{max} = \max\{p_i \quad | \quad i = 1...m\} \tag{4.11}$$

$$w_{min} = \min\{w_{ij} \quad | \quad i = 1...m, j = 1...n\} \tag{4.12}$$

$$CV(x,i) = \max(0, \sum_{j=1}^{n} w_{ij}x_j - C_i) \tag{4.13}$$

where p_{max} is the largest profit available (Eq. 4.11), w_{min} is the minimum resource consumption (Eq. 4.12) and CV is the maximum constraint violation for the ith constraint C_i. Note that $w_{min} \neq 0$ should be ensured, i.e., the resources have to consume something.

In the experiments were the MKP was used, one instance with $n = 250$ items and $m = 5$ constraints was selected from the *OR-Library*[1].

4.2 Duration

HH incurs a learning process with a high computational effort, as the quality of each generated strategy must be estimated by applying it to an optimization situation. This leads to the appearance of two contradictory forces that must be balanced. On the one hand, the off-line learning should not take too long, which implies relying on small instances and adopting parameters (e.g., population size, number of iterations) that minimize the computational overhead. On the other hand, the adoption of excessively simple conditions might compromise results.

The experiments described in this section aim to contribute to a better understanding of the impact that the number of generations and population size have in the learning phase, and ultimately in the structure and quality of the algorithms. Our aim is to identify useful guidelines that help to define better HH.

4.2.1 Grammar

This section defines a grammar whose words are EAs, i.e., the grammar must allow the generation of a complete algorithm, defining both its main components and its settings.

[1] http://people.brunel.ac.uk/ mastjjb/jeb/orlib/mknapinfo.html

The grammar is presented in Grammar 4.1, where $<$ *start* $>$ plays the role of the axiom. The grammar enforces a sequential construction of components, thereby modeling the overall structure of the evolved algorithms. The grammar describes algorithmic components, such as selection, variation operators and replacement strategy; likewise, it also specifies parameters including the number of individuals in the initial population, the number of offspring created at each generation and the probability of applying variation operators. An inspection of the right hand side of the first production clarifies how we constrain the general structure of the evolved solutions: first, parameters *mu* and *lambda* are specified, corresponding to the number of individuals in the initial population and number of offspring created at each generation, respectively. Both values are defined in proportion to the maximum population size granted to the algorithm. Afterwards, a cycle iterates over a predetermined sequence of typical EA operations: evaluation, selection, variation and replacement. The grammar defines alternatives for each component, thus allowing the GE to learn the most suitable combination for a given situation. It is worth noting that the grammar allows the generation of solutions without some components (option ϵ denotes the empty string). In addition, several recombination and mutation operators can simultaneously appear in the same EA. Two examples of this grammar words, i.e., EAs, are depicted in Algs. 8 and 9.

Algorithm 8 Example of an Evolutionary Algorithm evolved by Grammar 4.1

```
1: mu = 1
2: lambda = 1
3: while not termination condition do
4:     evaluate
5:     RouleteWheel()
6:     SinglePointCrossover(0.9)
7:     PointMutation(0.05)
8:     Elitist(0.01)
9: end while
```

4.2.2 Learning

The settings of the GE HH are presented in Table 4.2. To estimate the quality of evolved strategies: i) one single instance of moderate size is used to assign fitness; ii) only one run is performed; iii) the number of evaluations is kept low. To investigate how these

$N = \{$*start*, *proportion*, *selection*, *integer* $-$ *const*,
variation, *operator*, *recombination*, *mutation*, *prob* $-$ *recombination*, *prob* $-$ *mutation*,
replacement, *random* $-$ *per*$\}$

$T = \{$0.5, 0.7, 0.9, 1.0, 2.0, 5.0, 10.0, *SinglePointXover*, *NPointXover*, *UniformXover*,
PointMutation, *BinarySwapMutation*, *Generational*, *RankReplacement*, *RankReplacementNoDup*,
lambda, *mu*, *evaluate*, *integer* $-$ *const*, *random*01, *Elitist*
RouleteWheel, *SUS*, *RankBased*, *Tournament*, $(,), \epsilon, /, n\}$

$S = \{$*start*$\}$

And the production set P is:

⟨*start*⟩ ::= *mu* = ⟨*proportion*⟩
lambda = ⟨*proportion*⟩
while (not termination condition) **do**
evaluate
⟨*selection*⟩
⟨*variation*⟩
⟨*replacement*⟩
end while

⟨*proportion*⟩ ::= 0.25 | 0.5 | 0.75 | 1.0

⟨*selection*⟩ ::= RouleteWheel() | SUS() | Rank() | Tournament(⟨*integer-const*⟩) | ϵ

⟨*integer-const*⟩ ::= randominteger()

⟨*variation*⟩ ::= ⟨*operator*⟩ | ⟨*operator*⟩ ⟨*variation*⟩

⟨*operator*⟩ ::= ⟨*recombination*⟩ | ⟨*mutation*⟩

⟨*recombination*⟩ ::= SinglePointXover(⟨*prob-recombination*⟩)
| NPointXover(⟨*prob-recombination*⟩,⟨*integer-const*⟩)
| UniformXover(⟨*prob-recombination*⟩)
| ϵ

⟨*mutation*⟩ ::= PointMutation(⟨*prob-mutation*⟩) | BinarySwapMutation(⟨*prob-mutation*⟩)
| ϵ

⟨*prob-recombination*⟩ ::= 0.5 | 0.7 | 0.9 | 1.0 | ⟨*random-per*⟩

⟨*prob-mutation*⟩ ::= 1.0 / n | 2.0 / n | 5.0 / n | 10.0 / n | ⟨*random-per*⟩

⟨*random-per*⟩ ::= random01()

⟨*replacement*⟩ ::= Generational() | RankReplacement() | RankReplacementNoDup()
| Elitist(⟨*random-per*⟩)
| ϵ

Grammar 4.1: Grammar used to Evolve EAs for Knapsack Problems

Algorithm 9 Example of an Evolution Strategy evolved by Grammar 4.1

```
1: mu = 1
2: lambda = 0.25
3: while not termination condition do
4:     evaluate
5:     RouleteWheel()
6:     PointMutation(0.8)
7:     Generational()
8: end while
```

Table 4.2: Parameters of the GE Framework

Parameter	Value
One Point Crossover Probability	0.9
Bit Flip Mutation	0.01
Codon Duplication Probability	0.01
Codon Pruning Probability	0.01
Population Size	100
Selection	Tournament with size equal 3
Replacement	Steady State
Codon Size	8
Number of Wraps	3
Generations	50
Runs	30

design options impact the quality and structure of the solutions learned by the GE, we present the learning results obtained with different conditions. In concrete, we focus on the length of the run used to assign fitness and also on how the number of evaluations is split between generations and population size. We consider four settings, detailed in Table 4.3. The settings aim to understand how the learning is affected by the following conditions: a small population evolved for a small number of generations; a small population evolved for a large number of generations; a large population evolved for a small number of generations; a large population evolved for a large number of generations.

Table 4.3: Evaluation Learning Settings

Setting	**Population size (M)**	**Number of Generations (K)**	**Evaluations**
1	20	100	2000
2	20	250	5000
3	50	100	5000
4	50	250	12500

Table 4.4: Hyper Heuristic Framework Learning Results

Setting	Mean Best Fitness (MBF)	Best Hits
1	0.0047 ($\pm$**0.0017**)	0 / 30
2	0.0002 ($\pm$**0.0004**)	22 / 30
3	0.0003 ($\pm$**0.0003**)	20 / 30
4	0.0000 ($\pm$**0.0000**)	30 / 30

Results

The KP instance selected to evaluate the EAs that are generated by the GE is the KP1 (first instance of Table 4.1). All the results are presented in terms of the relative error:

$$error = \frac{|ObtainedFitness - BestSolution|}{BestSolution} \tag{4.14}$$

Table 4.4 summarizes the results of the off-line learning step. For each setting, column MBF displays the mean of the best strategies found in the 30 runs performed by the GE and the corresponding standard deviation (in brackets). Last column (Best Hits) presents the number of runs where the GE evolved strategies that were able to discover the optimal solution of the selected instance. The chart from Fig. 4.1 displays the evolution of MBF, for all settings, along the 50 GE generations. These results show that the framework gradually learns better optimization strategies. Moreover, the outcomes in the Table 4.4 reveal that evolved EAs are able to discover the optimum or near-optimum solutions. Results obtained with setting 1 are an exception to this general rule. The low number of evaluations granted to each EA to solve the KP instance (between 16% and 40% of the computational budget granted by the other settings) prevents the discovery of the highest quality solutions. Note that this does not necessarily imply that the GE with setting 1 is unable to find good optimization strategies. Learning is also occurring with setting 1 (see the corresponding line in Fig. 4.1) and the optimization ability of the best

strategies evolved in this scenario will be accessed in Section 4.2.3.

A detailed inspection reveals that different learning conditions (as defined by the 4 settings previously described) impact the structure of the evolved algorithms. For all settings, we selected the best EA learned in each run and created charts that measure the frequency of appearance of the main components (the numerical settings are not considered in this analysis). Fig. 4.2 contains 3 panels that group the components by type: panels a), b) and c) display selection options, replacement options and variation operators, respectively. Unlike selection and replacement, components in panel c) are not exclusive. Virtually all best learned strategies rely on RankReplacementNoDup, a replacement mechanism based on the rank of the solutions with elimination of duplicates. This is true for all settings and is in accordance with the literature that states that this mechanism outperforms all other considered replacement components [Raidl and Gottlieb, 1999]. In what concerns selection, there is not a clear winner, although roulette wheel and rank mechanisms are slightly prevalent.

Interesting patterns arise in the selection of variation operators. For all settings, uniform crossover and binary swap mutation achieve the highest percentage, suggesting a clear advantage over the other alternatives when exploring the search space of the KP instance selected for learning. However, a close inspection of panel c) reveals that as the number of generations increases, the frequency of appearance of binary swap mutation tends to decrease, whereas the frequency of the point mutation increases. This may be explained by the fact that as the number of generations increases, stagnation at local optima may become a problem to the EA. Since the binary swap mutation does not allow a full exploration of the search space (i.e., it cannot remove nor add new items to the knapsack), point mutation starts to be more useful, since it can remove or add new items, and thus escape local optima.

Additionally, panel c) reveals a remarkable difference between settings 2 and 3. In spite of both having the same computational budget (5000 evaluations to estimate the quality of each EA), the mutation operator is often disregarded in the best strategies learned with setting 3. This can be explained by examining how the computational budget is allocated. In setting 2, a low population size (20 individuals) is iterated for a considerably high number of generations (250), which might lead to premature convergence. In these conditions the mutation operator plays a crucial role in diversity maintenance, thus avoiding premature convergence. On the contrary, setting 3 has a higher population

size (50 individuals) coupled with a lower number of generations (100). Convergence is hardly a problem and, given the moderate size of the KP instance, the EA can rely just on uniform crossover to perform an appropriate sampling of the search space. Therefore, it is not surprising that many EAs learned with setting 3 prefer to not include mutation.

To confirm this hypothesis, we ran an additional set of experiments, with a slightly modified grammar, that only allows the appearance of a single variation operator in each evolved structure. This way, the framework has to select the most suitable operator (either crossover or mutation) to include in the optimization strategy, given specific learning conditions. We repeated the experiments for all four settings presented in Table 4.3. Fig. 4.3 displays the frequency of appearance of the variation operators using the modified grammar. An overview of the chart confirms our claims. Strategies evaluated with low population size (settings 1 and 2) need to incorporate mutation operators to prevent premature convergence. When the population size is high and the number of generations is low (setting 3), all the EA needs is crossover. Finally, setting 4 grants the EA a considerable computational budget to run, hence it is natural that strategies relying only on mutation appear. It is worth notice that, in the experiments performed with the modified grammar, binary swap mutation is completely absent from the best learned solutions. This is again related to the fact that this operator alone cannot modify the number of items in the knapsack, thereby preventing a full exploration of the search space.

Results presented in Figs. 4.2 and 4.3 reveal that different EA architectures emerge when different learning conditions are adopted. This a relevant contribution that sheds light on the impact of learning conditions adopted by a hyper-heuristics framework. To complete this section, we present the best evolved algorithm for each one of the settings, labelled according to the scenario where they were discovered in Algorithms EAL1, EAL2, EAL3 and EAL4. In agreement with the previous analysis, they share the same replacement method and tend to rely on different selection strategies. In what concerns the variation operators, EAL1, EAL2 and EAL4 include both crossover and mutation, whereas EAL3 relies solely on crossover.

4.2.3 Validation

We present now a set of experiments to analyze how the 4 best evolved algorithms (EAL1, EAL2, EAL3, EAL4) behave in KP instances that are different from the one used in

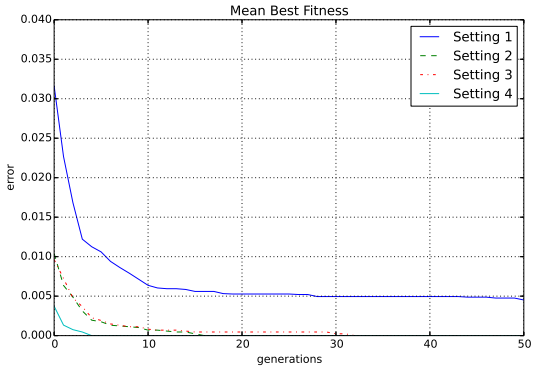
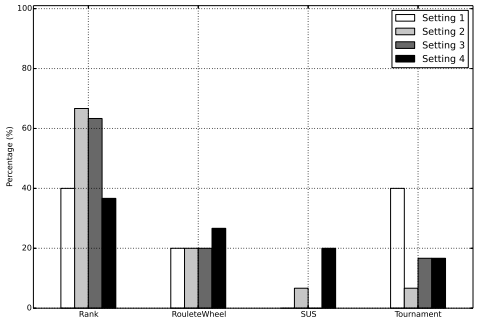
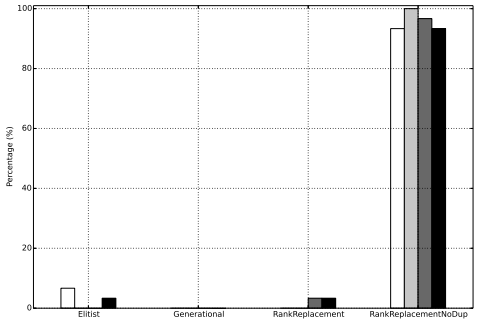


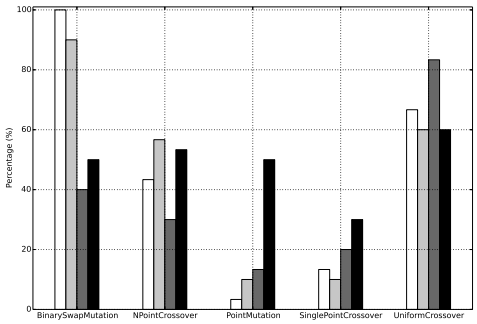
Figure 4.1: Evolution of the Learning Mean Best Fitness across the 4 Learning Settings



a) Selection Components



b) Replacement Components



c) Variation Operators

Figure 4.2: Frequency of components in the best evolved solutions: panels a), b) and c) display selection, replacement and variation operators, respectively.

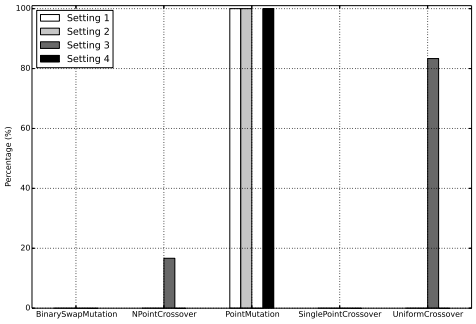


Figure 4.3: Frequency of components in the best evolved solutions with just one variation operator.

EAL1: Strategy Evolved using Setting 1

```
mu = 0.5
lambda = 1.0
while not termination condition do
    evaluate
    Tournament(2)
    UniformCrossover(1.0)
    BinarySwapMutation(2.0 / n)
    RankReplacementNoDup()
end while
```

EAL2: Strategy Evolved using Setting 2

```
1: mu = 1.0
2: lambda = 1.0
3: while not termination condition do
4:     evaluate
5:     Tournament(10)
6:     [1] UniformCrossover(0.1)
7:     [1] BinarySwapMutation(2.0 / n)
8:     [1] RankReplacementNoDup()
9: end while
```

EAL3: Strategy Evolved using Setting 3

```
1: mu = 1.0
2: lambda = 1.0
3: while not termination condition do
4:     evaluate
5:     Rank()
6:     UniformCrossover(1.0)
7:     RankReplacementNoDup()
8: end while
```

EAL4: Strategy Evolved using Setting 4

```
1: mu = 1.0
2: lambda = 1.0
3: while not termination condition do
4:     evaluate
5:     RouleteWheel()
6:     UniformCrossover(0.9)
7:     PointMutation(1.0 / n)
8:     RankReplacementNoDup()
9: end while
```

learning. Such study will help to gain insight into the optimization performance differences that may eventually arise between strategies learned in different conditions. Also, we will verify if the evolved EAs generalize well to unseen instances of the KP, and are competitive with hand design approaches regularly applied to the KP. Three hand-designed algorithms (HEA1, HEA2, HEA3), were considered in this study. The HEA1 and HEA2 are suited to work on the 0-1 Knapsack problem, whilst HEA3 works well when applied to the MKP problem [Chu and Beasley, 1998], [Raidl and Gottlieb, 2005]. All hand-designed methods adopt the RankReplacementNoDup replacement mechanism and tournament selection with tourney size 3. They differ in the variation operators and/or corresponding rate of application, as detailed in Table 4.5. The EAs were applied to different KP instances, with a number of items equal to 1000 and 3000 (instances 2 and 3 of Table 4.1).

To mimic the training conditions, we created four different validation scenarios in which we varied the population size and number of generations. The settings are detailed in Table 4.6. All EAs were applied to the two remaining KP instances (KP2, KP3) with each one of these settings. In every optimization scenario, 30 runs were performed and the

Table 4.5: Hand-designed EAs: Variation operators and rate of application.

ID	Variation Operator	Rate
HEA1	SinglePointCrossover	0.9
	PointMutation	1 / n
HEA2	UniformCrossover	0.9
	PointMutation	1 / n
HEA3	UniformCrossover	0.9
	BinarySwapMutation	1 / n

Table 4.6: Validation settings.

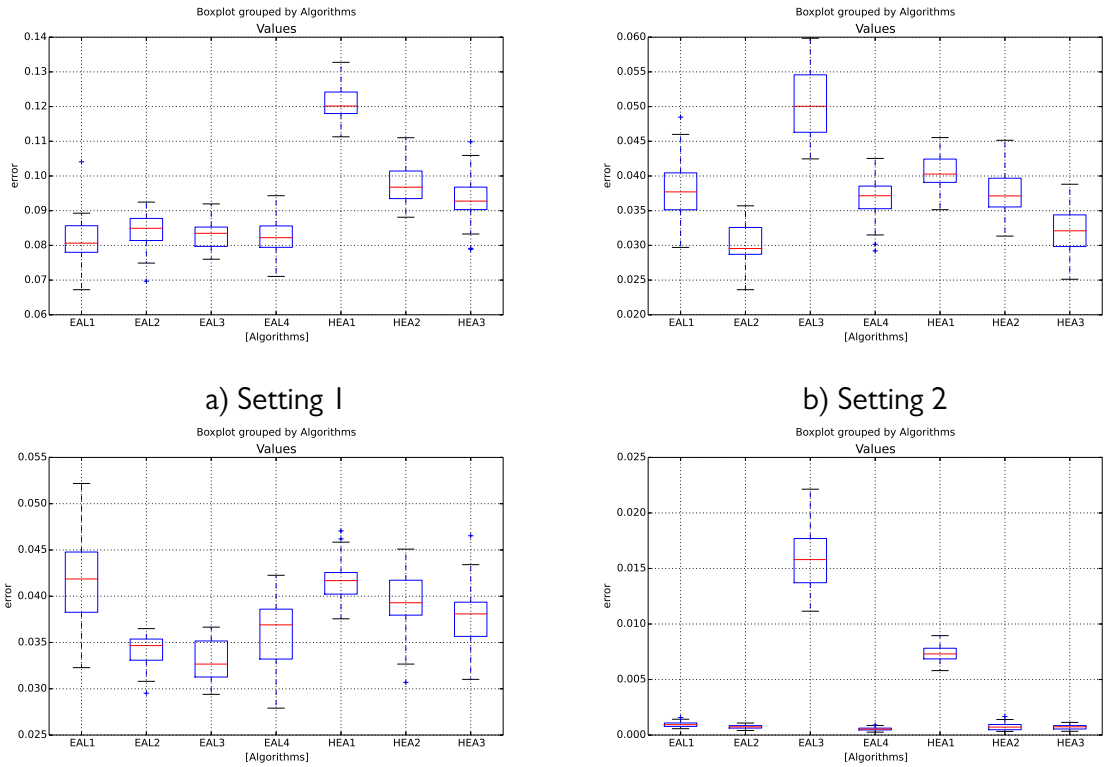
Setting	Population size	Number of Generations	Evaluations
1	50	100	5000
2	50	400	20000
3	100	200	20000
4	100	5000	500000

best solution found was recorded.

Figs. 4.4 and 4.5, contain the optimization results of the 7 EAs (both evolved and hand-designed), obtained in each one of the validation instances. Four panels, corresponding to each one of the validation scenarios, are displayed.

A brief perusal of the figures reveal that, in general, the learned EAs perform well across the different scenarios (as determined by the combination of a given instance and setting), suggesting that they are able to generalize, beyond the specific situation used for learning. Results also show that the effectiveness of the learned EAs is comparable to the hand-designed approaches, confirming that the GE framework was able to learn meaningful combinations of components and settings. Additionally, and even though differences are not always statistically significant, it is possible to say that EAL1 tend to achieve the best MBF in setting 1, EAL2 is the best method in setting 2, EAL3 is good for setting 3, and EAL4 is the best in setting 4. This confirms that evolved strategies contain specific features that allow them to excel in situations similar to those found during learning. Despite the good general behavior of the evolved strategies, there are some differences in performance that require a detailed analysis. The results from the Friedman's ANOVA test indicate that there are significant differences amongst the strategies in all the settings (Table 4.7).

Tables 4.8 and 4.9 presents the pairwise comparisons. The results show that the



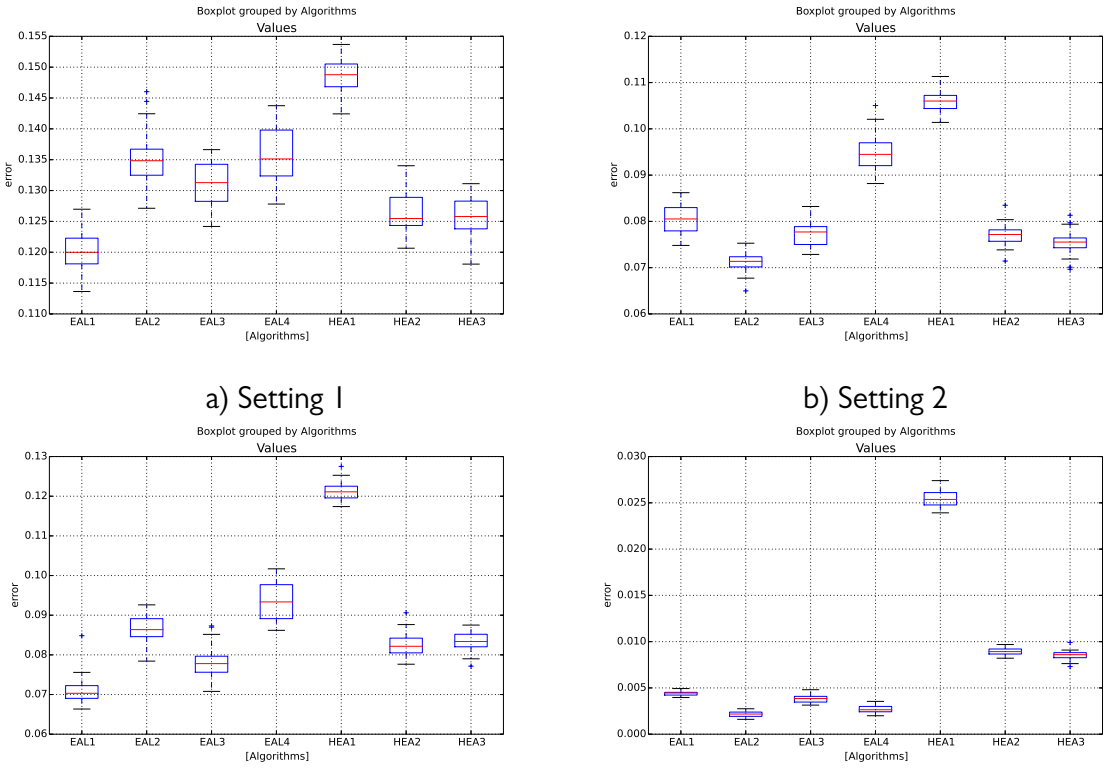
a) Setting 1 b) Setting 2

c) Setting 3 d) Setting 4

Figure 4.4: Box plot distribution of the MBF obtained by different EAs in the validation instance KP2. Each panel corresponds to a given scenario as described in Table 4.6.

Table 4.7: Friedman's ANOVA statistical test results

	Instance			
	*KP*2		*KP*3	
Setting	$\chi^2(6)$	p-value	$\chi^2(6)$	p-value
1	108.4	$< 2.2 * 10^{-16}$	144.0	$< 2.2 * 10^{-16}$
2	153.6	$< 2.2 * 10^{-16}$	155.9	$< 2.2 * 10^{-16}$
3	155.1	$< 2.2 * 10^{-16}$	154.6	$< 2.2 * 10^{-16}$
4	145.9	$< 2.2 * 10^{-16}$	173.4	$< 2.2 * 10^{-16}$



a) Setting 1

b) Setting 2

c) Setting 3

d) Setting 4

Figure 4.5: Box plot distribution of the MBF obtained by different EAs in the validation instance with size KP3. Each panel corresponds to a given scenario as described in Table 4.6.

Table 4.8: Statistical analysis between the learned architectures using the Wilcoxon Signed Rank Test for the KP2 instance

	Settings	**Alg. EAL1**	**Alg. EAL2**	**Alg. EAL3**	**Alg. EAL4**	**Alg. HEA1**	**Alg. HEA2**	**Alg. HEA3**
Alg. EAL1	1		∼	∼	∼	+++	+++	+++
	2		-	+++	∼	∼	∼	-
	3		-	-	-	∼	∼	∼
	4		-	+++	-	+++	-	-
Alg. EAL2	1	∼		∼	∼	+++	+++	+++
	2	+++		+++	+++	+++	+++	+++
	3	+++		-	++	+++	+++	+++
	4	+++		+++	-	+++	∼	∼
Alg. EAL3	1	∼	∼		∼	+++	+++	+++
	2	-	-		-	-	-	-
	3	+++	++		+++	+++	+++	+++
	4	-	-		-	-	-	-
Alg. EAL4	1	∼	∼	∼		+++	∼	-
	2	∼	-	+++		+++	-	-
	3	+++	-	-		+++	+++	∼
	4	+++	+++	+++		+++	++	+++

Table 4.9: Statistical analysis between the learned architectures using the Wilcoxon Signed Rank Test for the KP3 instance

	Settings	**Alg. EAL1**	**Alg. EAL2**	**Alg. EAL3**	**Alg. EAL4**	**Alg. HEA1**	**Alg. HEA2**	**Alg. HEA3**
Alg. EAL1	1		+++	+++	+++	+++	+++	+++
	2		-	-	+++	+++	-	-
	3		+++	+++	+++	+++	+++	+++
	4		-	-	-	+++	+++	+++
Alg. EAL2	1	-		-	∼	+++	-	-
	2	+++		+++	+++	+++	+++	+++
	3	-		-	+++	+++	-	-
	4	+++		+++	+++	+++	+++	+++
Alg. EAL3	1	-	+++		+++	+++	-	-
	2	+++	-		+++	+++	∼	∼
	3	-	+++		+++	+++	+++	+++
	4	+++	-		-	+++	+++	+++
Alg. EAL4	1	-	∼	-		++	-	-
	2	-	-	-		+++	-	-
	3	-	-	-		+++	-	-
	4	+++	-	+++		+++	+++	+++

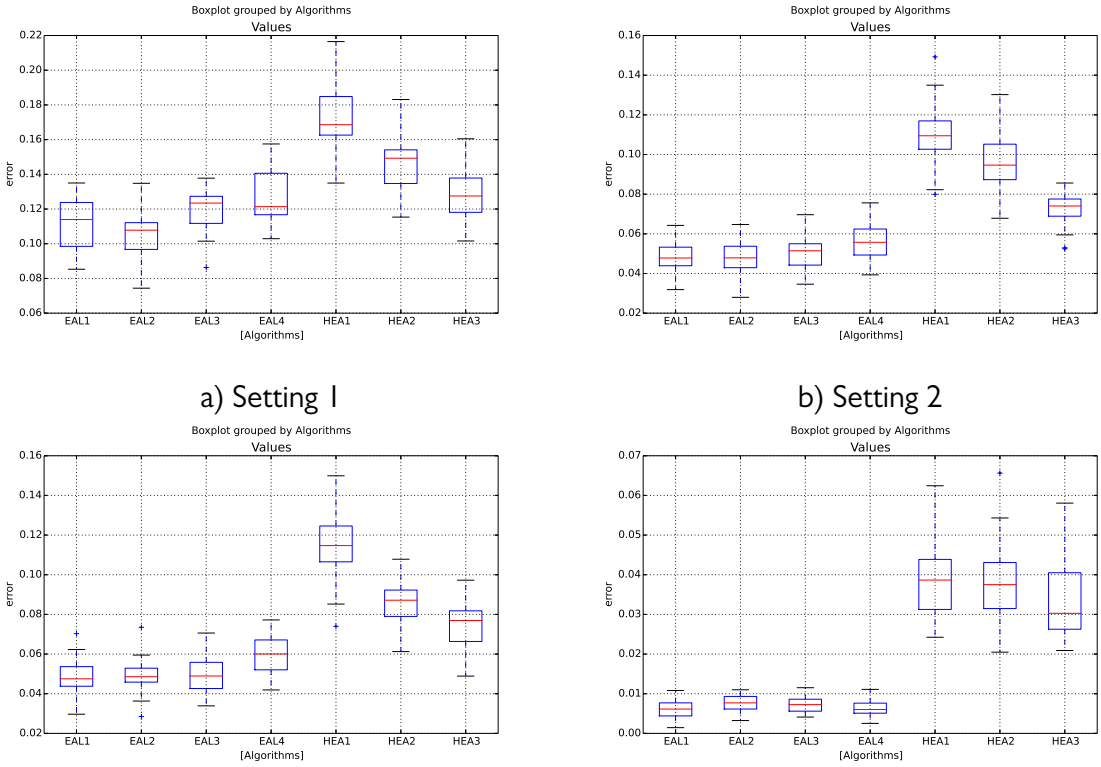
evolved algorithms maintain a reasonable effectiveness in unseen conditions. Specifically EAL2 and EAL4 seem to have an increased robustness, which allows them to adapt to different optimization scenarios. These outcomes allow us to conclude that the number of generations used to assign fitness in the off-line learning step is more important than the population size. The evolved strategies must be executed for a reasonable number of generations, in order to obtain an accurate estimate of their optimization ability. Finally it is worth noting that learning setting 2 is able to evolve effective and robust algorithms, even though it only needs a modest computational budget (40% of the computational budget to evaluate candidates when compared to setting 4).

Additionally, and comparing the evolved algorithms with the hand-design, it is possible to see that, in general, the evolved ones tend to outperform the hand-design. This happens in both the instances used for validation. Considering that larger instances are harder to solve, our framework is able to evolve strategies that are effective in finding good quality solutions. These results confirm the benefits of using a framework like the one presented in this work to learn algorithms to solve a specific task. It is important to say that the hand-design algorithms were implemented as they were presented in the literature. This means that we did not use any tuning mechanism in order to adjust the parameters for the instances used in our experiments.

To verify how the evolved algorithms worked in an different problem from the one used in learning, we applied the same algorithms (the 4 evolved, and the 3 hand-designed) to the MKP problem. The MKP problem is more difficult than the traditional KP, due to the number of constraints that have to be satisfied. This means that even a small instance can be hard to solve.

The results show that the learned strategies have a better performance on every setting, as can be seen in Fig. 4.6. The strategy EAL2 is exceptionally effective as it seems to always perform significantly better than all the hand-design approach. Moreover, it seems that algorithms with a low selective pressure, and with a slightly large mutation rate, work better on this problem.

To corroborate our analysis, we performed a statistical analysis. The results of Table 4.10 confirm that there are statistical differences between the algorithms. In Table 4.11 we present the results for the pairwise comparisons between the learned and the hand design algorithms. It confirms that EAL2 is very effective, as it is always better than the hand design. Although they are not as good as EAL2 the other evolved approaches (EAL1,



a) Setting 1 b) Setting 2

c) Setting 3 d) Setting 4

Figure 4.6: Box plot distribution of the MBF obtained by different EAs in the MKP. Each panel corresponds to a given scenario as described in Table 4.6.

EAL2, EAL3) have a very good performance.

These results confirm the advantages of adopting a HH framework like the one described in the previous chapter, since it can learn algorithms that can provide good results in a reasonable number of problems.

4.3 Evolution of Selection Strategies

In this section, we will look how different design options influence the learning of a specific component. In concrete, the framework will be used to evolve selection strategies. Our analysis will be focused on the capacity of the framework to evolve strategies with different selective pressures. The selective pressure is the driving force that guides the individuals to evolve in a certain direction. It establishes the individuals that will create

Table 4.10: Friedman's ANOVA statistical test results for the MKP

	MKP	
Setting	$\chi^2(6)$	p-value
1	112.0	$< 2.2 * 10^{-16}$
2	142.4.6	$< 2.2 * 10^{-16}$
3	136.7	$< 2.2 * 10^{-16}$
4	137.6	$< 2.2 * 10^{-16}$

Table 4.11: Statistical analysis using the Wilcoxon Signed Rank Test for the MKP.

	Settings	**Alg. HEA1**	**Alg. HEA2**	**Alg. HEA3**
Alg. EAL1	1	+	+++	+++
	2	+++	+++	+++
	3	+++	+++	+++
	4	+++	+++	+++
Alg. EAL2	1	+++	+++	+++
	2	+++	+++	+++
	3	+++	+++	+++
	4	+++	+++	+++
Alg. EAL3	1	++	+++	+++
	2	+++	+	+++
	3	+++	+++	+++
	4	+++	+++	+++
Alg. EAL4	1	+++	+++	$\sim$
	2	+++	++	+++
	3	++	+++	+++
	4	+++	+++	+++

offspring for the next generation. High selective pressure means that the best individuals have more chances of being selected for reproduction.

4.3.1 Grammar

For these experiments the grammar words are selection strategies. The grammar encodes modifications that aim to overcome some limitations that the BNF imposes, namely the lack of tools to allow non-terminal symbols repetition and ranges of alternative values. The first extension is the addition of the operator $\sim$ to signal the repetition of non-terminals. The full syntax is as follows: $\sim< a >< NT >$, where $< a >$ is an integer or terminal value, indicating that the non-terminal $< NT >$ should be repeated $< a >$ times. The second extension is the addition of valued range alternatives. A range of numeric alternative values can be compactly specified, using the operator $\&$. Thus $<$ *int* $>::= \mathbf{0}\&\mathbf{5}$ is equivalent to $<$ *int* $>::= \mathbf{0}|\mathbf{1}|\mathbf{2}|\mathbf{3}|\mathbf{4}|\mathbf{5}$. Taking these extensions into account, the grammar used in this work is described in Grammar 4.2. Assuming that POP_SIZE is equal to 50, an example of the grammar words is presented in Alg. 14. The POP_SIZE value was selected based on previous the analysis.

$$
\begin{aligned}
N &= \{\textit{start}, \textit{proportion}, \textit{selectionStrategy}, \textit{elements}, \textit{rank}, \textit{calculateParents}\} \\
T &= \{\mathbf{0}, \textit{POP\_SIZE}, \textit{parents}, =, \textit{numberOfParents}, \textit{random}\mathbf{01}, (,), *\} \\
S &= \{\textit{start}\}
\end{aligned}
$$

And the production set P is:

$\langle start \rangle$ ::= $\langle calculateParents \rangle$ $\langle selectionStrategy \rangle$
$\langle selectionStrategy \rangle$ ::= parents = { $\sim$ numberOfParents$\langle elements \rangle$ }
$\langle elements \rangle$ ::= getrank($\langle rank \rangle$)
$\langle rank \rangle$ ::= 0 & POP_SIZE
$\langle calculateParents \rangle$ ::= numberOfParents = (random01() * POP_SIZE)

Grammar 4.2: Grammar used to evolve selection strategies for Evolutionary Algorithms

The $<$ *start* $>$ symbol represents the grammar axiom. The grammar starts by calculating the number of parents that the strategy should select (*numberOfParents*), according to a percentage of the total individuals available (*POP_SIZE*). Since evolved strategies are

targeted for EAs with crossover, we enforce an even number of parents in the selection pool. Afterwards, a selection strategy to choose which individuals will appear in the selection pool is generated. The solutions from the current population are ranked by fitness and a selection strategy emerges by defining which ranks should be chosen as parents.

Algorithm 14 Example of a Selection Strategy generated by the Grammar 4.2

```
1: numberOfParents = (0.2 * 50)
2: parents = getrank(35), getrank(3), getrank(45), getrank(2), getrank(43), getrank(41), getrank(1),getrank(36),getrank(9),getrank(8)
```

4.3.2 Learning

The settings adopted by the GE for all the tests conducted are depicted in Table 4.12. To estimate the relevance of the selection strategy being evolved, one must access how they help an EA to solve a given problem. Therefore, each individual is implanted in a standard EA, which in turn will solve an instance of the KP problem.

We report experiments using three different EA settings as surrogates for the selection strategies. In all of them, the maximum population size (*POP_SIZE*) is set to 50 and the number of generations is set to 250. Three possible replacement strategies, *R1*, *R2*, and *R3*, are considered (see Table 4.13). *R1* corresponds to a standard generational EA, whereas the last two implement a steady-state architecture where descendants compete with existing individuals for survival based on the fitness criterion. Both *R1* and *R2* force the evolved selection strategies to select a number of parents that is equal to *POP_SIZE*, thus the grammar production $<$ *calculateParents* $>$ simply becomes $<$ *calculateParents* $>::=$ *numberOfParents* $=$ *POP_SIZE*. On the contrary, *R3* allows the selection strategy to choose a number of parents that is lower than *POP_SIZE*. All three replacement strategies consider uniform crossover with a rate of 0.9 and swap mutation with rate 0.01 as variation operators.

Results

The KP instance selected to evaluate learned EAs is the KP1 (Table 4.1). Once again all the results are presented in terms of the relative error (Eq 4.14). Table 4.14 summarizes the results of the off-line learning process. Every cell contains two values: the number of

Table 4.12: Parameter setting for the GE-based Hyper-Heuristic

Parameter	**Value**
One Point Crossover Probability	0.9
Bit Flip Mutation	0.01
Codon Duplication Probability	0.01
Codon Pruning Probability	0.01
Population Size	100
Selection	Tournament with size equal 3
Replacement	Steady State
Codon Size	8
Number of Wraps	3
Generations	50
Runs	30

Table 4.13: Replacement strategies used in the surrogate EAs. The column *Fixed* indicates whether the number of selected parents is fixed or not.

Setting	**Fixed**	**Replacement Strategy**
R1	Yes	Generational
R2		Steady State
R3	No	Steady State

Table 4.14: Selection Strategies learning results

	Replacement strategies		
	R1	**R2**	**R3**
Best Hits	30	30	30
MBF	0.000 ($\pm$**0.000**)	0.000 ($\pm$**0.000**)	0.000 ($\pm$**0.000**)

GE runs that discovered selection strategies that helped the EA to discover the optimum (*BestHits*) and the Mean Best Fitness (*MBF*) together with the corresponding standard deviation. The outcomes reveal that, for all training situations, the HH is able to learn selection strategies that help the surrogate EA to effectively solve the problem at hand.

Looking at the evolved strategies, it is possible to see that the different replacement strategies used in the surrogate EA lead to the appearance of selection methods with different selective pressure. The three lines from Fig. 4.7 (one from each replacement strategy) help to clarify this issue. For every setting we selected the best selection algorithm evolved in each run and created charts displaying the distribution of the appearance of the possible ranks (values displayed are averages of 30 runs). Note that rank 0 corresponds to the best individual and rank 49 to the worst. An inspection of the figure shows that selection strategies evolved inside a generational surrogate (*R1*) have a higher selective pressure than those that evolved in the steady state surrogates. In generational EAs, the whole population is replaced at each generation. The HH acknowledges the risk of losing good quality solutions and promotes the appearance of selection strategies with a high selective pressure, thereby maximizing the likelihood of passing information contained in good quality solutions to the next generations. On the other hand, in steady state surrogate EAs, the ranks are distributed more or less evenly. This results is not unexpected, since in this scenario, the greedy replacement mechanism already ensures selective pressure: an offspring only enters the population if it is better than its parents. Therefore the selection strategy in these EAs can act more like a diversity preservation mechanism. Finally, in Fig. 4.8 we exemplify the rank distribution of one of the best evolved strategies, using the *R1* setting.

4.3.3 Validation

The experiments described in this section aim to study how the best strategies discovered by the GE-based HH behave in KP instances that are different from the one used in

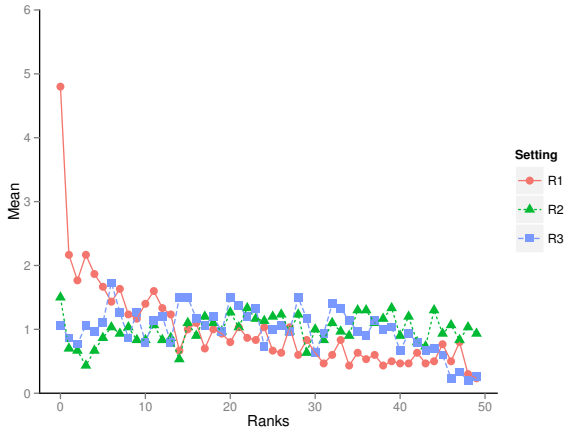


Figure 4.7: Rank distribution in the best evolved strategies with the three replacement settings.

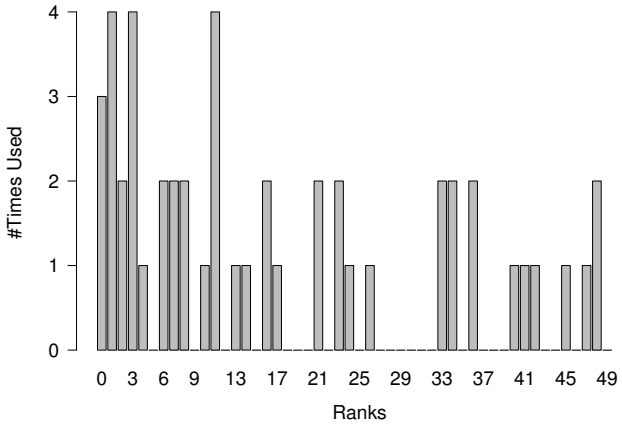


Figure 4.8: Example of the rank distribution of a selection strategy evolved with the *R1* setting.

learning. We selected three evolved strategies from each possible replacement strategy. In the remainder of this section these selection strategies are identified as *EAL*1 for the methods evolved with the *R*1 replacement strategy, *EAL*2 for *R*2 replacement strategy, and *EAL*3 for *R*3 replacement strategy. This experimental study will help to gain insight into the optimization performance of EAs that have the learned strategies as selection methods. Also, we will verify if the strategies generalize well to unseen instances and are competitive with standard hand-designed selection strategies. Three common selection options (Roulette Wheel, Tournament with size 2, and Tournament with size 3) are considered. We report results obtained with a generational and a steady-state surrogate EAs, both of them relying on uniform crossover with rate 0.9 and binary swap mutation with rate $1/n$ as variation operators, and with a population size of 50 individuals running for 5000 generations. The selection strategies were validated using different KP instances, with a number of items equal to 1000 and 3000 (Table 4.1). Figs. 4.9, 4.10 present the MBF box plot distribution of the 6 selected strategies (3 evolved and 3 hand-designed) for each validation scenario: Panel a) displays the results for the generational surrogate, whereas panel b) presents the results for the steady-state surrogate. Clearly, the performance of the evolved strategies is related to the configuration where they are applied. Strategies *EAL*1 was evolved with a generational EA surrogate and, as a consequence, they promote a considerable selection pressure. Therefore it is not a surprise that this strategy achives good results in a validation scenario where a generational surrogate is adopted (see Panel a) from Figs. 4.9, 4.10). On the contrary, strategies *EAL*2 and *EAL*3 have a low selective pressure and are inadequate for a generational EA environment.

An opposite situation arises in the steady-state validation surrogate (Panel b) from Figs. 4.9, 4.10). In this scenario, and given the fitness-based replacement strategy adopted, selection methods evolved in a generational environment tend to converge prematurely to sub-optimal regions of the search space. The remaining 2 evolved strategies were obtained in a scenario similar to the one used in this validation phase. For that reason, they contain features that help to maintain diversity and to effectively help the EA to discover the regions of the search space containing the best solutions. The distinction between these two sets of evolved selection methods confirms that the HH framework is able to generate strategies that are suited to the specific features of the training environment. To complement the results and confirm that the HH is able to evolve selection methods competitive with the hand-designed approaches, Table 4.15 presents the Fried-

Table 4.15: Friedman's ANOVA statistical test results

	Instance			
	*KP*2		*KP*3	
Setting	$\chi^2(5)$	p-value	$\chi^2(5)$	p-value
Generational	141.5	$< 2.2 * 10^{-16}$	141.8	$< 2.2 * 10^{-16}$
Steady-State	94.3	$< 2.2 * 10^{-16}$	95.8	$< 2.2 * 10^{-16}$

Table 4.16: Statistical analysis between the learned strategies and the hand-designed using the Wilcoxon Signed Rank Test for the KP2 instance

	Generational			**Steady-State**		
	Roulette Wheel	**Tournament(2)**	**Tournament(3)**	**Roulette Wheel**	**Tournament(2)**	**Tournament(3)**
EAL1	+++	++	+++	-	-	-
EAL2	-	-	-	+	$\sim$	$\sim$
EAL3	-	-	-	$\sim$	-	-

man's ANOVA test. The information present in this table show that there are significant differences between the strategies used in the comparison.

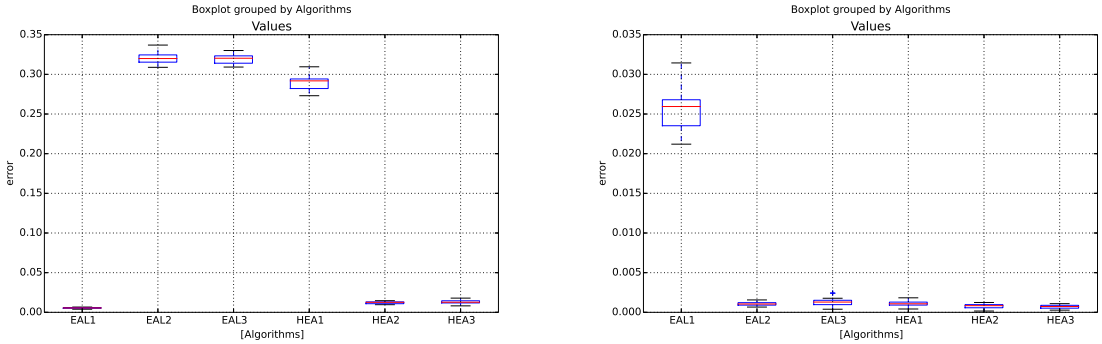
The information displayed in Tables 4.16, 4.17 help to further clarify the relative performance of learned strategies. The pairwise comparison confirms that the evolved strategies tend to perform better in situations resembling those found during learning. When the steady-state EA surrogate is adopted, it is possible to see that strategies *EAL2* and *EAL3* have a reasonable degree of effectiveness. The performance of methods evolved with the *EAL2* setting is particularly impressive, as each one of them outperforms all hand-designed selection mechanisms.

To complete our analysis we investigate if the evolved strategies generalize well to a problem different from that used in the learning step. We keep our focus on the KP class, but consider the Multiple Knapsack Problem (MKP) variant.

We maintain the 6 selection strategies adopted in the previous validation analysis and keep all other optimization conditions, including the two same surrogate EAs. Fig. 4.11 depicts the MBF box plot distribution of the selection methods, both for the generational

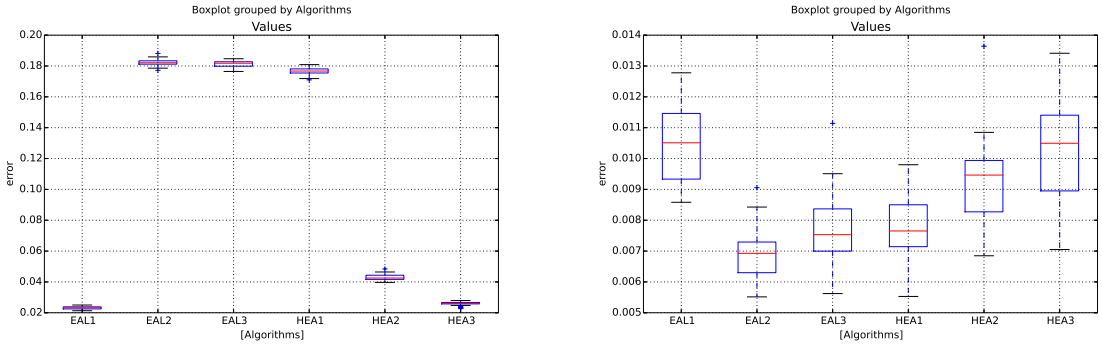
Table 4.17: Statistical analysis between the learned strategies and the hand-designed using the Wilcoxon Signed Rank Test ($\alpha = 0.05$) for the KP3 instance

	Generational			**Steady-State**		
	Roulette Wheel	**Tournament(2)**	**Tournament(3)**	**Roulette Wheel**	**Tournament(2)**	**Tournament(3)**
EAL1	+++	+++	+++	-	-	$\sim$
EAL2	-	-	-	+++	+++	+++
EAL3	-	-	-	$\sim$	+++	+++



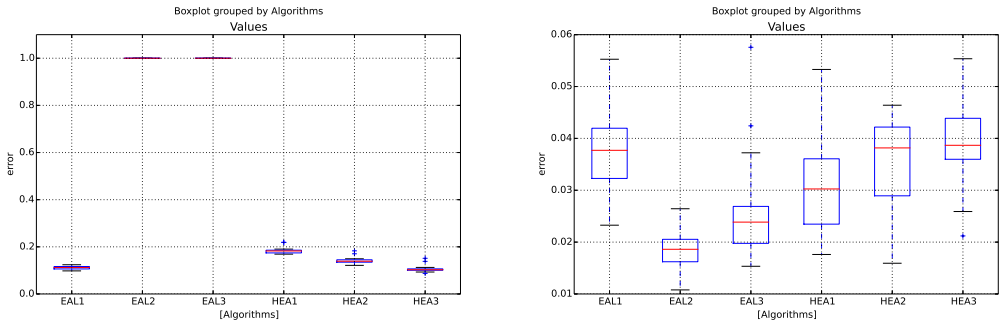
a) Generational b) Steady State

Figure 4.9: Optimization results of the 6 selection strategies chosen for the validation instance $n = 1000$: panels (a), (b), present the results obtained with the generational and steady state EAs, respectively.



a) Generational b) Steady State

Figure 4.10: Optimization results of the 6 selection strategies chosen for the validation instance $n = 3000$: panels (a), (b), present the results obtained with the generational and steady state EAs, respectively.



(a) Generational (b) Steady-State

Figure 4.11: MKP optimization results of the 6 selection strategies chosen for the generalization study: panels (a), (b), present the results obtained with the generational and steady state EAs, respectively.

Table 4.18: Statistical analysis between the learned strategies and the hand-designed using the Wilcoxon Signed Rank Test for the MKP (see text for details on the notation).

	Generational			**Steady-State**		
	Roulette Wheel	**Tournament(2)**	**Tournament(3)**	**Roulette Wheel**	**Tournament(2)**	**Tournament(3)**
EAL1	+++	+++	∼	∼	∼	∼
EAL2	-	-	-	+++	+++	+++
EAL3	-	-	-	∼	++	+++

(panel a)) and steady-state (panel b)) surrogates. In Table 4.18 we summarize the statistical comparison between the strategies considered in the generalization study. The analysis of the results reveals the exact same trend that was identified in the previous validation. Considering the performance of the evolved selection strategies, there is a clear correlation between the conditions found in the off-line learning step and those of the validation/generalization experiments. Additionally, optimization results are competitive with those achieved by hand-designed approaches: the *EAL1* method tends to outperform standard selection strategies in generational environments, whereas *EAL2* and *EAL3* excel in steady-state surrogates. These outcomes confirm that the framework was able to learn strategies that generalize well to different KP variants.

4.4 Summary

This Chapter presented a set of experiments to gain insight into the influence that learning conditions play in the effectiveness and robustness of evolved strategies, using the framework previously described.

The first part was dedicated to the analysis of different off-line learning settings, in what concerns the population size and the number of generations used to estimate the fitness of solutions evolved by the GE. The HH framework was executed in the different learning scenarios and the best evolved strategies were subsequently applied to unseen and larger KP instances. In the validation step, alternative optimization scenarios, with similar features to those adopted in the learning step, were considered. As a rule, evolved strategies obtained extremely good results in scenarios similar to those found during learning, showing that they contain features that are particularly suited for a specific environment. However, learning settings that allow the execution of a higher number of generations to estimate the fitness of solutions generated by the GE, allow the discovery of strategies with enhanced robustness. Lastly, we investigated how the evolved strategies worked in a problem different from that used in learning.

The second part of the chapter focused on the analysis of how different EA replacement methods influence the learning of rank-based selection strategies. We demonstrated the validity of the approach in the domain of different KP variants. Results obtained show that the HH framework adapts the selective pressure of the evolved mechanism, taking into account the specific features of the adopted surrogate. Moreover we showed that the framework was able to learn effective selection strategies, competitive with standard hand-designed mechanisms regularly adopted in the literature. Additionally, we devised a set of experiments to see if the evolved strategies were able to generalize to different variants of the problem considered in our study.

5

Optimization Ability of Learned Strategies

To keep the computational effort at a reasonable level, the evaluation step of an HH search engine relies on small instances and simplified fitness criteria. However, the previous chapter showed that the learning conditions impact the properties of the algorithms being evolved. Taking this into account, it is not clear if the limited evaluation conditions adopted by the HH frameworks compromise the accurate identification of the best optimization strategies, i.e., the best strategies discovered during learning might not be the ones with the best generalization abilities.

In this Chapter we address this question by investigating if the fitness criteria used in learning provide enough information to identify the most effective and robust strategies. The study is performed with the framework presented in [Tavares and Pereira, 2012]. There are two reasons to select a different framework. First it is a different example from the one used in the previous Chapter. Secondly, it has already been recognized by the scientific community, winning the award for best paper in the renowned conference EuroGP.

The computational model proposed in [Tavares and Pereira, 2012] is able to automatically generate complete Ant Colony Optimization (ACO) algorithms to tackle the traveling salesperson problem (TSP)(Section 5.1). A complete description of the grammar used is provided in Section 5.2. In Section 5.3 we provide a study to assess the optimization ability of the strategies learned by the HH.

Table 5.1: Summary of TSP instances used

Instance	Name	Number of Cities	Best Solution
1	pr76	76	108159
2	lin105	105	14379
3	pr136	136	96772
4	ts225	225	126643
5	pr226	226	80369
6	lin318	318	42029

5.1 Traveling Salesman Problem

The Traveling Salesman Problem (TSP) aims to find the shortest possible trip connecting a given set of customer cities. Note that the salesman starts and ends the trip in the same place and visits each city once. More formally, The TSP can be represented as an weighted graph $G = (N, A)$, where N is a set of nodes representing the cities, and A is a set of arcs connecting the cities. Each arc has an assigned value d_{ij}, $i, j \in N$, representing the distance between the cities i and j. The objective is to find the minimal length Hamiltonian circuit of the graph G. An Hamiltonian circuit is a closed tour, where each node is visited exactly once.

The TSP problem can be *symmetric* or *asymmetric*. In symmetric TSPs, the value of the distance d_{ij} is independent of the direction of the transversing, i.e., $d_{ij} = d_{ji}$. On the contrary, in asymmetric TSPs the distance might change with the direction of transversing, i.e., $d_{ij} \neq d_{ji}$.

In this chapter we will focus our attention on the symmetric TSP. We selected several TSP instances from the TSPLIB[1] for the experimental analysis. Table 5.1 summarizes the general properties (number of cities, optimal solution) of the used instances.

5.2 Design of Ant Algorithms

The framework originally proposed by Tavares *et al.* [Tavares and Pereira, 2012] to evolve fully-fledged ACO algorithms, will be used as the test case for our experiments. Ant Colony Optimization (ACO) algorithms are a set of population-based methods, loosely

[1] http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/

inspired by the behavior of ant foraging [Dorigo and Stützle, 2004]. Following the original Ant System (AS) algorithm proposed by Marco Dorigo in 1992, many other variants and extensions have been described in the literature. To help researchers and practitioners to select and tailor the most appropriate variant to a given problem, several automatic ACO design frameworks have been proposed in the last few years. The production set of the above mentioned framework defines the general architecture of an ACO-like algorithm, comprising an initialization step followed by an optimization cycle (See Grammar 5.1). The first stage initializes the pheromone matrix and other settings of the algorithm. The main loop consists of the building of the solutions, pheromone trail update and daemon actions. Each component contains several alternatives to implement a specific task. As an example, the decision policy adopted by the ants to build a trail can be either the random proportional rule used by AS methods or the q-selection pseudorandom proportional rule introduced by the Ant Colony System (ACS) variant. If the last option is selected, the GE engine also defines a specific value for the q-value parameter. The grammar allows the replication of all main ACO algorithms, such as AS, ACS, Elitist Ant System (EAS), Rank-based Ant System (RAS), and Max-Min Ant System (MMAS). Additionally, it can generate novel combinations of blocks and settings that define alternative ACO algorithms. Results presented in [Tavares and Pereira, 2012] show that the GE-HH framework is able to learn original ACO architectures, different from standard strategies. Moreover, results obtained in validation instances reveal that the evolved strategies generalize well and are competitive with human-designed variants (consult the aforementioned reference for a detailed analysis of the results).

5.3 Optimization Ability

Experiments described in this section aim to gain insight into the capacity of the GE-based HH to identify the most promising solutions during the learning step. In concrete, we determine the relation between the quality of strategies as estimated by the GE and their optimization ability when applied to unseen and harder scenarios. Such study will provide valuable information about the capacity of the GE to identify strategies that are robust.

In practical terms, we take all strategies belonging to the last generation of the GE and rank them by the fitness obtained in the learning evaluation instance. Since the GE relies

⟨*aco*⟩ ::= (aco ⟨*param-init*⟩ ⟨*optim-cycle*⟩)

⟨*param-init*⟩ ::= (init ⟨*trail-amount*⟩ ⟨*weights-init*⟩)

⟨*trail-amount*⟩ ::= (uniform ⟨*trail-min*⟩ ⟨*trail-max*⟩) | (tas) | (teas ⟨*rate*⟩) | (tras ⟨*rate*⟩ ⟨*weight*⟩) | (tacs) | (tmmas ⟨*rate*⟩)

⟨*weights-init*⟩ ::= (init-info ⟨*alpha*⟩ ⟨*beta*⟩)

⟨*optim-cycle*⟩ ::= (repeat-until ⟨*loop-ants*⟩ ⟨*update-trails*⟩ ⟨*daemon-actions*⟩)

⟨*loop-ants*⟩ ::= (foreach-ant make-solution-with ⟨*decision-policy*⟩ (if ⟨*bool*⟩ (local-update-trails ⟨*decay*⟩)))

⟨*decision-policy*⟩ ::= (roulette-select) | (q-select ⟨*q-value*⟩)

⟨*update-trails*⟩ ::= (progn ⟨*evaporate*⟩ ⟨*reinforce*⟩)

⟨*evaporate*⟩ ::= (do-evaporation
(if ⟨*bool*⟩ (full-evaporate ⟨*rate*⟩))
(if ⟨*bool*⟩ (partial-evaporate ⟨*rate*⟩ ⟨*ants-subset*⟩)))

⟨*reinforce*⟩ ::= (do-reinforce
(if ⟨*bool*⟩ (full-reinforce))
(if ⟨*bool*⟩ (partial-reinforce ⟨*ants-subset*⟩))
(if ⟨*bool*⟩ (rank-reinforce ⟨*many-ants*⟩))
(if ⟨*bool*⟩ (elitist-reinforce ⟨*weight*⟩))

⟨*daemon-actions*⟩ ::= (do-daemon-actions
(if ⟨*bool*⟩ (mmas-update-pheromone-limits)))

⟨*ants-subset*⟩ ::= ⟨*single-ant*⟩ | ⟨*many-ants*⟩

⟨*single-ant*⟩ ::= (all-time-best) | (current-best) | (random-ant) | (all-or-current-best ⟨*probability*⟩)

⟨*many-ants*⟩ ::= (all-ants) | (rank-ant ⟨*rank*⟩)

⟨*weight*⟩ ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | (n-ants)

⟨*trail-min*⟩ ::= 0.000001

⟨*trail-max*⟩ ::= 1.0

⟨*alpha*⟩ ::= 1 | 2 | 3

⟨*beta*⟩ ::= 1 | 2 | 3

⟨*q-value*⟩ ::= 0.7 | 0.75 | 0.8| 0.85 | 0.9 | 0.95 | 0.98 | 0.99

⟨*decay*⟩ ::= 0.01 | 0.025 | 0.05 | 0.075 | 0.1 | 0.2

⟨*rate*⟩ ::= 0.01 | 0.02 | 0.1 | 0.25 | 0.5 | 0.75 | 0.9

⟨*probability*⟩ ::= 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 | 0.7 | 0.8 | 0.9

⟨*rank*⟩ ::= 5 | 10 | 25 | 50 |75

⟨*bool*⟩ ::= t | nil

Grammar 5.1: Grammar used to evolve ACO algorithms [Tavares and Pereira, 2012]

Table 5.2: GE Learning Parameters: adapted from [Tavares and Pereira, 2012]

Runs	30
Population Size	64
Generations	40
Individual Size	25
Wrapping	No
Crossover Operator	One-Point with a 0.7 rate
Mutation Operator	Integer-Flip with a 0.05 rate
Selection	Tournament with size 3
Replacement	Steady State
Learning Instances	pr76, ts225

on a steady-state replacement method, the last generation contains the best optimization strategies identified during the learning phase. Then, these strategies are applied to unseen instances and ranked again based on the new results achieved. The comparison of the ranks obtained in different phases will provide relevant information in what concerns the generalization ability of the evolved strategies.

The GE settings used in the experiments are adapted from the original work of [Tavares and Pereira, 2012], and are outlined in Table 5.2. The population size is set to 64 individuals, each one composed by 25 integer codons, which is an upper bound on the number of production rules needed to generate an ACO strategy using the grammar adopted in this work. As this grammar does not contain recursive production rules, it is possible to determine the maximum number of values needed to create a complete phenotype. Also, wrapping is not necessary since the mapping process never goes beyond the end of the integer string.

Two different instances from Table 5.1 were selected to learn the ACO strategies: pr76 and ts225. Each ACO algorithm encoded in a GE solution is executed once during $K = 100$ iterations. The fitness assigned to this strategy corresponds to the best solution found. The strategies encode all the required settings to run the ACO algorithm, with the exception of the colony size (M), which is set to 10% of the number of cities (truncated to the closest integer).

In what concerns the validation step, the best ACO strategies are applied to four different TSP instances: lin105, pr136, pr226, lin318. In this phase, all ACO algorithms are run for 30 times and the number of iterations is increased to 5000. The size of the

Table 5.3: ACO Validation Parameters

Runs	30
Iterations	5000
Colony Size	10% of the Instance Size
Instances	lin105, pr136, pr226, lin318

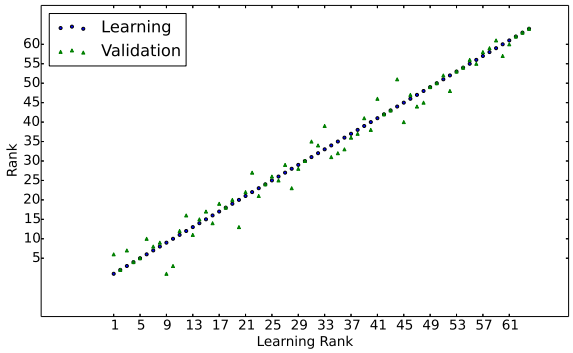
colony is the same (10% of the size of the instance being optimized). Table 5.3 summarises the parameters used. In both phases, the results are expressed as a normalized distance to the optimum.

Results

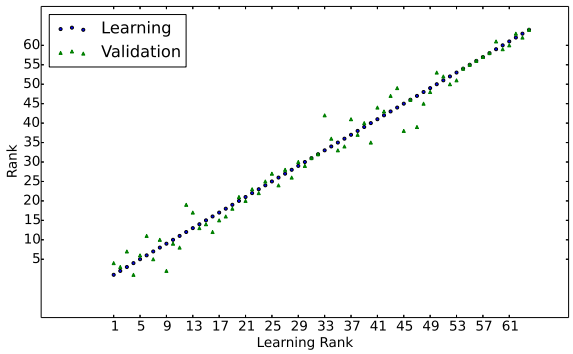
Fig. 5.1 displays the ranking distributions of the best ACO strategies learned with the pr76 instance. The 4 panels correspond to the 4 different validation instances. Each solution from the last GE generation is identified using an integer from 1 to 64, displayed in the horizontal axis. These solutions are ranked by the fitness obtained in training (solution 1 is the best strategy from the last generation, whilst solution 64 is the worst). The vertical axis corresponds to the position in the rank. Small circles highlight the learning rank and, given the ordering of the solutions from the GE last generation, we see a perfect diagonal in all panels. The small triangles identify the ranking of the solutions achieved in the 4 validation tasks (one on each panel). Ideally, these rankings should be identical to the ones obtained in training, i.e., the most promising solutions identified by the GE would be those that generalize better to unseen instances.

An inspection of the results reveals an evident correlation between the behavior of the strategies in both phases. An almost perfect line of triangles is visible in the 4 panels, confirming that the best strategies from training keep the good performance in validation. This trend is visible across all the validation instances and shows that, with the pr76 instance, training is accurately identifying the more robust and effective ACO strategies.

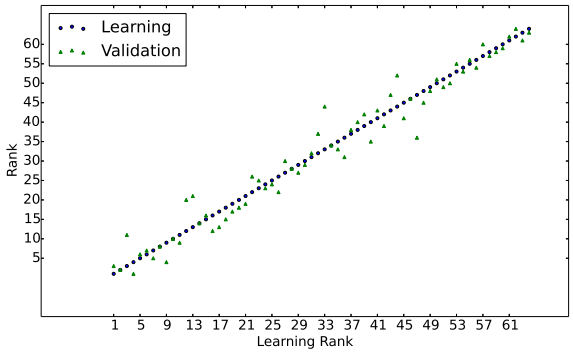
Fig. 5.2 displays the ranking distributions of the best ACO strategies learned with the ts225 instance. Although the general trend is maintained, a close inspection of the results reveals some interesting disagreements. The best ACO strategies learned with ts225 tend to have a modest performance when applied to small validation instances, such as lin105 and pr136. On the contrary, they behave well on larger instances (see, e.g., the results obtained with the validation instance from panel d)). This outcome confirms that



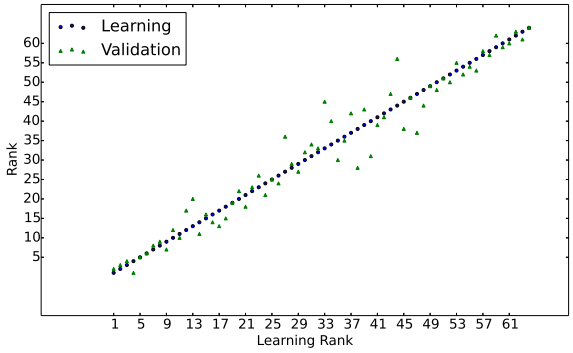
(a) lin105



(b) pr136

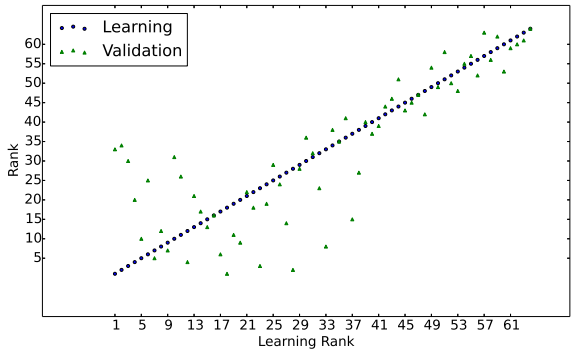


(c) pr226

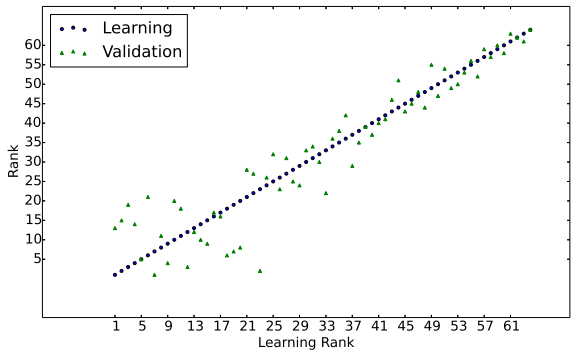


(d) lin318

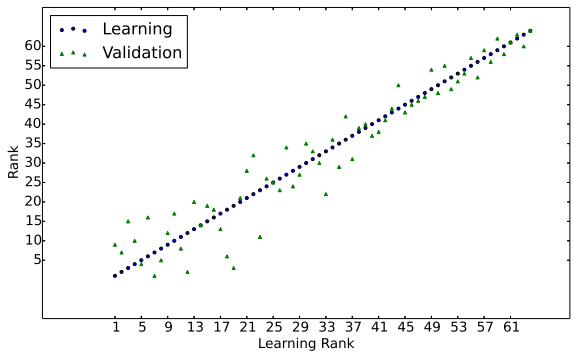
Figure 5.1: Ranking distribution of the best ACO strategies discovered with the pr76 learning instance.



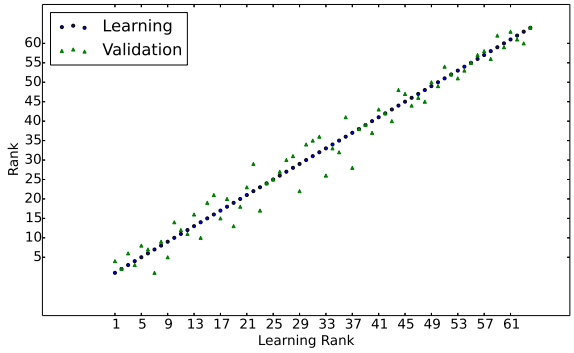
(a) lin105



(b) pr136



(c) pr226



(d) lin318

Figure 5.2: Ranking distribution of the best ACO strategies discovered with the ts225 learning instance.

the training conditions impact the structure of the evolved algorithmic strategies, which reinforces the results obtained in the previous Chapter. The ts225 instance is considered a hard TSP instance [Merz and Freisleben, 2001] and, given the results displayed in Fig. 5.2, it promotes the evolution of ACO strategies particularly suited for TSP problems with a higher number of cities. In the remainder of this section we present some additional results that help gain insight into these findings.

To authenticate the correlation between learning and validation we computed the Pearson correlation coefficient between the rankings obtained in each phase. This coefficient ranges between -1 and 1, where -1 identifies a completely negative correlation and 1 highlights a total correlation (the best strategies in learning are the best in validation). The results obtained are presented in Table 5.4. Columns contain instances used in learning, whilst rows correspond to validation instances. The values from the table confirm that there is always a clearly positive correlation between the two phases, i.e., the quality obtained by a solution in learning is an accurate estimator if its optimization ability. The lowest values of the Pearson coefficient are obtained by strategies learned with the ts225 instance and validated in small TSP problems, confirming the visual inspection of Fig. 5.2. In this correlation analysis we adopted a significance level of $\alpha = \mathbf{0.05}$. All the p-values obtained were smaller than α, thus confirming the statistical significance of the study.

To complement the analysis, we present in Fig. 5.3 the absolute performance of the best learned ACO strategies in the 4 selected validation instances. Each panel comprises one of the validation scenarios and contains a comparison between the optimization performance of strategies evolved with different learning instances (black mean and error bars are from ACO strategies trained with the pr76 instances, whilst the grey are from algorithms evolved with the ts225 instance). In general, for all panels and for strategies evolved with the two training instances, the deviation from the optimum increases with the training ranking, confirming that the best algorithms from phase 1 are those that exhibit a better optimization ability. However, the results reveal an interesting pattern in what concerns the absolute behavior of the algorithms. For the smaller validation instances (lin 105 and pr136 in panels a) and b)), the ACO strategies evolved by the smaller learning instances achieve a better performance. On the contrary, ACO algorithms learned with the ts225 instance are better equipped to handle the largest validation problem (lin318 in panel d)). This is another piece of evidence that confirms the impact of the training conditions on the structure of the evolved solutions. A detailed

analysis of the algorithmic structure reveals that the pr76 training instance promotes the appearance of extremely greedy ACO algorithms (e.g., they tend to have very low evaporation levels), particularly suited for the quick optimization of simple instances. On the contrary, strategies evolved with the ts225 training instance strongly rely on full evaporation, thus promoting the appearance of methods with increased exploration ability, particularly suited for larger and harder TSP problems.

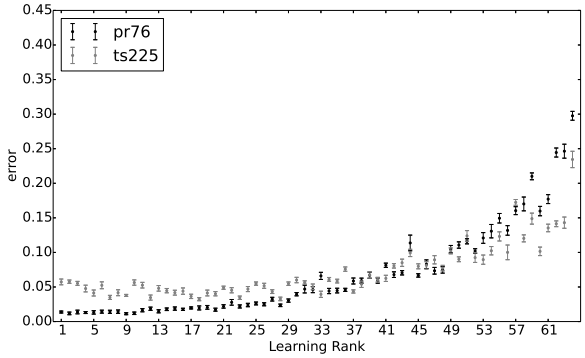
5.3.1 Measuring Overfitting

To complete our analysis we investigate the evolution of overfitting while learning ACO strategies. To estimate the occurrence of overfitting we selected one additional instance for each training scenario, with the same size of the instance used in training (eil76 and tsp225, respectively). In each GE generation, the current best ACO strategy is applied to this new test instance and the quality of the obtained solution is recorded (this value is never used for training).

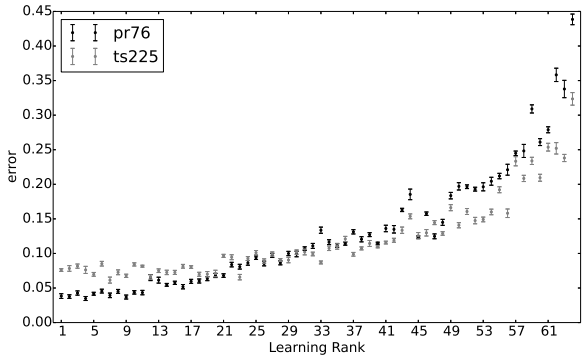
Fig. 5.4 and 5.5 present the evolution of the *Mean Best Fitness* (MBF) during the learning phase, respectively for the pr76 and ts225 instances. Both figures contain two panels: panel a) exhibits the evolution of the MBF measured by the learning instance, which corresponds to the value used to guide the GE exploration; panel b) displays the MBF obtained with the testing instance and it is only used to detect overfitting.

The results depicted in panels 5.4a and panel 5.5a show that the HH framework gradually learns better strategies. A brief perusal of the MBF evolution reveals a rapid decrease in the first generations, followed by a slower convergence. This is explained by the fact that in the beginning of the evolutionary process the GE combines different components provided by the grammar to build a robust strategy, whilst at the end it tries to fine-tune the numeric parameters. The search for a meaningful combination of components has a stronger impact on fitness than modifying numeric values.

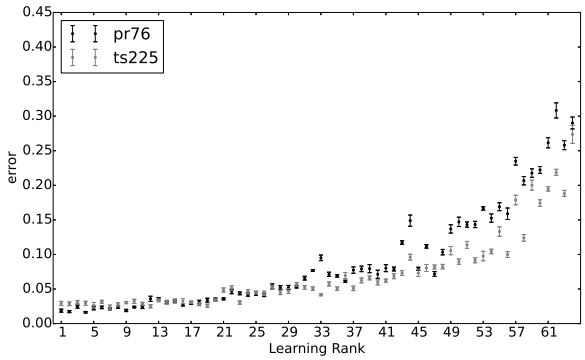
Overfitting occurs when the fitness of the learning strategies keeps improving, whilst it deteriorates in testing. Panels 5.4b and 5.5b show the MBF for the testing step. An inspection of the results shows that it tends to decrease throughout the evolutionary run. This shows that the strategies being evolved are not becoming overspecialized, i.e., they maintain the ability to solve instances different from the ones used in training.



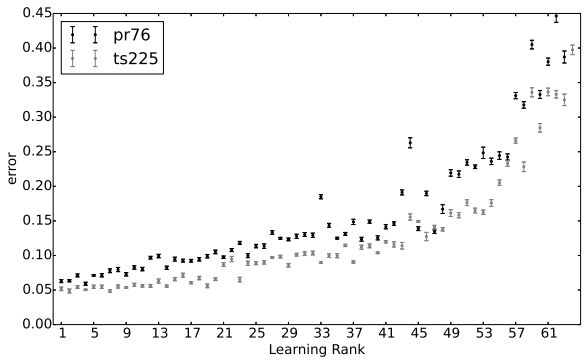
(a) lin105



(b) pr136

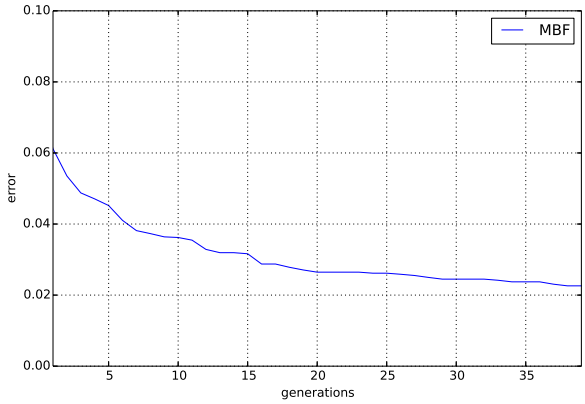


(c) pr226

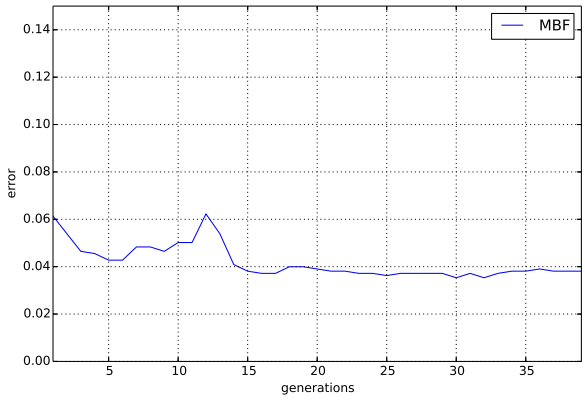


(d) lin318

Figure 5.3: MBF of the best evolved ACO strategies in the 4 validation instances. Black symbols identify results from strategies learned with the pr76 instance and grey symbols correspond to results from strategies obtained with the ts225 instance.



(a) Learning Fitness

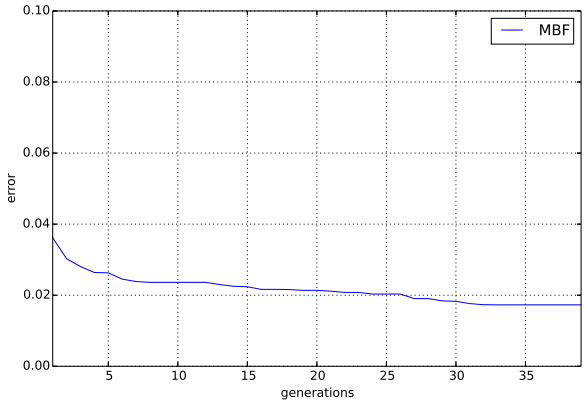


(b) Testing Fitness

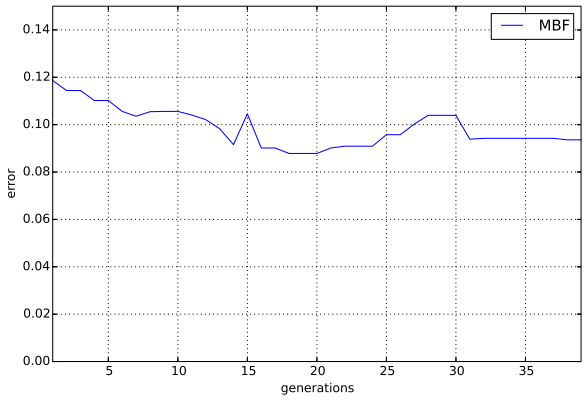
Figure 5.4: Evolution of the MBF for the pr76 learning instance and the corresponding eil76 testing instance

Table 5.4: Pearson correlation coefficients

	pr76	**ts225**
lin105	0.98	0.81
pr136	0.90	0.90
pr226	0.97	0.95
lin318	0.95	0.98



(a) Learning Fitness



(b) Testing Fitness

Figure 5.5: Evolution of the MBF for the ts225 learning instance and the corresponding tsp225 testing instance

5.4 Summary

In this Chapter we studied the correlation between the quality exhibited by strategies during learning and their effective optimization ability when applied to unseen scenarios. We relied on an existing GE-based HH to evolve full-fledged ACO algorithms to perform the analysis. Results revealed a clear correlation between the quality exhibited by the strategies in both phases. As a rule, the most promising algorithms identified in learning generalize better to unseen validation instances. This study provides valuable guidelines for HH practitioners, as it suggests that, in this case, the limited training conditions do not seriously compromise the identification of the algorithmic strategies with the best optimization ability. The outcomes also confirmed the impact of the training conditions on the structure of the evolved solutions. Training with small instances promotes the appearance of greedy optimization strategies particularly suited for simple problems, whereas larger (and harder) training cases favor algorithmic solutions that excel in more complicated scenarios. Finally, a preliminary investigation revealed that training seems to be overfitting free, *i.e.*, the strategies being learned are not becoming overspecialized to the specific instance used in the evaluation.

6

Structured Grammatical Evolution

In this Chapter we focus our attention on the HH search engine. Although Grammatical Evolution is flexible and allows the evolution of variable length programs, it has some known issues. These issues are issues related with locality and redundancy [Keijzer et al., 2002, Rothlauf and Oetzel, 2006, Thorhauer and Rothlauf, 2014]. A representation is said redundant when several different genotypes correspond to one phenotype. The original proposal of GE contains several experimental results corroborating the benefits of redundancy [O'Neill and Ryan, 2003]. In any case, this is a highly debated topic among researchers, and, clearly, excessive redundancy levels slow down evolution, thus decreasing the performance of EAs [Rothlauf, 2006].

Locality studies how variations performed in the genotype reflect on the phenotype. An EA has high locality if small modifications on the genotype result in small modifications in the phenotype, thus creating conditions for an effective sampling of the search space. If this condition is not satisfied, the exploration performed by an EA tends to resemble random search [Gottlieb and Eckert, 2000, Gottlieb and Raidl, 2000, Raidl and Gottlieb, 2005, Rothlauf, 2003]. In section 6.1 we provide further details on recent developments to improve GE in order to minimize its issues.

Structured Grammatical Evolution (SGE) is our proposal to address the limitations of GE. Its distinctive feature is having a one-to-one correspondence between genes and non-terminals of the grammar being used (see Section 6.2). The effectiveness of SGE is tested on a set of benchmarks problems, and results were encouraging, as the new representation was able to obtain better optimization results (Section 6.3).

Moreover, we present an in-depth analysis that helps to gain insight into the distinctive features of SGE and to understand why it outperforms the standard GE representation in most of the considered optimization scenarios. We rely on a set of static measures to characterize the interactions between the representation and variation operators and assess how they influence the interplay between the genotype-phenotype spaces (Section 6.4). Finally we study the impact of the choices that we made while designing the SGE (Section 6.5).

6.1 Background

There are many reports in the literature addressing the extension/improvement of the original GE framework (see, *e.g.*, [Keijzer et al., 2002, O'Neill and Brabazon, 2006, O'Neill and Brabazon, 2006, O'Neill et al., 2004]). Additionally there are several studies that aim to gain insight on how GE explores the search space [Rothlauf and Oetzel, 2006, Thorhauer and Rothlauf, 2014]. The contributions directly related with the current work are briefly reviewed in the next sections.

Mapping

The work of Keijzer et al. [Keijzer et al., 2002] is one of the first attempts to improve the mapping of standard GE by removing the bias that may exist when many rules from the grammar have the same number of production choices. Consider the following production set:

$$< \textit{bitstring} > ::= < \textit{bit} > \mid < \textit{bit} >< \textit{bitstring} >$$
$$< \textit{bit} > ::= 0 \mid 1$$

In this example there are two possible choices for each rule. The modulo operation will always select the first option ($< \textit{bit} >$ or 0) when an even codon is considered, whereas it will choose the second option ($< \textit{bit} >< \textit{bitstring} >$ or 1) when odd codons are used. This creates a linkage bias between different productions, which may decrease the GE effectiveness. To remove this effect they propose the *bucket rule*, a new mapping strategy that allows the same codon to select for different production choices.

Chorus [Azad, 2003, Ryan et al., 2002] pioneered the proposal of a position independent GE, whose representation is loosely inspired on protein production to regulate the

metabolic pathways of the cell. In this system the genotype is composed by 8-bit integers that represent the rules of the associated BNF grammar. The key difference from GE is that each gene value corresponds to a specific production rule. The modulo operation considers the total number of production rules in the grammar, so a specific allele always maps to the same rule, regardless of its position in the genotype. The analysis presented in the above mentioned reference reveals that Chorus obtains results comparable to standard GE in a set of selected optimization problems.

In [O'Neill et al., 2004], O'Neill et al. proposed πGE, another position independent alternative to GE. The standard GE mapping creates a positional dependency, as the derivation is always performed by expanding the leftmost non-terminal. πGE removes this bias by creating codons with two values: *nont* and *rule*. In this case, *nont* helps to select the next non-terminal NT to be expanded: $NT = nont\%count$, where ***nont*** is the value present in the genotype, and ***count*** is the number of non-terminals still in the derivation tree. The *rule* value of the codon pair, as in standard GE, selects which production rule should be applied from the selected non-terminal NT.

Fagan and coworkers [Fagan, 2013, Fagan et al., 2010] compared the performance of several mapping mechanisms. Besides the aforementioned πGE and the traditional depth-first expansion, they also considered breadth-first and a random expansion mechanism. Results revealed that πGE outperforms standard GE, confirming the relevance of investigating new, alternative, genotypic representations, and the corresponding mapping processes.

Representation and Search Operators

In 2002 Sullivan et al. [O'Sullivan and Ryan, 2002] compared the performance of GE-based systems, while using different search strategies. In concrete, they separately considered hill-climbing, simulated annealing, random search, and genetic algorithms as the GE search engine. Results obtained in several benchmark problems revealed that the genetic algorithm is the best option for achieving an enhanced performance. Additionally, the authors analysed the impact of different search operators and verified that crossover is critical for the success of the optimization.

Later, Rothlauf et al. [Rothlauf and Oetzel, 2006] pioneered the analysis of locality and redundancy in GE. Their research revealed that, in approximately 90% of the cases,

a genotypic mutation does not change the phenotype. This result shows that GE suffers from extremely high levels of redundancy, which is mostly a consequence of the many-to-one mapping that allows multiple codons to correspond to the same production rule. Another important result of this work is related with the remaining 10% of mutations. Specifically, when the genotype suffers one mutation, changes of one or more units occur at the phenotypic level. Units of change in this level are estimated by the tree edit distance [Zhang and Shasha, 1989], that considers the minimal deletions, insertions, or replacements needed to transform one phenotype into the other. Empirical results obtained in two optimization problems reveal that the locality of the genotype-phenotype mapping is low, as many genotypic neighbors originate highly dissimilar phenotypes. Recently Thorhauer et al. [Thorhauer and Rothlauf, 2014] extended the previous work and compared the locality of the standard GE operators with standard GP. They analyzed the locality on problems that rely on binary trees, by performing random walks through the search space and measuring the distance between parents and offspring. This research confirms the low locality of GE, reinforcing the relevance of developing alternative representations and associated operators that increase locality.

Byrne et al. [Byrne et al., 2009, Byrne et al., 2010] proposed two distinct GE mutation operators, with complementary effects on the locality level: nodal mutation is a high-locality operator that changes a single node labelling in the derivation tree; structural mutation is a low-locality operator that modifies the structure of the derivation tree. An experimental study confirms the complementary effect of the two operators, as structural mutation promotes the exploration of the search space, whereas nodal mutation focuses on the exploitation of the neighborhood of current solutions.

In [Castle and Johnson, 2010], the authors present an experimental study that identifies a positive correlation between the locus where variation operators are applied and their ability to change the phenotype. In concrete, results show that the application of crossover and mutation to the beginning of the genotype promotes low-locality, as it may alter the interpretation of all the remaining codons.

Hugosson et al. [Hugosson et al., 2010] analyze the impact of three different genotypic representations on GE effectiveness: binary, gray coding, and integer. Experimental results are inconclusive, as no representation clearly outperforms the others. According to the authors, this outcome suggests that the relevance of a given GE representation cannot be decoupled from the mapping process embedded in the grammar.

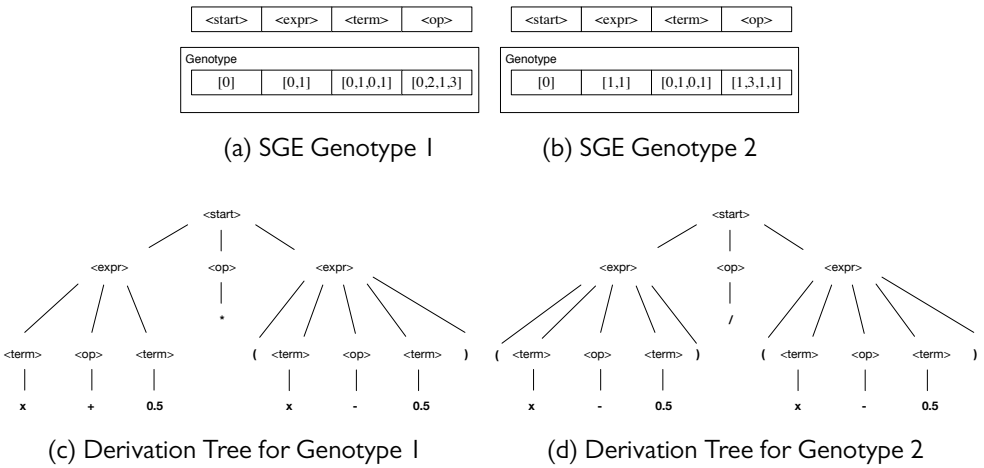
$< start >::=< expr >< op >< expr > | < expr >$
$< expr >::=< term >< op >< term > |(< term >< op >< term >)$
$< term >::= x|0.5$
$< op >::= +| - | * |/$

Figure 6.1: Example of a production set to create polynomial expressions

6.2 Structured Grammatical Evolution

Structured Grammatical Evolution (SGE) is a novel genotypic representation for Grammatical Evolution. In SGE each gene is linked to a specific non-terminal, and it comprises a list of integers used to select an expansion option. The length of each list is determined by computing the maximum possible number of expansions of the corresponding non-terminal (see details in section 6.2.1). This structure ensures that the modification of a gene does not affect the derivation options of other non-terminals, thus limiting the number of changes that can occur at the phenotypic level. The values inside each list are bounded by the number of possible expansion options of the corresponding non-terminal. Therefore, mapping does not rely on the modulo rule, thus reducing the redundancy associated with it.

As an example, consider the grammar depicted in Fig. 6.1. The set of non-terminals is $\{< start >, < expr >, < term >, < op >\}$. Then, the SGE genotype is composed by four genes, each one linked to a specific non-terminal. To determine the length of the gene's lists we compute the maximum number of expansions of a non-terminal. The $< start >$ symbol is expanded only once, as it is the grammar axiom. The $< expr >$ symbol is expanded, at most, twice, because of the rule $< expr >< op >< expr >$. The computation of the list size for $< term >$ establishes a direct dependence between this non-terminal and $< expr >$: each time $< expr >$ is expanded, $< term >$ is expanded twice (in the two possible expansion options). As the grammar allows a maximum of two $< expr >$ expansions, it immediately follows that the list size for the $< term >$ gene is four. Following the same line of reasoning, the list size for the $< op >$ gene is 3. Thus, the list sizes for each gene are: $< start >$: $1, < expr >$: $2, < term >$: $4, < op >$: 3. To complete the list inside each gene we take the number of derivation options c_N of the corresponding non-terminal, and assign a random value from the interval $[0, c_N - 1]$



(a) SGE Genotype 1 (b) SGE Genotype 2

(c) Derivation Tree for Genotype 1 (d) Derivation Tree for Genotype 2

Figure 6.2: Panels (a) and (b) exemplify two possible genotypes for the production set of Fig. 6.1. Panels (c) and (d) display the corresponding derivation trees.

to every position. The $<$ *start* $>$, $<$ *expr* $>$ and $<$ *term* $>$ symbols have $c_N = 2$, whereas $<$ *op* $>$ has $c_N = 4$. In Fig. 6.2 we exemplify two possible SGE genotypes for this example.

The translation process of the genotype from Fig. 6.2a is illustrated in Fig. 6.3. The operation starts at the $<$ *start* $>$ axiom and it proceeds in the standard way by expanding non-terminals in a left-first approach. The first unused integer of the $<$ *start* $>$ gene list is 0, which replaces the axiom with $<$ *expr* $><$ *op* $><$ *expr* $>$. Given the 0 value in the first position of the $<$ *expr* $>$ non-terminal, the expression is transformed into $<$ *term* $><$ *op* $><$ *term* $><$ *op* $><$ *expr* $>$. Next, $<$ *term* $>$ is replaced by *x* and the process continues until there are no more symbols to derive. The final expression that results from the mapping of the genotype displayed in Fig. 6.2a is "x+0.5*(x-0.5)". Following an identical approach, the genotype from Fig. 6.2b is translated into the phenotype "(x-0.5) * (x-0.5)".

6.2.1 Genotypic Structure

To establish the structure of the genotype for a given grammar, one must compute an upper bound for the number of non-terminal expansions, as this defines the list size for each gene. This is accomplished in a two-step process. Initially, Alg. 15 iterates through the grammar productions and records the maximum number of non-terminal references

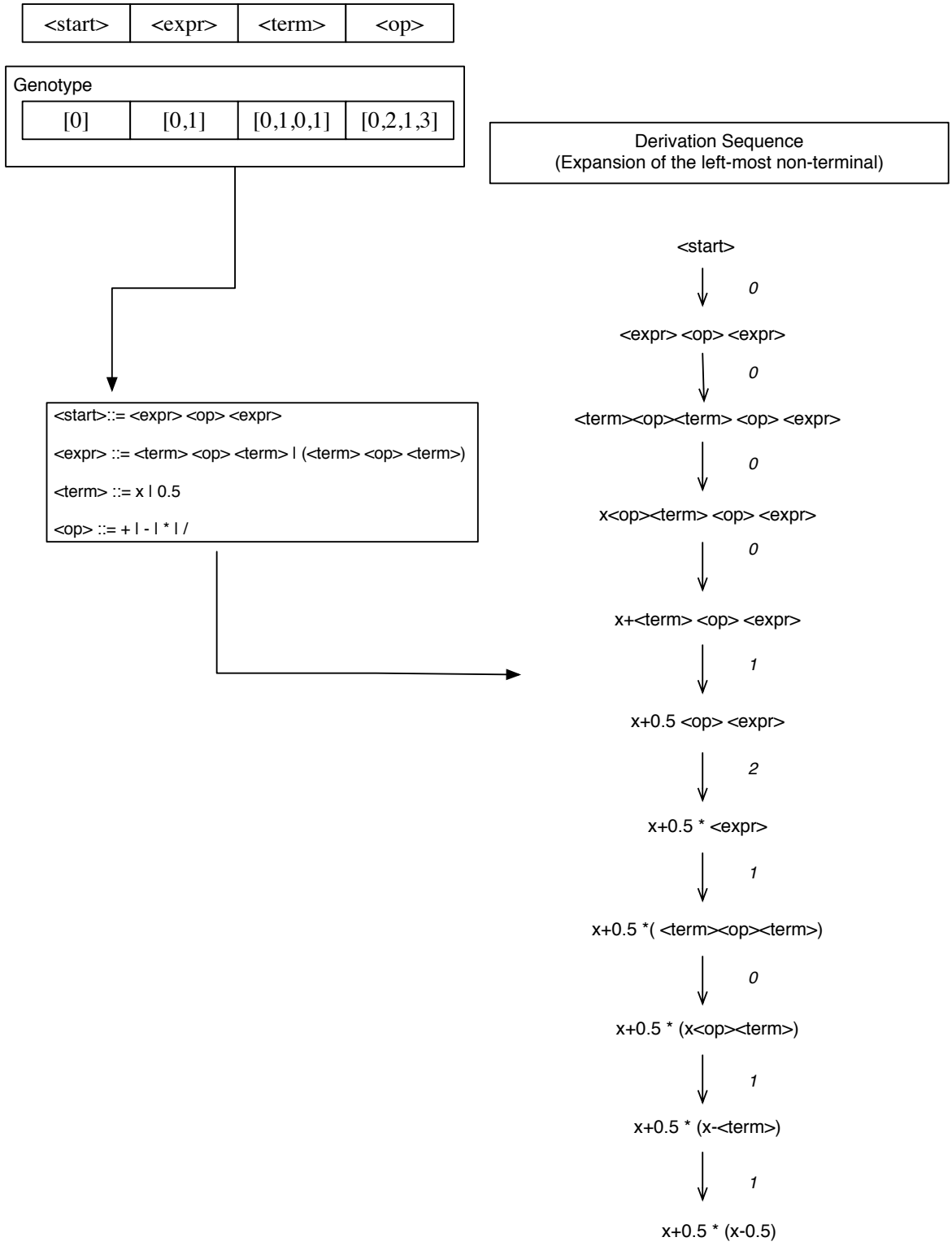


Figure 6.3: Translation of the genotype from Fig. 6.2a: Derivation sequence.

that occur in each choice. At the same time, a set establishing a relation between non-terminals is built. In step 2, Alg. 16 iterates the set of non-terminals and recursively determines an upper bound for the number of expansions of each non-terminal. This algorithm takes into account the dependences between the non-terminals (e.g., the dependence between $<$ *expr* $>$ and $<$ *term* $>$ in the production set of Fig. 6.1)

Algorithm 15 Calculation of the non-terminal references.

```
countReferences ← {}
isReferencedBy ← {}
for nt in nonTerminalsSet do
    for production in grammar[nt] do
        for option in production do
            if option ∈ nonTerminalsSet then
                isReferencedBy[option] ← nt
                count[option] ← count[option] + 1
            end if
        end for
    end for
    for key in count do
        countReferences[key][nt] ← max(countReferences[key][nt], count[key])
    end for
end for
```

6.2.2 Recursive Grammars

The pre-processing described in the previous section does not consider recursive grammars. Standard GE deals with recursion by always trying to perform the translation into an executable program. If it runs out of integers, GE assigns the worst possible fitness value to the individual.

SGE deals with recursion in a different way, as it contains a parameter that establishes the maximum level of recursion. This value is selected prior to the application of the algorithm to a specific problem and, given this approach, a set of intermediate symbols that mimic the levels of the recursion tree must be inserted in the grammar. The following example is an excerpt of a grammar for symbolic regression problems:

Algorithm 16 Calculation of the upper bound for non-terminals expansion.

```
function FINDREFERENCES(nt, isReferencedBy, countReferencesByProd)
    references ← getTotalReferencesOfCurrentProduction(countReferencesByProd, nt)
    results ← []
    if nt = startSymbol then
        return 1
    end if
    for ref in isReferencedBy[nt] do
        result.append(FINDREFERENCES(ref,isReferencedBy,countReferencesByProd))
    end for
    references ← references * max(result)

    return references
end function
```

$< start >::= < expr >$

$< expr >::= < expr >< op >< expr > | < var >$

$< op >::= + | - | * | /$

$< var >::= x$

As the $< expr >$ production is recursive, it needs to be rewritten before the SGE structure is determined. Assuming that 2 levels of recursion were established it becomes:

$$
\begin{aligned}
< start >::= & < expr > \\
< expr >::= & < expr\_lvl\_0 >< op >< expr\_lvl\_0 > \\
& | < var > \\
< expr\_lvl\_0 >::= & < expr\_lvl\_1 >< op >< expr\_lvl\_1 > \\
& | < var > \\
< expr\_lvl\_1 >::= & < var >< op >< var > | < var > \\
< op >::= & + | - | * | / \\
< var >::= & x
\end{aligned}
$$

The grammar transformation satisfies two conditions: first, there will be no invalid solutions, as the mapping process always ends; also, the grammar symbols maintain their

selection probability, since they are always replicated to each new added level.

All GP variants impose a constraint in the maximum program size, a mandatory step to prevent solutions from growing excessively and becoming computationally intractable. The constraint might be imposed in terms of tree depth, number of available nodes [Koza, 1992], or by imposing limits on the number of wrappings as performed in GE [Dempsey et al., 2009]. Following a similar line of procedure, SGE limits the maximum program size by imposing a limit on the number of recursive calls.

6.2.3 Genetic Operators

GE relies on standard operators to explore the search space, looking for promising solutions to the problem at hand. Two existing variation operators are adapted to work with SGE.

Recombination

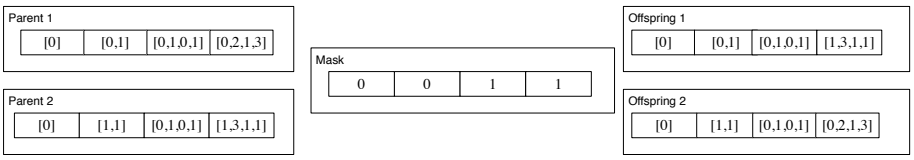
SGE recombination is based on the uniform crossover for binary representations. It generates a random binary mask and the offspring are created by selecting the parents genes' based on the mask values. Recombination does not modify the lists inside the genes. Fig. 6.4 illustrates an application of this operator.

Mutation

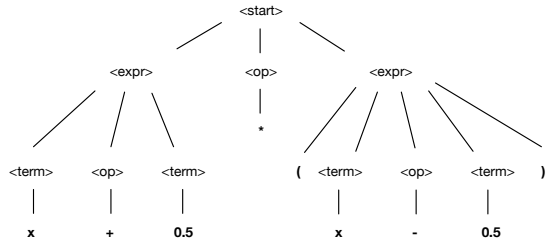
This operator has the ability to modify the lists inside the SGE genes. When applied to a specific list value, it changes it to a new random value from $[0, c_N - 1]$, where c_N is the number of derivations options of the non-terminal linked this gene. Fig. 6.5 illustrates an application of mutation.

6.3 Experimental Results

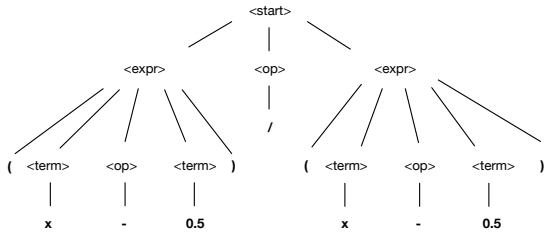
This section presents a set of experimental results to assess the optimization performance of SGE and to compare it with standard GE. We selected several benchmark problems according to the guidelines proposed by White et al. [White et al., 2013]. These problems



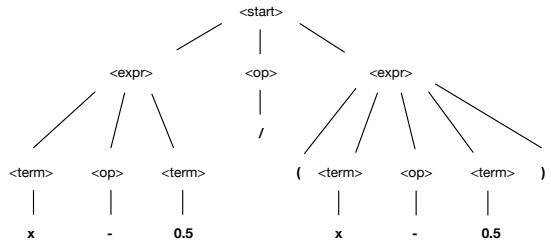
(a) Crossover Application



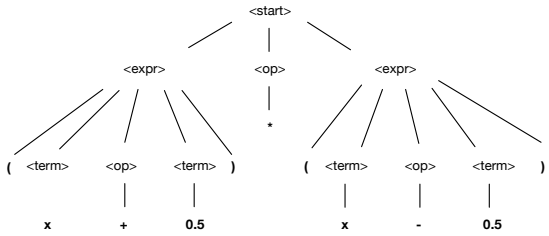
(b) Derivation Tree of Parent 1



(c) Derivation Tree of Parent 2



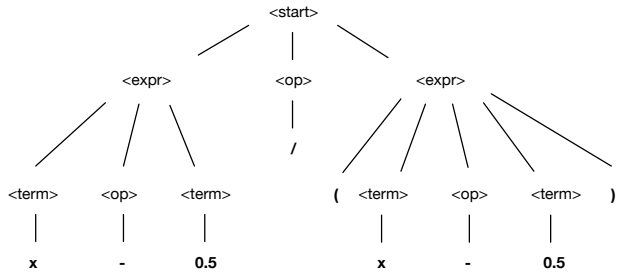
(d) Derivation Tree of Offspring 1



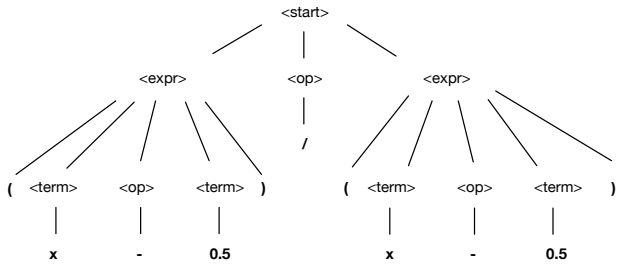
(e) Derivation Tree of Offspring 2

Figure 6.4: Application of the recombination operator. Panel (a) shows how to combination two parents two produce offspring. Panels (b) and (c) show the derivation trees of the two parents, and panels (d) and (e) show the derivation trees for the two generated offspring.

(a) Mutation Application



(b) Derivation Tree of Offspring 1 before mutation



(c) Derivation Tree of Offspring 1 after mutation

Figure 6.5: Application of the mutation operator. Panel (a) exemplifies how mutation changes the values inside each list. Panel (b) shows the derivation tree before the mutation, whilst panel (c) shows the derivation tree resulting from the application of the mutation.

Table 6.1: Settings for the Experimental Analysis

Parameter	GEVA	SGE
Initial Population	500	
Recombination rate	0.9	
Mutation rate	0.02	
Replacement	Steady-State with a generation gap of 0.9	
Selection	Tournament with size 3	
Generations	50	
Recombination Operator	Single Point Crossover	SGE Crossover
Mutation Operator	Integer Flip Mutation	SGE Mutation
Genotype Size	128 (Ramped Half and Half Initialization)	-
Wraps	3	-
Maximum Level of Recursion	-	6

are classified in three categories: symbolic regression, predictive modelling, and path finding.

The GEVA implementation of GE is adopted [O'Neill et al., 2008] in the experiments. As SGE does not have invalid individuals, GE sensible initialization is used to create equivalent initial populations. The complete experimental settings of both approaches are summarized in Table 6.1. The sum of the errors is the fitness functions adopted for all tests and 30 independent runs are performed in each optimization scenario. Results presented are always averages of the 30 runs.

6.3.1 Symbolic Regression

In this work we consider two instances: the harmonic curve and the pagie polynomial.

The main objective of the harmonic curve regression is to approximate the polynomial $\sum_i^x \frac{1}{i}$ in the interval $x \in [1, 50]$. In addition to the standard optimization task, this problem also considers a generalization step that extrapolates the fitting to the interval $x \in [51, 120]$. The grammar for the harmonic curve regression is defined in Grammar 6.1. Note that the terminal *inverse* is $1/x$.

In the pagie polynomial, the goal is to approximate the function defined by $\frac{1}{1+x^{-4}} + \frac{1}{1+y^{-4}}$, where x, y are sampled in the interval $[-5, 5]$, with a step $s = 0.4$. Even though it defines a smooth search space, the pagie polynomial has the reputation for being difficult

$$N = \{start, expr, op, pre\_op, var\}$$
$$T = \{+, -, inverse, sqrt, x\}$$
$$S = \{start\}$$

And the production set P is:

$\langle start \rangle$ ::= $\langle expr \rangle$

$\langle expr \rangle$ ::= $\langle expr \rangle \langle op \rangle \langle expr \rangle$ | ($\langle expr \rangle$) | $\langle pre\_op \rangle (\langle expr \rangle)$ | $\langle var \rangle$

$\langle op \rangle$ ::= + | *

$\langle pre\_op \rangle$ ::= + | - | inverse | sqrt

$\langle var \rangle$ ::= x

Grammar 6.1: Grammar used in the Harmonic Curve Regression

[White et al., 2013, Harper, 2012]. Like in the first task, this problem also considers a generalization step. Here, a compact grid of points, in which x, y are obtained from the same interval with a smaller step of $s = 0.1$, is defined. The grammar for this problem is defined in Grammar 6.2.

The optimization results obtained by SGE and GE in the two regression problems are displayed in Fig. 6.6. The evolution of the Mean Best Fitness (MBF) confirms that both approaches gradually discover better approximations for the polynomials under consideration. There are, however, important differences between GE and SGE. In both problems, the results obtained by SGE are consistently better than those of GE throughout the optimization run, revealing that the new representation is more effective in building models to accurately approximate the target polynomial in the given interval. Additionally, SGE can deliver good quality solutions faster than standard GE, revealing an enhanced efficiency.

To complement the analysis we verified how some evolved models generalize to the extended intervals. For each representation, we selected the best solutions from the initial population (GE1, SGE1), from the population of the 25th generation (G25, SGE25) and from the last population (G50, SGE50). The results of the application of these models to the generalization step are presented in the boxplots of Fig. 6.7. For the harmonic curve, the general trend is for an enhanced performance of models discovered in the last generations. This is a relevant outcome, as it reveals that, in the harmonic curve,

$$N = \{start, expr, op, pre\_op, var\}$$
$$T = \{+, -, /, sin, cos, exp, log, x, y\}$$
$$S = \{start\}$$

And the production set P is:

$\langle start \rangle$ ::= $\langle expr \rangle$

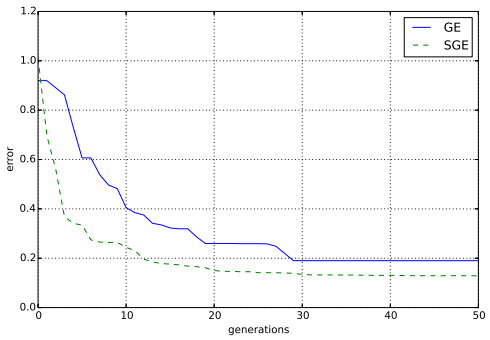
$\langle expr \rangle$ ::= $\langle expr \rangle \langle op \rangle \langle expr \rangle$ | ($\langle expr \rangle$) | $\langle pre\_op \rangle (\langle expr \rangle)$ | $\langle var \rangle$

$\langle op \rangle$ ::= + | - | * | /

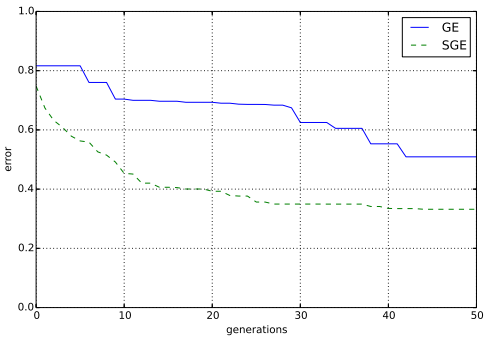
$\langle pre\_op \rangle$ ::= sin | cos | exp | log

$\langle var \rangle$ ::= x | y

Grammar 6.2: Grammar used in the Pagie Polynomial

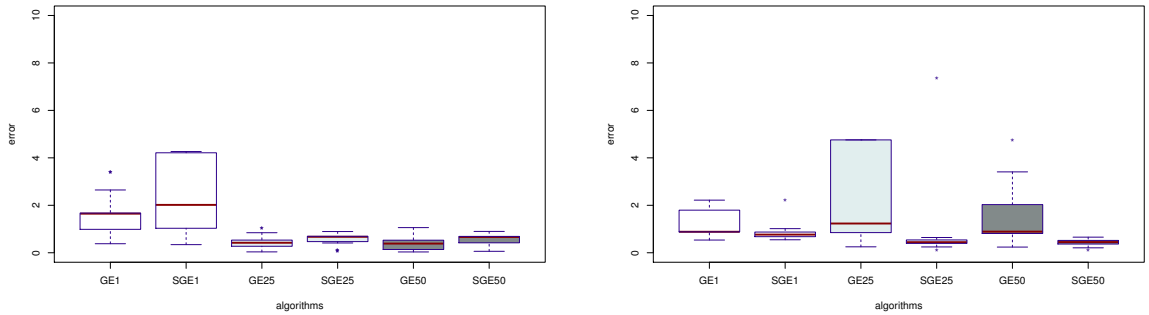


(a) Harmonic Curve



(b) Pagie

Figure 6.6: Evolution of the MBF in the two regression problems: (a) Harmonic Curve; (b) Pagie.



(a) Harmonic Curve (b) Pagie

Figure 6.7: Results obtained by selected evolved models in the generalization step of the regression problems: (a) Harmonic Curve; (b) Regression.

neither GE or SGE are overfitting. A closer inspection reveals that both GE variants obtain comparable results, and there are never statistically significant differences between pairs of models taken from the same generation. This outcome suggests that, in this particular problem, SGE and GE have similar generalization ability. The scenario on the Pagie polynomial generalization task is different. In this problem, SGE is able to clearly find strategies that generalize better than standard GE. Also, SGE variance is smaller, revealing a higher reliability. A direct comparison of the GE boxes in the generalization task discloses an overfitting situation, since the errors increase as we consider models obtained in later generations.

6.3.2 Path Finding

The Santa Fe Ant Trail is the selected path finding problem. It consists in defining a strategy that allows an artificial ant to collect 89 food pellets from a 32x32 toroidal grid in a limited number of steps.

The ant starts in the top-left corner of the grid and can turn left, right, move one square forward, and check if the square ahead contains food. Grammar 6.3 shows the grammar used in this problem.

The result obtained in this problem are visible in Fig 6.8. As the data reflects SGE is more effective than GE, since it can build programs that allow the ant to eat all the pieces of food in the grid. Also, we see again that SGE is more efficient since it finds good quality

$$N = \{start, code, line, op\}$$
$$T = \{ant.sense\_food, ant.turn\_left, ant.turn\_right, ant.move\_forward, (,)\}$$
$$S = \{start\}$$

And the production set P is:

⟨*start*⟩ ::= ⟨*code*⟩

⟨*code*⟩ ::= ⟨*line*⟩ |
⟨*code*⟩
⟨*line*⟩

⟨*line*⟩ ::= if ant.sense_food():
⟨*line*⟩
else:
⟨*line*⟩
| ⟨*op*⟩

⟨*op*⟩ ::= ant.turn_left() | ant.turn_right() | ant.move_forward()

Grammar 6.3: Grammar used in the Santa Fe Ant Trail

solutions faster.

6.3.3 Predictive Modelling

Predictive modelling considers sets of previously labelled data to create models that correctly predict the label of unseen data. For our experiments we selected Bio, PPB and LD50, three high-dimensional datasets from the pharmacokinetics domain. In the Bio dataset the goal is to predict the human oral bioavailability (represented as %F). %F is the parameter that measures the percentage of the initial orally submitted drug dose that effectively reaches the systemic blood circulation after passing through the liver. This dataset consists of 359 instances of 242 elements (241 descriptors that identify a drug, followed by the %F value for that drug). The aim of the PPB dataset is to predict the protein-plasma binding level (%PPB), *i.e.*, to quantify the initial dose of a drug that reaches the blood circulation and binds to the plasma proteins. PPB is composed by 131 instances of 627 descriptors (626 features that identify a drug, followed by the known %PPB value for that drug). Lastly, the goal LD50 is to build a model that predicts the median lethal

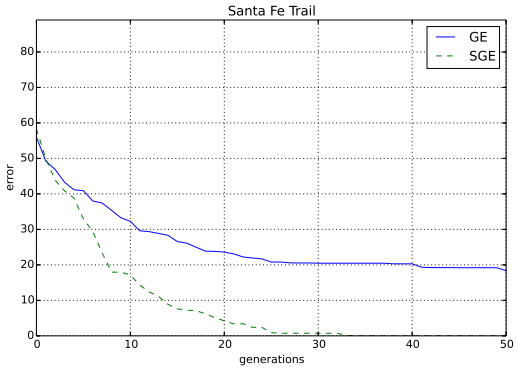


Figure 6.8: Santa Fe Ant Trail.

dose of a set of drug compounds. This dataset is composed by 234 instances where each is a vector of of 627 elements (626 features that identify a drug, followed by the known LD50 value for that drug). For a more detailed description of the datasets please refer to [Archetti et al., 2007, Gonçalves and Silva, 2013].

The experiments described in this section correspond to a typical machine learning task and the datasets are divided in two equal parts. 50% of the instances are used in training, which is when the GE variants try to evolve promising models. Afterwards, the remaining 50% are used to assess the generalization of the best evolved models. Following the guidelines of the aforementioned references, the number of generations of the evolutionary process is increased to 200 generations. The grammar for the predictive modeling is presented in Grammar 6.4.

Figs. 6.9a), 6.10a), 6.11a) present the evolution of the MBF during the training period, both for GE and SGE. The outcomes are in line with the optimization results obtained in previous problems and confirm SGE effectiveness and efficiency: the new representation is able to discover better solutions in a lower number of generations.

Once again we selected three models evolved with each GE variant to estimate the generalization ability: one from the beginning, one halfway and another at the final generation. The testing results are depicted in the boxplots from Figs. 6.9b), 6.10b),6.11b). An inspection of the results reveals two different scenarios: overfitting does not occur for the Bio and PPB datasets, as models from the final training stages generalize better. Furthermore, a direct comparison between GE200 and SGE200 demonstrates a clear advantage of the solutions evolved with SGE. In both datasets, the new representation

$$N = \{start, expr, op, var\}$$
$$T = \{+, -, /, x_i\}$$
$$S = \{start\}$$

And the production set P is:

⟨*start*⟩ ::= ⟨*expr*⟩

⟨*expr*⟩ ::= ⟨*expr*⟩⟨*op*⟩⟨*expr*>⟩ | (⟨*expr*⟩) | ⟨*var*⟩

⟨*op*⟩ ::= +|-|*|/

⟨*var*⟩ ::= x_i|-1.0|-0.5|0.0|0.5|1.0

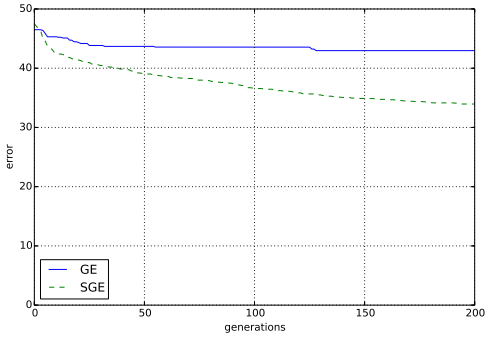
Grammar 6.4: Grammar used in the Predictive Modeling

promotes the discovery of models with increased reliability for labelling unseen data.

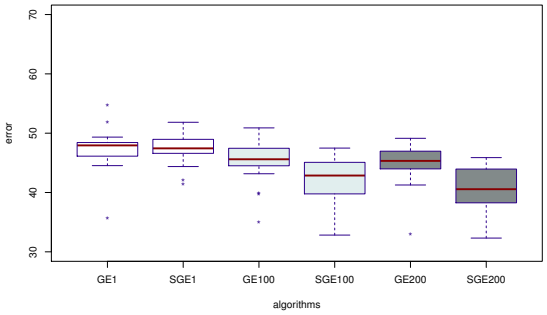
The situation for the LD50 dataset (Fig. 6.11b)) is different. The generalization behavior of models evolved with the two representations is similar, as the boxplots are leveled. Also, the solutions obtained in the last training stages are not clearly better than those discovered in the beginning, suggesting that the GE variants are not able to evolve reliable and general models. The LD50 dataset is considered as particularly challenging and standard GP-based approaches are unable to evolve models with good generalization ability [Gonçalves and Silva, 2013]. The solution to overcome difficulties is to rely on specific training strategies that help to control overfitting. Since our experiments did not consider such methods, it is not surprising that both GE variants have difficulties in evolving effective models and obtain comparable generalization results.

6.3.4 Statistical Validation

A statistical analysis was applied to confirm the relevance of the results and to assess if there are differences in the means and, if that is the case, how significant they are. Since the samples do not follow a normal distribution, the analysis is performed using non-parametric tests. Moreover, and since the two groups are unrelated, the Mann-Whitney test, at a $\alpha = 0.05$ level of significance, is the most appropriate. When significant differences exist, the effect size *r* estimates its magnitude [Field, 2009]: a +++ sign indicates

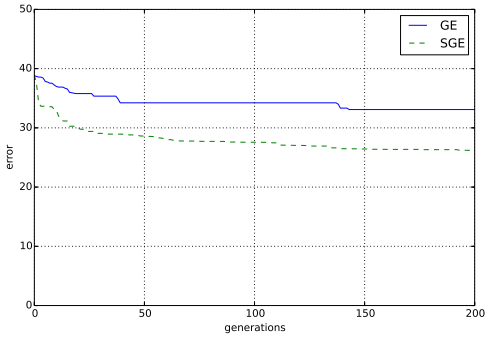


(a) Training

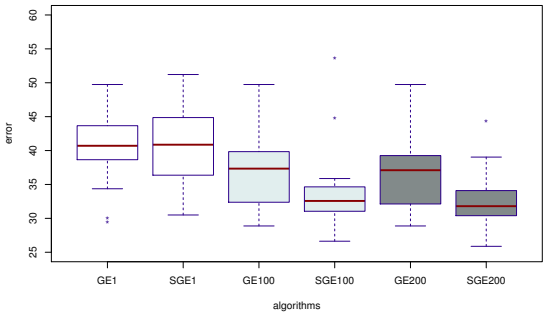


(b) Generalization

Figure 6.9: Results obtained with the Bio dataset: (a) Evolution of BMF during training; (b) Generalization ability of selected models.

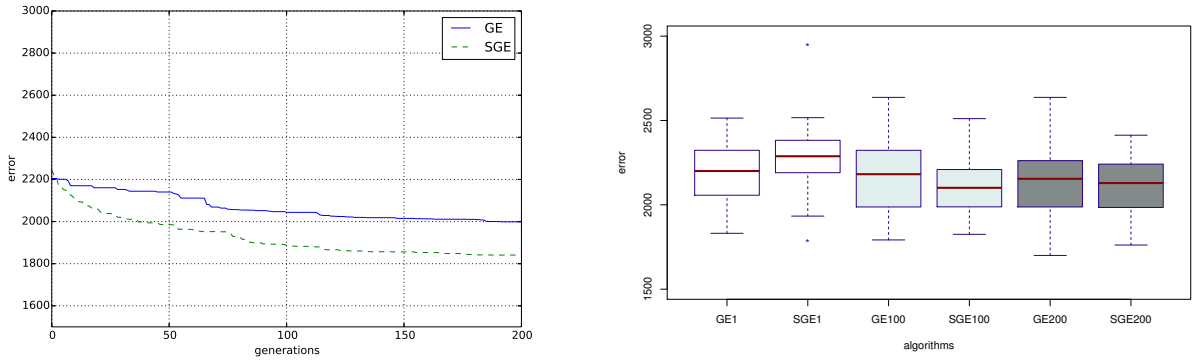


(a) Training



(b) Generalization

Figure 6.10: Results obtained with the PPB dataset: (a) Evolution of BMF during training; (b) Generalization ability of selected models.



(a) Training (b) Generalization

Figure 6.11: Results obtained with the LD50 dataset: (a) Evolution of BMF during training; (b) Generalization ability of selected models.

that the effect size is large ($r >= 0.5$), a ++ sign indicates that the effect size is medium ($0.3 <= r < 0.5$), whereas a + identifies a small effect size ($0.1 <= r < 0.3$). Additionally, $\sim$ is used to identify situations where no statistical differences exist.

Table 6.2 reports the statistical analysis for the optimization/training stages of the 6 selected problems. Columns *GE* and *SGE* display, respectively, the MFB and standard deviation of the solutions from the final generation. The last column (*StatisticalValidation*) shows the *p-values* obtained in the comparison, and the corresponding effect size. Table 6.3 compares the generalization ability of the best models obtained in the last generation of training. The Santa Fe ant trail is not included, as it does not have a generalization step. The results contained in both tables confirm the effectiveness of SGE. In the optimization/training stage, the new representation always finds solutions that significantly outperform standard GE. Even more important is the analysis summarized in Table 6.3, as it shows that the optimization effectiveness does not compromise the generalization ability. In three of the problems, SGE is able to evolve models that outperform standard GE, whereas in the remaining two the behavior of boths representation is equivalent.

6.4 Representation Analysis

There are several empirical tools that help to gain insight into the locality and redundancy of EAs. One of the first proposals was the Fitness Distance Correlation (FDC) [Jones

Table 6.2: Statistical analysis of Optimization/Training results: MBF, standard deviation, and statistical validation. Results are averages of 30 runs.

Problem	GE	SGE	Statistical Validation	
			p-value	Effect Size
Harmonic	0.20 ($\pm$**0.11**)	0.13($\pm$**0.05**)	$\mathbf{7.21 * 10^{-03}}$	++
Pagie	0.50 ($\pm$**0.26**)	0.29 ($\pm$**0.09**)	$\mathbf{2.20 * 10^{-06}}$	+++
BIO	42.97 ($\pm$**4.95**)	34.43 ($\pm$**3.57**)	$\mathbf{4.03 * 10^{-09}}$	+++
PPB	34.06 ($\pm$**5.06**)	26.61 ($\pm$**2.70**)	$\mathbf{8.02 * 10^{-10}}$	+++
LD50	2115.70 ($\pm$**143.97**)	1841.69 ($\pm$**160.33**)	$\mathbf{1.37 * 10^{-08}}$	+++
Santa Fe Ant Trail	21.40 ($\pm$**12.40**)	0.00 ($\pm$**0.00**)	$\mathbf{9.21 * 10^{-14}}$	+++

Table 6.3: Statistical analysis of Generalization/Testing results: MBF, standard deviation, and statistical validation. Results are averages of 30 runs.

Problem	GE	SGE	Statistical Validation	
			p-value	Effect Size
Harmonic	0.45 ($\pm$**0.36**)	0.56($\pm$**0.24**)	0.05	$\sim$
Pagie	0.90 ($\pm$**6.58**)	0.4352 ($\pm$**0.13**)	$\mathbf{3.02 * 10^{-09}}$	+++
BIO	45.05 ($\pm$**3.04**)	40.67 ($\pm$**3.84**)	$\mathbf{2.44 * 10^{-06}}$	+++
PPB	36.87 ($\pm$**5.33**)	32.35 ($\pm$**3.70**)	$\mathbf{1.96 * 10^{-04}}$	+++
LD50	2148.00 ($\pm$**202.17**)	2162 ($\pm$**352.49**)	0.69	$\sim$

and Forrest, 1995]. FDC estimates the relation between fitness values and the distance to the optimum. If the quality increases as the distance to the optimum decreases, then EAs are expected to have a good performance. However, FDC has some limitations, such as the requirement to know the global optima before hand, which is impossible for many situations.

In this work we adopt another framework, originally proposed by Raidl et al. [Raidl and Gottlieb, 2005]. This empirical model allows for an easy and accurate analysis of the interplay between representation and genetic operators and provides a reliable basis for assessing the performance of the search algorithm. Studies performed with this framework are based on distance measures applied to pairs of solutions linked by the application of variation operators. The distribution of the distance values helps to estimate the redundancy of the representation and to establish a relation between the exploration of genotypic space and how it reflects in the phenotypic space. In concrete, the framework relies on Mutation Innovation (MI) and Crossover Innovation (CI) measures. MI estimates how the mutation operator modifies the phenotype of an individual, whilst the CI approximates the novelty introduced by the recombination operator.

6.4.1 Basic Concepts

Spaces

SGE relies on a mapping between genotypes and phenotypes, thus requiring the explicit definition of the two spaces: the genotype space ϕ_g, and the phenotype space ϕ_p. The genetic operators are applied to solutions x, on ϕ_g. Each x is mapped to a program X, in the phenotype space, via a mapping function f_g, such that $f_g(x) \rightarrow X$. Finally, the quality of each program X is measured by a fitness function f, on ϕ_p: $f(X) : \phi_p \rightarrow \mathbb{R}$.

Distances

The genotypic distance, d^g, is the Hamming distance for integer representations, *i.e.*, the distance between two arbitrary solutions $x, y \in \phi_g$ is defined as the total number of positions in which they differ.

In GE the phenotypes are derivation trees, allowing the application of the edit distance to measure the phenotypic distance, d^p, between two programs. The edit distance calculates the minimum number of elemental operations required to transform one tree into the other. There are three elemental operations:

1. **Deletion**: A node is removed from the tree;
2. **Insertion**: A node is added;
3. **Replacement**: The label of a node is modified.

All operations have unitary cost. The tree edit measure has been widely used to estimate the similarity between trees in GP [Brameier and Banzhaf, 2002, Keller and Banzhaf, 1996, O'Reilly, 1997, Rothlauf and Oetzel, 2006].

Mutation Innovation

Let x be a solution in the genotype space and x^m the solution that results from the application of one mutation to x. Let $X, X^m \in \phi_p$ be the programs mapped by x and x^m, respectively. MI is the phenotypic distance between programs X and X^m:

$$MI = d^p(X, X^m) \tag{6.1}$$

The distribution of the MI variable discloses several important features concerning locality and redundancy. The application of a locally strong operator implies that a small modification in the genotype (such as the one originated by one mutation), should result in a limited phenotypic change, *i.e.*, the edit distance between the two involved programs should be small. On the contrary, mutation operators with weak locality induce large phenotypic jumps, compromising search space exploitation.

When $MI = 0$, the mutation was not able to modify the phenotype of an individual. The frequency of these events helps to gain insight into the redundancy level of the representation.

Crossover Innovation

Crossover plays an important role in mixing up relevant blocks of solutions, thereby fostering the appearance of novel strategies. When using crossover, an offspring x^c is created from two parent solutions x^{p1}, x^{p2} (without loss of generality, in our analysis we disregard the second solution that results from the application of crossover). Let X^c, X^{p1} and $X^{p2} \in \phi_p$ be the corresponding phenotypes. CI measures the phenotypic distance between an offspring and its phenotypically closer parent:

$$CI = min(d^p(X^c, X^{p1}), d^p(X^c, X^{p2})) \tag{6.2}$$

If $CI = 0$, then $X^c = X^{p1}$ or $X^c = X^{p2}$, indicating that crossover was not able to create a phenotypically different solution. Moreover, it is expected that CI is directly related to the distance $d^g(x^{p1}, x^{p2})$ between parents, *i.e.*, dissimilar parents should increase the likelihood of generating innovative solutions.

6.4.2 Results

This section presents a comparative study between SGE and standard GE, to better understand why SGE exhibits an enhanced performance. We rely on the previously defined MI and CI measures to estimate the locality and redundancy levels of both representations and consider a recursive grammar regularly used for symbolic regression problems (Grammar 6.5). To compute the phenotypic distances, we adopt the dynamic programming approach proposed by Zhang et al. [Zhang and Shasha, 1989].

$$N = \{expr, op, pre\_op, var\}$$
$$T = \{sin, cos, exp, log, +, -, *, /, x, 1.0, (,)\}$$
$$S = \{start\}$$

And the production set P is:

$\langle start \rangle$ ::= $\langle expr \rangle$
$\langle expr \rangle$ ::= $\langle expr \rangle \langle op \rangle \langle expr \rangle$ | ($\langle expr \rangle$) | $\langle pre\_op \rangle (\langle expr \rangle)$ | $\langle var \rangle$
$\langle op \rangle$ ::= + | - | * | /
$\langle pre\_op \rangle$ ::= sin | cos | exp | log
$\langle var \rangle$ ::= x | 1.0

Grammar 6.5: Grammar used in the locality analysis.

A few settings must be defined for each representation: standard GE requires 128 codons and the maximum number of wraps is set to 3, whereas the maximum recursion level of SGE is 6. These are the same settings that were adopted in the experimental analysis reviewed in Section 6.3.

Redundancy

To estimate redundancy we count how many genotypic variations result in $MI = 0$ or $CI = 0$, *i.e.*, do not lead to phenotypic modifications. The first set of experiments considers the application of one mutation to a randomly generated solution. This process was repeated 10000 times. Results reveal that, for GE, approximately 90.3% of the mutations do not lead to a phenotypic change, whereas in SGE this percentage drops to about 40%. The GE redundancy is in line with the values presented in [Rothlauf and Oetzel, 2006] and the disparity between the two representations is explained by the different mapping strategies: SGE eliminates all redundancy that results from the modulo rule, as there is a direct relation between the values inside the genes' lists and the alternative derivation options of the corresponding non-terminal.

To complement the analysis, we also measured the redundancy level after a sequence of mutations is applied to a solution. In concrete, we consider a random walk where we depart from a random solution and iteratively apply a mutation event. At each step, the

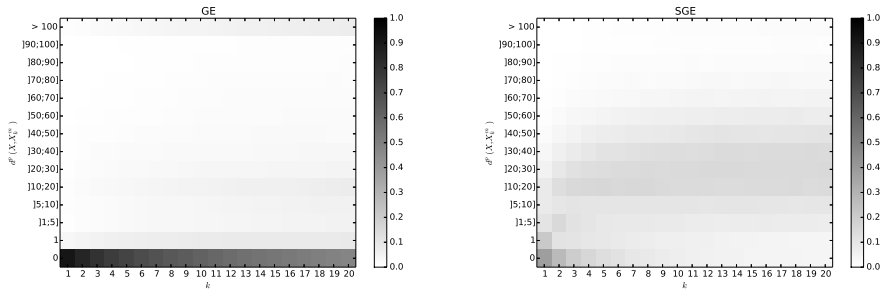
phenotypic distance between the original solution and the current mutant is calculated. Results presented are averages of 10000 random walks with $k = 20$ steps. To simplify the analysis, the phenotypic distances between the original solution and the successive mutants are grouped in different sets. Given a d^p distance between two solutions, the set D_i to which it is assigned, is determined in the following way: $\{D0 : d^p = 0; D1 : d^p = 1; D2 : 1 < d^p \leq 5; D3 : 5 < d^p \leq 10; D4 : 10 < d^p \leq 20; D5 : 20 < d^p \leq 30; D6 : 30 < d^p \leq 40; D7 : 40 < d^p \leq 50; D8 : 50 < d^p \leq 60; D9 : 60 < d^p \leq 70; D10 : 70 < d^p \leq 80; D11 : 80 < d^p \leq 90; D12 : 90 < d^p \leq 100; D13 : d^p \geq 100; \}$.

The boundaries selected for the intervals were adjusted in such a way that they help to emphasize the distribution of distances. The two panels of Fig. 6.12 contain the distribution of distances (vertical axis) over the 20 steps of the random walk (horizontal axis): panel (a) displays results obtained with standard GE, while panel (b) shows the outcomes of SGE. For each mutation step k, the gray shaded square represents the percentage of k-mutated individuals, whose phenotypic distance to the original solutions falls in that specific distance set. The darker the squares, the higher the percentage of individuals in that situation.

For GE, the application of a single mutation leads to a situation where about 90.3% of the individuals belong to the set $D0$, 4.5% belong to $D1$, 2.6% belong $D2$, and 1.1% belongs to $D3$, and so on. Results from panel (a) reveal that GE redundancy decreases, as mutations gradually start to accumulate. Nevertheless, nearly half of the random walks end in solutions whose phenotypic distance to the starting point is still 0, *i.e.*, 20 mutations were not enough to obtain a different derivation tree. It is obvious that the absolute values are dependent on several design options (e.g., the number of codons in the genotype). However, these results confirm that standard GE is vulnerable to extremely high level of redundancy, which clearly compromises search efficiency.

The evolution of the SGE redundancy levels is displayed in panel (b) of Fig. 6.12. The general trend is more in accordance to what is expected when mutations start to accumulate. After one step, around 40% of mutants remain identical to the original solutions, but redundancy gradually drops as successive mutations are applied and, when $k = 20$, the percentage is less then 10%. Moreover, the plot shows that the phenotypic distances are scattered across the defined sets, suggesting that mutation allows SGE to perform a meaningful exploration of the search space.

The CI measure helps to estimate how crossover impacts redundancy. In this exper-



(a) GE (b) SGE

Figure 6.12: Distribution of the phenotypic distances between an original solution and mutants iteratively generated over a random walk with 20 steps: (a): GE; (b): SGE. Results are averages of 10000 runs.

iment, one parent x^{p1} is randomly generated and kept unchanged throughout all the test. The second parent x^{p2} is obtained from the first by the application of $k > 0$ successive mutations. At each step, crossover is performed, one of the offspring is randomly chosen, and the corresponding CI is measured. The process ends after 20 steps. All results presented are averages of 10000 runs. Fig. 6.13 displays the probability of creating a phenotypically identical offspring ($CI = 0$), as the genotypic distance between parents increases (horizontal axis). For both representations, $P(CI_k = 0)$ naturally decreases with increasing k. However, for all k values, SGE clearly achieves higher innovation levels, thus fostering the discovery of novel solutions. Besides the existence of the modulo rule, another plausible explanation for the lower innovation of standard GE is related to the, possibly large, non-expressed area of its genotype. When the cut point falls in this area, the expressed phenotype will not change and the crossover operation will have no immediate impact in the discovery of novel solutions.

Locality

We will now focus in the subset of situations where $d^p > 0$, *i.e.*, where the application of variation operators originates a different phenotype. In Fig. 6.14 we present the phenotypic distances between an original random solution and mutants iteratively generated. Results are averages of 10000 random walks composed just by effective mutations ($MI > 0$). To support the analysis, the horizontal line in the chart displays an estimate

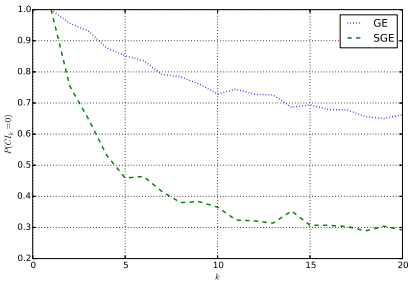


Figure 6.13: Evolution of $P(CI = 0)$, as the genotypic distance between parents increases. Results are averages of 10000 runs.

of the phenotypic distance between two random solutions (averaged over 10000 random pairs of individuals). Generically, as mutations accumulate, the resulting phenotypes gradually diverge from the derivation tree of the original solution. There are, however, important differences in what concerns the locality of the two representations. SGE promotes a gradual differentiation along the random walk, as every effective mutation slightly modifies the resulting phenotype. This behavior is illustrative of a representation with high locality and it supports an effective exploitation of neighbor solutions. When $k > 15$, the MI of SGE tends to stabilize. At this point, many individuals have reached the maximum possible size (given the predetermined recursion limit). Thus, most of the new trees are obtained by modifying leaf nodes labels and do not result from structural changes.

By contrast, effective mutations in GE immediately lead to the appearance of highly dissimilar derivation trees. The locality is clearly lower than that of SGE and, a small number of mutations obliterates the connection between solutions. Just after $k = 10$ mutation steps, the phenotypic distance is equivalent to the distance between two random solutions. The combined analysis of this result and the outcome of the previous section reveals that, while most mutations in GE are redundant, the few that are immediately effective compromise a meaningful exploration of the search space. The higher locality of SGE is a direct consequence of the genotype organization. By relying on a one-to-one correspondence between genes and non-terminals, one ensures that a mutation only modifies the derivation options of the mutated non-terminal, leaving the others unchanged. On the contrary, mutations in GE can modify the derivation options of several non-terminals, which may result in a substantially different tree.

To estimate how crossover impacts locality we repeated the experiment where we

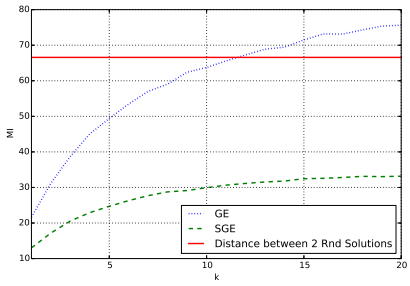


Figure 6.14: Evolution of MI over a random walk with 20 steps, composed just by effective mutations. Results are averages of 10000 runs.

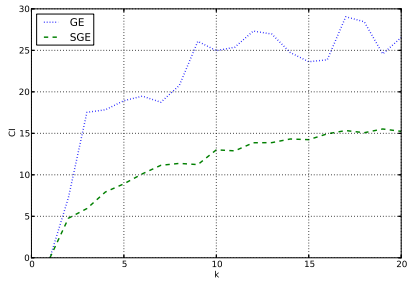


Figure 6.15: Evolution of CI, as the genotypic distance between parents increases. The study considers only situations where $P(CI > 0)$ and results are averages of 10000 runs.

fixed one of the parents and obtained the other by applying successive mutations $k = 1, 2, 3, 4, ...20$, allowing the generation of increasing distant pairs. In this section we ignore the applications of crossover that results in $CI = 0$. Results presented are averages of 10000 repetitions. Fig. 6.15 presents the evolution of CI, as the genotypic distance between parents increases. As expected, for both representations there is a direct correlation between the dissimilarity of parents and the offspring innovation level. In accordance with the results obtained with mutation, SGE promotes a gradual and sustained CI growth, whereas standard GE is unable to avoid an abrupt rise since the first steps of the random walk.

6.5 Design Choices

The analysis from last section helps to unveil why SGE has an increased performance, when compared to the standard GE representation. On the one hand, SGE is able to reduce the extremely high redundancy levels of GE, thus promoting search efficiency. Additionally, it exhibits a balanced locality, which fosters an appropriate sampling of the search space. There are several novel design options that differentiate SGE from the standard GE representation and, to complement the analysis, it is important to fully understand how these different components and settings impact SGE behavior.

The maximum recursion level is a key parameter to define when dealing with recursive grammars, as it determines the maximum tree size. If the limit is too low, then it might be impossible to find the optimal solution. This way the usual option is to define a safe value that allows for some redundancy, at the expense of having a larger search space to explore. Clearly, the definition of this setting is comparable to the specification of the maximum tree size in tree-based GP. To study the influence of this limit, we repeated the MI experiments detailed in section 6.4 with alternative levels of recursion: $\{\mathbf{4}, \mathbf{8}, \mathbf{10}\}$. In what concerns redundancy, the percentage of mutations that do not lead to an immediate phenotypic change ranges between 33% for 4 recursion levels and approximately 50% for 10 recursion levels. As expected, redundancy slightly increases as more recursive productions are added to the grammar, but the values are still clearly below those achieved by standard GE. Fig. 6.16 summarizes the MI evolution for effective mutations, *i.e.*, mutations that immediately change the phenotype. A brief perusal of the results confirms that the recursion level is directly correlated to locality. The general trend is the same for all lines displayed in the chart, but there is a clear hierarchy in what concerns the level of recursion: higher values allow for a more pronounced departure from the starting solution. This is an expected outcome, as higher recursion levels enable the appearance of larger derivation trees, thus creating room for higher degrees of innovation.

Given the MI disparity seen in Fig. 6.16, a question that immediately arises is how the variation in the recursion level, and as a direct consequence, in redundancy and locality, impacts SGE search effectiveness. To estimate this effect, we repeated all optimization experiments described in Section 6.3 with the additional recursion levels. Fig. 6.17 presents the results for the harmonic curve regression problem. The outcomes obtained by SGE with different recursion levels are similar and the small variations are never sta-

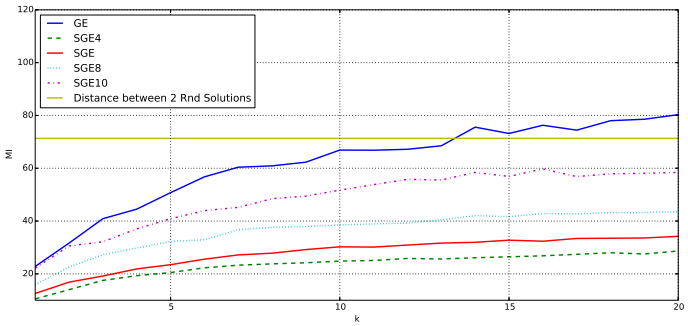
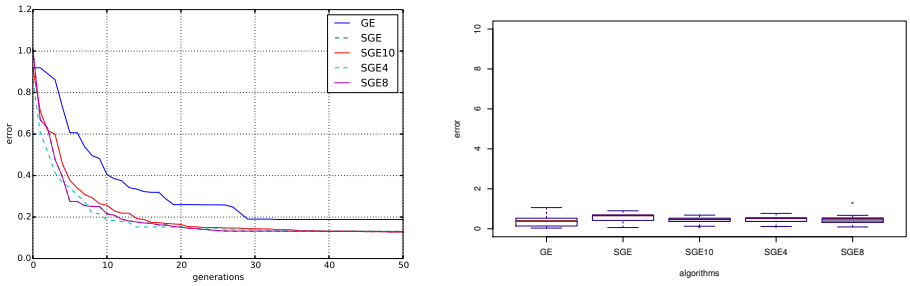


Figure 6.16: Evolution of MI over a random walk with 20 steps, composed just by effective mutations. Different levels of recursion are considered for SGE: $\{\mathbf{4}, \mathbf{6}, \mathbf{8}, \mathbf{10}\}$. Results are averages of 10000 runs.

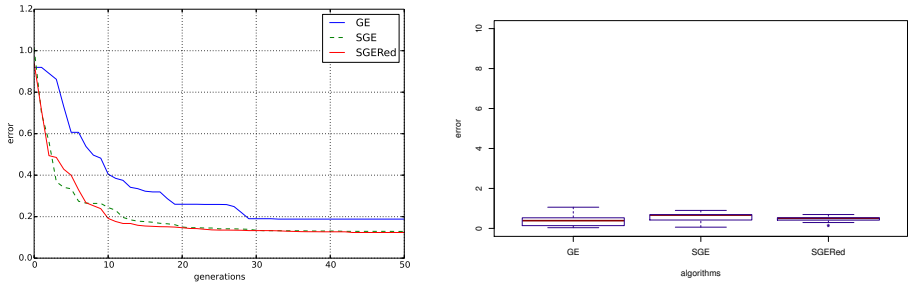


(a) Training Mean Best Fitness (b) Testing Mean Best Fitness

Figure 6.17: Optimization results for the harmonic curve regression problem. Different levels of recursion are considered for SGE. Results are averages of 30 runs.

tistically significant. Results obtained in the other optimization problems selected for this study follow the same trend, suggesting that SGE is robust to moderate variations in the recursion level without compromising effectiveness.

SGE new representation has a dual effect, as it minimizes redundancy while, at the same time, increases locality. We performed an additional test to analyze if any of these features is more important than the other in determining SGE behavior. Redundancy in standard GE is the result of non-expressed codons and of the application of the modulo rule to determine which option to use when expanding a non-terminal. SGE eliminates the last source of redundancy, while it keeps the first (it can also have non-expressed values in its genotype). We can easily create a redundant SGE variant (*SGERed*): when filling



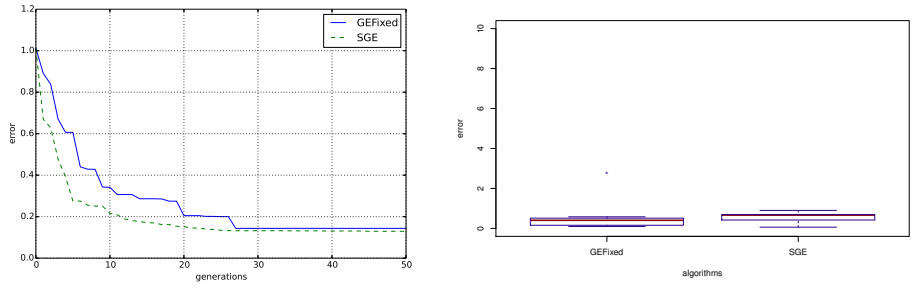
(a) Training (b) Testing

Figure 6.18: Optimization results for the harmonic curve regression problem. A redundant version of SGE is also considered (*SGERed*). Results are averages of 30 runs.

the lists belonging to the genes, integer values randomly drawn from [0,255] are selected and the number of options of the corresponding non-terminal are not considered. This way, SGERed also relies on the modulo rule to select the expansion alternative. Considering 6 recursion levels, SGERed redundancy after the application of one mutation raises to approximately 60%, above the 40% of SGE. The difference between the redundancy exhibited by SGERed and standard GE is now only a consequence of the genome size (for GE) and the recursion level (SGERed). The application of SGERed to an optimization task reveals an interesting feature. Fig. 6.18 displays the outcomes obtained in the harmonic curve regression problem. Results reveal that SGE and SGERed have an identical behavior, suggesting that locality is the main strength of SGE and is the key feature that allows the new representation to have an enhanced performance. The same trend is visible in other optimization problems.

It might be argued that standard GE is unfairly penalized in the analysis, since it does not have an explicit way to perform a meaningful definition of the genome size. If the genotype has too many codons, then redundancy is extremely high. On the contrary, small genotypes heavily rely in wrapping, thus compromising locality. A final set of results reveals that, although this limitation exists, the main weakness of standard GE is its low locality.

We selected the grammar used by SGE for the harmonic curve regression problem after applying the pre-processing step (considering 6 levels of recursion). Given this grammar we can easily define a fixed size for the GE genotype, as there are no recursive



(a) Training (b) Testing

Figure 6.19: Optimisation results for the harmonic curve regression problem. A fixed variant of GE is also considered (*GEFixed*). Results are averages of 30 runs.

productions. Also, wrapping is not needed, since mapping never goes beyond the final codon. This GE variant, which we identify as GEFixed, differs from SGE on two issues: the selection of an option when expanding a non-terminal and the non-existence of a direct correspondence between genes and non-terminals. We recall that these are the two key novel issues addressed in the definition of SGE.

Fig. 6.19 presents the results obtained by SGE and GEFixed in the optimization of the harmonic curve regression. The behavior of GEFixed is slightly better than that of standard GE. However, it is still outperformed by SGE in the training phase (differences are statistically significant). On the testing step, the behaviors are equivalent (just like in Section 6.3). These final results reinforce that the key factor for SGE success is its structured genotypic definition that allows for an increased locality of the representation and for a sustained reduction of redundancy.

6.6 Summary

SGE is a new genotypic representation for GE that explicitly considers the features of the grammar being used. The definition of the genotype requires two pre-processing steps: first, recursive productions are rewritten in a non-recursive format, which requires the addition of several new non-terminals; then, an upper bound for the maximum number of non-terminals expansion is computed. After pre-processing is over, the structured genotype is defined. Each gene links to a specific non-terminal and it encodes a list of

integers that help to determine the derivation options during mapping.

The effectiveness of the new representation was tested on a set of benchmarks and results were encouraging, as SGE was able to outperform standard GE in all selected problems. Moreover, it proved to be efficient, as it needed a lower number of evaluations to discover good quality solutions.

SGE was defined aiming to increase the locality and also at reducing the extremely high redundancy levels that are usually observed in standard GE representations. We considered the framework proposed by Raidl et al. [Raidl and Gottlieb, 2005] that relies on a set of static measures to determine the innovation of variation operators, and thus, estimate the redundancy and locality of a given representation. Empirical results confirm that SGE clearly reduces the redundancy level of standard GE, thus fostering search efficiency. When dealing with recursive grammars, SGE redundancy is directly correlated with the recursion level. However, since SGE does not rely on the modulo rule to perform mapping, redundancy is always clearly lower than the extremely high values exhibited by standard GE.

The locality of SGE was studied by considering the impact of effective genotypic variations in the phenotype. Results show that the new representation exhibits a balanced locality, as the genotypic changes that are iteratively accumulated, promote a gradual modification in the phenotypes. This behavior is in contrast with standard GE, where small changes in the genotype tend to create abrupt modifications in the derivation tree. SGE high locality fosters a meaningful exploration of the search space, providing an explanation for the increased optimization effectiveness. In Section 6.5 we defined several GE and SGE variants and provided a set of complementary results revealing that, although redundancy is useful to enhance search efficiency, the key feature that justifies the SGE optimization behavior is its ability to increase locality.

7

Conclusion and Future Work

"It is good to have an end to journey toward; but it is the journey that matters, in the end"

Ernest Hemingway

In this dissertation we show that it is possible to use an Evolutionary Algorithm as a meta-heuristic to achieve the automatic design of complete Evolutionary Algorithms. We provided evidence that this challenge is not only feasible, but also, under certain conditions, beneficial when compared with human designs. Due to the very nature of the problem, i.e., the fact that the meta-heuristic algorithm has to search the space of all possible evolutionary algorithms looking for promising candidates, we had to solve two major issues. First, we had to study the influence of the learning conditions on the performance of the final outcome, to overcome the tradeoff of overfitting/generalization with an acceptable computational cost. Second, we had to design an appropriate meta-heuristic algorithm to harness the complexity of the search space, proposing an empowered version of the standard Grammatical Evolution algorithm based on a novel representation.

7.1 Main Achievements

Learning is a critical step for the successful design of algorithms, as performed by the framework proposed in this work. Specific conditions are adopted to determine how learning occurs and they are regulated by two contradictory forces that must be carefully balanced: i) They must allow a clear/accurate identification of the most promising optimization strategies; ii) They should be computationally efficient. Learning efficiency was achieved by relying on an inexpensive evaluation of candidate solutions: a single run optimization of a modest size instance, coupled with limited settings for the definition of the population size and number of iterations.

As a rule, results obtained in different scenarios confirm that restricted learning conditions do not compromise the discovery of effective and robust optimization strategies. This is true, both for the evolution of complete algorithms, as well as for the design of specific components.

A detailed analysis of the results from Chapter 4 reveals that the structure of the evolved algorithms is influenced by the conditions found during learning: settings that contribute to premature convergence naturally foster the design of optimization strategies encompassing mechanisms for diversity maintenance. In turn, learning scenarios where convergence is hardly a problem, promote the appearance of agressive search methods that can quickly perform an effective sampling of the search space.

It should be emphasized that the number of iterations granted to the evaluation of the solutions generated by GE is the key parameter when defining specific learning conditions. Results analysed in Chapter 4 clearly reveal that a higher number of generations performed in the offline evaluation promote an accurate discovery of optimization strategies with enhanced robustness and effectiveness.

The two-stage framework has a potential drawback, as strategies learned offline might reveal its brittleness when later applied to unseen situations. The limited computational resources granted to learning might further compromise an effective identification of strategies with better generalization potential. The study presented in Chapter 5 reveals that this is not happening. Despite the simple conditions adopted, there is a strong correlation between learning and validation, as strategies that perform well in learning are also those that exhibit better robustness and generalization ability. These results provide valuable guidelines for HH practitioners, as they confirm the impact of the learning condi-

tions on the structure and behavior of the evolved solutions. Moreover, they reveal that simple learning conditions do not seriously compromise the identification of the algorithmic strategies with the best optimization ability. It is also worth noticing that the learning process seems to be overfitting free. The results presented are still preliminary, but they suggest that the strategies being learned are not becoming overspecialized to the specific conditions found in learning.

Grammatical Evolution is a powerful search engine for HH and promotes the automatic discovery of optimization strategies, competitive with state-of-art hand design approaches. Still, there are reports in the literature describing serious limitations of GE in what concerns its ability to perform a meaningful exploration of the search space (e.g., the works from [Rothlauf and Oetzel, 2006] and the recent work from [Whigham et al., 2015]). By taking this into account we present SGE, a new genetic encoding for GE. The structured organization of this novel representation promotes a higher locality and lower redundancy, thus creating conditions for an enhanced exploration of the space. Results obtained with several well-known GP benchmarks confirm the effectiveness of SGE and the advantages provided over the standard representation and create room for future improvement of GE-based HH.

7.2 Future Work

Even though the work presented in this dissertation sheds some light on how to build better HH frameworks, there is plenty to be done. One way to expand this work is to extend the analysis of the impact of the learning conditions to other parameters, such as the adoption of a single or several training instances with different properties.

When we hand design an EA we can greatly improve its performance by including knowledge about the problem itself in the design (e.g., in the representation and in the variation operators). We may extend our work along the same lines. One possible road to explore is the incorporation of semantic information in the grammar used.

The fact that we rely on a grammar is a positive aspect, for we can constrain the way the search space is explored. Nevertheless, in the context of HH, this can also have the negative effect of limiting the set of possible solutions. So, it will be interesting to explore some form of novelty search, including the possibility of introducing some degree of stochasticity into the grammar.

Another interesting path to follow it the application of SGE to the HH domain as the structured organization of the new representation might further enhance the ability of the framework to discover effective optimization strategies.

Alan Turing believed that by the end of the 20th century there would be machines able to think by themselves. Even though this completely autonomous AI, able to surpass the human intelligence is still fiction, we are making progress towards fully autonomous systems. But there is still plenty of work to be done.

To infinity and beyond
Buzz Lightyear, Toy Story

Bibliography

[Angeline, 1996] Angeline, P. J. (1996). Two self-adaptive crossover operators for genetic programming. In Angeline, P. J. and Kinnear, Jr., K. E., editors, *Advances in genetic programming*, pages 89–109. MIT Press.

[Archetti et al., 2007] Archetti, F., Lanzeni, S., Messina, E., and Vanneschi, L. (2007). Genetic programming for computational pharmacokinetics in drug discovery and development. *Genetic Programming and Evolvable Machines*, 8(4):413–432.

[Azad, 2003] Azad, R. M. A. (2003). *A Position Independent Representation for Evolutionary Automatic Programming Algorithms - The Chorus System*. PhD thesis, University of Limerick.

[Bäck, 1994] Bäck, T. (1994). Parallel optimization of evolutionary algorithms. In *Parallel Problem Solving from Nature - PPSN III*, volume 866 of *Lecture Notes in Computer Science*, pages 418–427. Springer Berlin / Heidelberg.

[Banzhaf et al., 1998] Banzhaf, W., Nordin, P., Keller, R. E., and Francone, F. D. (1998). *Genetic programming: an introduction*, volume 1. Morgan Kaufmann San Francisco.

[Beyer and Schwefel, 2002] Beyer, H. and Schwefel, H. (2002). Evolution strategies: A comprehensive introduction. *Natural Computing*, 1(1):3–52.

[Brameier and Banzhaf, 2001] Brameier, M. and Banzhaf, W. (2001). Effective linear genetic programming. Technical Report Reihe CI 108/01, SFB 531, Department of Computer Science, University of Dortmund, 44221 Dortmund, Germany.

[Brameier and Banzhaf, 2002] Brameier, M. and Banzhaf, W. (2002). Explicit control of diversity and effective variation distance in linear genetic programming. In *Genetic Programming*, volume 2278 of *Lecture Notes in Computer Science*, pages 37–49. Springer Berlin Heidelberg.

[Brameier and Banzhaf, 2007] Brameier, M. F. and Banzhaf, W. (2007). *Linear genetic programming*. Springer Science & Business Media.

[Burke et al., 2012] Burke, E., Hyde, M., and Kendall, G. (2012). Grammatical evolution of local search heuristics. *IEEE Transactions on Evolutionary Computation*, 16(3):406–417.

[Burke et al., 2010a] Burke, E., Hyde, M., Kendall, G., Ochoa, G., Ozcan, E., and Woodward, J. (2010a). A classification of hyper-heuristic approaches. *Handbook of Metaheuristics*, 57:1–21.

[Burke et al., 2010b] Burke, E., Hyde, M., Kendall, G., and Woodward, J. (2010b). A genetic programming hyper-heuristic approach for evolving 2-d strip packing heuristics. *IEEE Transactions on Evolutionary Computation*, 14(6):942–958.

[Byrne et al., 2009] Byrne, J., O'Neill, M., and Brabazon, A. (2009). Structural and nodal mutation in grammatical evolution. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation (GECCO)*, pages 1881–1882, New York, NY, USA.

[Byrne et al., 2010] Byrne, J., O'Neill, M., McDermott, J., and Brabazon, A. (2010). An analysis of the behaviour of mutation in grammatical evolution. In *Genetic Programming*, volume 6021 of *Lecture Notes in Computer Science*, pages 14–25. Springer Berlin Heidelberg.

[Castle and Johnson, 2010] Castle, T. and Johnson, C. (2010). Positional effect of crossover and mutation in grammatical evolution. In *Genetic Programming*, volume 6021 of *Lecture Notes in Computer Science*, pages 26–37. Springer Berlin Heidelberg.

[Chu and Beasley, 1998] Chu, P. C. and Beasley, J. E. (1998). A genetic algorithm for the multidimensional knapsack problem. *Journal of Heuristics*, 4(1):63–86.

[Cowling et al., 2001] Cowling, P. I., Kendall, G., and Soubeiga, E. (2001). A Hyper-Heuristic Approach to Scheduling a Sales Summit. In *Selected papers from the Third International Conference on Practice and Theory of Automated Timetabling III*, pages 176–190, London, UK, UK. Springer-Verlag.

[Davis, 1989] Davis, L. (1989). Adapting Operator Probabilities in Genetic Algorithms. In *Proceedings of the 3rd International Conference on Genetic Algorithms (ICGA)*, pages 61–69. Morgan Kaufmann.

[de Sá and Pappa, 2013] de Sá, A. G. C. and Pappa, G. L. (2013). Towards a method for automatically evolving bayesian network classifiers. In *Proceedings of the 15th annual Conference companion on Genetic and Evolutionary Computation (GECCO)*, pages 1505–1512. ACM.

[Dempsey et al., 2009] Dempsey, I., O'Neill, M., and Brabazon, A. (2009). *Foundations in Grammatical Evolution for Dynamic Environments*, volume 194. Springer.

[Denzinger et al., 1997] Denzinger, J., Fuchs, M., and Fuchs, M. (1997). High Performance ATP Systems by Combining Several AI Methods. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 102–107. Morgan Kaufmann Publishers Inc.

[Dioşan and Oltean, 2009] Dioşan, L. and Oltean, M. (2009). Evolutionary Design of Evolutionary Algorithms. *Genetic Programming and Evolvable Machines*, 10(3):263–306.

[Dorigo and Stützle, 2004] Dorigo, M. and Stützle, T. (2004). *Ant Colony Optimization*. Bradford Company, Scituate, MA, USA.

[Dorndorf and Pesch, 1995] Dorndorf, U. and Pesch, E. (1995). Evolution based learning in a job shop scheduling environment. *Computers and Operations Research*, 22(1):25–40.

[Drechsler et al., 1996] Drechsler, R., Göckel, N., and Becker, B. (1996). Learning Heuristics for OBDD Minimization by Evolutionary Algorithms. In *Parallel Problem Solving from Nature - PPSN IV*, pages 730–739.

[Eberhart et al., 2001] Eberhart, R. C., Shi, Y., and Kennedy, J. (2001). *Swarm Intelligence*. Morgan Kaufmann, 1 edition.

[Edmonds, 2001] Edmonds, B. (2001). Meta-Genetic Programming: Co-evolving the operators of variation. *Turkish Journal Electric Engineering*, 9(1):13–29.

[Eiben et al., 1999] Eiben, A., Hinterding, R., and Michalewicz, Z. (1999). Parameter Control in Evolutionary Algorithms. *IEEE Transactions on Evolutionary Computation*, 3(2):124 –141.

[Eiben and Smith, 2003] Eiben, A. E. and Smith, J. E. (2003). *Introduction to Evolutionary Computing*. Springer-Verlag.

[Fagan, 2013] Fagan, D. (2013). *An analysis of genotype-phenotype mapping in Grammatical Evolution*. PhD thesis, University College Dublin.

[Fagan et al., 2010] Fagan, D., O'Neill, M., Galván-López, E., Brabazon, A., and McGarraghy, S. (2010). An Analysis of Genotype-Phenotype Maps in Grammatical Evolution. In *Genetic Programming*, volume 6021 of *Lecture Notes in Computer Science*, pages 62–73.

[Fang et al., 1993] Fang, H., Ross, P., and Corne, D. (1993). A Promising Genetic Algorithm Approach to Job-Shop Scheduling, Rescheduling, and Open-Shop Scheduling Problems. In *Proceedings of the Fifth International Conference on Genetic Algorithms (ICGA)*, pages 375–382. Morgan Kaufmann.

[Field, 2009] Field, A. (2009). *Discovering statistics using SPSS*. Sage publications.

[Fisher and Thompson, 1963] Fisher, H. and Thompson, G. L. (1963). Probabilistic learning combinations of local job-shop scheduling rules. *Industrial scheduling*, 3:225–251.

[Floreano and Mattiussi, 2008] Floreano, D. and Mattiussi, C. (2008). *Bio-Inspired Artificial Intelligence: Theories, Methods, and Technologies*. The MIT Press.

[Fogel et al., 1966] Fogel, L. J., Owens, A. J., and Walsh, M. J. (1966). *Artificial Intelligence through Simulated Evolution*. John Wiley, New York, USA.

[Gonçalves and Silva, 2013] Gonçalves, I. and Silva, S. (2013). Balancing Learning and Overfitting in Genetic Programming with Interleaved Sampling of Training Data. In *Genetic Programming*, volume 7831 of *Lecture Notes in Computer Science*, pages 73–84. Springer Berlin Heidelberg.

[Gottlieb, 2001] Gottlieb, J. (2001). On the Feasibility Problem of Penalty-Based Evolutionary Algorithms for Knapsack Problems. In *Applications of Evolutionary Computing*, volume 2037 of *Lecture Notes in Computer Science*, pages 50–59. Springer Berlin Heidelberg.

[Gottlieb and Eckert, 2000] Gottlieb, J. and Eckert, C. (2000). A comparison of two representations for the fixed charge transportation problem. In *Parallel Problem Solving from Nature PPSN VI*, volume 1917 of *Lecture Notes in Computer Science*, pages 345–354. Springer Berlin Heidelberg.

[Gottlieb and Raidl, 2000] Gottlieb, J. and Raidl, G. (2000). Characterizing locality in decoder-based eas for the multidimensional knapsack problem. In *Artificial Evolution*, volume 1829 of *Lecture Notes in Computer Science*, pages 38–52. Springer Berlin Heidelberg.

[Grefenstette, 1986] Grefenstette, J. (1986). Optimization of control parameters for genetic algorithms. *IEEE Transactions on Systems Man and Cybernetics*, 16(1):122 –128.

[Harper, 2012] Harper, R. (2012). Spatial co-evolution: quicker, fitter and less bloated. In *Proceedings of the 14th annual Conference on Genetic and Evolutionary Computation (GECCO)*, pages 759–766. ACM.

[Holland, 1975] Holland, J. H. (1975). *Adaptation in natural and artificial systems*. University of Michigan Press, Ann Arbor.

[Hugosson et al., 2010] Hugosson, J., Hemberg, E., Brabazon, A., and O'Neill, M. (2010). Genotype representations in grammatical evolution. *Applied Soft Computing*, 10(1):36 – 43.

[Jones, 1994] Jones, T. (1994). A description of holland's royal road function. *Evolutionary Computation*, 2(4):409–415.

[Jones and Forrest, 1995] Jones, T. and Forrest, S. (1995). Fitness distance correlation as a measure of problem difficulty for genetic algorithms. In *Proceedings of the 6th International Conference on Genetic Algorithms (ICGA)*, pages 184–192, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Keijzer et al., 2002] Keijzer, M., O'Neill, M., Ryan, C., and Cattolico, M. (2002). Grammatical evolution rules: The mod and the bucket rule. In *Genetic Programming*, volume 2278 of *Lecture Notes in Computer Science*, pages 123–130. Springer Berlin Heidelberg.

[Keller and Banzhaf, 1996] Keller, R. and Banzhaf, W. (1996). Genetic programming using genotype-phenotype mapping from linear genomes into linear phenotypes. In *Proceedings of the 1st Annual Conference on Genetic Programming*, pages 116–122, Cambridge, MA, USA. MIT Press.

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

[Kramer and Koch, 2007] Kramer, O. and Koch, P. (2007). Self-adaptive partially mapped crossover. In *Proceedings of the 9th annual Conference on Genetic and Evolutionary Computation (GECCO)*, pages 1523–1523, New York, NY, USA. ACM.

[Kruisselbrink et al., 2011] Kruisselbrink, J. W., Li, R., Reehuis, E., Eggermont, J., and Bäck, T. (2011). On the log-normal self-adaptation of the mutation rate in binary search spaces. In *Proceedings of the 13th annual Conference on Genetic and Evolutionary Computation (GECCO)*, pages 893–900, New York, NY, USA. ACM.

[Langdon and Harman, 2013] Langdon, W. B. and Harman, M. (2013). Optimising existing software with genetic programming. *IEEE Transactions on Evolutionary Computation*, 19(1):33–41.

[Lohn et al., 2005] Lohn, J. D., Hornby, G. S., and Linden, D. S. (2005). An evolved antenna for deployment on nasa's space technology 5 mission. In *Genetic Programming Theory and Practice II*, pages 301–315. Springer.

[López-Ibánez et al., 2011] López-Ibánez, M., Dubois-Lacoste, J., Stützle, T., and Birattari, M. (2011). The irace package: Iterated racing for automatic algorithm configuration. Technical report, IRIDIA.

[Lourenço et al., 2014] Lourenço, N., Pereira, F. B., and Costa, E. (2014). Learning selection strategies for evolutionary algorithms. In *Artificial Evolution*, Lecture Notes in Computer Science, pages 197–208. Springer International Publishing.

[Lourenço et al., 2015a] Lourenço, N., Pereira, F. B., and Costa, E. (2015a). An inquiry into the properties of structured grammatical evolution. Technical report, ECOS-CISUC.

[Lourenço et al., 2015b] Lourenço, N., Pereira, F. B., and Costa, E. (2015b). The optimization ability of evolved strategies. In *17th Portuguese Conference on Artificial Intelligence (EPIA-2015)*, Lecture Notes in Computer Science. Springer International Publishing.

[Lourenço et al., 2012] Lourenço, N., Pereira, F., and Costa, E. (2012). Evolving evolutionary algorithms. In *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference companion*, pages 51–58. ACM.

[Lourenço et al., 2013] Lourenço, N., Pereira, F. B., and Costa, E. (2013). The importance of the learning conditions in hyper-heuristics. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pages 1525–1532. ACM.

[Machado and Cardoso, 2002] Machado, P. and Cardoso, A. (2002). All the truth about nevar. *Applied Intelligence*, 16(2):101–118.

[Marmion et al., 2013] Marmion, M.-E., Mascia, F., López-Ibánez, M., and Stützle, T. (2013). Automatic design of hybrid stochastic local search algorithms. In *Hybrid Metaheuristics*, pages 144–158. Springer.

[Marshall et al., 2014] Marshall, R. J., Johnston, M., and Zhang, M. (2014). Developing a hyper-heuristic using grammatical evolution and the capacitated vehicle routing problem. In *Simulated Evolution and Learning*, volume 8886 of *Lecture Notes in Computer Science*, pages 668–679. Springer International Publishing.

[Martin and Tauritz, 2013] Martin, M. A. and Tauritz, D. R. (2013). Evolving black-box search algorithms employing genetic programming. In *Proceedings of the 15th Annual Conference companion on Genetic and Evolutionary Computation (GECCO)*, pages 1497–1504. ACM.

[McKay et al., 2010] McKay, R. I., Hoai, N. X., Whigham, P., Shan, Y., and O'Neill, M. (2010). Grammar-based Genetic Programming: a survey. *Genetic Programming and Evolvable Machines*, 11(3-4).

[Merz and Freisleben, 2001] Merz, P. and Freisleben, B. (2001). Memetic algorithms for the traveling salesman problem. *Complex Systems*, 13(4):297–346.

[Michalewicz, 1996] Michalewicz, Z. (1996). *Genetic Algorithms + Data Structures = Evolution Programs (3rd ed.)*. Springer-Verlag, London, UK, UK.

[Miller and Thomson, 2000] Miller, J. F. and Thomson, P. (2000). Cartesian genetic programming. In *Genetic Programming*, volume 1802 of *Lecture Notes in Computer Science*, pages 121–132. Springer Berlin Heidelberg.

[Mitchell et al., 1991] Mitchell, M., Forrest, S., and Holland, J. H. (1991). The royal road for genetic algorithms: Fitness landscapes and ga performance. In *Proceedings of the First European Conference on Artificial Life*, pages 245–254. MIT Press.

[Oltean, 2002] Oltean, M. (2002). Multi expression programming. Technical report, Babeş-Bolyai University, Cluj-Napoca, Romania.

[Oltean, 2005] Oltean, M. (2005). Evolving evolutionary algorithms using linear genetic programming. *Evolutionary Computation*, 13(3):387–410.

[Oltean, 2007] Oltean, M. (2007). Evolving evolutionary algorithms with patterns. *Soft Computing*, 11(6):503–518.

[Oltean and Groşan, 2003] Oltean, M. and Groşan, C. (2003). Evolving Evolutionary Algorithms Using Multi Expression Programming. In *Advances in Artificial Life*, volume 2801 of *Lecture Notes in Computer Science*, pages 651–658. Springer Berlin / Heidelberg.

[O'Neill and Brabazon, 2006] O'Neill, M. and Brabazon, A. (2006). Grammatical differential evolution. In Arabnia, H. R., editor, *Proceedings of the 2006 International Conference on Artificial Intelligence (ICAI)*, volume 1, pages 231–236.

[O'Neill and Ryan, 2003] O'Neill, M. and Ryan, C. (2003). *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language*. Kluwer Academic Publishers, Norwell, MA, USA.

[O'Reilly, 1997] O'Reilly, U. M. (1997). Using a distance metric on genetic programs to understand genetic operators. In *IEEE International Conference on Systems, Man, and Cybernetics*, volume 5, pages 4092–4097.

[O'Sullivan and Ryan, 2002] O'Sullivan, J. and Ryan, C. (2002). An Investigation into the Use of Different Search Strategies with Grammatical Evolution. In *Genetic Programming*, volume 2278 of *Lecture Notes in Computer Science*, pages 268–277. Springer Berlin Heidelberg.

[O'Neill and Brabazon, 2006] O'Neill, M. and Brabazon, A. (2006). Grammatical swarm: The generation of programs by social programming. *Natural Computing*, 5(4):443–462.

[O'Neill et al., 2004] O'Neill, M., Brabazon, A., Nicolau, M., McGarraghy, S., and Keenan, P. (2004). pigrammatical evolution. In *Proceedings of the 6th Annual Conference on Genetic and Evolutionary Computation (GECCO)*, volume 3103, pages 617–629. Springer Berlin Heidelberg.

[O'Neill et al., 2008] O'Neill, M., Hemberg, E., Gilligan, C., Bartley, E., McDermott, J., and Brabazon, A. (2008). Geva - grammatical evolution in java (v 2.0). Technical report, Technical Report, UCD School of Computer Science.

[Pappa and Freitas, 2009] Pappa, G. L. and Freitas, A. (2009). *Automating the Design of Data Mining Algorithms: An Evolutionary Computation Approach*. Springer Publishing Company, Incorporated, 1st edition.

[Poli et al., 2005a] Poli, R., Chio, C. D., and Langdon, W. B. (2005a). Exploring extended particle swarms: A genetic programming approach. In *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation (GECCO)*, pages 169–176. ACM Press.

[Poli et al., 2005b] Poli, R., Langdon, W. B., and Holland, O. (2005b). Extending Particle Swarm Optimisation via Genetic Programming. In *Genetic Programming*, volume 3447 of *Lecture Notes in Computer Science*, pages 291–300. Springer Berlin Heidelberg.

[Poli et al., 2008] Poli, R., Langdon, W. B., and McPhee, N. F. (2008). *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`. (With contributions by J. R. Koza).

[Raidl and Gottlieb, 1999] Raidl, G. R. and Gottlieb, J. (1999). On the importance of phenotypic duplicate elimination in decoder-based evolutionary algorithms. In *Late Breaking Papers at the Genetic and Evolutionary Computation Conference (GECCO)*, pages 204–211.

[Raidl and Gottlieb, 2005] Raidl, G. R. and Gottlieb, J. (2005). Empirical analysis of locality, heritability and heuristic bias in evolutionary algorithms: A case study for the multidimensional knapsack problem. *Evolutionary Computation*, 13(4):441–475.

[Rothlauf, 2003] Rothlauf, F. (2003). On the locality of representations. In *Proceedings of the 5th Annual Conference on Genetic and Evolutionary Computation (GECCO)*, volume 2724 of *Lecture Notes in Computer Science*, pages 1608–1609. Springer Berlin Heidelberg.

[Rothlauf, 2006] Rothlauf, F. (2006). *Representations for genetic and evolutionary algorithms*. Springer.

[Rothlauf and Oetzel, 2006] Rothlauf, F. and Oetzel, M. (2006). On the Locality of Grammatical Evolution. In *Genetic Programming*, volume 3905 of *Lecture Notes in Computer Science*, pages 320–330. Springer Berlin Heidelberg.

[Ryan et al., 2002] Ryan, C., Azad, A., Sheahan, A., and O'Neill, M. (2002). No Coercion and No Prohibition, a Position Independent Encoding Scheme for Evolutionary Algorithms – The Chorus System. In *Genetic Programming*, volume 2278 of *Lecture Notes in Computer Science*, pages 131–141. Springer Berlin Heidelberg.

[Ryan et al., 1998] Ryan, C., Collins, J., and O'Neill, M. (1998). Grammatical evolution: Evolving programs for an arbitrary language. In *Lecture Notes in Computer Science*

1391, Proceedings of the First European Workshop on Genetic Programming, pages 83–95. Springer-Verlag.

[Schwefel, 1981] Schwefel, H. P. (1981). *Numerical Optimization of Computer Models*. John Wiley & Sons, Inc., New York, NY, USA.

[Schwefel, 1993] Schwefel, H. P. (1993). *Evolution and Optimum Seeking: The Sixth Generation*. John Wiley & Sons, Inc., New York, NY, USA.

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

[Smit and Eiben, 2010] Smit, S. K. and Eiben, A. E. (2010). Beating the 'world champion'evolutionary algorithm via revac tuning. In *IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8. IEEE.

[Smith and Eiben, 2009] Smith, S. and Eiben, A. (2009). Comparing parameter tuning methods for evolutionary algorithms. In *IEEE Congress on Evolutionary Computation (CEC)*, pages 399 –406.

[Spears, 1995] Spears, W. M. (1995). Adapting crossover in evolutionary algorithms. In *Proceedings of the Fourth Annual Conference on Evolutionary Programming*, pages 367–384. MIT Press.

[Spector and Robinson, 2002] Spector, L. and Robinson, A. (2002). Genetic programming and autoconstructive evolution with the push programming language. *Genetic Programming and Evolvable Machines*, 3(1):7–40.

[Tavares et al., 2004] Tavares, J., Machado, P., Cardoso, A., Pereira, F. B., and Costa, E. (2004). On the evolution of evolutionary algorithms. In *Genetic Programming*, volume 3003 of *Lecture Notes in Computer Science*, pages 389–398. Springer Berlin / Heidelberg.

[Tavares and Pereira, 2010] Tavares, J. and Pereira, F. B. (2010). Evolving strategies for updating pheromone trails: A case study with the tsp. In *Parallel Problem Solving from Nature - PPSN XI*, volume 6239 of *Lecture Notes in Computer Science*, pages 523–532. Springer Berlin Heidelberg.

[Tavares and Pereira, 2011a] Tavares, J. and Pereira, F. B. (2011a). Designing pheromone update strategies with strongly typed genetic programming. In *Genetic Programming*, volume 6621 of *Lecture Notes in Computer Science*, pages 85–96. Springer Berlin Heidelberg.

[Tavares and Pereira, 2011b] Tavares, J. and Pereira, F. B. (2011b). Towards the development of self-ant systems. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation (GECCO)*, pages 1947–1954, New York, NY, USA. ACM.

[Tavares and Pereira, 2012] Tavares, J. and Pereira, F. B. (2012). Automatic Design of Ant Algorithms with Grammatical Evolution. In *Genetic Programming*, volume 7244 of *Lecture Notes in Computer Science*, pages 206–217. Springer Berlin Heidelberg.

[Thorhauer and Rothlauf, 2014] Thorhauer, A. and Rothlauf, F. (2014). On the locality of standard search operators in grammatical evolution. In *Parallel Problem Solving from Nature - PPSN XIII*, volume 8672 of *Lecture Notes in Computer Science*, pages 465–475. Springer International Publishing.

[Wah et al., 1995] Wah, B. W., Ieumwananonthachai, A., Chu, L., and Aizawa, A. N. (1995). Genetics-based learning of new heuristics: Rational scheduling of experiments and generalization. *IEEE Transactions on Knowledge and Data Engineering*, 7(5):763–785.

[Whigham, 1995] Whigham, P. (1995). Grammatically-based genetic programming. In *Proceedings of the workshop on genetic programming: from theory to real-world applications*, pages 33–41.

[Whigham et al., 2015] Whigham, P. A., Dick, G., Maclaurin, J., and Owen, C. A. (2015). Examining the "Best of Both Worlds" of Grammatical Evolution. In *Proceedings of the 2015 on Genetic and Evolutionary Computation Conference (GECCO)*, pages 1111–1118, New York, NY, USA. ACM.

[White et al., 2013] White, D., McDermott, J., Castelli, M., Manzoni, L., Goldman, B., Kronberger, G., Jaskowski, W., O'Reilly, U. M., and Luke, S. (2013). Better gp benchmarks: community survey results and proposals. *Genetic Programming and Evolvable Machines*, 14(1):3–29.

[Wolpert and Macready, 1997] Wolpert, D. H. and Macready, W. G. (1997). No Free Lunch Theorems for Optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82.

[Woodward and Swan, 2011] Woodward, J. R. and Swan, J. (2011). Automatically designing selection heuristics. In *Proceedings of the 13th Annual Conference companion on Genetic and Evolutionary Computation (GECCO)*, pages 583–590. ACM.

[Woodward and Swan, 2012] Woodward, J. R. and Swan, J. (2012). The automatic generation of mutation operators for genetic algorithms. In *Proceedings of the 14th Annual Conference companion on Genetic and Evolutionary Computation (GECCO)*, pages 67–74. ACM.

[Zhang and Shasha, 1989] Zhang, K. and Shasha, D. (1989). Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal on Computing*, 18(6):1245–1262.

Appendices

A

Knapsack Problem Data

Listing A.1: Script used to generate the Knapsack instances used in the experiments.

```
# R --slave --no-restore --file=GenerateInstances.R --args 0.8 2 8 0 output.txt
args <- commandArgs(TRUE)
# correlation between table contributions
rho <- as.numeric(args[1])
# number of objectve functions
M <- as.numeric(args[2])
# size of the bit string
N <- as.numeric(args[3])
# cardinality
K <- as.numeric(args[4])
# seed number
s <- as.numeric(args[5])
# file name
fileName <- args[6]
# bound
bound <- 1000 - 100
library(MASS)
# seed the random generator number
set.seed(s)
# generation of matrix correlation
C       <- matrix(rep(rho,M), M, M)
diag(C) <- rep(1, M)
R <- 2 * sin(pi / 6 * C)
# pdf of multivariate normal law
```

```
25          fim <- pnorm(mvrnorm(n= N, mu = rep(0, M), Sigma = R))
26          # output
27          f <- file(fileName, open = "w")
28          cat(N, "\n", file = f)
29          cat(K, "\n", file = f)
30          for(i in seq(1, N)) {
31            for(m in seq(1,M)) {
32              #val = fim[i,m]*(bound-1)+1
33              val = 100 + fim[i,m]*(bound-1)+1
34              val2 = formatC(val, format = "f", digits = 0)
35              cat(val2, "", file = f)
36            }
37            cat("\n", file = f)
38          }
39          close(f)
```