THEORY GROUNDED DESIGN OF GENETIC PROGRAMMING AND PARALLEL EVOLUTIONARY ALGORITHMS

by

ANDREA MAMBRINI

A thesis submitted to
The University of Birmingham
for the degree of
DOCTOR OF PHILOSOPHY

School of Computer Science
College of Engineering and Physical Sciences
The University of Birmingham
April 2015

Abstract

Evolutionary algorithms (EAs) have been successfully applied to many problems and applications. Their success comes from being general purpose, which means that the same EA can be used to solve different problems. Despite that, many factors can affect the behaviour and the performance of an EA and it has been proven that there isn't a particular EA which can solve efficiently any problem. This opens to the issue of understanding how different design choices can affect the performance of an EA and how to efficiently design and tune one. This thesis has two main objectives. On the one hand we will advance the theoretical understanding of evolutionary algorithms, particularly focusing on Genetic Programming and Parallel Evolutionary algorithms. We will do that trying to understand how different design choices affect the performance of the algorithms and providing rigorously proven bounds of the running time for different designs. This novel knowledge, built upon previous work on the theoretical foundation of EAs, will then help for the second objective of the thesis, which is to provide theory grounded design for Parallel Evolutionary Algorithms and Genetic Programming. This will consist in being inspired by the analysis of the algorithms to produce provably good algorithm designs.

ACKNOWLEDGEMENTS

Firstly I want to acknowledge my parents. I couldn't have written this thesis without them supporting and encouraging me to pursue good education.

I am thankful to my supervisor, Prof. Xin Yao, for having given me the opportunity to start a PhD at the University of Birmingham together with the resources and the freedom necessary to produce quality research.

I want to express my gratitude to Dirk Sudholt for all the work we did together on parallel evolutionary algorithms. He also had a fundamental role as a co-supervisor throughout my PhD and he gave me a lot of suggestions to improve the final version of this thesis. I also want to thank Alberto Moraglio for having introduced me to Geometric Semantic Genetic Programming and to have given me the chance to work with him on many aspects of it, which are discussed in this thesis.

I am also thankful to the other co-authors of my publications: Luca Manzoni and Dario Izzo. Particularly Dario was the one who firstly introduced me to evolutionary computation when I worked with him on PaGMO before starting my PhD journey. I am also thankful to Carsten Witt, who helped me to prepare my first research proposal after I met him at Danmarks Tekniske Universitet, while I was an exchange student there. A special thanks to Peter Oliveto: unfortunately we never had the chance to work together but all the interesting discussions we had (mostly in front of a beer) have really broadened my knowledge on the field of evolutionary computation.

I strongly believe that quality work comes from an happy and fulfilling life and from a stimulating working environment. I want to thank all the people I met at the School of Computer Science in Birmingham which made these conditions possible. Particularly, among them: Antonio Lima, Luca Canzian, Luca Rossi, Manlio De Domenico, Olle Fredriksson, Thiago Silva and Veljko Pejovic.

Finally I want to thank all the people I met at conferences and during my visits at European Space Agency, Nanjing University and Honda Research Institute, who contributed to my personal and professional growth.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

PUBLICATIONS

This work is based on the following publications:

1. Andrea Mambrini, Dirk Sudholt, and Xin Yao. Homogeneous and heterogeneous island models for the set cover problem. In *12th international conference on Parallel Problem Solving from Nature*, PPSN'12, pages 11–20, 2012

2. Alberto Moraglio, Andrea Mambrini, and Luca Manzoni. Runtime analysis of mutation-based geometric semantic genetic programming on boolean functions. In *Foundations of Genetic Algorithms XII, FOGA'13*, pages 119–132. ACM, 2013

3. Andrea Mambrini, Luca Manzoni, and Alberto Moraglio. Theory-laden design of mutation-based geometric semantic genetic programming for learning classification trees. In *IEEE Congress on Evolutionary Computation*, pages 416–423. IEEE, 2013

4. Alberto Moraglio and Andrea Mambrini. Runtime analysis of mutation-based geometric semantic genetic programming for basis functions regressions. In *Genetic and Evolutionary Computation Conference, GECCO'13*, pages 989–996. ACM, 2013

5. Andrea Mambrini and Luca Manzoni. A comparison between Geometric Semantic GP and Cartesian GP for Boolean functions learning. In *Genetic and Evolutionary Computation Conference*, GECCO '14, pages 143–144, 2014

6. Andrea Mambrini and Dirk Sudholt. Design and analysis of adaptive migration intervals in parallel evolutionary algorithms. In *Genetic and Evolutionary Computation Conference*, GECCO '14, pages 1047–1054, 2014

Publication number 4 was nominated for a best paper award in the track Genetic Programming at GECCO'13, and publication 6 won a best paper award in the track Parallel Evolutionary Systems at GECCO'14. An extended version of 6 was invited and submitted to a Special Issue on "Theory of Evolutionary Computation" of the Evolutionary Computation Journal (ECJ).

Other publications produced during the PhD and not included in this thesis:

7. Andrea Mambrini and Dario Izzo. Pade: A parallel algorithm based on the MOEA/D framework and the island model. In *Parallel Problem Solving from Nature - PPSN XIII*, pages 711–720, 2014

CHAPTER 1

INTRODUCTION AND MOTIVATIONS

Evolutionary computation is a subfield of Artificial Intelligence which tries to solve computational problems mimicking Darwin's principle of survival of the fittest. Given a computational problem, an evolutionary algorithm (EA) samples an *initial population* as a set of points from the solution space. Each element of the population is referred to as an *individual.* It then applies *variation operators* to the initial population to produce new individuals and uses a *selection strategy* to select which individuals to include in the *offspring population.* It then applies again variation and selection using the offspring population as a base. The process of variation and selection, which produces an offspring population is called a *generation.* An evolutionary algorithm runs for many generations until a *stopping criterion* is satisfied. The stopping criteria can depend on the number of generations already computed or on some property of the population that is eventually achieved. Algorithm 1 provides a schematic of a general Evolutionary Algorithm.

Algorithm 1 An Evolutionary Algorithm

```
initialise the population $P$
g = 0
while (stopping criteria(P,g) == false) do
    N = variation(P) (generate new individuals trough variation)
    O = selection(P, N) (select the offspring population)
    P = O (set the current population as the offspring)
    g = g + 1 (increase the generations count)
end while
```

One of the most appealing characteristic of Evolutionary Algorithms is that they usually are *general purpose*, which means they are not tailored on a specific problem. It is thus possible to solve many different problems using the same algorithm. Despite this generality, many factors like the population size, the variation and selection method and the representation of individuals, can affect the behaviour and the performance of an evolutionary algorithm, leading to different types of evolutionary algorithms. So a natural question would be: what is the best evolutionary algorithm? In 1996 Yao wrote the following:

> *The answer to the question which EA is the best, is problem dependent. There is no universally best algorithm which can achieve the best result for all problems. One of the challenges to researchers is to identify and characterise which EA is best suited for which problem* [96].

That happens because of the No Free Lunch Theorem (NFL), which states that all the search heuristics are equivalent when averaging on *all* the problems their expected time to solve a problem [94].

Thus, despite EAs being general purpose algorithms they still need proper design and tuning to perform well on specific problems. Good design requires a way to be evaluated, so it is crucial to have a methodology to assert the performance of an algorithm. This has been done using different approaches.

The most common one consists in testing an EA against some benchmark problems. The idea behind this approach is that the NFL theorem applies when averaging the performances on all possible problems, but just a subset of them are of practical interest. So, benchmark problems should be representative enough of the subset of “problems of practical interest” and, assuming the performance of an EA being similar when tested on similar problems, it would be possible to generalise that an EA performing well on a set of well designed benchmarks would also perform well on real applications. Unfortunately this locality of performance assumption cannot be taken as a general true statement. Moreover, since this approach doesn’t give any insight on why a particular approach has good or poor performances, the search for a performing design is usually based on trial and error, and consisting in trying to blindly modify different parts of the algorithm until good performances on desired benchmarks are achieved. Automatic tuning [24, 46] can automatise this process and it is a very useful method to tune parameters of existing algorithms on a specific instance of a problem. Despite that, it still consists in trying different variants of an algorithm on benchmark problems and it is thus still open to the issue of non-locality of performances discussed above when used as a general design tool.

Another approach is to provide claims on performances that are mathematically proven, which usually consists in providing upper and lower bounds of the expected time an EA takes to solve a specific problem together with a mathematical proof for that. That is usually hard to do and requires to model the behaviour of an EA and to develop techniques to prove runtime results. Despite that the idea developed a lot in the last 20 years [69, 4, 27, 68]. Together with being rigorous this approach also allows to get insights on how evolutionary algorithms work and how different aspects of them affect their performances on specific problems. This opens the possibility to do design guided by theory, that is to make design choices inspired by the analysis of the algorithm, that provably lead to good performance.

The theoretical foundation of evolutionary algorithms is a relatively young field and much work has still to be done. Particularly there are many open questions regarding Parallel Evolutionary Algorithms and Genetic Programming. This thesis will have the double aim of advancing the theoretical knowledge about these two topics, filling the existing gaps in the theory, and to use this novel knowledge to guide the design of new efficient algorithms and operators.

Chapter 2 provides a literature review on the theoretical foundation of Genetic Programming and Parallel Evolutionary Algorihtms and highlights the existing gaps.

On Chapters 3, 4 and 5 we will advance the theoretical knowledge about genetic programming providing design and analysis of operators for a novel theory-friendly kind of genetic programming called *Geometric Semantic Genetic Programming*. The analysis and the design provided will range between different class of problems: boolean functions learning (Chapter 3), classification trees learning (Chapter 4) and basis function regression

(Chapter 5).

In Chapters 6 and 7 we will deal with Parallel Evolutionary Algorithms. We will introduce the island model, a common way of parallelising an evolutionary algorithm and in Chapter 6 we will design and analyse a heterogeneous island model finding good approximations for the NP-Hard problem SetCover. Chapter 7 will deal with an open problem in the design of island models, which is how to set the migration interval. We will design two methods to efficiently adapt the migration interval throughout the run, and we will propose an analysis of them on different common test functions.

In Chapter 8 we will finally summarize the contributions of the thesis and provide directions for future work.

CHAPTER 2

LITERATURE REVIEW

2.1 Theoretical foundation of Evolutionary Algorithms

In the previous chapter we explained how, in order to provide provable claims on the performances of evolutionary algorithms, it is necessary to build a mathematical model of the behaviour of an EA and to develop techniques to measure the number of generations that an EA requires to solve a problem (runtime). In this chapter we will give an overview of the history of the theoretical foundation of evolutionary algorithms. Starting from the early attempts with Schema Theory and moving to the recent developments of Runtime Analysis.

2.1.1 Early work

The first attempts to build a theoretical foundation of evolutionary algorithms were concerned with analysing their behaviour rather than their performance. The schema theory was one of the first attempt to analyse evolutionary algorithms [21]. It consists in dividing the search space into subsets (schemata) sharing some syntactic features. Schema theorems describe how the number of individuals belonging to each schemata vary from one generation to the another. Even if it is considered an important framework to understand the behaviour of evolutionary algorithms, it failed to produce quantitative convergence and runtime results.

The main problems of Schema theorems is that they divide individuals into subsets according to their genotype and not taking into consideration their fitness. In this way it is not possible to describe how the fitness of a population (or of the best individual of the population) evolves during time, and thus it is not possible to derive runtime results.

In the nineties Markov Chain theory gave the first convergence results. Rudolph [80] stated that an algorithm which always keeps the best found solution in its population (elitism) and which uses variation operators such that there is a non-zero probability to reach any point in the search space from any other point, will always converge to the optimum. Unfortunately this result does not give any information on the time required to reach it (runtime).

2.1.2 Runtime analysis

After establishing that an EA will converge to the optimum for a specific problem it is interesting to know how long it will take to do so. That will help in the process of choosing the right EA for a particular problem and in understanding how different design choices affect the performance of an EA.

The runtime of an algorithm is traditionally defined as the number of primitive steps to solve a problem. For evolutionary algorithms a good definition of a "primitive step" is a call to the fitness function. The runtime is expressed in asymptotic notation, and since at each generation the number of fitness function calls is usually constant, another common definition of the runtime is the *number of generations to reach the optimum.* Moreover since EAs are randomized algorithms, they will thus perform differently in different runs on the same instance of the problem, we are interested in *expected runtime.*

OneMax, (1+1)-EA and RLS

The first runtime results started to appear in the nineties [64, 79]. OneMax is a simple binary problem in which solutions are bitstring of length n and the fitness of a solution (to maximise) is the total number of 1s in the bitstring. Formally the fitness function for OneMax is:

$$\text{OneMax}(x) = \sum_{i=1}^{n} x_i$$

(1+1)-EA is a simple evolutionary algorithm keeping a population of size 1 (see Algorithm 2). At each generation a new individual is generated flipping each bit of the current individual independently with probability p. Then the fittest between the parent and the offspring is kept for the next generation. In 1998 by Droste, Jansen and Wegener, analysed the runtime of (1+1)-EA (with flipping probability $p = 1/n$) on separable Boolean functions such as OneMax, proving a lower-bound for the runtime of $n \log n - O(n \log \log n)$ and an upper bound of $O(n \log n)$ [17].

Algorithm 2 (1+1)-EA

```
Initialise $x$ with a $n$-bitstring $x \in \{0,1\}^n$
$g = 0$
while $x$ is not optimum do
  Create the offspring $x'$ by flipping each bit of $x$ independently with probability $p$
  if $x'$ has better fitness than $x$ then
    $x = x'$
  end if
  $g = g + 1$ (increase the generations count)
end while
```

Despite this analysis being on a simplified EA and on an easy problem, it inspired a general technique that was used to analyse more complicated EAs on more interesting problems. *Artificial Fitness Levels* consist of partitioning the search space based on the fitness function. Given a fitness function taking m different values, m partitions $A_1, \ldots, A_m$ are created such that each partition contains all the search points with the same fitness and for all the points $a \in A_i$ and $b \in A_j$ it happens that $f(a) < f(b) \Leftrightarrow i < j$. We can then denote as p_i a lower bound for the probability that from a point $x_i \in A_i$

an offspring $x' \in A_{i+1} \cup \ldots \cup A_m$ is produced. Then the expected running time of a stochastic search algorithm that works at each time step with a population of size 1 and produces at each time step a new solution from the current solution is upper bounded by $\sum_{i=1}^{m-1}(1/p_i)$ [68].

It is then very easy to calculate an upper bound for the expected runtime of (1+1)-EA on OneMax in the following way. We partition the search space in $m = n+1$ partitions such that $x \in A_i$ if OneMax$(x) = i$. Then, assuming the current best solution being in A_i, the probability p_i of increasing its fitness is at least the probability of flipping at least a 0-bit without flipping any 1-bit. This probability is then $p_i \geq \frac{n-i}{n}(1-\frac{1}{n})^{n-1} > \frac{n-i}{en}$, where we used the fact that $(1-1/n)^{n-1} > 1/e$ for $n > 1$, being e the Euler's number. The expected runtime is thus $T = \sum_{i=0}^{m-1} 1/p_i \leq \sum_{i=0}^{m-1} \frac{en}{n-i} = \sum_{i=0}^{n-1} \frac{en}{n-i} = en\sum_{i=0}^{n-1} \frac{1}{n-i} = en\sum_{i=1}^{n} \frac{1}{i} = O(n \log n)$, given that $\sum_{i=1}^{n} \frac{1}{i} = O(\log n)$.

The same analysis can be done for Random Local Search (RLS) a variation of (1+1)-EA in which the global mutation operator is replaced by a local one (thus the offspring is always an hamming neighbour of the parent). Random Local Search is described in Algorithm 3. In RLS on OneMax the probability of increasing the fitness at level i is $p_i = \frac{n-i}{n}$, which is the probability of flipping a 0-bit (since there is just one flip we don't need to assure that no 1-bits are flipped). Thus the runtime is $T = \sum_{i=0}^{m-1} 1/p_i = \sum_{i=0}^{m-1} \frac{n}{n-i} = O(n \log n)$.

It has been proved that these bounds are also tight [17] so the expected runtime of (1+1)-EA and RLS on OneMax is actually $\Theta(n \log n)$.

Algorithm 3 Random Local Search (RLS)

```
Inizialise $x$ with a $n$-bitstring $x \in \{0,1\}^n$
$g = 0$
while $x$ is not optimum do
    Create the offspring $x'$ by selecting a random bit of $x$ and flipping it
    if $x'$ has better fitness than $x$ then
        $x = x'$
    end if
    $g = g + 1$ (increase the generations count)
end while
```

We also introduce a lazy version of RLS that we will need in the rest of the thesis. It differs from RLS just for the mutation operator. In lazy-RLS (see Algorithm 4) the offspring is generated by selecting a random bit of the parent and forcing it to 0 with probability 0.5 or to 1 with the same probability.

Algorithm 4 Lazy Random Local Search (lazy-RLS)

```
Inizialise $x$ with a $n$-bitstring $x \in \{0,1\}^n$
$g = 0$
**while** $x$ is not optimum **do**
    Create the offspring $x'$ by selecting a random bit of $x$ and forcing it 0 or 1 with equal probability
    **if** $x'$ has better fitness than $x$ **then**
        $x = x'$
    **end if**
    $g = g + 1$ (increase the generations count)
**end while**
```

The effect of this operator is that, with probability 0.5 the mutation is not changing the parent, while with the same probability the operator of the non-lazy RLS is applied. It is easy to show that, in expectation, the runtime of lazy-RLS is by a factor of 2 slower than the runtime of RLS.

Further Developments

In the following years further techniques have been developed to analyse the complexity of evolutionary algorithms [4]. We have here presented previous work on the runtime analysis of simple EAs using just mutation and having population size equal to one. Afterwards results on more complex EAs performing crossover operations have appeared [85, 29] and the effect of the interaction of different components of the algorithms has been studied [45]. Moreover techniques to analyse population-based EAs have been developed [23] and applied to derive considerations on the effect of the population size on the runtime [13, 14].

Problems more complex than OneMax have also been analysed. More results have been obtained for various classes of problems [92, 63], popular combinatorial problems [68] and some effort has gone into explaining where the hard instances of problems are [22, 12].

2.2 Genetic programming

Genetic programming is a subclass of genetic algorithms in which the individuals are programs or functions (instead of vectors of numbers). It is widely used in practice with applications ranging from symbolic regression to financial trading and arts [74]. Traditionally programs are represented as syntax trees [31], even though other representations have been proposed: for example Linear GP represents programs as sequence of commands in an imperative language[10], while Cartesian GP represents them as circuits [55].

Variations operators act on the representation of the individuals, i.e. their *syntax*. For example subtree-swap crossover, a common crossover operator, gets two parents from a population of programs represented as trees, then it selects a crossover point (a node) in each tree and creates the offspring as a tree obtained copying the first parent and replacing the subtree rooted in the crossover point of the first parent with the subtree rooted in the crossover point of the second parent. A common mutation operator is subtree mutation, which select a random node from the parent and replaces the subtree rooted in that node with a randomly generated subtree.

Selection is done using a fitness functions evaluating the input-output behaviour of the program, i.e. its *semantics*, which usually means that the program is evaluated from the way it responds to a set of possible inputs sampled from the universe of all its possible inputs. This is one of the main differences between evolutionary algorithms and genetic programming. In fact, while individuals in evolutionary algorithms are usually evaluated according to some feature of the structure of the genotype (i.e. their syntax) in genetic programming individuals are evaluated according to their semantics, i.e. by their behaviour when executed on a particular input. Despite that, variation operators in traditional genetic programming still act on the syntax of individuals. This discrepancy is one of the reason why it is harder (compared to evolutionary algorithms) to develop a theoretical framework for GP leading to runtime results, as we will explain in the rest of this chapter.

2.2.1 Theoretical foundation of Genetic Programming

In Section 2.1.2 we explained how knowing how a variation operator affects the fitness of an individual helps in analysing the expected runtime to reach the optimum. Particularly we showed that having a lower bound p_i for the probability that an individual at a fitness level i will produce an offspring with better fitness, it is possible, using the artificial fitness levels method, to calculate an upper bound of the expected runtime to reach the optimum.

Compared to evolutionary algorithms evolving bitstrings, in Genetic Programming it is more difficult to predict how a variation operator affects the fitness of an individual (i.e. the genotype-phenotype mapping is hard). In fact variation operators in GP operate on the syntax of the individuals while the fitness is calculated according to their semantics (this is similar to what happens in Evolutionary Artificial Neural Networks [95] where it is not easy to predict how a variation operator applied to neural network affects its behaviour).

Because of this the work available on theoretical foundations of Genetic Programming is much less advanced than the one for evolutionary algorithms. Moreover the runtime is not the only important factor to measure in GP. Since the individuals in GP can have variable length, analysing how their size grow throughout the run is also important.

In the rest of the section we will first go through the different approaches that have been proposed to explain and analyse the growth in the size of individuals and the runtime to find a good or optimal solution.

Size of individuals

A common phenomenon in GP is that the average size of a program remains almost static for several generations, and then starts to grow rapidly. This phenomenon is called *bloat*.

There is a lack in quantitative theory about why bloat occurs. However some qualitative theories have been produced. The oldest theory is the *protection against crossover* theory [86]. It divides the code of a program into *inactive code*, that is code that is not executed and thus does not affect the output of the program, and *active code*, which is actually executed. The theory states that GP produces some *inactive code* that protects the useful code from being modified by crossover operations and doing so the size of the program grows. This theory is far from being rigorous and doesn't really explain bloat. In fact it is not clear why GP should produce inactive code to "protect" the active one.

Another theory is the *removal bias theory* [84]. It states that inactive code resides in small subtrees and that those subtrees are replaced, by crossover, with bigger inactive subtrees making the program to grow. Still the reason why this happens is not clear.

Finally the *nature of program search spaces theory* [36] states that the distribution of fitness does not vary with size (above a certain size). Since there are more long programs than short ones, the number of long programs of a given fitness is higher than the number of short programs with the same fitness. So GP selects long programs just because there are more of them.

All those theories lack in rigour and in quantitative results. It is not clear after which size the size of the program will start to grow faster, and how to prevent it (some methods try to stop it but can't prevent it from happening [77]). The root problem is that we don't understand enough how GP works. Particularly we don't know how a particular operator affects the size and the fitness of the offspring.

Runtime

As for simpler evolutionary algorithms the first contribution to the theory of GP were based on schema theory. Poli *et al.* firstly provided an analysis considering GP with just one-point crossover [73] and they then extended it to the class of homologous crossover [78] and then to any type of crossover which swaps subtrees [75, 76]. Recently, following what has already been done for genetic algorithms, some work has started on the runtime analysis of GP. Neumann *et al.* [67] and Durret *et al.* [19] present the runtime analysis of mutation-based GP with a tree representation in which trees do not represent functions (i.e., objects that return different output values for different input values) but, rather, structures (i.e., objects whose fitness depends on some structural properties of the tree representation). This deviates quite significantly from the very essence of GP, which is about evolving functions. The Max Problem considered by Kötzing *et al.* [30] is more similar to a real GP since the fitness depends on the computed function. However this dependence is very weak, such that the space of possible inputs can be partitioned into just two subsets such that for every input in a subset, the optimal solution to the problem is the same. In [28] Kötzing *et al.* analyse the learning of a linear function under the PAC-learning framework. However the algorithm analysed is far from a classical GP algorithm since it evolves the coefficients of a fixed size linear function, rather than evolving a function.

Concerning the theoretical foundation of Genetic Programming we have thus two leading theories. The first, based on schema theory and Markov chains try to explain how proper genetic programming (tree based, swap tree crossover) works in general but fails to provide quantitative results on the runtime. The second, inspired by the runtime analysis of evolutionary algorithms, provides runtime results but it is not general enough since it refers to very simplified versions of GP.

2.3 Semantic Genetic Programming

In the previous section we have seen that it is hard to obtain runtime results for genetic programming because it is difficult to predict how a variation operator would affect the fitness of an individual. This happens because the traditional GP operators act on the syntax of the individuals, while the fitness is calculated on the semantics.

In this section we introduce Semantic Genetic Programming, a new kind of GP that will try to tackle this problem, proposing operators that have a predictable behaviour on the semantics of the offspring generated.

We will first go through the early works on semantic Genetic Programming and then we will introduce Geometric Semantic Genetic Programming.

In Chapters 3, 4, 5 we will show successful examples of runtime results obtained for geometric semantic genetic programming, and how these analyses have produced theoretical-driven guidelines to improve the initial algorithms.

2.3.1 Previous attempts of semantic GP

Traditional Genetic Programming searches the space of functions/programs by using search operators that manipulate their syntactic representation, regardless of their actual semantics/behaviour. For instance, subtree swap crossover is used to recombine functions represented as parse trees, regardless of trees representing boolean expressions, mathematical functions, or computer programs. Although this guarantees that offspring are always syntactically well-formed, there is no reason to believe that such a blind syntactic search can work well for different problems and across domains.

In recent literature, there are a number of approaches that use semantics to guide the search in the attempt to improve on GP with purely syntactic operators, as follows. Many individuals may encode the same function (i.e., they may have the same semantics). It is possible to enforce semantic diversity throughout evolution, by creating semantically unique individuals in the initial population [8, 25], and by discarding offspring of crossover and mutation when semantically coinciding with their parents [9, 7].

The semantics of a program can be directly and uniquely represented by enumerating the input-output pairs making up the computed function, or equivalently, by the vector of all output values of the program for a certain fixed order of all possible input values. Quang Uy *et al.* [88] have proposed a probabilistic measure of semantic distance between individuals based on how their outputs differ for the same set of inputs sampled at random. This distance is then used to bias semantically the search operators: mutation rejects offspring that are not sufficiently semantically similar to the parent; crossover chooses only semantically similar subtrees to swap between parents.

Geometric crossover and geometric mutation [62, 57] are formal and representation-independent search operators that can be, in principle, specified to any search space and representation, once a notion of distance between individuals is provided. Simply stated, the offspring of geometric crossover are in the segment between parents, and the offspring of geometric mutation are in a ball around the parent, w.r.t. the considered distance. Many crossover and mutation operators across representations are geometric operators (w.r.t. some distance). Krawiec *et al.* [32, 35] have used a notion of semantic distance to propose a crossover operator for GP trees that is approximately a geometric crossover in the semantic space (i.e., a geometric semantic crossover). The operator was implemented approximately by using the traditional sub-tree swap crossover, generating a large number of offspring, and accepting only those offspring that were sufficiently "semantically intermediate" with respect to the parents. An analogous approach can be used to implement a geometric semantic mutation, with offspring lying in a small ball around the parent in the semantic space. Krawiec *et al.* [33] propose a semantic crossover-like operator that, given a pair of parents, finds a semantically intermediate procedure from a previously prepared library and in [34] they show how this operator leads to significant increase in search performance when compared to standard subtree-swapping crossover.

Whereas the semantically aware methods above are promising, as they have been shown to be better than traditional GP on a number of benchmark problems, their implementations are very wasteful as heavily based on trial-and-error: search operators are implemented via acting on the syntax of the parents to produce offspring, which are accepted only if some semantic criterion is satisfied. More importantly from a theoretical perspective, these implementations do not provide insights on how syntactic and semantic searches relate to each other.

Geometric Semantic Genetic Programming (GSGP) introduced by Moraglio *et al.* [59, 58] is a form of genetic programming that uses geometric semantic crossover and geometric semantic mutation to search *directly* the semantic space of functions/programs. This is possible because, seen from a geometric viewpoint, the genotype-phenotype mapping of GP becomes surprisingly easy, and allows us to derive explicit algorithmic characterizations of geometric semantic operators for different domains following a simple formal recipe, which was used to derive specific forms of GSGP for a number of classic GP domains (i.e, Boolean functions, arithmetic functions and classifiers).

The fitness landscape seen by the geometric semantic operators is *always* a cone by construction, as the fitness of an individual is its semantic distance to the optimum (i.e., its fitness is the distance between its output vector and the output vector of the target function). This has the consequence that GP search on functions with geometric semantic operators is formally *equivalent* to a GA search on the corresponding output vectors with standard crossover and mutation operators. For example, for Boolean functions, geometric semantic GP search is equivalent to GA search on binary strings on the OneMax landscape, for *any* Boolean problem.

The equivalence between GSGP and GA is very attractive from a theoretical point of view as it opens the way to a rigorous theoretical analysis of the optimisation time of GSGP by simply reusing known runtime results for GAs on OneMax-like problems. This analysis obtained is general, as it applies to all problems of a certain domain (e.g., all Boolean functions are seen as OneMax by GSGP).

In the rest of the chapter we will formally define geometric semantic genetic programming and we will propose a formal recipe to design GSGP operators for different domains.

2.3.2 Geometric Semantic Genetic Programming

In the following we describe the GSGP framework formally reporting and expanding on the relevant results from [59].

A search operator $CX : S \times S \to S$ is a *geometric crossover* w. r. t. the metric d on S if for any choice of parents p_1 and p_2, any offspring $o = CX(p_1, p_2)$ is in the segment

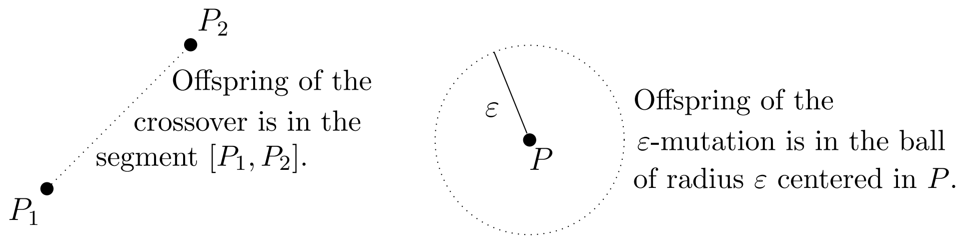

P_2
Offspring of the crossover is in the segment $[P_1, P_2]$.
P_1
ε
P
Offspring of the ε-mutation is in the ball of radius ε centered in P.

Figure 2.1: A visualization of the geometric crossover (left) and of the geometric mutation (right).

$[p_1, p_2]$ between parents, i.e., it holds that $d(p_1, o) + d(o, p_2) = d(p_1, p_2)$. A search operator $M : S \to S$ is a *geometric ε-mutation* w. r. t. the metric d if for any parent p, any of its offspring $o = M(p)$ is in the ball of radius ε centered in the parent, i.e., $d(o, p) \leq \varepsilon$ (See Fig. 2.1).

Given a fitness function $f : S \to \mathbb{R}$, the geometric search operators induce or see the fitness landscape identified by the triple (f, S, d). Many well-known recombination operators across representations are geometric crossovers [57]. For example, for binary strings, one-point crossover is a geometric crossover w. r. t. Hamming distance (HD), and point mutation is geometric 1-mutation w. r. t. Hamming distance [57].

For many applications, genetic programming can be seen as a supervised learning method. Given a training set made of fixed input-output pairs $T = \{(x_1, y_1), ..., (x_N, y_N)\}$ (i.e., fitness cases), a function $h : X \to Y$ within a certain fixed class H of functions (i.e., the search space specified by the chosen terminal and function sets) is sought that interpolates the known input-output pairs. I.e., for an optimal solution h it holds that $\forall (x_i, y_i) \in T : h(x_i) = y_i$. The fitness function $F_T : \mathrm{H} \to \mathbb{R}$ measures the error of a candidate solution h on the training set T. Compared to other learning methods, two distinctive features of GP are (i) it can be applied to learn a wider set of functions, and (ii) it is a black-box method, as it does not need explicit knowledge of the training set, but only of the errors on the training set.

We define the *genotype-phenotype mapping* as the function $P : \mathrm{H} \to Y^{|X|}$ that maps a representation of a function h (i.e., its genotype) to the vector of the outcomes of the application of the function h to all possible input values in X (i.e., its phenotype), i.e.,

$P(h) = (h(x_1), ..., h(x_{|X|}))$. We can define a *partial* genotype-phenotype mapping by restricting the set of input values X to a given subset X' as follows: $P_{X'} : \mathrm{H} \rightarrow Y^{|X'|}$ with $P_{X'}(h) = (h(x_1), ..., h(x_{|X'|}))$ with $x_i \in X'$. Let $I = (x_1, ..., x_N)$ and $O = (y_1, ..., y_N)$ be the vectors obtained by splitting inputs and outputs of the pairs in the training set T. The output vector of a function h on the training inputs I is therefore given by its partial genotype-phenotype mapping $P_I(h)$ with input domain restricted to the training inputs I, i.e., $P_I(h) = (h(x_1), ..., h(x_N))$. The training set T identifies the partial genotype-phenotype mapping of the optimal solution h restricted to the training inputs I, i.e., $P_I(h) = O$.

Traditional measures of error of a function h on the training set T can be interpreted as distance between the target output vector O and the output vector $P_I(h)$ measured using some suitable metric D, i.e., $F_T(h) = D(O, P_I(h))$ (to minimise). For example, when the space H of functions considered is the class of Boolean functions, the input and output spaces are $X = \{0, 1\}^n$ and $Y = \{0, 1\}$, and the output vector is a binary vector of size $N = 2^n$ (i.e., Y^N). A suitable metric D to measure the error as a distance between binary vectors is the Hamming distance.

We define the *semantic distance* SD between two functions $h_1, h_2 \in \mathrm{H}$ as the distance between their corresponding output vectors measured with the metric D used in the definition of the fitness function F_T, i.e., $SD(h_1, h_2) = D(P(h_1), P(h_2))$. The semantic distance SD is a genotypic distance induced from a phenotypic metric D, via the genotype-phenotype mapping P. As P is generally non-injective (i.e., different genotypes may have the same phenotype), SD is only a pseudometric (i.e., distinct functions can have distance zero). This naturally induces an equivalence relation on genotypes. Genotypes h_1 and h_2 belong to the same semantic class $\overline{h}$ iff their semantic distance is zero, i.e., $h_1, h_2 \in \overline{h}$ iff $SD(h_1, h_2) = 0$. Therefore the set of all genotypes H can be partitioned in equivalence classes, each one containing all genotypes in H with the same semantics. Let $\overline{H}$ be the set of all semantic classes of genotypes of H. The set of semantic classes $\overline{H}$ is by construction in one-to-one correspondence with the set of phenotypes (i.e., output vectors). Then, as

D is a metric on the set of phenotypes, it is naturally inherited as a metric on the set $\overline{H}$ of semantic classes.

We define *geometric semantic crossover and mutation* as the instantiations of geometric crossover and geometric mutation to the space of functions H endowed with the distance SD. E.g., geometric semantic crossover SGX on Boolean functions returns offspring Boolean functions such that the output vectors of the offspring are in the Hamming segment between the output vectors of the parents (w. r .t. all $x_i \in X$). I.e., any offspring function $h_3 = SGX(h_1, h_2)$ of parent functions h_1 and h_2 meets the condition $SD(h_1, h_3) + SD(h_2, h_3) = SD(h_1, h_2)$ which for the specific case of Boolean functions becomes $HD(P(h_1), P(h_3)) + HD(P(h_2), P(h_3)) = HD(P(h_1), P(h_2))$. The geometric crossover SGX can be also seen as a geometric crossover on the space of semantic classes of functions $\overline{H}$ endowed with the metric D. From the definition of SGX above, it is evident that if $h_3 = SGX(h_1, h_2)$ then it holds that $h'_3 = SGX(h'_1, h'_2)$ for any $h'_1 \in \overline{h_1}, h'_2 \in \overline{h_2}, h'_3 \in \overline{h_3}$ because $P(h_1) = P(h'_1), P(h_2) = P(h'_2)$ and $P(h_3) = P(h'_3)$. In words, the result of the application of SGX depends only on the semantic classes of the parents $\overline{h_1}$, $\overline{h_2}$ and not directly on the parents' genotypes h_1, h_2, and the returned offspring can be any genotype h_3 belonging to the offspring semantic class $\overline{h_3}$. Therefore, SGX can be thought as searching directly the semantic space of functions.

2.3.3 Search equivalence: GSGP $\approx$ GA

When the training set covers all possible inputs, the *semantic fitness landscape* seen by an evolutionary algorithm with geometric semantic operators is, from the definition of semantic distance, a particularly nice type of unimodal landscape in which the fitness of a solution is its distance in the search space to the optimum [1] (i.e., a *cone landscape*). This observation is general, as it holds for any domain of application of GP (e.g., Boolean, Arithmetic, Program), any specific problem within a domain (e.g., Parity and Multiplexer

[1]Notice that the optimum in the space of function classes is unique, as the output target vector is unique (since the training set is fixed prior to evolution), and it identifies uniquely the target function class.

problems in the Boolean domain) and for any choice of metric for the error function.

GP search with geometric operators w. r .t. the semantic distance SD on the space of functions H is thus formally equivalent to EA search with geometric operators w. r .t. the distance D on the space of output vectors. This is because: (i) semantic classes of functions are in bijective correspondence with output vectors, as "functions with the same output vector" is the defining property of a semantic class of function; (ii) geometric semantic operators on functions are isomorphic to geometric operators on output vectors, as SD is induced from D via the genotype-phenotype mapping P (see also diagram (2.1) and explanation in Section 2.3.4).

Equivalence example: Boolean Functions

For the specific case of Boolean functions with n input variables and a single output variable, GSGP search with a training set of size $N = 2^n$ encompassing all the possible inputs, is equivalent to GA search with standard mutation and crossover on binary strings of length N. When the training set covers all possible inputs, the fitness landscape seen by the GA is OneMax because minimising the error means minimising the Hamming distance between the output vector of a candidate solutions and the target output vector. That is equivalent to maximising the number of the right outputs, which on binary strings is equivalent to maximising the number of ones. When the training set covers only a subset of all possible inputs, the fitness landscape seen by the GA is OneMax on τ "active" bits that contribute to the fitness, and it is neutral on the remaining bits that do not affect the fitness.

All Boolean functions are seen as equivalent from GSGP search. This is because, whereas any distinct target training set gives rise to a different fitness landscape whose optimum is a different target string, any unbiased black-box search algorithm [44] does not assume a priori the knowledge of the location of the optimum and sees all these landscapes as equivalent.

2.3.4 Construction of GSGP operators: a formal recipe

The commutative diagram below illustrates the relationship between the geometric semantic crossover GX_{SD} on genotypes (e.g., trees) on the top, and the geometric crossover (GX_D) operating on the phenotypes (i.e., output vectors) induced by the genotype-phenotype mapping P, at the bottom. It holds that for any $T1, T2$ and $T3 = GX_{SD}(T1, T2)$ then $P(T3) = GX_D(P(T1), P(T2))$.

$$\begin{tikzcd}
T1 \arrow[d, "P"] & \times & T2 \arrow[d, "P"] \arrow[r, "GX_{SD}"] & T3 \arrow[d, "P"] \\
O1 & \times & O2 \arrow[r, "GX_D"] & O3
\end{tikzcd} \tag{2.1}$$

The problem of finding an algorithmic characterization of geometric semantic crossover can be stated as follows: given a family of functions H, find a recombination operator GX_{SD} (unknown) acting on elements of H that induces via the genotype phenotype mapping P a geometric crossover GX_D (known) on output vectors.

E.g., for the case of Boolean functions (individuals are $h : \{0,1\}^n \to \{0,1\}$) with fitness measure based on Hamming distance, output vectors are binary strings corresponding to the output column of their truth table (i.e. $P(h) \in \{0,1\}^n$ and $P(h) = (h(x_1), \dots, h(x_{2^n})))$. If we want to design a geometric semantic operator GX_{SD} inducing on the output vector a GX_D equivalent to the mask based crossover, we need to design an operator $GX_{SD}(h_1, h_2)$ which combines syntactically the two parent boolean expressions h_1 and h_2 in such a way that the offspring boolean expression $h_3 = GX_{SD}(h_1, h_2)$ has, as output vector, the result of a mask-based crossover of the output vectors of the two parents. More formally that means that GX_{SD} is in such a way that

$$h_3 = GX_{SD}(h_1, h_2) \Rightarrow P(h_3) = GX_D(P(h_1), P(h_2))$$

with GX_D being a mask-based crossover for binary strings.

Note that there is a different type of geometric semantic crossover for each choice of space H and distance D. Consequently, there are different semantic crossovers for different GP domains. In the following chapters (Chapter 3, 4, 5) we will use the formal recipe presented above to design GSGP operators for the domain of boolean functions, classification trees, and real functions, respectively.

2.4 Parallel Evolutionary algorithms

Due to the current development in computer architecture and the steeply rising number of processors in modern devices, parallelisation is becoming a more and more important topic, being a cost-effective approach to solve problems in real time and for tackling large-scale problems.

There are many variants of parallel evolutionary algorithms, from parallelising function evaluations on multiple processors to fine-grained models such as cellular EAs and coarse-grained EAs such as island models [47]. In the latter approach, multiple populations evolve independently for a certain period of time. Every τ generations, for a parameter τ called the *migration interval*, individuals migrate between islands to coordinate their searches. Communication takes place according to a spatial structure, a topology connecting populations. Common topologies include rings, two-dimensional grids/toroids, hypercubes, or complete graphs [87]. Compared to panmictic populations, this decreases the spread of information. A slower spread of information can increase the diversity in the whole system, and by choosing the right topology and the frequency or probability of migration, the communication effort can be tuned. Moreover Island models are popular optimisers for several reasons:

- Multiple communicating populations can make the same progress as a single population in a fraction of the time, speeding up computation.
- Small populations can be simulated faster than large populations, reducing execution times [1].

- Periodic communication only requires small bandwidth, leading to low communication costs.
- Solution quality is improved as different populations can explore different regions of the search space.

The usefulness of parallel populations has been demonstrated in many successful applications ranging from language tagging, circuit design, scheduling and planning to bioinformatics [47, 2].

2.4.1 Theoretical foundation of Parallel EAs

Despite being applied and researched intensively, the theoretical foundation of parallel EAs is still in its infancy. Present theoretical studies include takeover times and growth curves (see, e. g., [81] or [47, Chapter 4]). Recently the expected running time of parallel EAs has been studied, leading to a constructed example where island models excel over panmictic populations [37, 38] and examples where the diversity in island models makes crossover a powerful operator [66]. Also the speedup in island models has been studied rigorously: how the number of generations can be decreased by running multiple islands instead of one. Studies include pseudo-Boolean optimisation [39, 40] and polynomial-time solvable problems from combinatorial optimisation [41].

Design and Adaptive Schemes

Designing an effective parallel evolutionary algorithm can be challenging as the method and amount of communication needs to be tuned carefully. Too frequent communication leads to high communication costs, and it can compromise exploration. Too little communication means that the populations become too isolated and unable to coordinate their searches effectively. There is agreement that even the effect of the most fundamental parameters on performance is not well understood [47, 2]. Skolicki and De Jong [83] investigated the impact of the migration interval and the number of migrants on performance, while different migration policies were compared by Araujo and Merelo [3]. Bravo, Luque and Alba [11] studied the effect of the migration interval when tackling dynamic optimization problems. They found that the dynamic behaviour is not just the result of the number of exchanged individuals, but it results from several phenomena. For frequent migrations the effect of varying the migration interval is much stronger than that of varying the number of migrants. Performance degrades when the number of migrants approaches the population size of islands. And performance may degrade in the presence of large migration intervals if the algorithm stops prematurely.

Osorio, Luque, and Alba [71, 70] presented adaptive schemes for the migration interval, which aim for convergence at the end of the run (for runs of fixed length). The migration interval is set according to growth curves of good individuals and the remaining number of generations; migration intensifies towards the end of a run. They obtained competitive performance results, compared to optimal fixed parameters, for MAX-SAT instances [70].

Finally, Lässig and Sudholt [40] presented schemes for adapting the number of islands during a run of an island model (and offspring populations in a $(1+\lambda)$ EA). Scheme A doubles the number of islands if no improvement was found in a generation. Otherwise, the number of islands drops to one island. Scheme B also doubles the number of islands when no improvement is found, and halves it otherwise. Both schemes achieve optimal or near-optimal parallel running times, while not increasing the total number of function evaluations by more than a constant factor.

2.5 Inadequacies of the state of the art

As stated in the introduction the double aim of the thesis is to advance the theoretical knowledge of Parallel Evolutionary Algorithms and Genetic Programming and to use this novel knowledge to produce new theory grounded efficient designs. In this chapter we have presented a review on the state of the art of the theoretical foundation of Genetic

Programming and Parallel Evolutionary Algorithms. There are many gaps to fill.

Concerning the theoretical foundation of Genetic Programming, we have stated that there are two leading theories. The first, based on schema theory and Markov chains analyses proper Genetic Programming evolving functions (i.e. tree based GP using swap tree crossover) but fails in providing quantitative results on the runtime. The second, inspired to the runtime analysis of evolutionary algorithms, can provide runtime results but it fails in doing that for proper GP, since it takes into consideration problems in which the fitness of an individual does not depend on its computed function (i.e. its semantics) but rather on its structure (i.e. its syntax). Geometric Semantic Genetic Programming (GSGP) gives the opportunity to fix the inadequacies of both the leading theories, thus allowing us to produce analysis leading to runtime results for problems in which individuals are treated as functions rather than as structures. Still GSGP has not been analysed rigorously yet and it also leaves open the problem of designing efficient operators for different domains that are *provably good*. Our contribution in Chapters 3, 4 and 5 will provide rigorous analysis of GSGP and design of provably efficient operators for the domains of boolean functions learning, classification trees learning and basis function regression.

Concerning Parallel Evolutionary algorithms there is no rigorous analysis on how island models perform on NP-hard problems, or how they deal with multi-objective fitness functions. Moreover previous studies took into consideration just homogeneous island in which all islands run the same algorithm and they did not consider heterogeneous models where each island can run different algorithms with different parameters, different operators, and even different fitness functions. This gap will be filled in Chapter 6 where an homogeneous island model in which each island solve a multi-objective formulation of the NP-hard problem SetCover is analysed. The analysis will inspire the design of a more efficient heterogeneous island model which assigns to each island a different single-objective problem representing a subset of the whole search space.

For Parallel Evolutionary algorithms another open problem is how to set parameters.

Particularly the works available on the migration interval are mainly experimental and a rigorous approach analysing schemes to adapt it throughout the run is missing. This gap will be filled in Chapter 7, where two new adaptive schemes for the migration interval are proposed. We will show, through rigorous analysis, that these new schemes can perform as well as a scheme fixing the migration interval to its best value in terms of upper bounds for the parallel runtime, while sometime guaranteeing even better upper bounds for the communication effort.

CHAPTER 3

GEOMETRIC SEMANTIC GENETIC PROGRAMMING FOR BOOLEAN FUNCTIONS

In the previous chapter we introduced geometric semantic genetic programming in a general way. In this chapter we will design geometric semantic operators for the domain of Boolean functions and we will analyse the runtime of GSGP using these operators solving the problem of black box Boolean functions learning.

This chapter will provide the first example in this thesis of theory-driven design. We will first introduce and analyse a point mutation operator (see Definition **??**), which was firstly introduced in [59]. The analysis will show the inadequacies of this operator and will guide towards the design of new *block operators*, which we will prove to be more efficient.

This chapter is based on [61, 49]. My contribution consisted in the design of all the novel mutation operators, in the statements and proofs of Theorems 2, 3, 4, 5 and in the design and implementation of the experiments in Section 3.5 concerning Cartesian GP.

3.1 Black box Boolean learning problem

Given a complete truth table $C = \{(x_1, y_1), ..., (x_N, y_N)\}$ consisting in the complete description of the input-output behaviour of a fixed Boolean function $\overline{h} : \{0,1\}^n \rightarrow \{0,1\}$ in n variables ($N = 2^n$). A training set T consisting in $\tau \leq N$ test cases $T \subset C = \{(x_1, y_1), ..., (x_\tau, y_\tau)\}$ is sampled from the truth table uniformly at random without replacement.

The aim is to use just the training set T to learn a Boolean function $h : \{0,1\}^n \rightarrow \{0,1\}$ matching as well as possible the input-output behaviour described by C.

In this and in the following Chapter our algorithms will try to do that finding an expression matching the input-output behaviour described by training set, thus minimising the *training error*

$$\varepsilon_t(h) = \frac{1}{\tau} \sum_{(x_i,y_i)\in T} I[h(x_i) \neq y_i]$$

where $I[\cdot]$ is the indicator function that is 1 if its inner expression is true and 0 otherwise.

The problem has the additional constraint of being black-box. Which means that the learning algorithm has no direct access to T. It has instead access to an oracle that, given a candidate Boolean expression X will return how well it matches the input-output behaviour described by T.

When $\tau = N$, the training set encompasses all the possible input-output cases. In this situation finding an expression fitting the training set T will also lead to an expression fitting the complete set, and thus the original Boolean function. When, on the other hand, the training set is smaller than the complete set, fitting the training set would lead to a generalization error which can be defined as

$$\varepsilon_g(h) = \frac{1}{N} \sum_{(x_i,y_i)\in C} I[h(x_i) \neq y_i]$$

In the following sections we will consider just the problem of minimizing the training error $\varepsilon_t(h)$, thus we will considered the black box Boolean learning problem "solved" when an expression matching completely the training set is found. We will give considerations on the generalization error in Section 4.5.

3.2 Geometric Semantic Operators for the Boolean domain

As explained in Section 3.2, in order to design geometric semantic operators we need to derive recombination and mutation operators acting on Boolean expressions corresponding to geometric recombination and mutation operator in the output vector. Particularly we will we first present a crossover operator (SGXB) corresponding to a mask-based crossover on the output vectors and a mutation operator (SGMB) corresponding to a forcing point mutation on the output vector. These two operators have been firstly introduced in [59].

Definition 1. ***Boolean semantic crossover:*** *Given two parent functions* $T1, T2 : \{0,1\}^n \to \{0,1\}$*, the recombination SGXB returns the offspring Boolean function* $T3 = (T1 \wedge TR) \vee (\overline{TR} \wedge T2)$ *where* TR *is a randomly generated Boolean function (see Fig. 3.1).*

The random function TR can be generated in many different ways. This gives rise to different implementations of the semantic operator.

Definition 2. ***Forcing Point Mutation:*** *Given a parent function* $T : \{0,1\}^n \to \{0,1\}$*, the mutation SGMB returns the offspring Boolean function* $TM = T \vee M$ *with probability* 0.5 *and* $TM = T \wedge \overline{M}$ *with probability* 0.5 *where* M *is a random minterm of all input variables.*

A minterm is a conjunction of all the input variables that can be negated or not (i.e. given $n = 3$ and an input set $\{X_1, X_2, X_3\}$, a possible minterm is $M = X_1 \wedge \overline{X_2} \wedge X_3$)

Theorem 1. *SGXB is a geometric semantic crossover for the space of Boolean functions with fitness function based on Hamming distance, for any training set and any Boolean problem. SGMB is semantic* 1*-geometric mutation for Boolean functions with fitness function based on Hamming distance.*

The proof of the previous theorem can be found in [59]. In the following, we give an example to illustrate the theorem for the crossover. Let us consider the 3-parity problem, in which we want to find a Boolean function $F(X_1, X_2, X_3)$ that returns 1 when an odd

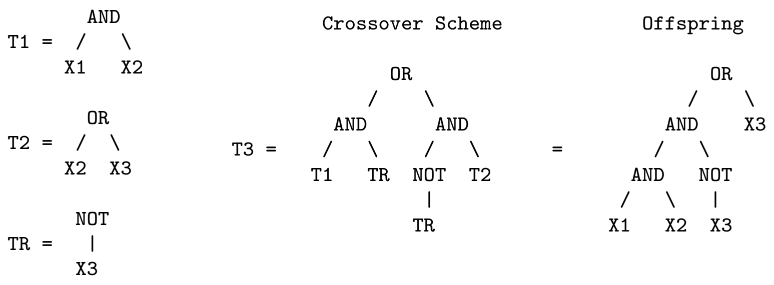


Figure 3.1: T1 and T2 are parent functions and TR is a random function. The offspring T3 is obtained by substituting T1, T2 and TR in the crossover scheme and simplifying algebraically.

number of input variables is 1 and 0 otherwise. Its truth table is in Table 3.1 (first 4 columns). As we have 3 input variables, there are 2^3 possible input combinations (first 3 columns of Table 3.1). In this example, we consider the training set to be made of all 8 entries of the truth table. However, normally the training set comprises only a small subset of all input-output pairs. The target output vector Y is the binary string 01101001 (column 4 of Table 3.1). For each tree representing a Boolean function, one can obtain its output vector by querying the tree with all possible input combinations. The output vectors of the trees in Figure 3.1 are in the last 4 columns of Table 3.1. The fitness $f(T)$ of a tree T (to minimise) is the Hamming distance between its output vector $P(T)$ and the target output vector Y (restricted to the outputs of the training set), e.g., the fitness of parent $T1$ is $f(T1) = HD(P(T1), Y) = HD(00000011, 01101001) = 4$. The semantic distance between two trees $T1$ and $T2$ is the Hamming distance between their output vectors $P(T1)$ and $P(T2)$, e.g., the semantic distance between parent trees $T1$ and $T2$ is $SD(T1, T2) = HD(P(T1), P(T2)) = HD(00000011, 01110111) = 4$. Let us now consider the relations between the output vectors of the trees in Table 3.1. The output vector of TR acts as a crossover mask to recombine the output vectors of $T1$ and $T2$ to produce the output vector of $T3$ (in $P(TR)$, a 1 indicates that $P(T3)$ gets a bit from $P(T1)$ for that position, and 0 that the bit to $P(T3)$ is from $P(T1)$). This crossover on output vectors is a geometric crossover w.r.t. Hamming distance, as $P(T3)$ is in the Hamming segment between $P(T1)$ and $P(T2)$ (i.e., it holds that

Table 3.1: Truth table of 3-parity problem (first 4 columns). Output vectors of trees in Figure 3.1 (last 4 columns): of parents T1 and T2, of random mask TR, and of offspring T3.

X_1	X_2	X_3	Y	$P(T1)$	$P(T2)$	$P(TR)$	$P(T3)$
0	0	0	0	0	0	1	0
0	0	1	1	0	1	0	1
0	1	0	1	0	1	1	0
0	1	1	0	0	1	0	1
1	0	0	1	0	0	1	0
1	0	1	0	0	1	0	1
1	1	0	0	1	1	1	1
1	1	1	1	1	1	0	1

$HD(P(T1), P(T3)) + HD(P(T3), P(T2)) = HD(P(T1), P(T2)))$, as we can verify on the example:
$HD(00000011, 01010111) + HD(01010111, 01110111) = 3+1 = 4 = HD(00000011, 01110111)$. This shows that, in this example, the crossover on trees in Figure 3.1 is a geometric semantic crossover w.r.t. Hamming distance.

Intuitively, the reason the theorem holds in general is that the crossover scheme in Figure 3.1 describes, using the language of Boolean functions, the selecting action of the recombination mask bit on the corresponding bits in the parents to determine the bit to assign to the offspring (i.e., it is a 1-bit multiplexer function piloted by the mask bit).

As the syntax of the offspring of SGMB (mutation) contain one parent plus an additional term, the size of individuals grows *linearly* in the number of generations. On the other hand, for SGXB (crossover), since the syntax of the offspring contains the syntax of both parents, the size of it grows *exponentially* in the number of generations.

3.3 Runtime analysis

In this section we formally analyse the runtime of a mutation only GSGP using the previously introduced Forcing Point Mutation, solving the black box Boolean learning problem. To do that we will rely on the search equivalence described in Section 2.3.3. We will find out the runtime is exponential even when the size of the training set is small.

We will then introduce new operators. The operators are presented divided into two categories: *pointwise mutations operators* can change single entries independently, while *blockwise mutation operators* make the same change to entire blocks of entries. We will show that blockwise mutation operators can be efficient in solving the black box Boolean learning problem when the size of the training set is small enough.

3.3.1 Pointwise Mutation Operators

Let us first consider the case in which the training set encompasses all possible input-output pairs of a Boolean function with n input variables. The size of the output vectors is $N = 2^n$, which is also the number of examples in the training set. A natural definition of *problem size* is the number of input variables n of the functions in the search space. Let us also assume that we have a GSGP (e.g., searching the space of functions represented as trees) that is exactly equivalent to a (1+1) EA (with mutation probability $1/N$) or to a RLS on the space of output vectors. The runtime of GSGP would be the same as of (1+1) EA or RLS on OneMax on the space of output vectors, which is $\Theta(N \log N) = \Theta(n2^n)$ (see Section 2.1.2). This highlights a first issue: although the fitness landscape seen by GSGP is OneMax, since the size of the output vectors N is exponentially long in the problem size n (the truth table of a n-variables Boolean expression has length $N = 2^n$), the runtime of GSGP is exponential in the problem size n.

A second issue is that with an exponential size of the training set in n, each single fitness evaluation takes exponential time as it requires to evaluate all the outputs of a function against the target vector on exponentially many fitness cases. Naturally, in practice, the training set encompasses only a small fraction of all the input-output pairs of a function. To be able to evaluate the fitness of a function in polynomial time the size of the training set needs to be polynomial in n. In the following, we will consider the size of the training set (τ) to be polynomial in n and less than N, thus $\tau = \text{poly}(n) < N$. This transforms the problem seen by the EA on output vectors into a "sparse" version of OneMax in which most of vector entries are neutral (i.e. they don't affect the fitness), and the remaining entries, whose locations in the output vector are unknown, give each a unitary contribution to the fitness. Even with a training set of polynomial size, it is easy to see that both RLS and (1+1) EA (with mutation probability $1/N$) take exponential time to find the optimum. This is because at each generation the probability of mutating a non-neutral position of the output vector is exponentially small (since it is equal to the ratio $\frac{\tau}{N} = \frac{\text{poly}(n)}{2^n}$), hence it takes exponential expected time to get an improvement. What would then be a mutation operator that gives rise to a polynomial runtime on the sparse OneMax problem?

Initialisation operator

Before attempting answering this question, let us consider another issue with semantic mutation. Can we actually implement semantic mutation operators corresponding to (1+1) EA and RLS efficiently? Even implementing the initialisation operator that generates a function uniformly at random in the sematic space takes exponential space and time. That happens due to the fact that doing that is equivalent to sampling a random binary string exponentially long in n. Fortunately, the problem is easily solved by starting from an arbitrary initial solution which admits a short representation (e.g., the True function) rather than from a random one. This does not affect the runtime analysis since the $O(n \log n)$ bound for OneMax holds for any choice of starting point.

3.3.2 Point mutations

The forcing point mutation presented in Section 3.2 is reported below.

Definition 2. ***Forcing Point Mutation*** *Given a parent function $X : \{0,1\}^n \to \{0,1\}$, the mutation returns the offspring Boolean function $X' = X \vee M$ with probability 0.5, and $X' = X \wedge \overline{M}$ with probability 0.5, where M is a random minterm of all input variables.*

This operator forces to a random value exactly one randomly selected entry of the

Table 3.2: First three columns on the left: truth table of 2-parity problem with inputs X_1 and X_2, and output Y. Three rightmost columns: output vectors of the random minterm M, of the parent P and of the produced offspring O obtained by applying the bit-flip point mutation.

X_1	X_2	Y	M	P	O
1	1	0	**1**	**1**	**0**
1	0	1	0	1	1
0	1	1	0	1	1
0	0	0	0	0	0

parent output vector. This is because adding ($\vee$) a minterm to a Boolean expression has the effect of forcing the corresponding single entry in the truth table to 1, and multiplying it ($\wedge$) by a negated minterm forces the corresponding entry to 0. This operator can be also rephrased as: it flips a randomly selected entry with probability 0.5 and it does not change anything with probability 0.5. So this operator is equivalent, on the output vector to lazy-RLS (see Section 2.1.2). The runtime is thus $\Theta(N \log N) = \Theta(n2^n)$ which is exponential in the problem size n.

The semantic mutation below (bit-flip point mutation) induces exactly point mutation on output vectors, so it induces on the output vector a search that is equivalent of that of RLS on OneMax. It is also interesting to notice that this mutation can be seen as crossover of the parent with the negation of itself with a crossover mask which selects all bits but one from the parent.

Definition 3. ***Bit-flip point mutation:*** *Given a parent function* $X : \{0,1\}^n \rightarrow \{0,1\}$ *the mutation returns the offspring Boolean function* $X' = \left(X \wedge \overline{M}\right) \vee \left(M \wedge \overline{X}\right)$*, where* M *is a random minterm of all input variables.*

Seen on the output vectors, this operator flips the output of the parent function corresponding to the combination of the input variables that makes the random minterm M true. Let us illustrate this mutation with an example. Let us consider the 2-parity problem, so $n = 2$ input variables. Its truth table is in Table 3.2, in the first three columns on the left. Let us consider the following application of the bit-flip point mutation operator:

Random minterm:	$\mathrm{M} = X_1 \wedge X_2$
Parent:	$\mathrm{P} = (X_1 \wedge X_2) \vee (X_1 \vee X_2)$
Offspring:	$O = (P \wedge \overline{M}) \vee (\overline{P} \wedge M)$

The three rightmost columns of Table 3.2 report the corresponding output vector view of the application of the bit-flip point mutation above. Note that the output vector of the offspring is obtained by flipping the bit of the output vector of the parent corresponding to the '1' in the output vector of the random minterm (boldface in Table 3.2).

Since this operator is equivalent on the output vector to RLS solving OneMax (see Section 2.1.2), the runtime is $\Theta(N \log N) = \Theta(n2^n)$, which is exponential in the problem size n.

3.3.3 Bitwise mutation operators

Let us now consider bitwise mutation, which flips each bit independently with a certain probability (as in (1+1)-EA). Before considering a semantic mutation that induces bitwise mutation on the output vectors, we show that bitwise mutation with an adequate mutation probability can lead to a polynomial runtime on the sparse OneMax with exponentially long chromosome.

Theorem 2. *On the sparse OneMax problem with an exponentially long chromosome with $N = 2^n$ entries and with $\tau < N$ non-neutral entries, (1+1) EA with bitwise mutation with $p = 1/\tau$ finds the optimum in time $O(\tau \log \tau)$. In particular, when τ is polynomial in n, the runtime is polynomial.*

Proof. We define the potential k as the number of incorrect bits. At each iteration the potential decreases if an incorrect bit is flipped. This happens with probability

$$p_k \geq k\frac{1}{\tau}\left(1 - \frac{1}{\tau}\right)^{\tau-1} > \frac{k}{e\tau}$$

Since the potential can decrease at most τ times and the expected time for the potential to decrease is $1/p_k$, the expected time to reach $k = 0$ (and finding the solution to the problem is)

$$E(T) \leq \sum_{i=1}^{\tau} 1/p_k = \sum_{i=1}^{\tau} \frac{e\tau}{i} = O(\tau \log \tau)$$

$\square$

How can we implement a semantic mutation that corresponds to bitwise mutation with mutation probability p on the output vectors?

Definition 4. ***Bitwise mutation:*** *Given a parent function* $X : \{0,1\}^n \to \{0,1\}$ *the bitwise mutation operator does the following:*

- *Sample an integer number* x *from* $x \sim Bin(p, N)$
- *Generate* x *minterms uniformly at random without repetitions* $\{M_1, \cdots, M_x\}$
- *The offspring is* $X' = \left(X \wedge \overline{M}\right) \vee \left(M \wedge \overline{X}\right)$*, where* $M = M_1 \vee \cdots \vee M_x$.

Unfortunately, also this operator has a problem. Using a probability of mutation $p = \frac{1}{\tau}$, which by theorem 2 makes the runtime of GSGP efficient for a training set size τ polynomial, this implementation becomes exponential in space because the expected number of minterms making up M is pN, which is exponential in n. So, by changing the probability p, what it is gained in terms of runtime of GSGP, it is then lost in terms of efficiency of the implementation of a single application of the mutation operator, and vice versa. The challenge is therefore finding a semantic mutation operator that, at the same time, (i) can be implemented efficiently and (ii) makes the runtime of GSGP polynomial in n for any Boolean problem.

3.3.4 Blockwise Mutations

The challenge is to find an operator flipping more than one bit in the output vector (ideally flipping in expectation N/τ bits as from Theorem 2), while keeping the size of the offspring small.

In this section, we consider four semantic mutations trying to do so, which extend the forcing point mutation introduced in the previous section. That is obtained by replacing the mutating random minterm M with an *incomplete minterm* which may include only a subset of all the input variables. The four mutations differ on the family of incomplete minterms considered and their probability distributions. Using an incomplete, rather than a full, minterm in the forcing mutation has the effect to force more than an entry of the output vector to the same value, i.e., it forces a block of entries to the same value, hence the name block mutations. These operators could work well on the sparse OneMax problem on exponentially long strings. As a first approximation, the idea behind these operators is that, by using sufficiently few variables in the incomplete minterm, the mutation would affect sufficiently many entries of the output vector, so that typically a non-neutral bit will be affected after a single application of the mutation operator. This, in effect, would be equivalent to searching the OneMax problem on the non-neutral entries, which are polynomially many, hence leading to a polynomial optimisation time. However, the analysis of block mutations is complicated by the fact that, unlike traditional mutations, they force *dependencies* between values at different entries of the string, as they cannot act separately on a single entry.

Fixed Block Mutation

Definition 5. ***Fixed Block Mutation (FBM):*** *Let us consider a* fixed *set of $v < n$ variables (fixed in some arbitrary way at the initialisation of the algorithm). FBM draws uniformly at random an incomplete minterm M comprising all fixed variables as a base for the forcing mutation.*

Fixing v of the n input variables induces a *partition* of the output vector into $b = 2^v$ blocks each covering 2^{n-v} entries of the output vector (which has a total of 2^n entries). There is a one-to-one correspondence between the set of all incomplete minterms M made up of the fixed v variables and the set of all the blocks partitioning the output vector. The effect of mutation FBM on the output vector is that of selecting one block uniformly

Table 3.3: Example of FBM. First four columns: truth table of 3-parity problem. Remaining columns: output vectors of the drawn random incomplete minterm $M = X_1 \wedge \overline{X_2}$, of the parent $P = X_2 \vee (X_1 \wedge \overline{X_2} \wedge X_3)$, and of the offspring $O = P \vee M = X_1 \wedge X_2$. Horizontal lines separate blocks of the partition of the output vectors obtained by fixing variables X_1 and X_2.

X_1	X_2	X_3	Y	M	P	O
0	0	0	0	0	0	0
0	0	1	1	0	0	0
0	1	0	1	0	1	1
0	1	1	0	0	1	1
1	0	0	1	**1**	**0**	**1**
1	0	1	0	**1**	**1**	**1**
1	1	0	0	0	1	1
1	1	1	1	0	1	1

at random and forcing all the entries belonging to the block to the same value, 0 or 1, selected at random with equal probability. Table 3.3 shows an example of application of FBM.

GSGP with FBM can only find functions whose output vectors have the same output values for all entries of each block. We call *the search space seen by an operator* (or more simply *the search space of an operator*) the subset of the search space of the problem that can actually be explored by the operator.

Since the search space seen by FBM is smaller than the search space of the problem it happens that for some training sets of some Boolean problems, the optimal function satisfying the training set is not reachable for some choice of the fixed variables.

This happens when at least two training examples with different outputs belong to the same block (e.g., in Table 3.3 the values of the entries in the first block of the optimum output vector Y are 0 and 1. This solution is therefore not reachable as reachable output vectors have both these entries set at zero or at one).

The particular settings of the analysis are as follows. As explained in Section 3.1, we make the standard Machine Learning assumption that the training set T is sampled uniformly at random from the set of all input-output pairs of the Boolean function P at hand (defined by the hidden Boolean function h). The training set is sampled only once, before the search for a function satisfying it has started, and, in particular, it remains unchanged during evolution. Therefore, from an optimisation viewpoint, the training set T defines the specific instance of the Boolean problem P being tackled by the search. The reachability of the optimum is uniquely determined when the training set T of a problem P is fixed, and when also the set V of variables inducing the partition of the output vector is fixed. We are interested in the probability p that GSGP with FBM initialised with an arbitrary fixed set of variables V can reach the optimum on a training set sampled uniformly at random from the problem at hand [1]. Furthermore, when GSGP with FBM can reach the optimum, we are interested in the expected runtime to reach the optimum.

Theorem 3. *For any Boolean problem P, given a training set T of size τ sampled uniformly at random on the set of all input-output pairs of P, GSGP with FBM with an arbitrarily fixed set of v variables finds the optimal Boolean function satisfying the training set T in time $O(b \log b)$, with probability $p \approx e^{-\frac{\tau^2}{2b}}$ for $b \gg \tau$, where $b = 2^v$ is the number of blocks partitioning the output vector.*

Proof. GSGP with FBM can reach the optimum provided that each distinct training example of the sampled training set T belongs to a distinct block. This holds irrespective of the prescribed output of the training examples, hence on any problem P. The optimum can be reached because with each training example in a different block, one application of FBM can flip independently the output value of each training example in the current solution. So, a function with any configuration of the outputs on the training set can be reached by the search.

The τ training examples are sampled uniformly at random on the set of all input-output pairs, i.e., uniformly on the output vector. The output vector is partitioned in b blocks of equal size. The probability of having each training example in a distinct block can be calculated by looking at it as a balls and bins process: blocks are bins of equal

[1]This is different from the traditional notion of probability of success of a search algorithm, as the source of the probabilistic outcome is the randomisation on the sampled problem instance (i.e., sampled training set T from P), and not the randomisation of the search algorithm.

size, training examples are balls thrown uniformly at random on the bins. The probability of having at most one ball in each bin is the probability of throwing the first ball in an empty bin (1) times the probability of throwing the second ball in a bin left empty after the first throw ($(b-1)/b$), and so on, until all balls have been thrown. So we have:

$$P(b,\tau) = \prod_{i=1}^{\tau} \frac{b-(i-1)}{b}$$

When the number of bins is much larger than the number of balls, i.e., $b \gg \tau$, $\frac{i-1}{b}$ is very small, as $e^x \approx 1+x$ for x close to 0, then $P(b,\tau) \approx \prod_{i=1}^{\tau} e^{-\frac{i-1}{b}} = e^{-\frac{\tau(\tau-1)}{2b}} \approx e^{-\frac{\tau^2}{2b}}$.

When each example of the training set belongs to a distinct block, GSGP with FBM, which at each generation creates an offspring by forcing all the entries of an entire block to the same value, can modify independently each single output value of the current solution corresponding to an entry in the training set. This search is therefore equivalent to that of lazy-RLS on binary strings of length b on a sparse OneMax problem with τ non-neutral entries the runtime is thus $O(b\log b)$. In fact at each generation, given i mismatched outputs, the fitness is improved with probability $p_i = \frac{\tau}{b}\frac{i}{\tau}\frac{1}{2}$ which is the probability of selecting a block containing a training point times the probability that training point has a mismatched output, times the probability of forcing that output to the correct value. The runtime then can be easily calculated as $\sum_{i=1}^{\tau} 1/p_i = O(b\log\tau) = O(b\log b)$, since $b \geq \tau$. $\square$

The number of blocks b partitioning the output vector is critical for the performance of the search. On the one hand, from the theorem above, the runtime of GSGP with FBM becomes larger for a larger number of blocks. Therefore, we would like to have as few blocks as possible. On the other hand, the probability of success becomes higher as b gets larger [1]. So, in this respect, the more blocks the better. The number of blocks b is an indirect parameter of the algorithm that can be chosen by choosing the number v of

[1]This can be intuitively understood, as increasing the number of bins (i.e., blocks) while keeping the number of balls unchanged (i.e., training examples) increases the chance of getting each ball in a separate bin.

variables in the initial fixed set, as $b = 2^v$. Furthermore, the number of blocks b should be chosen relative to the size of the training set τ (e.g., we must have $b \geq \tau$ to have enough bins to host all balls individually). The question is therefore if we can always choose the number of blocks relative to the number of training examples such that we have both a polynomial runtime and high probability of success. It turns out that this is always possible, as stated in the theorem below.

Theorem 4. *Let us assume that the size of the training set τ is a polynomial n^c in the number of input variables n, with c a positive constant. Let us choose the number of fixed variables v logarithmic in n such that $v > (2c + \varepsilon)\log_2(n)$ (for some $\varepsilon > 0$ constant in n). Then, GSGP with FBM finds a function satisfying the training set in polynomial time with high probability of success, on any problem P, and training set T uniformly sampled from P.*

Proof. The number of blocks $b = 2^v$ is then a polynomial in n with degree strictly larger than $2c+\varepsilon$, i.e., $b = n^{2c+\varepsilon}$. From Theorem 3, we have a success probability of $p \approx e^{-\frac{\tau^2}{2b}} = e^{-\frac{n^{2c}}{2n^{2c+\varepsilon}}} = e^{-\frac{1}{2}n^{-\varepsilon}}$. As the argument of the exponent approaches 0 as n grows, we can use the expansion $e^x \approx 1 + x$ obtaining $p \approx 1 - \frac{1}{2}n^{-\varepsilon}$, which tells us that the runtime holds with high probability for any $\varepsilon > 0$. Again for Theorem 3 the running time is $O\left(n^{2c+\varepsilon}\log n\right)$, which is polynomial in the size of the problem, assuming ε being constant in n. $\square$

Varying Block Mutation

The Fixed Block Mutation operator is an unnaturally "rigid" operator as it requires to fix a set of variables at the beginning of the search in some arbitrary way (e.g., selecting them at random), which are then used throughout evolution as sole components of the incomplete minterms used in the forcing mutation. On the one hand, this operator is appealing because it fixes a partition structure on the output vectors that remains unchanged during evolution, and that makes it particularly suitable for theoretical analysis. On the other hand, fixing the partition structure may restrict the search space too much and make the optimum function lie outside the search space of the operator, hence unreachable. In the following, we introduce a more "flexible" mutation operator that extends the Fixed Block Mutation, and enlarges the search space seen by it.

Definition 6. ***Varying Block Mutation (VBM):*** *Let $v < n$ be a parameter. VBM draws uniformly at random an incomplete minterm M made of v of the n input variables as a base for the forcing mutation.*

The VBM operator can be thought as constructing an incomplete minterm in two stages. First, it draws uniformly at random v distinct variables from the set of n input variables to form a set V of variables. Then, it draws uniformly at random an incomplete minterm M comprising all variables in V as a base for the forcing mutation. The effect on the output vector of a single application of the VBM operator is, therefore, as follows: first, it draws uniformly at random a partitioning of the output vector with $b = 2^v$ blocks, each covering 2^{n-v} entries of the output vector; then, it selects one block of the current partitioning uniformly at random and it forces all the entries belonging to the block to the same value, 0 or 1, selected at random with equal probability.

Since when using VBM the partition structure on the output vector is not fixed, the search space seen by GSGP with this operator is larger than that with FBM, and in particular, the former search space covers completely the latter. The reason for that is that GSGP with VBM has always a chance to select the fixed variable set used by FBM to feed to VBM, and explore this part of the search space.

We say that an operator is *more expressive* than another when the search space seen by the former includes the search space seen by the latter. In this case VBM is more expressive than FBM. When the optimum of a certain Boolean problem is within the search space of a less expressive operator is also within the search space seen by a more expressive search operator, but the viceversa is not true in general. From its definition, the probability of success of an operator is higher than the probability of success of all operators less expressive than it. We say that an operator is completely expressive when its seen search space covers all solutions of all Boolean problems and all training sets, thus its probability of success is 1. Since VBM is more expressive than FBM its probability of success is higher. However, as with FBM, VBM is also not able to always reach the optimum for any choice of Boolean problem and training set: there exists certain training set configurations such that for all partitioning induced by any choice of v input variables, there are always at least two training examples with different output values belonging to the same block.

Proposition 1. *Consider GSGP with VBM using $v < n$. Then there exists a training set of size $\tau = n + 1$ of the Boolean parity problem on which this algorithm cannot find the optimum.*

Proof. Consider the training set with input entries $\mathcal{T} = \{x \in \{0,1\}^n \mid \exists! i \in \{1, ..., v\}\ x_i = 1\} \cup \{(0, \dots, 0)\}$ and as output values those of the parity problem (i. e. $(0, \cdots, 0)$ has output 0, while all the other vectors in $\mathcal{T}$ have output 1, as in Example 1).

Let M be an incomplete minterm of $v < n$ variables. If the all-zero vector satisfies M, then there exists another row, with output value 1 in $\mathcal{T}$, that satisfy M (again see Example 1). Since VBM forces one whole block (specified by an incomplete minterm) to a specific values, and we have showed that any incomplete minterm selecting the first row (having output 0) must also select another row with output 1, it is not possible to obtain a perfect score on the training set τ. $\square$

Example 1. *We will illustrate the training set of Proposition 1 for $n = 3$ variables. In this case the training set is:*

	v_1	v_2	v_3	$f(v_1, v_2, v_3)$
$x_1 =$	0	0	0	0
$x_2 =$	1	0	0	1
$x_3 =$	0	1	0	1
$x_4 =$	0	0	1	1

where v_1, v_2, v_3 are variables and $x_1, \dots, x_4$ elements of the training set. Note that for any choice of two variables we cannot separate x_1 from another element of the training set: with v_1 and v_2 we cannot separate x_1 and x_4, with v_1 and v_3 we cannot separate x_1 and x_3, and with v_2 and v_3 we cannot separate x_1 and x_2.

We have previously shown that GSGP with FBM, when it can reach the optimum, it finds it, in expected polynomial time. Unfortunately, GSGP with VBM requires superpolynomial time to find the optimum in the worst case.

Proposition 2. *Consider GSGP with VBM using $v = c \log n < n$ variables for some constant $c > 0$. Then there exists a training set of size $\tau = n^c$ of the Boolean parity problem on which this algorithm needs superpolynomial time in n to find the optimum.*

Proof. Consider the training set with input entries $\mathcal{T} = \{x \in \{0,1\}^n \mid \forall i \in \{v+1, \dots, n\}\ x_i = 0\}$ and as output values those of the parity problem (see Example 2). Note that the number of entries in the training set is $\tau = 2^v = n^c$.

For every selection of variables different from the first v variables, any incomplete minterm M made with those variables will be such that the subset $\mathcal{T}'$ of $\mathcal{T}$ of all training instances that satisfy M contains exactly $|\mathcal{T}'|/2$ instances with output value 1 and $|\mathcal{T}'|/2$ instances with output value 0. Since VBM is a forcing mutation, using M as the incomplete minterm for the mutation it is not possible to increase the fitness (i. e. if all output are forced to 1 then we obtain the incorrect value for half of the instances in $\mathcal{T}'$, the same if we force 0).

There is only one selection of variables that allows to increase the fitness (i. e. the first v variables). As the selection is uniform across all the subsets of v variables, in expectation only one step every $\binom{n}{v} \geq \left(\frac{n}{v}\right)^v = \frac{n^{c \log n}}{(c \log n)^{c \log n}}$ steps can produce an individual with a better fitness and that can be accepted by the mutation. $\square$

Example 2. *Consider the following training set in four variables:*

	v_1	v_2	v_3	v_4	$f(v_1, v_2, v_3, v_4)$
$x_1 =$	0	0	0	0	0
$x_2 =$	1	0	0	0	1
$x_3 =$	0	1	0	0	1
$x_4 =$	1	1	0	0	0

where $v_1, \ldots, v_4$ are variables and $x_1, \ldots, x_4$ elements of the training set. Note that the only choice of variables that allow us to select elements of the training set with equal output value (and thus increase the fitness) is $\{v_1, v_2\}$ However there are $\binom{4}{2} = 6$ different possible subsets of two variables. Hence the fitness increase just once every 6 generations, in expectation.

Fixed Alternative Block Mutation

In the following we introduce a mutation operator which is half-way between FBM and VBM, which has higher probability of success than FBM and finds the optimum in expected polynomial time in the worst case.

Definition 7. ***Fixed Alternative Block Mutation (FABM):*** *Let $v < n$ be a parameter. Let us consider a* fixed *partition of the set of the n input variables (fixed in some arbitrary way at the initialisation of the algorithm) into n/v groups of v variables each. These groups of variables are the set of fixed alternatives of the mutation operator. FABM selects uniformly at random a group of variables among the fixed alternatives, and then draws uniformly at random an incomplete minterm M comprising all the variables in that group as a base for the forcing mutation.*

The FABM operator is half-way between FBM and VBM: as FBM, it fixes the choice of (groups of) variables at the initialisation, however, as VBM, it can use each time different (groups of) variables to construct the incomplete minterm to feed to the forcing mutation. From their definitions we have that FABM is more expressive than FBM, but less expressive than VBM. So, the probability of success of FABM is bounded below by the probability of success of FBM, and bounded above by the probability of success of VBM. We saw in the previous section that VBM is not completely expressive as there are some problems it cannot solve. Therefore, FABM is also not completely expressive as it is less expressive than VBM. The following theorem relates the performance of GSGP with FABM to those with FBM.

Theorem 5. *For any Boolean problem of size n on which GSGP with FBM with v fixed variables finds an optimal solution with probability p in time T, GSGP with FABM with groups of v variables finds an optimal solution in time $T' = O\left(\frac{n}{v}T\right)$ with success probability $p' \geq 1 - (1-p)^{n/v}$.*

Proof. For GSGP with FABM with groups of v fixed variables, it holds that:

- if FBM finds the optimum for a given problem and training set then there exist a group of variables among the alternatives for which we can always improve any current non-optimal solution;
- the worst case happens when there is only one group of variables that can be used to improve on the current solution. In this case the runtime is n/v times slower than the runtime of GSGP with FBM as, in expectation, the algorithm can draw a group of variables that allows for an improvement only once every n/v trials. Thus the running time of GSGP with FABM is at most $T' = \frac{n}{v}T$.

GSGP with FABM tries to find the optimum using n/v disjoint groups of the input variables. The probability that the optimum cannot be found using any of those subsets is bounded from above by $(1-p)^{n/v}$, thus the probability of success of FABM is $p' \geq 1 - (1-p)^{n/v}$. □

As a corollary, as GSGP with FBM finds an optimal solution in polynomial time with high probability, GSGP with FABM finds it with higher probability and higher, but still polynomial, expected time.

Multiple Size Block Mutation

The block mutation operators considered so far cannot guarantee to find the optimum in all cases, but when they do they may find it in polynomial time. Instead, pointwise mutation operators can always find the optimum as they can act on the value of any entry of the output vector independently from any other entry. However, they need exponential time to find the optimum on any problem and any choice of training set. In the following, we introduce a mutation operator that combines both blockwise and pointwise mutations, which attempts to preserve the benefits of both.

Definition 8. ***Multiple Size Block Mutation (MSBM):*** *The operator MSBM samples the number of variables v to consider uniformly at random between 0 and n. Then, it selects v variables at random from the set of n input variables, and it generates uniformly at random an incomplete minterm M using those variables, which is then used as a base for the forcing mutation.*

The effect on the output vectors of the feature of MSBM that the number of variable v is not fixed but it can be any number between 0 to n at each application is that the number of blocks partitioning the output vectors can vary from 1 block with 2^n entries to 2^n blocks with a single entry each. On the one hand, as for pointwise mutation, this allows GSGP with MSBM to always reach the optimum as each single entry of the output vector can be acted upon independently by the mutation. On the other hand, GSGP with MSBM can solve efficiently any problem that can be solved efficiently by GSGP with the block mutation VBM. This is because MSBM can simulate VBM on v variables in time which is in the worst case n times larger (as the probability of selecting exactly v variables by MSBM is $1/n$). However, the time needed by GSGP with MSBM to reach an optimal solution can be exponential on some training set, as shown below.

Proposition 3. *There exists a training set for which GSGP with MSBM takes expected exponential time to find the optimum.*

Proof. Consider the training set with input entries $\mathcal{T} = \{x \in \{0,1\}^n \mid \exists! i \in \{1, ..., v\}\ x_i =$

$1\} \cup \{(0,\dots,0)\}$ and as output values those of the parity problem. Except for the all-zero vector, that has output value 0, all the others output values are 1 (See Example 1).

By the same reasoning as the proof of Proposition 1, the only way to reach the optimum is to obtain a minterm M that is satisfied by the all-zero vector and no other vector. Note that there exists only one minterm of n variables with this property.

Since there are 2^n complete minterm and we select each of them with equal probability, selecting the correct one requires an exponential number of trials in expectation. $\square$

Note that for all the block mutation considered, the length of the Boolean function found as optimum is bounded above by the time complexity. In fact, as all the block mutations considered add a single (incomplete) minterm at each generation of GSGP, the number of minterms forming the optimum is bounded from above by the runtime, so it is polynomial. Thus, when the time complexity is polynomial, the length of the Boolean function found is also polynomial.

3.4 Experimental comparison of the four block operators

In this section, we present an experimental investigation of the time to reach the optimum and the success rate for GSGP with the four block mutations introduced before. The theoretical analysis has focused on worst case analysis and asymptotic behaviour. The empirical investigation complements the theoretical one focusing instead on the average case for growing finite problem size.

A problem instance is a pair of a Boolean function h and a training set $\mathcal{T}$. We set the size of the training set τ equal to the number on input variables n of the Boolean function, i.e., the problem size. At each run, a randomly selected training set of a randomly selected function is generated and tested on GSGP with the four mutations. For each problem size from 8 to 48 with step 8, 100 runs were performed. For all mutations except MSBM, the number of variables selected was $v = 2\lceil \log_2 n \rceil$. We used a GP with population of one, initialised with a random minterm, mutation applied with probability 1, and the offspring replaced the parent only if it had better fitness (i.e., a higher number of correct outputs on the training set). A run was stopped when either the optimal solution was found or 10^5 generations had passed

The results on the success rates for the different mutations is presented in Table 3.4. As for FBM, with the chosen v, the theoretical analysis predicts that asymptotically GSGP has a constant probability of success different from 1. The fluctuations and deviations from constancy for growing problem size seen experimentally are due to the rounding in the used expression for v and to the fact that the the theoretical result holds asymptotically in the problem size. As for both VBM and FABM, from theory we expect an asymptotical rate of convergence higher than for FBM. This is confirmed experimentally. It is also not surprising that the rate of convergence approaches 1 for increasing problem size for these two mutations, as the chosen v is a threshold point for the asymptotic behaviour of FBM between constant probability of success different from one, and probability of success one. As expected from the theory, MSBM converged to the optimum at all times given enough time.

The results on the number of generations to reach an optimal solution are presented in Table 3.5. A plot of the optimisation time for increasing problem size is presented in Fig. 3.2. Experimentally FBM, VBM and FABM have very similar performance. The experiments estimate the average-case performance which draw quite a different picture from the worst-case performance determined theoretically in which FBM and FABM have a polynomial worst case, and VBM an exponential worst case. The rather non-smooth shape of the performance curves for these three mutations is caused by the rounding effect in the used expression for v. Furthermore, it is striking that MSBM performs significantly better than the other mutations, and, unlike those, MSBM seems to present a linear trend between optimisation time and problem size. Again, the experimental average-case picture is different from the theoretical one, which prescribes an exponential worst case for this mutation.

Table 3.4: Success rate of GSGP on random Boolean problems for different problem sizes and mutations.

	Problem Size					
	8	16	24	32	40	48
FBM	0.95	0.76	0.93	0.74	0.87	0.88
VBM	0.95	0.99	1.00	1.00	1.00	1.00
FABM	1.00	1.00	1.00	1.00	1.00	1.00
MSBM	1.00	1.00	1.00	1.00	1.00	1.00

Table 3.5: Minimum (min), median (med) and maximum (max) number of generations for GSGP to reach the optimum on random Boolean problems.

		Problem Size					
		8	16	24	32	40	48
	Min	34	294	1659	2353	14445	10434
FBM	Med	207	1330	5862	6544	28456	27916
	Max	649	4959	14229	17068	69617	61842
	Min	44	471	1815	1736	11970	11225
VBM	Med	235	1531	5406	6432	29358	28437
	Max	759	3841	17355	22863	60965	56365
	Min	18	438	2316	2289	9900	11901
FABM	Med	254	1244	5756	6719	25827	28499
	Max	863	3925	13675	16792	81845	66043
	Min	5	117	182	477	1072	2291
MSBM	Med	115	547	1053	2222	3333	4928
	Max	450	2174	2677	8229	11907	14044

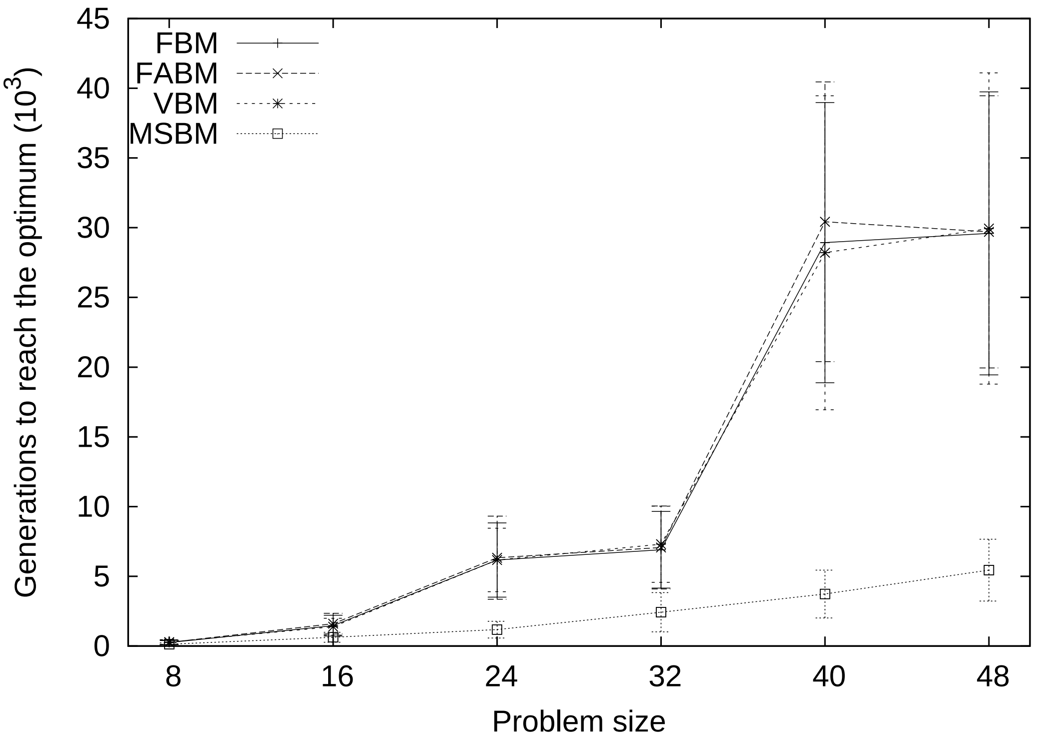


Figure 3.2: Average number of generations GSGP took to attain an optimal solution on random Boolean problems. The error bars represents one standard deviation above and below the average.

3.5 Comparison of GSGP with Cartesian GP

The experimental work presented in the previous section compares the 4 blocks mutation on random Boolean problems. Previous work [59] has also compared Forcing Point Mutation with traditional tree-based GP on common single-output Boolean problems (Parity, Multiplexer, Comparator).

In this section we will further expand the experimental investigation comparing GSGP using Forcing Point Mutation (GSGP-SGMB, from now on refereed as just "GSGP") and Multiple Size Block Mutation (GSGP-MSBM, from now on refereed as "GSGP with block mutation") against a well-known kind of GP not based on trees: *Cartesian GP* (CGP)[55]. The comparison will be performed using a comprehensive set of Boolean benchmarks that includes both traditional ones (e. g. k-even parity and k-multiplexer) and new ones (e. g. digital adder and multiplier), thus further expanding the experiments provided for Forcing Point Mutation in [59], which includes just single-output problems, and the experiments for the block mutations provided before, which are done just on random Boolean functions. The inclusion of multiple outputs problems follows [53] and [91], which suggests to start using benchmark problems that are more difficult than the traditional ones. The results will show that for both kinds of benchmarks, GSGP and GSGP with block mutation outperform CGP, confirming the efficacy of GSGP in solving Boolean problems even when compared with non-tree-based GP and on multiple-output benchmarks.

We will first introduce Cartesian GP and then we will proceed in presenting the experiment settings and results.

3.5.1 Cartesian GP

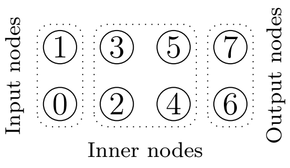
The idea of CGP appeared at first in works about the synthesis of digital circuits (see [54] for an example), and was then developed to a general technique and applied to many other domains. Here we will briefly detail CGP; for a comprehensive overview we refer the reader to [55].

Differently from tree-based GP, each CGP individual is represented as a grid of r rows and c columns (where both r and c are parameters of the algorithm). Each position (i, j) on this grid contains a functional unit (or node) that gets its inputs from some of the columns having index below j or from the input nodes.

The representation used by CGP is usually in the form of a list of integers representing both the different actions that the functional units can perform and the connections between them. Hence, there are two kind of genes: *function genes*, that encodes the actions that are performed by the different functional units using a gene mapping table, and *connection genes*, that encodes, for each functional unit, the list of other functional units or input nodes that are used as inputs. An example of the grid and the gene mapping table used by CGP are given in Fig. 3.3.

An individual of CGP can be seen as a list:

$$\left(\underline{n_1}, c_{1,1}, \ldots, c_{1,k} \ ; \ \underline{n_2}, c_{2,1}, \ldots, c_{2,k} \ ; \ \ldots \ ; \ o_1, \ldots, o_\ell\right)$$



Function gene	Action
0	$\wedge$
1	$\vee$
2	$\otimes$

Figure 3.3: The grid in which the functional units are placed with the position numbered (on the left) and the gene mapping table (on the right).

$$(\underline{0}01 \ ; \ \underline{0}11 \ ; \ \underline{1}23 \ ; \ \underline{2}22 \ ; \ 54)$$

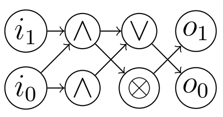


Figure 3.4: An example of a chromosome of CGP and, below it, the individual it encodes (assuming the grid and gene mapping table are the ones of Fig. 3.3).

where c_i for $1 \leq i \leq r \cdot c$ are function genes and, for each i, $c_{i,1}, \ldots, c_{i,k}$ represents the inputs of the i-th functional unit. Furthermore, $o_1, \ldots, o_\ell$ denotes the nodes from which the outputs are obtained. Fig. 3.4 shows the relation between a CGP's chromosome and the acyclic graph it encodes.

The presence of multiple output nodes makes CGP particularly suitable in solving multiple outputs problems, since no modification is required in the definition of the individuals.

3.5.2 Experimental Settings

In this section we describe in detail the benchmark problems and the settings for both CGP and GSGP.

Benchmark Problems

The benchmark problems can be divided into two classes. The former comprises Boolean problems traditionally used as benchmarks. The latter focus on problems with multiple outputs, which have been recently proposed as new, more meaningful, benchmarks [91].

Single-output problems are widely used as benchmarks [53] despite having some drawbacks that prevent them from being considered modern and effective benchmarks [91].

Being single-output problems they can be easily solved with non-GP methods, moreover they usually happen to be too easy for GP when the problem size is small. Nonetheless, we will still use them in order to make our results comparable to those from older works:

- **k-even parity**. This problem has k bits as input. The output is 1 if and only if the k-bits input has an even number of bits set to 1. We have performed experiments for $k = 4$, $k = 5$, and $k = 6$.
- **k-multiplexer**. For this problem the input consists in k bits, and k must be in the form $h + \log_2 h$. The first $\log_2 h$ bits represent an address on the remaining bitstring. The output is the value of the bit addressed by the first $\log_2 h$ bits. We tested this problem for $k = 3$ and $k = 4$.

To overcome the limitations of single-output problems, a new set of benchmarks has been proposed. In particular the *digital multiplier* is considered a modern and meaningful benchmark extremely difficult to be solved by GP [89]:

- **Digital Adder**. The inputs are two n-bitstrings representing two n-bits integers plus one carry bit. The output is a carry bit and a n-bitstring representing the sum of the two inputs integers. We used digital adders with 2 and 3 bits in the experiments.
- **Digital Multiplier**. The inputs are two n-bitstrings representing two n-bits integers. The output is a $2n$-bitstring integer representing the product of the two inputs integers. We used 2 and 3 bits digital multipliers as benchmarks.

Settings the algorithms

For CGP we have used as a base the implementation by Julian Miller[1]. The parameter settings were derived from [56], in which an high number of nodes and a low mutation probability were found to be effective. Hence, we used 4000 nodes with a mutation rate

[1]`http://www.cartesiangp.co.uk/resources/CGP-version1_1.7z`

Table 3.6: Parameters used for Cartesian GP

Population size	5
Mutation rate	1%
Number of nodes	4000
Functions set	AND, OR, NOR, NAND

of 1%. The population size was 5 and each individual could use AND, OR, NOR, and NAND as actions for the functional units (see Table 3.6).

For GSGP we have used a $(1+1)$EA setting [68]: the population is composed of one individual, at every generation the unique individual in the population is mutated and, if the resulting tree has a better fitness than its parent, it replaces the current individual. Since the population has size 1, we don't perform crossover. The starting individual is an empty tree, returning "false" on all inputs. For all the tests we have used our lisp implementation of GSGP. To produce multiple outputs with GSGP we have represented every individuals as an array of standard (i. e. single-output) GSGP trees, each one providing one output. The mutation is performed by randomly selecting a position of the array and then mutating the tree in that position. In this way at most one output bit is mutated.

We have performed 100 independent runs for every benchmark problem using a training set comprising all possible inputs (i. e. a complete training set). We have used as a fitness the fraction of correct output bits. Notice that, for multiple-outputs problems, this fitness does not always correspond to the fraction of correct outputs (since one output consist in more than a bit). The evolution was stopped after reaching a perfect score, i. e. all correct outputs, on the training set.

3.5.3 Experimental Results

In this section we present and discuss the results of the experiments that we performed. The results are grouped by benchmark problem and a general discussion of the results is presented at the end of the section.

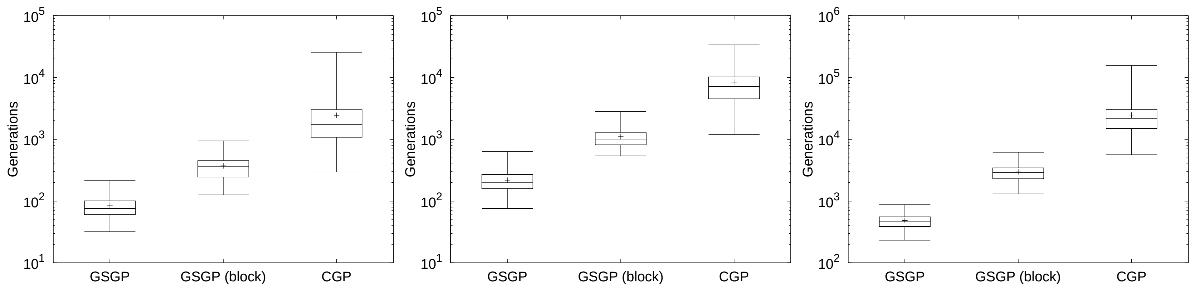


Figure 3.5: The box plot of the results obtained on the 4,5,6-even parity problem.

We report the average, the standard deviation, the median, and the 90th percentile of the number of generations needed to reach the optimum (see Table 3.7). We also report the results in the form of box plots, where the top and bottom bars indicate the 75th and 0.25th percentile, respectively. The central bar represents the median and the cross the average. The two whiskers indicates the maximum and the minimum of the number of generations needed to reach the optimum. Notice that the box plots are in logarithmic scale.

To test the existence of a statistically significant difference between the different methods, we have performed a Mann-Whitney U-test (see [93] for a description) with a significance level of 0.01 under the alternative hypothesis that the GSGP (resp., GSGP with block mutation) finds the optimum before CGP with probability greater than one half.

k-even Parity

The results for the k-even parity problem are reported in Fig 3.5 for $k = 4$, $k = 5$ and $k = 6$. For the smallest value of k, it is possible to observe a large difference between the three methods. On average, GSGP needs less than 100 generations to reach the optimum. This value grows to a little less than 400 for GSGP with block mutation and it is more than 2400 generations for CGP. Even when considering medians, the gap between the three methods remains wide. It is interesting to observe that the median for CGP is more than 700 generations lower than the average. The raw data in fact shows that CGP is penalized by the presence of a small number of long runs that increase the average.

For $k = 5$ GSGP remains the best performer and CGP the worst. In fact, GSGP with block mutation requires on average about 5 times more generations than GSGP to reach the optimum, while CGP requires a little less than 8 times more generations than GSGP with block mutation. As in the previous case, the gap between the three methods persists also when comparing medians. It is interesting to note that, for CGP, the median is more than one thousand of generations lower than the average.

Finally, for the $k = 6$ case, the situation remains similar to the previous two cases, with GSGP requiring less than 500 generations to find the optimum (on average). GSGP with block mutations requires about 6 times that value. CGP remains the worst performer with more than $2 \cdot 10^4$ generations to reach the optimum. As in the previous cases, the median number of generations is lower than the average for all methods, in particular for CGP (about three thousands generations lower).

For all values of k tested, the statistical tests have confirmed that both GSGP and GSGP with block mutation find the optimum earlier than CGP.

k-multiplexer

The results for the 3-multiplexer and 6-multiplexer problems are presented in Fig. 3.6. The results for the 3-multiplexer problem show that GSGP with both kinds of mutation needs, on average, less than 50 generations in order to reach the optimum. On the other hand, CGP requires more than 160 generations on average. If we consider the medians, the gap between the two variants of GSGP and CGP shrinks, however, the difference in performances remains statistically significant.

The results for the 6-multiplexer are similar to the ones for $k = 3$. On average the two variants of GSGP requires less than half the number of generations that CGP needs in order to reach the optimum. As before, when considering the median the gap between CGP and the other two methods shrinks, but even for the 6-multiplexer problem the statistical tests have confirmed that GSGP and GSGP with block mutation finds the optimum earlier than CGP.

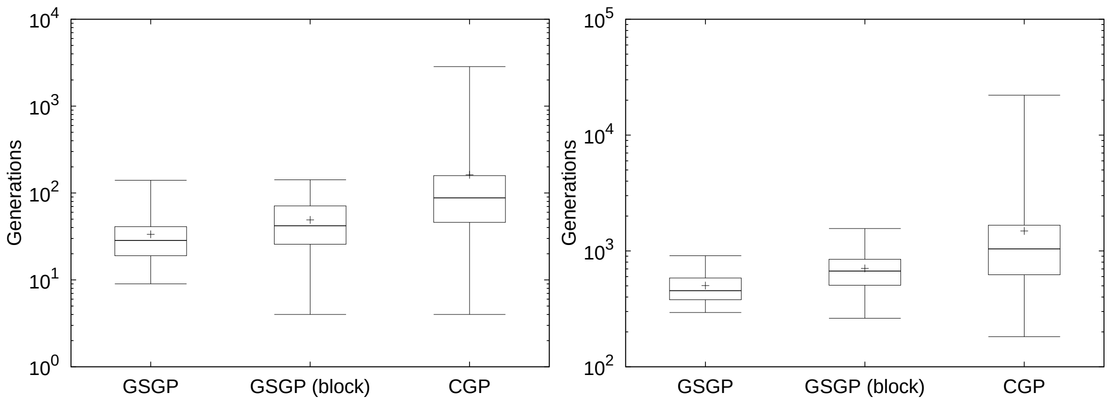


Figure 3.6: The box plot of the results obtained on the 3 (left) and 6 (right) multiplexer problem.

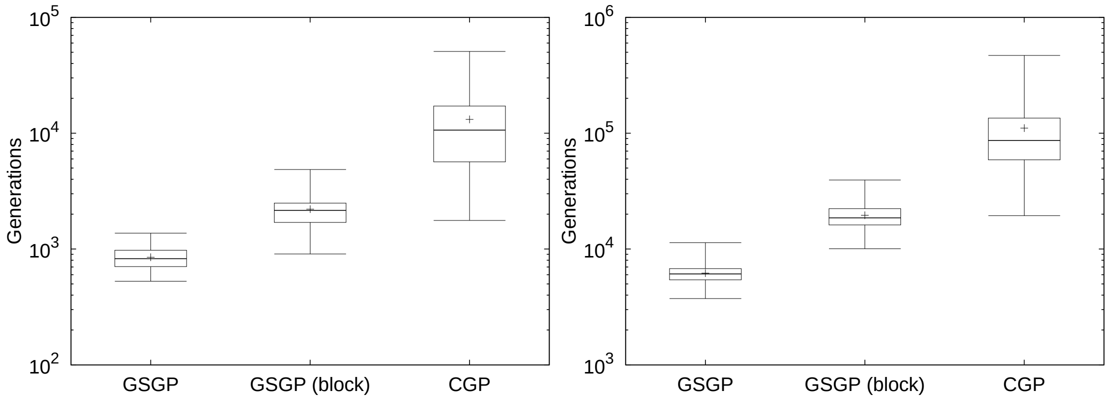


Figure 3.7: The box plot of the results obtained on 2,3 bits digital adder problem.

Digital Adder

The results for the 2-bits and 3-bits digital adder are presented in Fig. 3.7. For the 2-bits case, both GSGP and GSGP with block mutation requires, on average a number of generations to find the optimum that is from about 6 to 15 times smaller than CGP. Since this is the first multiple-outputs problem among the ones considered, it is a important to notice that the gap between the two variants of GSGP and CGP is still present. As in the previous benchmark problems, GSGP with block mutation needs more generations to find the optimum than GSGP with the original mutation.

The situation remains similar in the 3-bits case, in which GSGP requires, on average, a little more than 6000 generations to reach the optimum, GSGP with block mutation requires about three times more generations, while CGP remains the worst performer

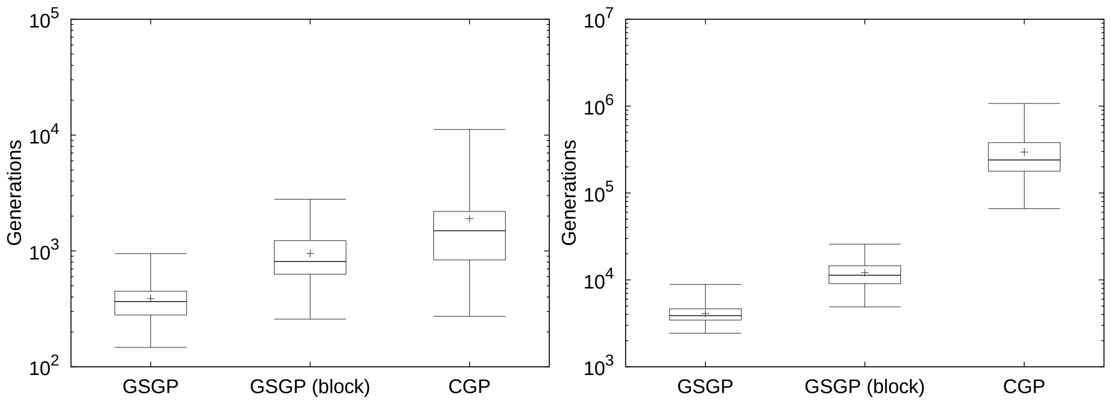


Figure 3.8: The box plot of the results obtained on 2,3 bits digital multiplier problem.

requiring, on average, more than 10^5 generations. As in the previous cases, the gap between the median and the average is higher for CGP.

For all tested values of k, the difference between CGP and the other two methods is confirmed by the statistical tests.

Digital Multiplier

Fig. 3.8 show the results for the 2-bits and 3-bits digital multiplier. For $k = 2$ the gap of performance between CGP and GSGP is lower than for the previous problems. In fact, CGP requires, on average, just twice the generations of GSGP with block mutation to reach the optimum. When considering the median the difference between the three methods is even smaller. However, the superiority of GSGP over CGP is still statistically significant.

For the 3-bits multiplier problem the difference between GSGP (resp., GSGP with block mutation) is of more than 10^5 generations (resp., $9 \cdot 10^4$). It is interesting to notice that for both semantic methods the worst run (the one requiring the higher number of generations) is better than the best run attained by CGP. The difference between the two semantic methods and CGP is then clearly statistically significant.

		4-even parity	**5-even parity**	**6-even parity**	**3-multiplexer**	**6-multiplexer**	**2-bits adder**	**3-bits adder**	**2-bits multiplier**	**3-bits multiplier**
GSGP	*Average*	85.8	217.9	484.4	33.5	502.2	849.3	6200.5	387.7	4099.2
	Std. Dev.	38.0	88.7	128.5	21.2	154.8	194.6	1100.4	144.8	1049.3
	Median	76.0	198.5	473.0	28.5	453.5	825.0	6096.0	365.0	3874.5
	90*th* *percentile*	136.0	323.0	628.5	57.5	765.0	1107.5	7521.0	583.0	5313.5
GSGP (block)	*Average*	372.1	1102.1	2967.6	49.0	706.6	2209.8	19580.5	949.9	12116.9
	Std. Dev.	163.6	419.1	860.4	30.4	284.7	743.0	5259.7	471.5	4445.3
	Median	360.0	977.0	2916.0	42.0	670.0	2157.5	18590.0	810.5	11312.5
	90*th* *percentile*	585.5	1715.0	4027.0	93.5	1117.0	3146.5	26964.5	1552.5	18597.5
CGP	*Average*	2450.7	8447.3	24722.5	161.8	1487.5	13147.2	110693.0	1898.3	296260.0
	Std. Dev.	2848.8	5764.8	17486.0	314.0	2304.6	10012.8	80568.4	1754.4	169132.0
	Median	1727.0	7164.5	21886.0	88.0	1041.0	10628.5	86477.0	1493.5	240308.0
	90*th* *percentile*	4970.0	16658.5	39388.5	322.5	2509.0	26457.5	219518.0	3708.0	521820.5

Table 3.7: A summary of the results. For each problem and for each method, the average, standard deviation, median, and the 90th percentile of the number of generations needed to reach the optimum are reported.

3.5.4 Results Discussion

In all problems tested we have observed the same pattern of behavior from the three tested methods. GSGP was the best performer in all problems, GSGP with block mutation the second best, and CGP was in all cases the worst performer. From the high standard deviation and the gap between median and average we can observe the strong presence of outliers in CGP. Such problem is absent in both GSGP and GSGP with block mutation, for which the number of generations is closer to the average. It is interesting to note that, in all benchmarks, as the problem difficulty increases the gap between GSGP (resp., GSGP with block mutation) gets wider, i. e. the two semantic methods seems to scale better.

Predictable runtime

Since GSGP is easy to study from a theoretical point of view, we can have an estimation of its expected performance. In Section 3.3.2 we have shown that Forcing Point Mutation (SGMB) is equivalent to a one-bit forcing mutation operator on the output vector, and the search for a Boolean function fitting the truth table is equivalent to that of an EA over a binary string of length equal to the number of the rows of the truth table. For example learning a 4 variables Boolean function using GSGP with the setting used for the experiments, has the same expected running time of a lazy-RLS solving the OneMax problem for a binary string of length 2^4. The expected runtime in that case is bounded above by $2n\log(n)$, where the 2 factor comes from the fact that half of the time in expectation the forcing point mutation is not producing any change in the output vector.

We have compared the worst-case expected runtime given by the theoretical study to the experimental results that we obtained (see Table 3.8). For the single-output problems the expected runtime is $2\cdot 2^n \log(2^n)$, where n is the number of input bits (thus for example $2 \cdot 2^6 \cdot \log(2^6) = 532.33$ for the 6-parity problem). For the multiple-outputs problems is $2 \cdot o \cdot 2^n \log(2^n \cdot o)$, where o is the number of output bits (so for example for 3-adder is $2\cdot(3+1)\cdot 2^{2\cdot 3+1}\cdot\log((3+1)\cdot 2^{2\cdot 3+1}) = 6388.0$. We can observe that the runtime is different

	Theoretical Bound	**Experimental measure**	**Factor**
4-*even parity*	88.72	85.8	1.03
5-*even parity*	221.80	217.9	1.02
6-*even parity*	532.33	484.4	1.10
3-*multiplexer*	33.27	33.5	0.99
6-*multiplexer*	532.33	502.2	1.06
2-*bits adder*	876.35	849.3	1.03
3-*bits adder*	6388.0	6200.5	1.03
2-*bits multiplier*	532.33	387.7	1.37
3-*bits multiplier*	4570.1	4099.2	1.11

Table 3.8: A comparison between the theoretical upper bounds on the number of generations for GSGP and the experimental measures.

from the theoretical upper bounds, on average, just by of factor between 0.99 and 1.37. This results highlights another advantage of GSGP with respect to CGP: given a Boolean problem, it is possible to estimate the runtime of GSGP by means of GA theory. The same cannot be done for CGP since no theoretical bound on the runtime is known.

3.6 Summary of the results and contributions

The novel contribution of this chapter is to provide a rigorous analysis of the geometric semantic mutation operator for boolean domain introduced in [59] and in being inspired by the issues highlighted by this analysis to propose (and analyse) new bitwise and block-wise mutation operators. Moreover this chapter proposes an experimental comparison on random boolean problem between the 4 novel block mutation operators, and an experimental comparison between GSGP-FBM, GSGP-MSBM and Cartesian GP on single and multiple output boolean benchmarks.

There are a number of peculiar issues arising with GSGP, which required a careful design of mutation operators to obtain an efficient algorithm. The fitness landscape seen by GSGP is a heavily neutral extension of OneMax on exponentially long bit strings, in which only a polynomial number of entries contribute to the fitness. Standard GA mutation operators, i.e., Forcing Point Mutation and Bitwise Mutation, give rise to an exponential runtime. Furthermore, Bitwise Mutation does not admit an efficient implementation on Boolean functions (i.e., may require exponential time for generating a single offspring). Blockwise mutations are mutation operators that can be implemented efficiently and that at each application force a whole block of bits to a random value. We proposed four block mutations, two of which (i.e. FBM and FABM) reach the optimum in polynomial time with high probability on any Boolean problem, when the size of the training set is polynomial. Experiments testing the average-case complexity of the block mutations when the size of the training set is equal to the number of input variables, have shown that one of the mutations (i.e. MSBM) seems, on the average case, much superior to the others in this setting, as it finds the optimum all the times and its runtime grows only linearly in the problem size.

We have then compared GSGP with Cartesian GP for the case in which the training set encompasses all the possible input-output pairs of the boolean function to learn. The experiments give indications of the superiority of GSGP to CGP on this setting. Particularly:

- In all the considered problems GSGP is able to find the optimum using less generations than the other methods.

- GSGP with block mutation needs more generations than GSGP with the original mutation operator. However, it still provides better performances than CGP. The worst method in all the tests is CGP. The performance gap between the three methods is statistically significant for all the considered benchmark problems.

- The existing theoretical studies allow to estimate the convergence time of GSGP, that cannot be done yet for CGP. In our experiments the convergence for GSGP is lower than the theoretical upper bounds by a factor of at least 3.

Being Cartesian GP an established method for solving Boolean problem, this result gives indication of the goodness on those problems. Moreover, differently from CGP, the runtime of GSGP is predictable, as shown in Section 3.5.4.

CHAPTER 4

GEOMETRIC SEMANTIC GP FOR LEARNING CLASSIFICATION TREES

In the previous chapter we gave a first example of theory-driven design providing design and analysis of geometric semantic mutation operators for the domain of Boolean functions. In this chapter we will do the same for the domain of Classification Trees.

We design new mutation operators and present a runtime analysis of GSGP on the class of all classification tree learning problems. This is a large class of problems whose functions to learn have categorical input variables, i.e., they take values from a limited fixed set of discrete or symbolic values, and categorical outputs. Boolean learning problems can be seen as a specific subclass of classification problems with two categorical values. However, classification problems and Boolean problems differ in the way solutions are normally represented in GP, which are classification trees for the former, and parse trees of Boolean expressions using AND, OR, and NOT operators for the latter. The different representations give rise to different issues in the design of efficient semantic operators.

This chapter is based on [50]. My contribution consisted in the design of all the novel operators, in the statements and proofs of Theorems 7, 8, 9, 10 and in the generalization framework proposed in Section 4.5.

4.1 Representations of Classification Functions

A classification function can be represented in various ways. For example it can be represented by explicit enumeration of its output values for any combination of the values of the input variables (also known as attributes), analogously to a truth table representation for Boolean functions. Similarly, it can be represented by its output vector (i.e., the output column of its truth table) when the input combinations are considered in some arbitrary but fixed order.

It can also be represented as a nested structure of `IF-THEN-ELSE` statements, whose conditions can be arbitrary Boolean expressions combining atomic conditions on input variables, and whose `THEN` and `ELSE` clauses are nested `IF-THEN-ELSE` statements or output symbols.

Perhaps, the most used and most natural representation of a classification function is a classification tree. In this representation, non-terminal nodes correspond to input variables and branching edges from a node represent the value that an input variable takes. Terminal nodes correspond to output symbols. The output for a given input variables setting is determined by starting from the root node and following a path in the tree dictated by the input combination to reach a terminal node containing the output value. Classification trees are closely related to the `IF-THEN-ELSE` representation. The nested `IF-THEN-ELSE` representation with atomic conditions correspond graphically to binary classification trees (i.e., trees whose all non-terminal nodes have two sub-trees). Classification trees correspond to nested `SWITCH` statements that can be expanded into nested `IF-THEN-ELSE` statements. Furthermore, classification trees can be represented more compactly using a directed acyclic graph representation analogous to Binary Decision Diagrams in which subtrees that occur more than once in the original decision tree are not duplicated but referred using an edge pointing to the first occurrence of the subtree when re-occurring.

In the following, we design semantic operators to evolve efficiently, both in time and space, optimal classification functions *represented as standard classification trees* as this is the most natural and used representation.

4.2 Construction of Semantic Operators for Classifiers

As explained in Section 2.3.4 the problem of finding an algorithmic characterization of geometric semantic crossover for a specific domain can be stated as follows: given a family of functions H, find a recombination operator GX_{SD} (unknown) acting on elements of H that induces via the genotype-phenotype mapping P a geometric crossover GX_D (known) on output vectors.

For the case of classification functions output vectors are vectors of symbols and GX_D is the discrete recombination that uses a random mask to recombine vectors of symbols, which is known to return offspring vectors on the Hamming segment between parent vectors [57]. We want to find a geometric semantic recombination operator for classification functions. I.e. we want to derive a recombination operator acting on classification functions that corresponds to the discrete recombination on their output vectors.

Definition 9. *Given two parent classifiers* `T1`, `T2` $: IS_1 \times \ldots \times IS_n \to OS$*, the recombination SGXP returns the offspring classifier* `T3 = IF CONDR THEN T1 ELSE T2` *where* `CONDR` *is a random Boolean condition on the input variables. Given a parent classifier* `T`*, the mutation SGMP returns the offspring classifier* `T' = IF CONDR THEN OUTR ELSE T` *where* `CONDR` *is a random condition which is true only for a single input combination of all input variables (complete condition), and* `OUTR` *is a random output symbol.*

Theorem 6. *SGXP and SGMP are, respectively, a geometric semantic crossover and a semantic* 1*-geometric mutation for the space of classifiers with fitness function based on Hamming distance, for any training set and any problem.*

The proof of the previous theorem can be found in [59]. In the following, we give an example to illustrate the theorem for the SGMP mutation. The input/output pairs describing the classifier are in the first three columns of Tab. 4.1. The classifier has $n = 2$ input variables, with input sets $IS_1 = \{1, 2, 3\}$ and $IS_2 = \{1, 2\}$, and output set $OS = \{1, 2, 3, 4\}$ (we are using sets of integer numbers as input and output symbol alphabet). Let us consider a parent classifier `T = IF (`X_1 `== 1) THEN 2 ELSE 3`. Its fitness is $f(P) = 3$, which is the Hamming distance between the output vector $P(T)$ (column 4 in Tab. 4.1) and the output vector O (column 3 in Tab. 4.1). The output vector $P(T)$ is obtained by querying `T` for all input combinations of X_1 and X_2. Say we apply to `T` the mutation SGMP with random condition parameter `CONDR = (`X_1 `== 2 AND` X_2 `== 2)` and random output parameter `OUTR=4`. The offspring `T'` is obtained by substituting `CONDR`, `OUTR` and `T` in the mutation scheme `T' = IF CONDR THEN OUTR ELSE T` obtaining `T' = IF (`X_1 `== 2 AND` X_2 `== 2) THEN 4 ELSE (IF (`X_1 `== 1) THEN 2 ELSE 3)`. The output vector $P(T')$ is in column 5 in Tab. 4.1. Note that the semantic distance between parent and offspring is $SD(T, T') = HD(P(T), P(T')) = 1$ as their output vectors differ only at one position. This shows that in this particular case SGMP is a semantic 1-geometric mutation. This holds generally for each choice of the parent `T`, complete condition `CONDR` and output value `OUTR`.

Table 4.1: Training set for the classifier problem (first 3 columns). Output vectors of parent classifier T and offspring T' after mutation with $CONDR = (X_1 == 2$ and $X_2 == 2)$ and $OUTR = 4$ (column 4 and 5)

X_1	X_2	O	$P(T)$	$P(T')$
1	1	2	2	2
2	1	1	3	3
3	1	3	3	3
1	2	2	2	2
2	2	4	**3**	**4**
3	2	1	3	3

As the offspring classifier of SGMP is obtained by the application of a functional form (i.e., mutation scheme) to its parent classifier, the semantics (i.e., underlying function) of the offspring does not depend on the actual representation of its parent (i.e., its syntax) but only on its semantics. This applies to SGXP as well, and it is in fact a characteristic of all geometric semantic operators [59]. Consequently, the search of semantic GP is not affected by how functions are actually represented. However, the representation plays an important role in terms of space efficiency of semantic GP, as we illustrate next. Continuing the example above, Fig. 4.1(left) shows the parent `T` represented as a classification tree. Fig. 4.1(centre) shows the offspring `T'` represented as a classification tree after expanding the condition `CONDR`, i.e., `T' = IF (`X_1` == 2 AND `X_2` == 2) THEN 4 ELSE T` which expanded becomes `T' = IF (`X_1` == 2) THEN (IF (`X_2` == 2) THEN 4 ELSE T) ELSE T`. Fig. 4.1(right) shows the completely expanded offspring. Using the classification tree representation the size of the offspring `T'` is more than double the size of its parent `T`: the size of the offspring grows exponentially in the number of mutation applications, which is a serious drawback of this representation. There are a number of remedies: (i) use a representation with more complex conditions so that the expansion of the condition is not required; (ii) use a direct acyclic graph representation; (iii) apply algebraic simplification to obtain smaller classification trees with the same semantics. Options (i) and (ii) have the drawback of not giving a classification tree as a solution, which is what we would like to have in the end. When the optimum is found, we could always convert it into a classifier, but of exponential size. Option (iii) may be computationally expensive, and it does not guarantee to counteract the exponential growth. We pursue a further option which consists in designing search operators that while operating on the classification tree representation still guarantee a polynomial growth.

4.3 Runtime Analysis

In Chapter 3 we have analysed GSGP for the Boolean domain. As Boolean functions can be regarded as a special type of classifiers, and the analysis does not depend on the solution representation, it turns out that the result can be readily generalised to classification problems with categorical inputs and outputs. We present the generalisation in section 4.3.1. As we are interested in evolving classifiers represented as classification

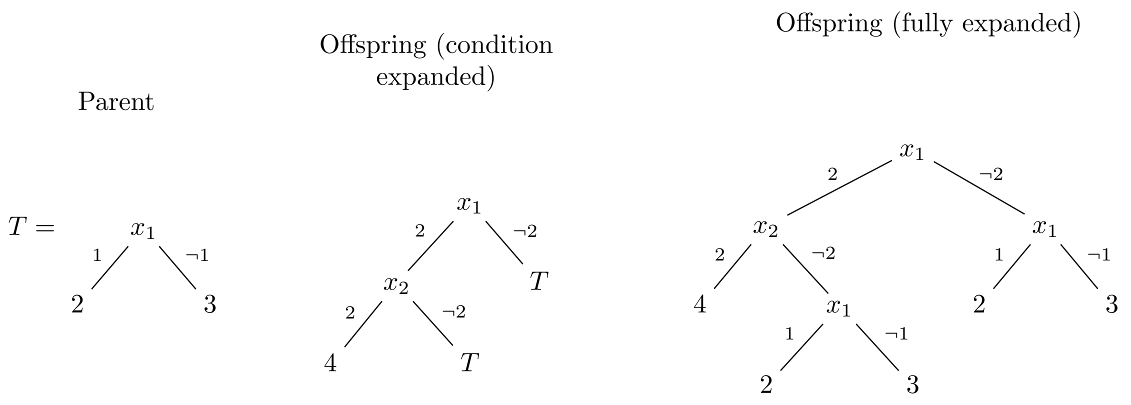


Figure 4.1: T is the parent function. The offspring T' is obtained applying the mutation operator SGMP with `CONDR = (`X_1` == 2 AND `X_2` == 2)` and `OUTR=4`.

trees, in section 4.3.2 we present mutation operators that guarantee polynomial growth of classification trees.

4.3.1 Generalisation of GSGP Runtime of Boolean Problems

The problem class we consider is *general black-box classification*, defined as follows. Let $p : IS_1 \times IS_2 \times \cdots \times IS_n \to OS$ be an unknown classifier of n input variables (n is the size of the problem). The domains IS_i of the input variables x_i and OS of the output are finite set of symbols (of constant size in n). Let $T = \{(i_1^1, \cdots, i_n^1, o^1), ..., (i_1^k, \cdots, i_n^k, o^k)\}$ be a fixed set of *polynomially many* input/output pairs of the function p where the inputs were sampled *uniformly at random* in the input domain. The pair (p, T) is an instance of the problem. We will define as runtime the time to find a classifier h that best approximates p on the training examples, in the *black-box setting*, thus without knowing directly the training set, but only via consulting an oracle which returns the error of a candidate solutions on the training set. We will not take into consideration how the classifier obtained generalizes on unknown test cases.

The mutation SGMP, by construction, corresponds to a variant of point mutation on the output vector space, that forces a randomly chosen entry of the output vector to a randomly selected output symbol (forcing mutation). Thus SGMP is equivalent, on the output vector to lazy-RLS (see Section 2.1.2).

For any choice of problem p and training set T, the fitness landscape seen by GP with geometric semantic operators is always a cone landscape, in which the fitness of a classifier (to minimise) is the Hamming distance of the output vector of the classifier queried on the training inputs to the vector of the training outputs. The size N of the output vector is exponential in the problem size n (i.e., it is $N = \prod_{i=1}^{n} |IS_i|$ where $|IS_i|$ is the cardinality of the i-th input alphabet). The fitness landscape can be understood as a generalisation of the OneMax black-box class to finite alphabet elements on exponentially long strings in which most vector entries are neutral, and the remaining entries (corresponding to elements of the training set), whose locations in the output vector are unknown, give each a unitary contribution to the fitness ("sparse" OneMax problem).

Theorem 7. *A mutation only GSGP using as mutation operator SGMP finds a classifier tree matching the training set in time $O(|OS| N \log N)$ where N is the length of the complete classification table and $|OS|$ is the cardinality of the output set OS. In particular, the runtime is exponential in n.*

Proof. We define the potential k as the number of unmatched training points. At each iteration the potential decreases if an incorrect bit is flipped to the correct one. This happens with probability $p_k = \frac{k}{N|OS|}$. Since the potential can decrease at most N times and the expected time for the potential to decrease is $1/p_k$, the expected time to reach $k = 0$ (an finding the solution to the problem is)

$$E(T) = \sum_{k=1}^{N} 1/p_k = \sum_{k=1}^{N} \frac{N|OS|}{k} = O(|OS| N \log N)$$

$\square$

In order to solve the sparse OneMax problem in polynomial time on the training set size, hence in $poly(n)$, we need to randomly change the value, in expectation, of at least $1/q(n)$ positions of the output vector associated with the training set at each generation, where $q(n)$ is any constant polynomial. Thus at each generation in expectation $\Theta(\frac{N}{q(n)\tau})$ values in the vector need to be changed at randomly chosen positions. Applying the point mutation operator (SGMP) many times is not feasible since it would require an exponential number of operations at each generation. In Chapter 3, for the case of Boolean functions, this problem is solved introducing a *block mutation operator*, which allows to change exponentially many values in a single mutation operation. The idea is to use an incomplete condition (i.e. a condition including just $v < n$ input variables) as parameter of the mutation.

Definition 10. ***Fixed Block Mutation (FBM)*** *Let's consider a fixed set of $v < n$ input variables (fixed in some arbitrary way at the initialisation of the algorithm). Given a parent classifier `T`, the mutation FBM returns the offspring classifier `T' = IF CONDR THEN OUTR ELSE T` where `CONDR` is an incomplete random condition comprising all v fixed variables, and `OUTR` is a random output symbol.*

Fixing v of the n input variables induces a *partition* of the output vector into $b = \prod_{i \in V} |IS_i|$ blocks each covering N/b entries of the output vector (which has a total of N entries). Each incomplete condition `CONDR` made up of the fixed v variables corresponds to a block partitioning the output vector. On the output vector, the effect of FBM is to select one block uniformly at random and to force all the entries belonging to that block to `OUTR`. Table 4.2 shows an example of application of FBM.

GSGP with FBM can only search the space of functions which have the same output values for each entry of the output vector in the same block. That means that there are classifier problem instances which cannot be solved optimally for some choice of the fixed variables using FBM, as the optimal solution lies outside the search space. This happens when at least two training examples with different outputs belong to the same block (e.g., in Table 4.2 the values of the entries in the first block of the target output vector Y are 4, 2 and 1. This solution is therefore not reachable as reachable output vectors have all these entries set to the same value).

For the following analysis, we make the standard Machine Learning assumption that the training set T is sampled uniformly at random without replacement from the set of all input-output pairs of the Classification problem p at hand and we give a bound on the

Table 4.2: Example of FBM. The first four columns represent the truth table of the classifier to learn. The fifth and sixth columns are the output vector of the parent T and of the offspring T' after applying FBM with `CONDR = (`$X_1 = 2$ `AND` $X_2 = 1$`)` and `OUTR=3`. Horizontal lines separate blocks of the partition of the output vectors obtained by fixing variables X_1 and X_2.

X_1	X_2	X_3	Y	$P(T)$	$P(T')$
1	1	1	4	4	4
1	1	2	2	2	2
1	1	3	1	2	2
1	2	1	4	4	4
1	2	2	2	4	4
1	2	3	2	3	3
2	1	1	2	**1**	**3**
2	1	2	3	**4**	**3**
2	1	3	3	**2**	**3**
2	2	1	2	2	2
2	2	2	1	3	3
2	2	3	3	3	3

probability of success (i.e., probability that the search algorithm can reach the optimum with the training set T). Furthermore, when GSGP with FBM can reach the optimum, we are interested in an upper-bound w.r.t. all Classification problems (any p) and all problem instances (any T of p) of the expected runtime to reach the optimum.

Theorem 8. *For any Classification problem p, given a training set T of size τ sampled uniformly at random on the set of all input-output pairs of p, GSGP with FBM with an arbitrarily fixed set of v variables finds the optimal classification function satisfying the training set T in time $O(|OS|b \log b)$, with probability $pr \approx e^{-\frac{\tau^2}{2b}}$ for $b \gg \tau$, where $b = \prod_{i \in V} |IS_i|$ is the number of blocks partitioning the output vector and $|OS|$ is the cardinality of the output set.*

Proof of Theorem 8. The proof of Theorem 8 is a simple generalisation of the proof of Theorem 3 in Chapter 3. For the probability of success the only difference is that the number of blocks b in which the search space is partitioned is equal to $\prod_{i \in V} IS_i$ (instead of 2^v). However, when $b \gg \tau$ the same reasoning applies: the asymptotic probability to find an optimal solution is $e^{-\frac{\tau^2}{2b}}$. For the runtime, again as in Theorem 3 in Chapter

3, searching for the optimum using GSGP with FBM is equivalent to solving a sparse version of OneMax on a string of length b (in this case over an alphabet of size $|OS|$) with τ active bits using only forcing mutation. Hence the expected time to reach the optimum is $O(|OS| b \log b)$ (see Theorem 3 in Chapter 3 and Theorem 7 in this Chapter). $\square$

The number of blocks b partitioning the output vector is critical for the performance of the search. On the one hand we need to keep b small to have a short runtime, on the other hand having many blocks increase the probability of success. The number of blocks b is an indirect parameter of the algorithm that can be chosen by fixing the number v of variables in the initial fixed set, as $b = \prod_{i \in V} |IS_i|$. The theorem below shows how to fix v to have both a polynomial runtime on the number of variables n and high probability of success.

Theorem 9. *GSGP with FBM on a problem with n variables and on an uniformly sampled training set of size τ bounded above by n^c, can find the optimum with high probability in polynomial time provided that the number v is fixed as $v = \lceil (2c + \varepsilon) \log_{r_m}(n) \rceil$ (for some $\varepsilon > 0$ constant in n), where r_m is the minimum size of the input alphabet across all variables.*

Proof. The number b of blocks in which the possible inputs are partitioned is at least r_m^v, hence, since $v > (2c + \varepsilon) \log_{r_m}(n)$, we have $b \geq r_m^v > n^{2c+\varepsilon}$. Therefore Theorem 8 is applicable to obtain a success probability that is, asymptotically, at least $e^{-\frac{n^{2c}}{2n^{2c+\varepsilon}}} = e^{-\frac{1}{2}n^{-\varepsilon}}$. As the argument of the exponent approaches 0 as n grows, we can use the expansion $e^x = 1 + x$ obtaining $p = 1 - \frac{1}{2}n^{-\varepsilon}$, which tells us that the runtime holds with high probability for any $\varepsilon > 0$.

Recall that the number of blocks b is bounded above by r_M^v, where r_M is the maximum size of the input alphabet across all variables. Then $b \leq r_M^v \leq r_M^{(2c+\varepsilon)\ \log_{r_m}(n)+1} \leq r_M n^{(2c+\varepsilon) \log_{r_m}(r_M)}$. Hence, by Theorem 8, the optimization time is $O(|OS| b \log b) = O(|OS| r_M n^{(2c+\varepsilon) \log_{r_m}(r_M)} \log n)$, which, assuming the size of the input alphabet not depending from n (i.e., the number of values that an input variable can assume does not depend on the number of inputs), is polynomial in n. $\square$

4.3.2 Space-Efficient Mutations for Classification Trees

GSGP with the FBM operator with an adequate parameter setting can find an optimal classifier, given a polynomial size training set, in polynomial time with high probability. Nevertheless, when classifiers are represented using classification trees, the classifier evolved with FBM grows exponentially in the number of generations (see Fig. 4.1). In the following, we introduce two mutation operators that have polynomial runtime and that lead to a polynomial growth of the evolving classifier by exploiting the specific structure of classification trees.

We know that given a complete classification table it is possible to build a classifier tree having height equal to the number of input variables. This is achieved assigning to each level of the tree an input variable to check. Nevertheless the number of nodes of the tree is exponential on the height of the tree and is thus exponential on the problem size. But since we want a classification tree representing a polynomial training set (and thus not the whole classification table) we might obtain a classification tree with a polynomial number of nodes. Particularly using random conditions containing just a fixed logarithmic-size set of input variables as stated in Theorem 9 it must be possible to build a classification tree of logarithmic height and thus polynomial number of nodes in the problem size n. The following improved version of FBM forces the tree to assign each level to a particular input variable, obtaining a polynomial size tree as explained above.

Definition 11. ***Improved Fixed Block Mutation (IFBM)*** *Let be $V = (X_1, \cdots, X_v)$ a subset of the set of all the input variables containing $v < n$ input variables that is fixed and ordered in some arbitrary way at the initialisation of the algorithm. Given a parent program `P`, the mutation IFBM (Improved Fixed Block Mutation) generates a random incomplete condition `CONDR` comprising all fixed variables as a base for the forcing mutation and `OUTR`, a random output symbol. Then it tries to explore the tree to match*

`CONDR` without using the else conditions. One of the following can happen:

1. *It successfully matches `CONDR` ending up on a leaf node. Then it substitutes the node with a leaf containing `OUTR` (Fig. 4.2, case 1)*

2. *It partially matches some conditions ($X_1 = i_1, \cdots, X_p = i_p$) but it ends up on a internal node containing the input variable X_{p+1} without the possibility to match the condition $X_{p+1} = i_{p+1}$ because a subtree for that condition is missing. Then it adds a child to that node, with an edge representing the condition $X_{p+1} = i_{p+1}$, and containing the subtree `ST = IF (`$X_{p+2} = i_{p+2}$ `AND` $\cdots$ `AND` $X_v = i_v$`) THEN OUTR ELSE OUT` (Fig. 4.2, case 2).*

Using this operator it is possible to obtain the same runtime of FBM but avoiding redundancy in the tree and keeping its size minimal. The tree is forced to keep an input variable for each level. Since FBM use a logarithmic number of variables (as suggested by Theorem 9) the evolved classification tree has logarithmic height and thus polynomial number of nodes in the problem size n.

The block mutation operators considered so far (i.e. FBM, IFBM) cannot find an optimal solution for every problem instance. Nevertheless when they do, they find it in polynomial time. Instead SGMP can always find the optimum as it can independently force each entry to any value. However, it needs exponential time to find the optimum on any problem and any choice of training set. In the following, we present a mutation operator which can always find the optimum, it can do it in polynomial time in many case, even though it takes exponential time in the worst case.

Definition 12. ***Multiple Size Block Mutation (MSBM)*** *Let $V = (x_1, \cdots, x_n)$ be randomly ordered list of all the input variables. Then at each iteration MSBM samples v uniformly at random from 1 to n, it builds a random condition `CONDR` using the variables $(x_1, \cdots, x_v)$ and it samples a random output symbol `OUTR`. Then it tries to explore the tree to match `CONDR` without using the else conditions. One of the following can happen:*

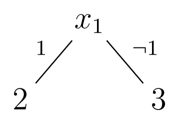
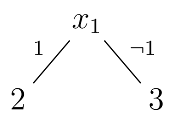
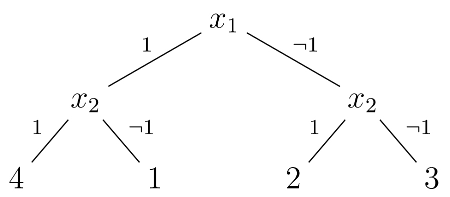
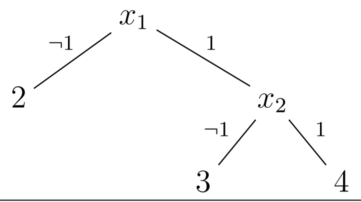
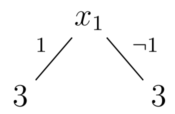
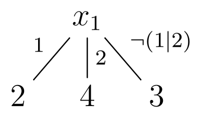
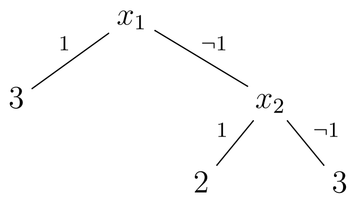
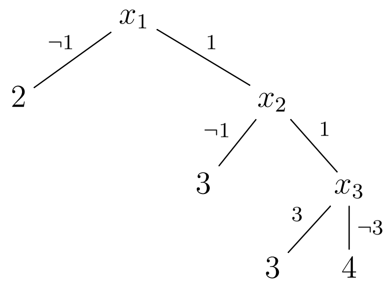
	Case 1	Case 2	Case 3	Case 4
Parent				
Offspring				
	CONDR: $(X_1 = 1)$, OUTR=3	CONDR: $(X_1 = 2)$, OUTR=4	CONDR: $(X_1 = 1)$, OUTR=3	CONDR: $(X_1 = 1,\, X_2 = 1,\, X_3 = 3)$, OUTR=3

Figure 4.2: Examples of mutation cases for IFBM (case 1-2) and MSBM (case 1-4)

1. *It successfully matches* `CONDR` *ending up on a leaf node. Then it substitutes the node with a leaf containing* `OUTR` *(Fig. 4.2, case 1)*

2. *It partially matches some conditions ($X_1 = i_1, \cdots, X_p = i_p$) but it ends up on a internal node containing the input variable X_{p+1} without the possibility to match the condition $X_{p+1} = i_{p+1}$ because a subtree for that condition is missing. Then it adds a child to that node, with an edge representing the condition $X_{p+1} = i_{p+1}$, and containing the subtree* `ST = IF (`$X_{p+2} = i_{p+2}$ `AND` $\cdots$ `AND` $X_v = i_v$`) THEN OUTR ELSE OUT`. *(Fig. 4.2, case 2)*

3. *It successfully matches* `CONDR` *ending up on an internal node. Then it substitutes the node with a leaf containing* `OUTR` *and deletes all the subtrees of the old node. (Fig. 4.2, case 3)*

4. *It partially matches some conditions ($X_1 = i_1, \cdots, X_p = i_p$) but it ends up on a leaf containing* `OUT` *without matching the renaming variables ($X_{p+1} = i_{p+1}, \cdots X_v = i_v$). Then it replaces the current leaf with the tree* `ST = IF (`$X_{p+1} = i_{p+1}$ `AND` $\cdots$ `AND` $X_v = i_v$`) THEN OUTR ELSE OUT`. *(Fig. 4.2, case 4)*

Since the number of variables to produce the random condition `CONDR` can be between 1 and n the length of the blocks that are forced in the output vector is not fixed. As for SGMP, this allows semantic GP with MSBM to always reach the optimum as each single entry of the output vector can be acted upon independently by the mutation (when the length of `CONDR` is 1). Moreover when the set of variables used by FBM is one of the subsets used by MSBM, then MSBM can solve efficiently any problem that can be solved efficiently with the block mutation FBM. In fact MSBM can simulate FBM on the set of v variables in common in time which is in the worst case n times larger (since the probability of selecting exactly v variables by MSBM is $1/n$).

Even if the classification tree evolved with MSBM use all n variables, the number of nodes is still kept polynomial, when the runtime is polynomial:

Theorem 10. *Consider a classification problem with n variables. An individual of GSGP with MSBM mutation grows of $O(n)$ nodes at each generation.*

Proof. Consider the possible cases described in Fig. 4.2. The increase in the number of nodes for each case is the following:

1. One node is replaced. The number of nodes remains the same;
2. At most n condition, each with two children, are added. Hence, the increase in the number of nodes is bounded by $O(n)$.
3. The number of nodes is reduced by at least 2;
4. By the same reasoning of case 2, the increase in the number of nodes is bounded by $O(n)$.

The four cases of Fig. 4.2 are exhaustive. That is, any tree transformation of MSBM is one of those cases. Therefore, the statement of the theorem follows immediately. □

The previous theorem lead to the following corollary.

Corollary 1. *If the tree is evolved in a polynomial number of generations, then the tree has polynomial size.*

That means that a polynomial upper bound for the running time also guarantees a polynomial space complexity.

4.4 Summary of the results and contributions

In this chapter, Geometric Semantic GP was applied to classification problems. GP with the syntactic search operators obtained using this framework corresponds, in the semantic space (i.e., on the output vectors), to a GA with standard search operators solving a "sparse" version of OneMax (on a string with finite alphabet), in which only a limited number of entries contribute to the fitness.

The novel contribution of this chapter is to provide an analysis of the existing GSGP operator for classification trees introduced in [59] and to be inspired by this analysis to propose (and analyse) three new block mutation operators: FBM, IFBM and MSBM.

We have considered the situation in which the training set has polynomial size in the number of input variables of the classifier to learn. In this setting, we have proposed and analysed three efficient geometric semantic mutations. FBM can find the optimum in expected polynomial time, but in some cases it cannot find it. However if the size of the training set is polynomial and the size of the blocks is set properly according to the size of the training set (see Theorem 9), FBM can solve a randomly selected boolean problem with high probability. The issue of FBM is that it grows the size of the classification trees exponentially at each generation.

From the analysis on the growing size of the individuals with FBM we got inspiration to design IFBM which has the same expressivity of FBM (thus it can solve all the problems that FBM can solve and it cannot solve all the problems that FBM cannot solve), but grows the classification trees linearly at each generation. Finally we introduced MSBM which can always find the optimum even if, for some combinations of training sets and problem instances, it may require exponential time.

4.5 On the generalization ability of GSGP

In this and in the previous chapters we have analysed the expected time for GSGP to fit the training set in the case of classification trees and Boolean domain respectively. As explained in Section 3.1, if the training set does not encompass all the possible inputs, this would lead to a generalization error $\varepsilon_g(h)$.

In this section we will propose a framework to measure the generalization ability of a general evolutionary algorithm (including GP), which is the ability of an EA to find expressions leading to a small generalization error.

We will just propose the framework and we will leave the calculation of the generalization ability for specific GSGP operators as future work. We will use for simplicity the boolean domain as example. All the definitions presented in this section can be easily generalized for classification functions at the cost of an heavier notation.

4.5.1 The learning problem

Recall from Section 3.1 that given a complete truth table $C = \{(x_1, y_1), ..., (x_N, y_N)\}$ describing $\overline{h} : \{0,1\}^n \to \{0,1\}$ and training set T consisting in $\tau \leq N$ ($N = 2^n$) test cases $T \subset C = \{(x_1, y_1), ..., (x_\tau, y_\tau)\}$ sampled uniformly at random from the truth table C, the aim of the black-box Boolean learning problem is to use just the training set T to learn a Boolean function $h : \{0,1\}^n \to \{0,1\}$ matching as well as possible the input-output behaviour described by C, i.e., to minimise the *generalisation error*,

$$\varepsilon_g(h) = \frac{1}{N} \sum_{i=1}^{N} I[h(x_i) \neq y_i]$$

The problem is black-box so the algorithm has no direct access to the training set T, but just to an oracle that will return how well a candidate function X matches the input-output behaviour described by the training set (i.e. the training error).

4.5.2 Fitness functions

In order to highlight the black-box constraint we will define two different fitness functions. The *fitness on the training set*, $f_T(X)$ is the training error, which measures how well a given Boolean expression X matches the training set T. It is defined as

$$f_T(X) = \varepsilon_t(X) = \frac{1}{\tau} \sum_{i=1}^{\tau} I[X(x_i) \neq y_i] \ , \ \forall (x_i, y_i) \in T$$

and it is equal to zero when an individual perfectly matches the training set T, while it has its worst value of 1 when the training set is completely unmatched. This fitness function represents the oracle described above and it is the fitness function used by the learning algorithm for selection.

The *real fitness* $f_C(X)$ is instead the generalisation error, measuring how well a given Boolean expression X matches the complete truth table C. It is defined as follows

$$f_C(X) = \varepsilon_g(X) = \frac{1}{N}\sum_{i=1}^{N} I[X(x_i) \neq y_i] \ , \ \forall (x_i, y_i) \in C$$

Its value is equal to 0 when the individual perfectly matches the complete truth table C, while it has its worst value of 1 when the truth table is completely unmatched. Notice that the learning algorithm has *no access* to $f_C(X)$. It can use the fitness on the training set, $f_T(X)$, to measure the fitness of the current individual X but it cannot get the value of $f_C(X)$. This fitness function will be used in the following to define the generalisation ability of an algorithm.

4.5.3 Learning performance measures

With f_T being the fitness on the training set, and f_C being the real fitness we define the *generalization ability* of an evolutionary algorithm $\mathcal{A}$ as

$$G(\mathcal{A}) = E[1 - f_C(\widetilde{X})]$$

where $\tilde{X}$ is the individual obtained after $\mathcal{A}$ has converged. The generalisation ability is in the range $[0, 1]$, and is the larger the better.

Notice that the definition of the generalisation ability depends just on f_C while the algorithm has just access to f_T. This means that the best generalisation performance will not necessarily be achieved by the algorithm which better optimise f_T. This is common knowledge in machine learning, where the effect of maximising f_T regardless of f_C would be considered as *overfitting*.

The works in the previous and this Chapter have provided upper bounds for the *expected runtime* to fit the training set, which in this framework can be defined as:

$$R(\mathcal{A}) = E[\mathcal{T}(f_T(X) = 0)]$$

which is the expected time for the algorithm to find an individual with zero error on the training set since $\mathcal{T}(f_T(X) = 0)$ is the number of generations until the first individual with zero training error is produced.

Significance of the measures

It is important to understand the significance and the limits of the two measures.

The expected runtime gives guarantees on the convergence time of the algorithm. It measures how fast a solution is found but it doesn't give any indication on the quality of solution for the learning problem. On the other hand the generalization ability measures the goodness of the algorithm as a learning tool providing indication on the quality of the solutions produced by it. It does not give any indication, though, on the time the algorithm takes to get to that solution. The two measures are both important and ideally they should both be taken into consideration when designing an algorithm. A good algorithm for a specific problem is one that has an high generalization ability and low (not more than polynomial) running time on that problem.

In this thesis we contributed towards the analysis of the expected runtime of GSGP. We also used the insights coming from the analysis of the runtime to design operators which are better from a runtime point of view. The problem of analysing the generalization ability of these operators is left open for future contributions.

CHAPTER 5

GEOMETRIC SEMANTIC GP FOR BASIS FUNCTIONS REGRESSION

In the previous chapters (Chapter 3, 4) we gave two examples of theory-driven design providing design and analysis of geometric semantic mutation operators for the domain of Boolean Functions and Classification Trees respectively. In the case of Boolean functions, theory helped in realizing that the first geometric semantic operators proposed would take exponential time in solving instances of the black box Boolean learning problem, even with a training set of polynomial size. We then built on that consideration new *blockwise mutation operators*, we analysed them, and we found out the conditions under which they perform well.

For the domain of Classification Trees we did the same. Moreover the theoretical analysis gave insights to derive block operators with an improved space efficiency.

In this chapter we will leave the integer domain and start to consider the continuous domain. We will in fact design and analyse Geometric Semantic Genetic Programming operators for Basis Functions Regression.

This is a general form of regression in which the regression function obtained comes from a class of functions expressible as a linear combination of the basis functions $\{g_j\}_{i\in\mathbb{N}}$. By using different families of basis functions, one obtains well-known specific regression classes such as simple linear regression, polynomial regression, trigonometric regression and more.

This chapter is based on [60]. I have contributed on the design of the IGSM mutation operator, on the statements and proofs of Theorem 12 and Lemma 1, on the experiments in section 5.4 and partially on the analysis in Section 5.3.1.

5.1 Problem Definition

The problem class we consider is a general form of regression – basis functions regression – defined as follows. Let $g_j : \mathbb{R}^n \to \mathbb{R}$ be a family of m basis functions, and H be the class of functions expressible as linear combination of the basis functions $\{g_j\}$. Let $p : \mathbb{R}^n \to \mathbb{R}$ be an unknown target function, and $T = \{(i_1, o_1), ..., (i_k, o_k)\}$ be a fixed set of input/output pairs of p. The aim is to find a function $h \in H$ that best approximates p on the training examples. I.e., $h^* = \text{argmin}_{h \in H}\{ED(P_I(h), O)\}$. By using different families of basis functions, one obtains well-known specific regression classes. E.g., for polynomial regression with a single input variable, one can choose $g_j = x^j$.

The general problem class can be solved directly, without search, using the *least squares method*. A function $h \in H$ can be written as $h(x_1, \dots, x_n) = c_1 \cdot g_1(x_1, \dots, x_n) + \dots + c_m \cdot g_m(x_1, \dots, x_n)$ and it is identified in H, given $\{g_j\}$, by the vector of coefficients $C = (c_1, \dots, c_m)$. The optimal vector of coefficients can be determined solving the system of simultaneous linear equations obtained by using the training set T, i.e., $o_z = c_1 \cdot g_1(i_z) + \dots + c_m \cdot g_m(i_z)$ with $z = 1, \dots, k$. In matrix form is $O' = GC'$ with O being the vector of training set outputs and G being the matrix $\{g_j(i_z)\}_{j,z}$. The solution is $C' = G^+O'$ where G^+ is the Moore-Penrose pseudoinverse of G that is a generalisation of inverse matrix that returns a well-defined result when the system is underdetermined ($m > k$) or overdetermined ($m < k$). The optimal approximating function so computed minimises the Euclidean distance to the target output vector.

GSGP goes beyond the least square method as it can solve the basis function regression problem in the black-box setting: it does not use the knowledge of the training set, but only the errors of candidate solutions on the training set.

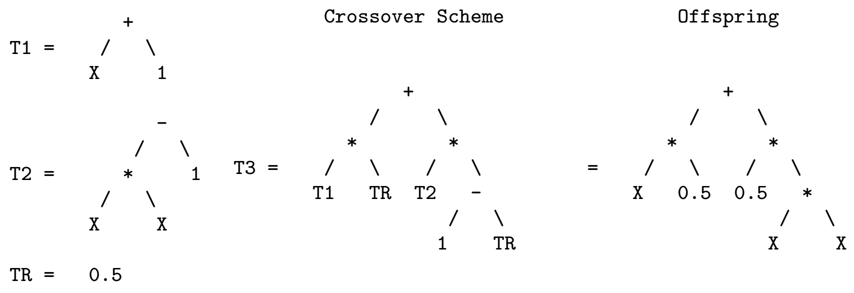


Figure 5.1: T1 and T2 are parent functions and TR is a random value in $[0, 1]$. The offspring T3 is obtained by substituting T1, T2 and TR in the crossover scheme and simplifying algebraically.

5.2 Construction of Semantic Operators for Real Functions

As explained in Section 2.3.4 in order to design geometric semantic crossover for a specific domain we need to find GX_{SD} acting on elements of the domain that induces a geometric known crossover GX_D on output vectors. For the case of Real functions with fitness measure based on Euclidean distance, output vectors are real vectors and GX_D should be a line crossover that returns offspring vectors on the (Euclidean) line segment between parent vectors. We want to derive a recombination operator GX_{SD} acting on Real functions that corresponds to a line crossover on their output vectors.

5.2.1 Naive Operators

The following two Geometric Semantic operators have been firstly proposed in [59].

Definition 13. *Given two parent functions $T1, T2 : \mathbb{R}^n \to \mathbb{R}$, the recombinations SGXE and SGXM return the real function $T3 = (T1 \cdot TR) + ((1 - TR) \cdot T2)$ where TR is a random real constant in $[0, 1]$ (SGXE) (see Fig. 5.1), or a random real function with codomain $[0, 1]$ (SGMX). Given a parent function $T : \mathbb{R}^n \to \mathbb{R}$, the mutation SGMR with mutation step ms returns the real function $TM = T + ms \cdot (TR1 - TR2)$ where $TR1$ and $TR2$ are random real functions.*

Theorem 11. *SGXE and SGXM are geometric semantic crossovers for the space of real functions with fitness function based on Euclidean and Manhattan distances, respectively, for any training set and any real problem. SGMR is a semantic ε-geometric mutation for real functions with fitness function based on Euclidean and Manhattan distances. The mean of its probability distribution is the parent, and ε is proportional to the step ms.*

The proof of the previous theorem can be found in [59]. In the following, we give an example to illustrate the theorem for the SGXE crossover. Let us consider the simple symbolic regression problem, in which, we want to find an expression whose values match those on the quadratic polynomial $x^2 + x + 1$ in the range $[-1, +1]$. Let us say the target function values for $x \in \{-1, -0.5, 0, +0.5, +1\}$ are given as training set (see two leftmost columns of Table 5.1, X inputs and Y outputs). The target output vector Y is the real vector $(1, 0.75, 1, 1.75, 3)$ (column 2 of Table 5.1). For each tree representing a real function, one can obtain its output vector by querying the tree on the inputs X. The output vectors of the trees in Figure 5.1 are in the last 4 columns of Table 5.1. The fitness $f(T)$ of a tree T (to minimise) is the Euclidean distance between its output vector $P(T)$ and the target output vector Y (restricted to the outputs of the training set), e.g., the fitness of parent $T1$ is $f(T1) = ED(P(T1), Y) = ED((0, 0.5, 1, 1.5, 2), (1, 0.75, 1, 1.75, 3)) \simeq 1.436$. The semantic distance between two trees $T1$ and $T2$ is the Euclidean distance between their output vectors $P(T1)$ and $P(T2)$, e.g., the semantic distance between parent trees $T1$ and $T2$ is $SD(T1, T2) = ED(P(T1), P(T2)) = ED((0, 0.5, 1, 1.5, 2), (0, -0.75, -1, -0.75, 0)) \simeq 3.824$. Let us now consider the relations between the output vectors of the trees in Table 5.1. This crossover on output vectors is a geometric crossover w.r.t. Euclidean distance, as $P(T3)$ is in the Euclidean segment between $P(T1)$ and $P(T2)$ as $P(T3)$ is obtained as a convex combination of $P(T1)$ and $P(T2)$. We can also verify this on the

Table 5.1: Training set for the polynomial regression problem (first 2 columns). Output vectors of trees in Figure 5.1 (last 4 columns): of parents T1 and T2, of random value TR, and of offspring T3.

X	Y	$P(T1)$	$P(T2)$	$P(TR)$	$P(T3)$
-1	1	0	0	0.5	0
-0.5	0.75	0.5	-0.75	0.5	-0.125
0	1	1	-1	0.5	0
+0.5	1.75	1.5	-0.75	0.5	0.375
+1	3	2	0	0.5	1

example using the distance relation for the line segment:

$$\begin{aligned}
&ED((0, 0.5, 1, 1.5, 2), (0, -0.125, 0, 0.125, 1))+\\
&+ ED((0, -0.125, 0, 0.125, 1), (0, -0.75, -1, -0.75, 0)) =\\
&= 1.912 + 1.912 = 3.824 =\\
&= ED((0, 0.5, 1, 1.5, 2), (0, -0.75, -1, -0.75, 0)).
\end{aligned}$$

This shows that, in this example, the crossover on trees in Figure 5.1 is a geometric semantic crossover w.r.t. Euclidean distance.

As the syntax of the offspring of semantic operators contains at least one parent, the size of individuals grows quickly in the number of generations. To keep their size manageable during evolution, we need to simplify algebraically offspring sufficiently and efficiently without changing the computed function. The search of semantic crossover and semantic mutation is unaffected by the simplification, which can then be done at any moment and in any amount. In practice, we need to work with "well-behaved" families of functions whose offspring can be easily simplified. In this chapter, we focus on linear combinations of basis functions.

Furthermore, as the semantic operators generate offspring as a functional combination of their parents, it does not matter how functions are actually represented (e.g., trees, graphs, sequences) as the representation does not affect the search behaviour. We have represented functions as trees only to compare this framework with traditional GP.

5.2.2 Design of an efficient mutation operator

In [26], a runtime analysis of (1+1)-ES with adaptive isotropic Gaussian mutation on the sphere function is reported showing that (1+1)-ES is efficient on the sphere. This analysis applies unchanged to the Euclidean cone landscape.

In the previous section we claimed that (SGMR) is a semantic ε-mutation. That means that the Euclidean distance between the output vector of the parent and the output vector of the offspring is bounded by some fixed constant ε. That does not say anything on the way the output vector of the offspring is mutated, and particular it does not guarantee that the change in the output vector is isotropic Gaussian.

So, in this section we will designing a semantic mutation operator for real functions that on the output vector space always corresponds to an isotropic Gaussian mutation. We will then be able to reduce the runtime analysis of GSGP for any basis functions regression problem, i.e., any choice of basis functions g_j, any choice of function to approximate p, and any choice of training set T, to the above settings, getting the same guarantees on the runtime provided by the analysis of (1+1)-ES with adaptive isotropic Gaussian mutation.

The semantic mutation we consider is of the form $o(x) = p(x) + ms \cdot r(x)$ where $p(x) \in H$ is the parent function, $ms \in \mathbb{R}^+$ is the mutation step, $r(x)$ is the perturbing function which is sampled according to some probability distribution over the functions search space H. As $o(x)$ is a linear combination of elements of H it is also in H. For a fixed number of basis functions m, we can represent functions in H by their (fixed-length) vectors of coefficients in the linear combination. The semantic mutation on this representation becomes $c_o = c_p + ms \cdot c_r$ where $c_o, c_p, c_r \in \mathbb{R}^m$ denote the vectors of coefficients of $o(x)$, $p(x)$ and $r(x)$. Simplification of the offspring in this representation is implicitly achieved by algebraic operations on real vectors. The genotype-phenotype mapping on this representation becomes a function $P : \mathbb{R}^m \to \mathbb{R}^k$ that maps vectors of coefficients to output vectors.

Lemma 1. *The genotype-phenotype mapping P is a linear map. So, it holds that $P(c_1 + c_2) = P(c_1) + P(c_2)$ and $P(\lambda c) = \lambda P(c)$ for all $c_1, c_2, c \in \mathbb{R}^m$ and $\lambda \in \mathbb{R}$.*

Proof. As explained in Section 5.1, given:

- Function: $F(X) = c_1 g_1(x) + \ldots + c_m g_m(x)$
- Coefficient vector $C = (c_1, \ldots c_m)$
- Output vector: $O = (o_1, \ldots, o_k) = (F(X_1), \cdots, F(X_k))$

The mapping P from C to O can be expressed in matrix form as $O' = GC'$

$$(o_1, \ldots, o_k) = \begin{pmatrix} g_1(X_1) & g_2(X_1) & \cdots & g_m(X_1) \\ g_1(X_2) & g_2(X_2) & & \\ \vdots & \vdots & \ddots & \vdots \\ g_1(X_k) & g_2(X_k) & \cdots & g_m(X_k) \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_m \end{pmatrix}$$

From linear algebra, we know that this is a linear map. □

Consequently, line segments and circles on the space of vectors of coefficients (genotypes) are projected via P to, respectively, line segments and (possibly rotated and rescaled) ellipses on the output vectors space (phenotypes). To obtain a circle on phenotype space, one needs an ellipse in the genotype space that compensates for the transformation P. In terms of semantic operators, this means that a line crossover on genotypes induces a line crossover on phenotypes. A carefully chosen non-isotropic Gaussian mutation on the space of genotypes corresponds to an isotropic Gaussian mutation on the space of phenotypes. We prove the last statement formally.

Theorem 12. *The non-isotropic Gaussian mutation on genotypes $c_o = c_p + ms \cdot c_r$ with $c_r \sim N(0, G^+ I G'^+)$ induces the isotropic Gaussian mutation on phenotypes $P(c_o) = P(c_p) + ms \cdot P(c_r)$ with $P(c_r) \sim N(0, I)$.*

Proof. $c_o = c_p + ms \cdot c_r$; $P(c_o) = P(c_p + ms \cdot c_r)$; For linearity of P: $P(c_o) = P(c_p) + ms \cdot P(c_r)$; $P(c_r)' = Gc_r'$ is a linear transformation of the multivariate Gaussian c_r: $P(c_r) \sim N(0, G \cdot G^{+}IG'^{+} \cdot G') = N(0, I \cdot I \cdot I') = N(0, I)$ $\square$

The implementation of the semantic mutation requires to sample c_r from $N(0, G^{+}IG'^{+})$. This can be done by multiplying G^{+} by a vector formed by sampling k times the normal distribution $N(0, 1)$.

Using this we can design an operator that, acting on the vector of coefficients of the parent function, produces an Isotropic Gaussian Mutation on its output vector. The operator (IGSM) is presented in the following:

Definition 14. *Given a function $F : \mathbb{R}^n \to \mathbb{R}$ in the class H of functions generated by the functions base $(g_1(x), \ldots, g_m(x))$, and $C = (c_1, \ldots, c_m)$ the vector of coefficients defining F in H (thus $F(X) = c_1 g_1(x) + \ldots + c_m g_m(x)$). The Isotropic Gaussian Semantic Mutation (IGSM) returns as offspring the function $F' \in H$ defined by the vector of coefficients C' such that $C' = C + C_r$ and $C_r \sim N(0, G^{+}IG'^{+})$.*

5.3 Runtime Analysis

The main problem in implementing the IGSM presented above is that, in order to sample $C_r \sim N(0, G^{+}IG'^{+})$, the algorithm needs to know the matrix G, which depends on the training inputs (but not on the training outputs).

We will start analysing GSGP on a semi-black-box setting (Section 5.3.1), in which the algorithm needs to know the training inputs to calculate the matrix G, but not the output corresponding to each input (so given $T = \{(i_1, o_1), ..., (i_k, o_k)\}$, the algorithm has just access to $(i_1, \ldots, i_k)$). In Section 5.3.2 we propose a method to calculate the matrix G without knowing the training inputs, thus in a real black-box scenario.

5.3.1 Analysis with known training inputs

In [26], an asymptotic runtime analysis to obtain an approximated solution within ε from the optimum [1] of the standard (1+1)-ES with the 1/5-rule (Algorithm 5) on the Euclidean cone is reported. It was found that it takes $\Theta(n)$ generations (where n is the dimension of the search space) to halve the distance $|c|$ from the initial point c to the optimum. Consequently, it takes $\Theta(n \log_2(\frac{|c|}{\varepsilon}))$ generations to find an approximated solution within ε from the optimum. When the search space is a hyper-box with length of the sides constant in n and ε, the maximum distance of the initial point to the optimum is given by the length of the longest diagonal of the hyper-box, which is $\Theta(\sqrt{n})$. The runtime then becomes $\Theta(n \log_2(\frac{\sqrt{n}}{\varepsilon}))$. The algorithm is very efficient both in n for any fixed ε, as the runtime is $\Theta(n \log n)$, and in $\frac{1}{\varepsilon}$ for a fixed n, as the runtime is $\Theta(\log(\frac{1}{\varepsilon}))$.

Algorithm 5 (1+1)-ES with 1/5-rule for adaptation

```
1: initialise $curr \in \mathbb{R}^n$ with a starting point
2: $\sigma := f(curr)/n$; $gen := 0$; $succ := 0$
3: while $f(curr) > \varepsilon$ do
4:    choose mutation vector $mut \in \mathbb{R}^n$ randomly $\sim N(0, I)$
5:    create offspring vector $off := curr + \sigma \cdot mut$
6:    if $f(off) \leq f(curr)$ then $curr := off$; $succ := succ + 1$ endif
7:    if $gen\%n = 0$ then
8:       if $succ < n \cdot 1/5$ then $\sigma := \sigma/2$ else $\sigma := \sigma \cdot 2$ endif
9:       $succ := 0$
10:   end if
11:   $gen := gen + 1$
12: end while
13: return $curr$
```

The GSGP algorithm that corresponds to Algorithm 5 on the space of output vectors can be obtained from it with the following modifications. The search space is the space of functions in H represented by vectors of coefficients, so $curr, mut \in \mathbb{R}^m$, and n is m. The fitness function $f(x)$ returns the Euclidean distance of the output vector of the function x to the target output vector. The mutation vector mut is sampled from $N(0, G^{+}IG'^{+})$. The pseudo-inverse matrix G^{+} is needed to sample mut, which is pre-computed in the

[1]In continuous domains, there is zero probability of hitting exactly the optimum.

initialisation of the algorithm. This algorithm is only *partly black-box* because it uses knowledge of the inputs of the training set (but not of the outputs) in the computation of G^+. In the next Section, a complete black-box algorithm is presented.

When the unknown target function p belongs to the search space H or when the number of basis functions m is larger or equal to the number of points in the training set k, GSGP may in the limit find a function h^* that passes through all points of the training set, i.e., $f(h^*) = 0$. When $p \notin H$ and $m < k$, the best approximating function $h^* \in H$ has non-zero fitness on the training set. This fitness value is the smallest error ε_{min} GSGP searching H can achieve.

From a runtime viewpoint, the search done by GSGP with IGSM is equivalent to the search of (1+1)-ES on the output vector space. The dimension of the space is the size k of the training set, which is therefore a natural definition of problem size. The size of the hyper-box delimiting the output vector space depends on the endpoints of the interval co-domains of the basis functions $\{g_j\}$, the endpoints of the ranges of the coefficients of linear combinations, which we consider both constant w.r.t. k and ε, and on the number of basis functions m, which we consider to be some function of k (e.g., $m = k$ to find an interpolating function). The length of the sides of the hyper-box is therefore $\Theta(m)$, and the runtime to find a function h with $f(h) - f(h^*) < \varepsilon$ is $\Theta(k \log_2(\frac{m\sqrt{k}}{\varepsilon}))$. In particular, the runtime is constant in the number of input variables n of the functions searched.

5.3.2 Using surrogate training inputs

In Machine Learning, when one can choose the training examples, they are normally selected to evenly cover the input domain X of the functions $h \in H$. In this case, the training inputs are known, and the GSGP analysis in the previous section applies. When one cannot choose the training examples, the standard assumption is that they were sampled uniformly at random on the input domain X. In this case to generate the matrix G we can use a *surrogate grid* of inputs, which is a set of points that cover evenly the input domain X so that the distance between any pair close points is the same. For example if the input space is a unidimensional segment $X = [0, 1]$ then the points in the grid are $A = \{\frac{1}{k+1}, \frac{2}{k+1}, \ldots, \frac{k}{k+1}\}$. Or, for the 2-dimensional case $X = [0, 3] \times [0, 3]$ and $k = 4$ the 4 points are $A = \{(1, 1), (1, 2), (2, 1), (2, 2)\}$.

The idea behind using a surrogate grid is that as the number of input points sampled uniformly at random grows, their spatial distribution tends to approximate better and better a grid on X. Thus the runtime of GSGP with IGSM using the grid to calculate the matrix G would converge, for growing number of training inputs, to the runtime of GSGP with IGSM which knows the training inputs and which is calculated in the previous section (Section 5.3.1). This statement is formally proved in [60].

5.4 Experiments

In the following, we compare experimentally GSGP and the standard (1+1)-ES with 1/5-rule for adaptation on the vector of coefficients to search the space of functions (hereinafter (1+1)-ES). These algorithms differ only on the mutation operator: (1+1)-ES uses an isotropic Gaussian mutation on the vector of coefficients, while GSGP uses a non-isotropic Gaussian mutation on the vector of coefficients which corresponds to a isotropic Gaussian mutation on the output vector.

We also compare GSGP, that uses the knowledge of training inputs in the mutation operator, with GSGPg that replaces it with a surrogate regular grid of inputs points covering the input space. The asymptotical runtime of GSGP and GSGPg are the same, but they may be different for finite training set size k.

The reason for the experimental investigation is that there are no theoretical bounds for a (1+1)-ES searching on the vector of coefficients. In fact its analysis is hard since its mutation operator is distorted by the genotype-phenotype mapping, and it corresponds to a non-isotropic mutation on the space of output vectors.

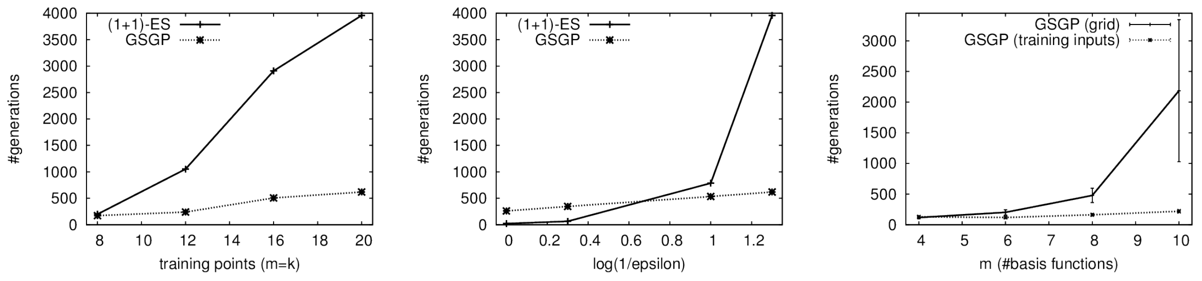


Figure 5.2: Comparison of GSGP and (1+1)-ES to reach an approximation $\varepsilon = 0.05$ (left). Comparison of GSGP and (1+1)-ES with $m = k = 20$ to reach different levels of approximation (centre). Comparison of GSGP using grid and GSGP using the training input for $k = 18$ and different values of m (right).

5.4.1 Experimental setting

We test the algorithms on randomly generated problem instances. A random problem p is a polynomial of degree m with coefficients generated uniformly at random in $[-1, 1]$. A random instance of the problem p is a set T of k input/output pairs obtained by querying p with inputs sampled uniformly at random in $[-1, 1]$. Each point in the graphs in Figure 5.2 reports the average number of generations of 100 runs, each on a new random instance of a new random problem.

Figure 5.2 (left) compares GSGP and (1+1)-ES in terms of the number of generations (on the ordinates) to get a solution within tolerance $\varepsilon = 0.05$ from the optimum, for increasing size of the training set k (data points for $k = 8, 12, 16, 20$ on the abscissa).

Figure 5.2 (centre) compares GSGP and (1+1)-ES in terms of the number of generations (on the ordinates) to get increasingly better approximations to the optimum, measured as $\log\left(\frac{1}{\varepsilon}\right)$ on the abscissa (data points for $\varepsilon = 1, 0.5, 0.1, 0.05$). The size of the training set k and the number of basis functions m are kept fixed as $m = k = 20$. GSGP scales linearly in $\log\left(\frac{1}{\varepsilon}\right)$, in line with the theory. (1+1)-ES reaches a certain approximation to the optimum quickly, after that it takes increasingly longer time to get better approximations, i.e., its approximation performance does not scale well.

Figure 5.2 (right) shows the effect on the performance of GSGP and GSGPg of the number of basis functions m (data points for $m = 4, 6, 8, 10$) for a fixed training set size

$k = 18$, and a fixed approximation ε. GSGP and GSGPg have similar performances, as they have asymptotically by the theory, when k is large w.r.t. m. Their performance difference becomes marked as m gets closer to k. The reason for that is that we are using polynomial basis functions ($g_i(x) = x^i$) thus when the number of basis functions grows, the transformation matrix G gets more sensible to small variations in x. Thus even if for large k the training inputs tends to be close to the grid, when m grows even the small difference between the training inputs and the grid creates a large difference in the transformation matrix G and thus affects the runtime.

5.5 Summary of the results and contributions

We have considered GSGP for Basis Functions Regression, which is a large class of regression problems. Problems in this class can be solved directly by the least squares method. However, GSGP can solve them in a black-box setting. We have shown a tight link between GSGP for real functions and (1+1)-ES.

The main novel contribution of this Chapter is the design of a semantic mutation operator making the search of GSGP for any basis functions regression problem equivalent to standard (1+1)-ES with isotropic mutation on the sphere function and to transfer the known runtime results for the (1+1)-ES to GSGP with this semantic mutation. GSGP is very efficient in terms of scalability in both the size of the training set k ($\Theta(k \log k)$) and the target approximation to the optimum ε ($\Theta(\log(\frac{1}{\varepsilon}))$). This operator requires the algorithm to know the training inputs (but not the outputs) thus breaking the black-box constraint. We have then proposed to replace the training inputs with a surrogate grid of points in the input space. Preliminary experiments have shown that GSGP with the new semantic mutation performs much better than a (1+1)-ES evolving the vector of coefficients and thus inducing a non-isotropic Gaussian mutation on the output vector. The experiments have also shown that the surrogate grid method is a viable option when the training inputs are not known and the number of basis functions m is small.

CHAPTER 6

HOMOGENEOUS AND HETEROGENEOUS ISLAND MODELS FOR SET COVER

In this and in the following chapter we will move from the domain of Genetic Programming to the domain of Parallel Evolutionary algorithms. We will keep in mind the two main aims of this thesis stated in the introduction: to advance the theoretical understanding of evolutionary algorithms, filling the gaps existing particularly in the sub-fields of genetic programming and parallel evolutionary algorithms, and to use this novel knowledge to guide the design of new algorithms and operators.

Parallelisation is becoming a more and more important issue, due to the current development in computer architecture and the steeply rising number of processors in modern devices. Evolutionary algorithms (EAs) can be parallelised by using island models, also called coarse-grained EAs or multi-deme models [47, 65]. Several subpopulations are evolved on different processors. Subpopulations coordinate their search by a process called migration, where selected individuals, or copies thereof, are sent to other islands.

As explained in Section 2.5 the theoretical foundation of Parallel EAs is still at its beginning. Particularly there are no works analysing the runtime of parallel EAs on NP-Hard problems. Moreover all the analyses proposed before deal with homogeneous island models, in which each island is performing the same task, while it can be appropriate to discuss whether an heterogeneous island model, in which each island is assigned a different behaviour, could perform better than an homogeneous one for some particular tasks.

In this chapter we propose and analyse homogeneous and heterogeneous island models for the SetCover problem. Given a set S with m elements and a collection of n subsets of S with associated costs, the SetCover problem asks for a selection of subsets of S that cover the whole set and have minimum cost. This classic NP-hard problem is one of the most fundamental problems in computer science. Friedrich *et al.* [20] studied a SEMO algorithm on a biobjective formulation of the problem and showed that SEMO efficiently computes an H_m-approximation, where $H_m = \sum_{i=1}^{m} \frac{1}{i}$ is the m-th Harmonic number.

In this chapter we study a parallel version of this algorithm where each island runs an instance of SEMO. Each island use the same bi-objective fitness functions to be minimised: one criterion counting the number of uncovered elements and the other representing the cost of the selection. Each island stores a population of non-dominated solutions. At the end of each generation migration occurs transmitting a copy of the whole population to all neighbouring islands. We will call this *homogeneous island model* since each island is running the same algorithm.

Our analysis will show that this leads to significant speedups, depending on the topology and the migration probability, for probabilistic migration policies. However, the analysis will also highlight that this island model has large communication cost as whole populations are exchanged between islands. To this end, we propose a heterogeneous island model that has a lower communication cost and in which islands run simpler algorithms.

The *heterogeneous island model* consists of $m+1$ islands using different single-objective fitness functions. Each island stores one individual and runs a (1+1) EA (or RLS that just differs in using local instead of global mutation). The fitness functions are such that on island i only selections covering i elements are feasible. Therefore, each island i keeps the best individual covering i elements of S. The island model can be implemented on fewer than $m+1$ processors by running multiple islands on each processor. We show that the collection of islands is able to guarantee the same performance and approximation quality as in the homogeneous model, but with lower communication costs and simpler operations. We also study different migration policies for the heterogeneous model and show how the migration policy affects running time and communication costs.

This chapter is based on [52]. My contribution consisted in the design and analysis of the novel heterogeneous island model, so in the model presented in Algorithm 7 and the analysis in Section 6.3, including statements and proofs of Theorems 15, 16, 17, 18. I've also contributed in implementing both the heterogeneous and homogeneous island models to run simulations that helped in the design and the analysed showed in this chapter.

6.1 Preliminaries

Let $S = \{s_1, \cdots, s_m\}$ be a set containing m elements and $C = \{C_1, \cdots, C_n\}$ be a collection of non-empty sets such that $C_i \subset S$ for $1 \le i \le n$ and $\bigcup_{i=1}^{n} C_i = S$. Each set C_i has a cost $c_i > 0$. We call $X = x_1 \cdots x_n$ a *selection* of C and we say that C_i is in the selection X iff $x_i = 1$. An optimal solution to the SetCover problem is an X such that $\bigcup_{i:x_i=1} C_i = S$ and $\sum_{i:x_i=1} c_i$ is minimum.

We define the following measures:

- $c(X) = |\bigcup_{i:x_i=1} C_i|$ is the number of covered elements of the selection.
- $|X|_1 = \sum_{i=1}^{n} x_i$ is the number of selected sets of a selection.
- $\text{cost}(X) = \sum_{i:x_i=1} c_i$ is the cost of a selection.
- $c_{\max} = \max_i c_i$ is the maximum cost of a set.
- $c_e(C_i, X) = \frac{\left|C_i \setminus \bigcup_{j:x_j=1} C_j\right|_1}{c_i}$ is the cost-effectiveness of a set w. r. t. X.

The *homogeneous island model* consists of an archipelago of μ islands each one running the SEMO algorithm, minimising the multi-objective fitness function $f(X) = (m - c(X), \text{cost}(X))$. In multi-objective optimisation, if $u = (u_1, \ldots, u_k)$ and $v = (v_1, \ldots, v_k)$ are two objective vectors, we say that u *dominates* v if $u_i \le v_i$ for $i = 1, \ldots, k$ and the strict inequality sign holds for at least one objective. SEMO always maintains a set of non-dominated search points, which is the set of points that are not dominated by any other point in the population. New solutions are created by selecting uniformly a search point from the current population and mutating it. The offspring is added to the current population and then all dominated search points are removed. SEMO uses local mutations: one bit is chosen uniformly at random and then flipped. A variant called *global SEMO* uses standard bit mutations instead (called *global mutations*), flipping each bit independently with probability $1/n$. In the homogeneous island model based on SEMO or global SEMO (see Algorithm 6), each island maintains such a population. For migration, a copy of this whole set is transmitted to all neighbouring islands. The union of this set with the target island's set is considered and then all dominated solutions are removed. This way, the best solutions among source and target islands are maintained and combined.

Algorithm 6 Homogeneous island model based on (global) SEMO

```
 1: Initialise $P^{(0)} = \{P_1^{(0)}, \dots, P_\mu^{(0)}\}$, where $P_i^{(0)} = \{0^n\}$ for $1 \leq i \leq \mu$. Let $t := 0$.
 2: repeat forever
 3:     for each island $i$ do in parallel
 4:         Simulate one generation of (global) SEMO, updating $P_i^{(t)}$.
 5:         Send a copy of the population $P_i^{(t)}$ to all neighbouring islands.
 6:         Unify $P_i^{(t)}$ with all populations received from other islands.
 7:         Remove all dominated search points from $P_i^{(t)}$.
 8:     end for
 9:     Let $t := t + 1$.
10: end repeat forever
```

The *heterogeneous island model* consists of a fully connected archipelago of $m + 1$ islands indexed $0, \dots, m$. Each island stores just one individual and runs an (1+1) EA (or RLS) using a single-objective function that is different on each island. For island i we define the fitness function (to maximise) as:

$$f_i(X) = \begin{cases} nc_{\max} - \text{cost}(X) & \text{if } c(X) = i \\ -|c(X) - i| & \text{if } c(X) \neq i \end{cases}$$

The idea is that island i stores an individual that represents the so far best selection covering i elements (referred to as *feasible*). If the solution does not cover i elements, the fitness is negative and hints are given towards covering i elements[1]. Each island is thus assigned a different part of the search space to optimise. This is similar to what happens in dynamic programming [16].

The heterogeneous island model is shown in Algorithm 7. Note that the heterogeneous island model can be easily implemented on $\mu \leq m$ processors by running up to $\lceil \frac{m+1}{\mu} \rceil$ islands on each processor.

Both island models are initialised with empty selections. This is a sensible strategy for SetCover and theoretical results [20] as well as preliminary experiments have shown that this only speeds up computation.

Algorithm 7 Heterogeneous island model based on (1+1) EA (or RLS)

```
1: Initialise the island individuals $X_0^{(0)}, \dots, X_m^{(0)}$ to $0^n$. Let $t := 0$.
2: **repeat forever**
3:     **for** each island $i$ **do in parallel**
4:         Produce a global (or local) mutation $\tilde{X}_i^{(t)}$ of the individual $X_i^{(t)}$.
5:         Send a copy of $\tilde{X}_i^{(t)}$ to each other island.
6:         Choose $X_i^{(t+1)}$ with maximal $f_i$-value among $X_i^{(t)}$, $\tilde{X}_i^{(t)}$ and all immigrants.
7:     **end for**
8:     Let $t := t + 1$.
9: **end repeat forever**
```

The homogeneous and heterogeneous island models differ fundamentally in their search behaviour. Following Skolicki [82], we distinguish *intra-island evolution* (the evolution within each island) and *inter-island evolution* (evolution among and between islands). The homogeneous model uses intra-island evolution to generate improvements by mutation, and migration helps to propagate these improvements to other islands. The heterogeneous island model strongly relies on inter-island evolution; in fact, beneficial mutations as in the homogeneous model yield solutions that are only feasible on other islands. The two island models also differ in the population size. In the heterogeneous model the population of each island consists of just one individual, while in the homogeneous model the population size of each island is upper bounded by m. This generally means that the time and

[1]Our analysis holds for any negative function for the second case of f_i.

space required to compute a generation in the homogeneous model is larger than in the heterogeneous one.

We define the *parallel running time* as the number of generations of an island model until it has found a satisfactory solution, in our case an H_m-approximation. We also refer to the *sequential running time* as the product between the parallel running time and the number of islands. This represents the computational effort to simulate the model on a single processor. The *speedup* of an island model with μ islands is defined as the rate between the expected parallel running time of the island model and the expected running time of the same EA using only a single island. This kind of speedup is called *weak orthodox speedup* in Alba's taxonomy [1]. If the speedup is of order $\Theta(\mu)$, we speak of a *linear speedup*. Furthermore, we also consider the effort for performing migration. We define the *communication effort* as the total number of individuals sent between islands, throughout a run of an island model. The (expected) communication effort is given by the (expected) parallel time, multiplied by the number of islands and the (expected) number of emigrants sent by one island.

We will analyse both the models for the following migration policies:

- *Complete Migration*: each island sends migrants to all other islands.
- *Uniform Probabilistic*: each island sends migrants to every other island independently with a migration probability p.

In Section 6.3.3 we will introduce and analyse two additional migration policies especially tailored for the heterogeneous island model.

6.2 Analysis of the Homogeneous Island Model

We first consider the homogeneous model with uniform probabilistic migration as complete migration is a particular case of probabilistic migration with migration probability equal to one. In their analysis of SEMO, Friedrich *et al.* [20] consider the time until SEMO finds an empty selection, and how long it takes to get a H_m-approximate solution from there. H_m is the m-th harmonic number, defined as $\sum_{k=1}^{m} \frac{1}{k}$. Their results are as follows.

Theorem 13 (Friedrich *et al.* [20])**.** *For any initialisation and every* SetCover *instance, SEMO and global SEMO find an H_m-approximate solution in $O(m^2 n + mn \log(nc_{\max}))$ expected generations. When starting with a population containing only an empty selection, the time bound is $O(m^2 n)$ generations.*

The following lemma is at the heart of their–and our–analysis. It goes back to Chvatal's analysis of the greedy algorithm [15]. Starting with an empty set, the greedy algorithm subsequently adds the most cost-effective set to the current solution. When k elements are covered, for some $0 \leq k \leq m$, the cost of this partial solution is at most $\operatorname{cost}(X) \leq (H_m - H_{m-k})\,\mathrm{OPT}$, where OPT denotes the cost of an optimal solution. For $k = m$ this gives an H_m-approximation.

Lemma 2. *Let* OPT *be the cost of an optimal set cover and X be such that $c(X) = k$ (with $k < m$) and* $\operatorname{cost}(X) \leq (H_m - H_{m-k})\,\mathrm{OPT}$*. Adding the most cost-effective set to X creates X' with $c(X') = k'$ and* $\operatorname{cost}(X') \leq (H_m - H_{m-k'})\,\mathrm{OPT}$.

Proof. The selection X leaves $m - k$ elements of S uncovered. These elements can be covered at cost OPT since the optimal cover covers the whole set. Then there is a set with cost-effectiveness at least $\frac{m-k}{\mathrm{OPT}}$. Let i be the number of newly covered elements by adding this set, then after adding the set we get a solution covering $k' = k + i$ elements at cost no more than

$$\left(H_m - H_{m-k} + \frac{i}{m-k}\right) \cdot \mathrm{OPT} \leq (H_m - H_{m-k'}) \cdot \mathrm{OPT}\,. \qquad \square$$

This behaviour can be mimicked by SEMO [20] and the homogeneous island model. Friedrich *et al.* [20] define the *potential* of the population of the archipelago as the largest k such that there is an individual in the population that covers k elements and costs at most $(H_m - H_{m-k}) \cdot \mathrm{OPT}$. The potential can never decrease as SEMO always keeps some solution with k covered elements in the population. Starting with empty selections, the initial potential is at least 0.

The probability of increasing the potential is at least $1/((m+1)en)$ for the following reasons. It is sufficient to select the solution defining the potential and to add a set with maximum cost-effectiveness (Lemma 2). The population contains at most $m+1$ individuals, so the probability of selecting the right parent is at least $1/(m+1)$. The probability of a specific 1-bit mutation is at least $1/n \cdot (1-1/n)^{n-1} \geq 1/(en)$ for both local and global SEMO.

This analysis can be transferred to our homogeneous island model using the general method by Lässig and Sudholt [39] based on fitness levels. Assume the search space can be partitioned into fitness-level sets ordered w. r. t. fitness such that an EA never decreases its current level. If we have lower bounds on the probability that the EA will leave a current level towards a better fitness-level set, we get an upper bound on the expected hitting time of the final level. For island models we get upper bounds on the expected parallel running time that depend on the topology at hand and the probability that migration successfully transmits information about the current best fitness level. A rapid spread of information enables more islands to search on the current best fitness level, which gives better performance guarantees than a slow spread of information.

In [39] upper bounds are stated for common topologies: ring graphs, torus or grid graphs, and the complete topology. In our case instead of using fitness levels, we argue with the potential of islands. As seen above, the potential can never decrease. We have $m+1$ potential values, and the probability of increasing the potential on any island is at least $1/((m+1)en)$. Plugging this into the results from [39, 43], we get the following bounds on the expected parallel time. The expected communication effort is by a factor of $pd(m+1)\mu$ larger than the expected parallel time, where d is the degree of any node in the topology.

Theorem 14. *For the homogeneous island model based on (global) SEMO on μ islands and migration probability $p > 0$ the expected parallel time until an H_m-approximation for*

SetCover is found is bounded by

- $O\Big(\frac{n^{1/2}m^{3/2}}{p^{1/2}} + \frac{nm^2}{\mu}\Big)$ *for any ring topology,*
- $O\Big(\frac{n^{1/3}m^{4/3}}{p^{2/3}} + \frac{nm^2}{\mu}\Big)$ *for any undirected* $\sqrt{\mu} \times \sqrt{\mu}$ *grid or torus graph*
- $O\Big(\frac{m}{p} + \frac{nm^2}{\mu}\Big)$ *for the complete topology* K_μ.

The expected communication effort is $O\big(p^{1/2}\mu n^{1/2}m^{5/2} + pnm^3\big)$ *for rings,* $O\big(p^{1/3}\mu n^{1/3}m^{7/3} + pnm^3\big)$ *for grids and* $O(\mu^2m^2 + p\mu nm^3)$ *for* K_μ.

The upper bounds are asymptotically minimised for choosing the number of islands as $\mu = \sqrt{pnm}$, $\mu = (pnm)^{2/3}$, and $\mu = pnm$, respectively. A larger number of islands will not improve the upper bounds. That's why we call these bounds *best bounds*[1] and we will refer to the values of μ smaller than these as *values of mu leading to linear speedup.* With these choices we get expected parallel times of $O(n^{1/2}m^{3/2}/p^{1/2})$, $O(n^{1/3}m^{4/3}/p^{2/3})$, and $O(m/p)$, respectively (see Table 6.1 in Section 6.4). The expected communication effort is $O(pnm^3)$, $O(pnm^3)$, and $O(p^2n^2m^4)$, respectively. Multiplying all parallel times by μ, we see that the expected sequential time is bounded by $O(nm^2)$ in all three cases. This asymptotically matches the upper bound from Theorem 13 for initialisation with empty selections. This means that, apart from constant factors hidden in the asymptotic notation, in these cases parallelization does not increase the (upper bounds on the) total running time, but the (upper bounds on the) parallel time can decrease significantly. In fact, all numbers of islands up to the values mentioned above yield linear speedups—for cases where the $O(nm^2)$-bound for a single (global) SEMO is asymptotically tight.

As remarked in [40], the bound for the complete topology with $p = 1$ also applies to an offspring population-version of SEMO where λ offspring are created and added to the population, before removing dominated solutions.

[1]Notice that these *best bounds* are not guaranteed to be the optimal values for μ.

6.3 Analysis of the Heterogeneous Island Model

For the heterogeneous model based on (1+1) EA or RLS we first present an analysis for the complete migration policy.

6.3.1 Complete Migration

Theorem 15. *The heterogeneous island model with complete migration finds an H_m-approximate solution for* SetCover *in an expected parallel time of $O(n \cdot \min(m, n))$. The expected communication effort is $O(nm^2 \cdot \min(m, n))$.*

Proof. As in Theorem 14 we calculate the expected time to produce a solution that is at least as good as the greedy solution, starting from 0^n and always adding the most cost-effective set. We define again the potential of the population of the archipelago as the largest k such that there is an individual in the population that covers k elements and costs at most $(H_m - H_{m-k}) \cdot \mathrm{OPT}$. At the end of each generation (after migration and selection) the potential can't decrease. In fact the individual X^k on island k can only be replaced by an individual with the same number of covered elements but a lower cost (and that would not affect the potential). Instead the potential can be increased to k' mutating X^k such that the most cost-effective set is added. That would produce an individual $\tilde{X}^k$ such that $c(\tilde{X}^k) = k' > k$ and $\mathrm{cost}(\tilde{X}^k) \leq (H_k - H_{m-k'}) \cdot \mathrm{OPT}$ (Lemma 2).

After migration and selection this individual will replace the individual on the island k' (which had higher cost and therefore lower fitness). This specific 1-bit mutation happens with probability at least $1/n \cdot (1 - 1/n)^{n-1} \geq 1/(en)$ for both local and global mutation. At most n sets can be included in a selection but, if $n > m$, at most m of them will be selected since each most cost-effective set covers at least one new element (otherwise its cost-effectiveness would be 0). So after $O(n \cdot \min(m, n))$ expected generations $k = m$ and then on island m we get an $H_m - H_{m-m} = H_m$-approximate solution. $\square$

Comparing this time with [20] and assuming $n = O(m)$, we get that the upper bound for the parallel time is by a factor of $\Theta(m)$ lower, while we get the same upper bound for the sequential running time.

6.3.2 Uniform Probabilistic Migration

For uniform probabilistic migration with migration probability $p < 1$, the island model only increases the potential if migration happens on the edge that links the two islands involved (k and k'). The probability estimate for this event decreases by a factor of p, and the waiting time thus increases by $1/p$, leading to the following result:

Theorem 16. *The heterogeneous island model with uniform probabilistic migration and migration probability p finds an H_m-approximate solution for* SetCover *in an expected parallel time of $O(n \cdot \min(m, n)/p)$. The expected communication effort is $O(nm^2 \cdot \min(m, n))$.*

We see that our estimate of the communication effort has not improved. This is not surprising as we only rely on inter-island evolution for making progress. A uniform migration probability delays the inter-island evolution and the reduced communication effort in a single generation is nullified by a larger parallel running time.

6.3.3 Other migration policies

In the heterogeneous island model each island is designed to *prefer* a different section of the original search space. It might thus be a good idea to design the migration policies to send individuals to islands which are likely to like them. In the following we present two migration policies following this idea: : *Non-Uniform Probabilistic* and *Smart Migration.*

Non-Uniform Probabilistic

With non-uniform probabilistic migration, each island i sends migrants to every other island $(i + k) \bmod (m + 1)$ independently with probability $1/k$.

In this case the chance of making the right migration is generally higher than for uniform migration probabilities. Typically only few new elements are covered, when adding a most cost-effective set. A large number of new elements implies that we make large progress. This balances out a small migration probability: if adding the most cost-effective set covers j new elements, the probability of making this move is at least $1/j \cdot 1/(en)$. In expectation, the potential increases by at least $j \cdot 1/j \cdot 1/(en) = 1/(en)$, regardless of j. A straightforward drift analysis gives the following.

Theorem 17. *The heterogeneous island model with non-uniform probabilistic migration finds an H_m-approximate solution for* SetCover *in an expected parallel time of enm. The expected communication effort is at most enm^2H_m.*

Proof. In contrast to the previous proof, we consider the expected increase of the potential. Let P_i be the island whose solution defines the current potential. We already know that there is a 1-bit mutation that increases the potential to $i+k$ by adding a set covering k new elements, for some k. Note that this k might change over time as the solution on P_i might be replaced by another one where k is different.

For every k as above, the probability of increasing the potential from i to $i+k$ is at least $1/(en) \cdot 1/k$ as $1/(en)$ is a lower bound for making the right mutation and $1/k$ is the probability that the mutant is migrated to island P_{i+k}. The expected increase of the potential in a single generation is therefore at least

$$k \cdot \frac{1}{en} \cdot \frac{1}{k} = \frac{1}{en}.$$

This bound holds regardless of the specific value of k. Thus the expected number of generations for increasing the potential from 0 to m is at most enm. □

Smart migration

With smart migration each island i sends migrants to island $c(\tilde{X}_i)$, where $\tilde{X}_i$ is the offspring generated on island i. Smart migration sends emigrants only to the unique island where they are considered feasible. The proof of Theorem 15 only relies on such migrations. Hence the upper bound also holds for smart migration.

Theorem 18. *The heterogeneous island model with smart migration finds an H_m-approximate solution for* SetCover *in an expected parallel time of $O(n \cdot \min(m, n))$. The expected communication effort is $O(nm \cdot \min(m, n))$.*

In our setting, smart migration outperforms all other migration policies as it leads to the best upper bound for the communication effort.

6.4 Summary of the results and contributions

The main contribution of this chapter is to propose and analyse two parallel EAs for the SetCover problem that provably find good approximations. Table 6.1 gives an overview of our results, regarding parallel and sequential expected running times as well as the communication effort. In order to fairly compare heterogeneous and homogeneous models we consider them running on μ processors. For the heterogeneous model this means (for $\mu \leq m$) running up to $\lceil \frac{m+1}{\mu} \rceil$ islands on the same processor and thus increasing the parallel running time by a factor of $\Theta(\frac{m}{\mu})$.

For the homogeneous model based on (global) SEMO, the topology determines how many islands still give a linear speedup. For dense topologies more islands can be used. The migration probability gives a smooth trade-off between this maximum number of islands and the communication effort. For large migration probabilities the heterogeneous island model based on the (1+1) EA (or RLS) has lower communication costs, when comparing complete topologies or using the right migration policies. It is also easier to implement as unlike for the SEMO-based model it is not necessary for each island to handle large populations and to remove many dominated solutions. Thus the heterogeneous model is also faster when considering the time and space required to compute a generation.

The discussion on migration policies has revealed how adding more knowledge about the problem can decrease the communication effort. The complete migration and uniform migration policies do not require any knowledge about the problem at hand, while non-uniform migration only needs a sensible ordering of islands to work. This ordering should

Algorithm	parallel time bounds general b.	$\rightsquigarrow$ best bound	seq. time	comm. effort
Non-parallel SEMO	$O(nm^2)$	$\rightsquigarrow O(nm^2)$	$O(nm^2)$	0
Homogeneous island model based on (global) SEMO and topology…				
– ring ($\mu \leq \sqrt{pnm}$)	$O\left(\frac{nm^2}{\mu}\right)$	$\rightsquigarrow O\left(\frac{n^{1/2}m^{3/2}}{p^{1/2}}\right)$	$O(nm^2)$	$O(pnm^3)$
– grid ($\mu \leq (pnm)^{2/3}$)	$O\left(\frac{nm^2}{\mu}\right)$	$\rightsquigarrow O\left(\frac{n^{1/3}m^{4/3}}{p^{2/3}}\right)$	$O(nm^2)$	$O(pnm^3)$
– complete ($\mu \leq pnm$)	$O\left(\frac{nm^2}{\mu}\right)$	$\rightsquigarrow O\left(\frac{m}{p}\right)$	$O(nm^2)$	$O(p^2n^2m^4)$
Heterogeneous island model with $\mu \leq m$ based on (1+1) EA (or RLS) and policy…				
– complete	$O\left(\frac{nm^2}{\mu}\right)$	$\rightsquigarrow O(nm)$	$O(nm^2)$	$O(nm^3)$
– uniform prob.	$O\left(\frac{nm^2}{\mu p}\right)$	$\rightsquigarrow O\left(\frac{nm}{p}\right)$	$O\left(\frac{nm^2}{p}\right)$	$O(nm^3)$
– non-uniform prob.	$O\left(\frac{nm^2}{\mu}\right)$	$\rightsquigarrow O(nm)$	$O(nm^2)$	$O(nm^2 \log m)$
– smart migration	$O\left(\frac{nm^2}{\mu}\right)$	$\rightsquigarrow O(nm)$	$O(nm^2)$	$O(nm^2)$

Table 6.1: Upper bounds on expected parallel times (general bounds and bounds for best μ), expected sequential times and expected communication effort for homogeneous island models with various migration topologies and for heterogeneous island models with various migration policies, until an H_m-approximation is found for any SetCover instance with m elements and n sets. p denotes the migration probability. We simplified $\min(n, m) \leq m$ and we constrained μ to yield linear speedups.

be consistent with the similarity between different islands. We believe that this approach can be fruitful for other heterogeneous island models. Smart migration requires knowledge about the problem at hand since it needs to inspect the genotype to determine the island to send it to. But it leads to the best performance guarantees among all considered policies.

CHAPTER 7

SCHEMES FOR ADAPTING MIGRATION INTERVALS IN ISLAND MODELS

As discussed in Section 2.4.1 designing an effective parallel evolutionary algorithm can be challenging as the method and amount of communication needs to be tuned carefully. Too frequent communication leads to high communication costs, and it can compromise exploration. Too little communication means that the populations become too isolated and unable to coordinate their searches effectively. There is agreement that even the effect of the most fundamental parameters on performance is not well understood [47, 2], which makes the tuning of parallel EAs really hard.

In this chapter we make a contribution towards finding good values for the migration interval, the parameter describing the frequency of migration. We propose adaptive schemes that adjust the migration interval, depending on whether islands have managed to find improvements during the last migration interval or not. The goal is to reduce communication, while not compromising the exploitation of good solutions. The main idea is: if an island has improved its current best fitness, migration is intensified to spread this solution to other islands. Otherwise, islands decrease the frequency of migration in order to avoid large communication costs.

Two different adaptive schemes are proposed. In both of them islands have individual migration intervals which are adapted throughout the run. In Scheme A if an island has not improved its current best fitness during the last migration interval, its migration interval is doubled. Once an improvement is found, the migration interval is set to 1 to communicate this new solution quickly. In Scheme B an island also doubles the migration interval when no improvement was found, while when an improvement is found, it halves τ at the end of the current migration interval.

We show that doubling the migration interval guarantees for elitist EAs that the number of migrations from an island is logarithmic in the time this island spends on a certain fitness level, i. e. for any value of the current best fitness.

We contribute a rigorous analytical framework that yields upper bounds on the expected optimisation time and the expected communication effort, defined as the total number of migrants sent. This is done for fixed migration intervals in Section 7.2, Scheme A in Section 7.3, and Scheme B in Section 7.4. Our adaptive schemes are then compared in Section 7.5 against the best fixed values of the migration interval for classical test problems. The results reveal that our adaptive schemes are able to match or even outperform the best fixed migration intervals with regard to the upper bounds of the expected parallel time and the upper bounds of the expected communication effort.

This chapter is based on [51]. My contribution consisted in inventing the design of the two adaptive schemes and in the results on common test functions presented in Section 7.5 and summarized in Table 7.1. I have also contributed in the statement and proofs of Theorems 20, 21, 22 for the case of complete topologies.

7.1 Preliminaries

We define the parallel EAs considered in this chapter, which contain our adaptive schemes. Our analytical framework is applicable to all kinds of elitist EAs: EAs that do not lose their current best solution. We define our schemes for maximisation problems.

Scheme A (Algorithm 8) maintains a migration interval τ_i for each island. As soon as the current best fitness on an island has improved through evolution, the island communicates this solution to its neighbouring islands. In this case, or when the best fitness increases after immigration, the migration interval for that island drops to 1. This implies that copies of a high-fitness immigrant are propagated to all neighbouring islands in the next generation. If no improvement of the current best fitness is found after τ_i generations, the migration interval τ_i doubles.

Algorithm 8 Elitist parallel EA with adaptive Scheme A

```
1: Initialize $\lambda$ islands $P_1^1, \dots, P_1^\lambda$ uniformly at random and let $\tau_i := 1$ and $u_i := 1$ for all $1 \leq i \leq \lambda$.
2: Let $f_i^*$ be the best fitness value for each island $P_1^1, \dots, P_1^\lambda$.
3: for $t := 1$ to $\infty$ do
4:   for all $1 \leq i \leq \lambda$ in parallel do
5:     Simulate one generation of the EA and create a new population $P_{t+1}^i$.
6:     Update $P_{t+1}^i$ by selecting a new population from the union of $P_t^i$ and $P_{t+1}^i$, keeping a best individual. Let $f_i^{*\prime}$ be the best fitness value in $P_{t+1}^i$.
7:     if $u_i \geq \tau_i$ or $f_i^{*\prime} > f_i^*$ then
8:       Send a copy of a fittest offspring in $P_{t+1}^i$ to all neighbouring islands.
9:     end if
10:    Update $P_{t+1}^i$ by selecting a new population from the union of $P_{t+1}^i$ and all immigrants, keeping a best individual. Let $f_i^{*\prime}$ be the best fitness value in $P_{t+1}^i$.
11:    if $u_i \geq \tau_i$ or $f_i^{*\prime} > f_i^*$ then
12:      if $f_i^{*\prime} > f_i^*$ then
13:        Let $\tau_i := 1$ and $f_i^* := f_i^{*\prime}$
14:      else
15:        Let $\tau_i := \tau_i \cdot 2$
16:      end if
17:      Let $u_i := 0$
18:    end if
19:    $u_i = u_i + 1$
20:  end for
21: end for
```

For the purpose of a theoretical analysis, we assume that all islands run in synchronicity: the t-th generation is executed on all islands at the same time. However, this is not a restriction of our adaptive scheme as it can be applied in asynchronous parallel architectures using message passing for implementing migration.

Inspired by [40], we also consider a Scheme B (see Algorithm 9) where the migration interval is being halved (instead of being set to 1) once an improvement has been detected. In contrast to Scheme A, this change is not implemented immediately, but only after the current migration period has ended. A flag "success$_i$" is used to indicate whether a success

Algorithm 9 Elitist parallel EA with adaptive Scheme B

```
1: Initialize $\lambda$ islands $P_1^1, \ldots, P_1^\lambda$ uniformly at random and let $\tau_i := 1$, $u_i := 1$, and
   success$_i$ := false for all $1 \leq i \leq \lambda$.
2: Let $f_i^*$ be the best fitness value for each island $P_1^1, \ldots, P_1^\lambda$.
3: for $t := 1$ to $\infty$ do
4:   for all $1 \leq i \leq \lambda$ in parallel do
5:     Simulate one generation of the EA and create a new population $P_{t+1}^i$.
6:     Update $P_{t+1}^i$ by selecting a new population from the union of $P_t^i$ and $P_{t+1}^i$, keeping
       a best individual.
7:     if $u_i \geq \tau_i$ then
8:       Send a copy of a fittest offspring in $P_{t+1}^i$ to all neighbouring islands.
9:     end if
10:    Update $P_{t+1}^i$ by selecting a new population from the union of $P_{t+1}^i$ and all immi-
       grants, keeping a best individual. Let $f_i^{*\prime}$ be the best fitness value in $P_{t+1}^i$.
11:    if $f_i^{*\prime} > f_i^*$ then
12:      Let $f_i^* := f_i^{*\prime}$ and success$_i$ := true
13:    end if
14:    if $u_i \geq \tau_i$ then
15:      if success$_i$ then
16:        Let $\tau_i := \lceil \tau_i/2 \rceil$
17:      else
18:        Let $\tau_i := \tau_i \cdot 2$
19:      end if
20:      Let $u_i := 0$ and success$_i$ := false
21:    end if
22:    $u_i = u_i + 1$
23:  end for
24: end for
```

on island i has occurred in the current migration period. The advantage of Scheme B is that it uses less communication than Scheme A, and if there is a good region in the parameter space of τ, our hope is that it will maintain a good parameter value in that region over time.

We provide general methods for analysing the expected parallel time and the expected communication effort for arbitrary elitist EAs that migrate copies of selected individuals (the original individuals remain on their island). As in Section 6.1, the parallel time is defined as the number of generations until a global optimum is found and denoted T^{par}, while the communication effort T^{com} is defined as the total number of individuals migrated until a global optimum is found. For simplicity and ease of presentation, we assume that each migration only transfers one individual; if $\nu > 1$ individuals migrate, the communication effort has to be multiplied by ν.

In order to demonstrate and illustrate this approach, we consider one simple algorithm in more detail: following [40], the parallel (1+1) EA is a special case where each island runs a (1+1) EA.

In terms of communication topologies, for Scheme A we consider general graphs on λ vertices and the following common graphs. A unidirectional ring is a ring with edges going in one direction. A grid graph contains undirected edges with vertices arranged on a 2-dimensional grid. A torus can be regarded a grid where edges wrap around horizontally and vertically. A hypercube graph of dimension d contains 2^d vertices. Each vertex has a d-bit label, and vertices are neighboured if and only if their labels differ in exactly one bit. The complete graph contains all possible edges. For Scheme B we only consider complete topologies as on sparse topologies migration intervals may adapt too slowly (see Section 7.4). Then all islands use the same migration interval.

The diameter $\mathrm{diam}(G)$ of a graph G with λ vertices is defined as the largest number of edges on any shortest path between two vertices. The unidirectional ring has the largest diameter of λ. The diameter of any grid or torus graph with side lengths $\sqrt{\lambda} \times \sqrt{\lambda}$ is at most $2\sqrt{\lambda}$. The diameter of a $(\log \lambda)$-dimensional hypercube is $\log \lambda$, and that of a complete topology is 1.

7.2 Fixed Migration Intervals

In order to compare our adaptive schemes against fixed migration intervals, we first need to investigate the latter. Lässig and Sudholt [39, 42] presented general upper bounds for the parallel optimisation time of island models with different topologies. Their method is based on the so-called fitness-level method, also known as fitness-based partitions [90].

We use a special case of this method: without loss of generality consider a problem with fitness values $1, \dots, m$. Consider fitness-level sets $A_1, \dots, A_m$ such that A_i contains all points with fitness i. In particular, A_m contains all global optima. We further assume that, if the current best individual of a population is in A_i, there is a lower bound s_i for the probability of finding a higher fitness level $A_{i+1} \cup \cdots \cup A_m$ in one generation. It is easy to show that then $\sum_{i=1}^{m-1} 1/s_i$ is an upper bound for the expected running time of an elitist EA.

Lässig and Sudholt [39, 41] showed how upper bounds on the parallel optimisation time can be derived through general functions on these success probabilities $s_1, \ldots, s_{m-1}$. They considered migration in every generation ($\tau = 1$) [41] as well as probabilistic migration, where every island independently decides for each neighbouring island whether migration occurs, and the probability for a migration is a fixed parameter p [42].

The following theorem provides results for periodic migration with migration interval τ. The proof is not included here since it is an adaptation of the one for probabilistic migration in [42]. The results for the expected communication effort on a topology with edge set E follow from multiplying the expected parallel time by $|E|/\tau$, as this term reflects the average number of migrated individuals across the topology in one generation. The upper bounds on the expected parallel time can be derived as in [42].

Theorem 19. *Consider an island model with λ islands where each island runs an elitist EA. Every τ iterations each island sends a copy of its best individual to all neighbouring islands. Each island incorporates the best out of its own individuals and its immigrants. For fitness-level sets $A_1, \ldots, A_m$, A_i containing all points of the i-th fitness value, let s_i be a lower bound for the probability that in one generation an island in A_i finds a search point in $A_{i+1} \cup \cdots \cup A_m$. Then the expected parallel optimization time $\mathrm{E}\left(T^{\mathrm{par}}\right)$ and the expected communication effort $\mathrm{E}\left(T^{\mathrm{com}}\right)$ are at most*

$$\mathrm{E}\left(T^{\mathrm{par}}\right) \leq 2\tau^{1/2} \sum_{i=1}^{m-1} \frac{1}{s_i^{1/2}} + \frac{1}{\lambda} \sum_{i=1}^{m-1} \frac{1}{s_i}$$

$$\mathrm{E}\left(T^{\mathrm{com}}\right) \leq 2\lambda\tau^{-1/2} \sum_{i=1}^{m-1} \frac{1}{s_i^{1/2}} + \frac{1}{\tau} \sum_{i=1}^{m-1} \frac{1}{s_i}$$

for every unidirectional ring,

$$\mathrm{E}\left(T^{\mathrm{par}}\right) \leq 3\tau^{2/3} \sum_{i=1}^{m-1} \frac{1}{s_i^{1/3}} + \frac{1}{\lambda} \sum_{i=1}^{m-1} \frac{1}{s_i}$$

$$\mathrm{E}\left(T^{\mathrm{com}}\right) \leq 12\lambda\tau^{-1/3} \sum_{i=1}^{m-1} \frac{1}{s_i^{1/3}} + \frac{4}{\tau} \sum_{i=1}^{m-1} \frac{1}{s_i}$$

for an undirected grid or torus with side lengths $\sqrt{\lambda} \times \sqrt{\lambda}$,

$$\mathrm{E}\left(T^{\mathrm{par}}\right) \leq \tau m + \tau \sum_{i=1}^{m-1} \log\left(\frac{1}{\tau s_i}\right) + \frac{1}{\lambda} \sum_{i=1}^{m-1} \frac{1}{s_i}$$

$$\mathrm{E}\left(T^{\mathrm{com}}\right) \leq \lambda(\log\lambda)m + \lambda(\log\lambda) \sum_{i=1}^{m-1} \log\left(\frac{1}{\tau s_i}\right) + \frac{\log\lambda}{\tau} \sum_{i=1}^{m-1} \frac{1}{s_i}$$

for the $(\log\lambda)$-dimensional hypercube, and

$$\mathrm{E}\left(T^{\mathrm{par}}\right) \leq m\tau + m + \frac{1}{\lambda} \sum_{i=1}^{m-1} \frac{1}{s_i}$$

$$\mathrm{E}\left(T^{\mathrm{com}}\right) \leq \lambda(\lambda-1)m + \frac{\lambda(\lambda-1)m}{\tau} + \frac{\lambda-1}{\tau} \sum_{i=1}^{m-1} \frac{1}{s_i}$$

for the complete topology K_λ.

These bounds match the ones from [43] for the case of probabilistic migration, if we compare τ against a migration probability of $p = 1/\tau$; the constant factors here are even better. The constants for probabilistic migration are higher to account for the variation in the spread of information. Periodic migration is more reliable in this respect since information is guaranteed to be spread every τ generations.

7.3 Adaptive Scheme A

In this section we analyse Scheme A on different topologies, including those from Theorem 19. Note that whenever an island improves its current best solution, a copy of this solution is being spread to all neighbouring islands immediately. Thus, good fitness levels spread in the same way as a migrating in every generation would do, i. e., using a global parameter $\tau = 1$. This means that the upper bounds from Theorem 19 apply for $\tau = 1$.

Theorem 20. *For Scheme A on topologies from Theorem 19, the expected parallel optimisation time is bounded from above as in Theorem 19 with $\tau = 1$.*

Note that the bounds on the expected parallel time from Theorem 19 are minimised for $\tau = 1$. This implies that we get upper bounds on the expected parallel time equal to the best upper bounds for any fixed choice of the migration interval. In case these bounds are asymptotically tight, this means that our adaptive Scheme A never increases the expected parallel running time asymptotically.

The intended benefit of Scheme A comes from a reduced communication effort as all islands decrease communication while no improvement is encountered through either variation or immigration. The expected communication effort is bounded from above in the following theorem. The main observation is that for each fitness level, the number of migrations from an island is logarithmic in the time it remains on that fitness level. For an upper bound we consider the expected worst-case time spent on a fitness level A_i, where the worst case is taken over all populations with its best individual in A_i.

Theorem 21. *Consider Scheme A on an arbitrary communication topology $G = (V, E)$ with diameter $\mathrm{diam}(G)$. Let $\mathrm{E}\left(T_i^{\mathrm{par}}\right)$ be (an upper bound on) the worst-case expected number of generations during which the current best search point in the island model is on fitness level i. Then the expected communication effort is at most*

$$\mathrm{E}\left(T^{\mathrm{com}}\right) \leq |E| \sum_{i=1}^{m-1} \log\left(\mathrm{E}\left(T_i^{\mathrm{par}}\right) + \mathrm{diam}(G)\right).$$

Proof. Initially, and after improving its current best fitness, an island will double its migration interval until its current best fitness improves again. If the current best fitness does not improve for t generations, the island will perform at most $\log t$ migrations.

Consider an island v after reaching fitness level i for the first time, either through variation or immigration. If no other island has found a better fitness level, the random parallel time for some island finding such an improvement is given (or bounded) by T_i^{par}. Then this solution (or another solution of fitness better than i) will be propagated through the topology, advancing to neighbouring islands in every generation. Hence, some solution on a better fitness level than i will eventually reach v within the following $\text{diam}(G)$ generations. The latter also holds if some island has already found a better fitness level than i at the time v reaches fitness level i. In any case, the total time v will spend on fitness level i is at most $T_i^{\text{par}} + \text{diam}(G)$.

An island v in the topology with outdegree $\deg^+(v)$ will send $\deg^+(v)$ individuals during each migration. Hence, the total number of migrated solutions in t generations on fitness level i is at most

$$\sum_{v \in V} \deg^+(v) \cdot \log t = |E| \log t.$$

The expected communication effort, therefore, is at most

$$\begin{aligned}
\mathrm{E}\left(T^{\text{com}}\right) &\leq |E| \cdot E[\log(T_i^{\text{par}} + \text{diam}(G))] \\
&\leq |E| \cdot \log\left(E[T_i^{\text{par}} + \text{diam}(G)]\right) \\
&= |E| \log\left(E[T_i^{\text{par}}] + \text{diam}(G)\right),
\end{aligned}$$

where the inequality follows from Jensen's inequality and the fact that log is a concave function. □

The communication effort is proportional to the logarithm of the expected time spend on each fitness level. For functions that can be optimised by the island model in expected polynomial time, and for polynomial λ (note $\text{diam}(G) \leq \lambda$), this logarithm is always at most $O(\log n)$. Then Theorem 21 gives the following upper bound.

Corollary 2. *Consider a fitness function f with m fitness values, such that f is being optimised by the island model in an expected polynomial number of generations, for every* *initial population. If also* $\lambda = n^{O(1)}$*, we have*

$$\mathrm{E}\left(T^{\mathrm{com}}\right) \le O(|E|m \cdot \log n).$$

The expected parallel time on a fitness level can be smaller than a polynomial: if sufficiently many islands are being used, and the topology spreads information quickly enough, it can be logarithmic or even constant. For specific topologies we get the following results by combining Theorem 21 with bounds on the parallel time from Theorem 19.

Theorem 22. *Given success probabilities* $s_1, \dots, s_{m-1}$ *as in Theorem 19, the expected communication effort for Scheme A is bounded from above for certain topologies with* λ *islands as follows:*

1. $\lambda \sum_{i=1}^{m-1} \log\left(\frac{1}{s_i^{1/2}} + \frac{1}{\lambda s_i} + \lambda\right)$ *for a unidirectional ring,*
2. $4\lambda \sum_{i=1}^{m-1} \log\left(\frac{1}{s_i^{1/3}} + \frac{1}{\lambda s_i} + 2\sqrt{\lambda}\right)$ *for every undirected grid or torus with side lengths* $\sqrt{\lambda} \times \sqrt{\lambda}$*,*
3. $\lambda(\log\lambda) \sum_{i=1}^{m-1} \log\left(\log\left(\frac{1}{s_i}\right) + \frac{1}{\lambda s_i} + \log\lambda\right)$ *for the* $(\log\lambda)$*-dimensional hypercube,*
4. $\lambda(\lambda-1) \sum_{i=1}^{m-1} \log\left(2 + \frac{1}{\lambda s_i}\right)$ *for the complete graph.*

We demonstrate the application of this theorem in Section 7.5.

7.4 Adaptive Scheme B

Scheme A resets the migration interval of an island to 1 every time an improvement is found. We propose Scheme B which halves this value instead. This may be advantageous if there is a "Goldilocks region" of good values for the migration interval. In contrast to Scheme A, Scheme B should be able to maintain a value in that region over several fitness levels.

When improvements are being found with probability p, good parameter values are close to $\tau \approx 1/(\lambda p)$, as then we find one improvement in expectation. If the current migration interval is much smaller, chances of finding an improvement are small, and τ is likely to increase. Likewise, if τ is large, the island will find an improvement with high probability and halve τ. Thus, the migration interval will reach an equilibrium state close to $1/(\lambda p)$.

Scheme B might have a smaller communication effort as it does not reset the migration interval to 1, where communication is most frequent. This, however, only holds if Scheme B does not lead to an increase in the parallel time. In fact, for sparse topologies G such as the unidirectional ring there is a risk that improvements may take a very long time to be communicated. If, say, all islands had the same migration interval τ, and one island found a new best solution, it may take up to $\text{diam}(G) \cdot \tau$ generations for this information to arrive at the last island.

Scheme B therefore makes more sense for dense topologies such as the complete graph, where $\text{diam}(G) = 1$ and decreases of τ quickly propagate to all islands.

We follow the analysis of Scheme B for adapting the number of islands in [40]. In both approaches over some time span a resource is being doubled and halved, depending on whether an improvement of the best fitness has been found. In [40] this resource is the number of islands, and hence the number of function evaluations executed in one generation. Here it is the number of generations within one migration period. The time span in [40] is just one generation, leading to the parallel time in [40] as performance measure. In our case the time span is the current migration interval, leading to the communication effort as performance measure.

For $\lambda = 1$ the parallel time in our work equals the sequential time in [40], and the communication effort equals the parallel time in [40]. However, a difference emerges for $\lambda > 1$ as in our scenario an island has $\lambda\tau$ trials to find an improvement, so the resources for finding improvements are by a factor of λ higher, compared to [40].

We adapt the analysis from [40] to accommodate this difference. The following lemma is a straightforward adaptation of parts of [40, Lemma 1], hence a proof is omitted. It estimates the expected communication effort for finding a single improvement, based on a given initial migration interval τ_0. We abbreviate $\max\{x, 0\}$ by $(x)^+$.

Lemma 3. *Assume an island model with a complete topology starts with a migration interval of τ_0 and that in each generation each island finds an improvement over the current best individual with probability at least p. Let T^{com} be the random number of individuals migrated between islands. Let τ^* be the migration interval at this time (before it is halved). Then for every $\alpha \in \mathbb{N}_0$*

1. $\Pr\left(\log(\tau^*) - \log(\tau_0) > (\log(1/(\lambda p)) - \log(\tau_0))^+ + \alpha\right) \leq \exp(-2^\alpha)$,
2. $\mathrm{E}\left(T^{\text{com}}\right) \leq \lambda(\lambda-1) \cdot \left((\log(1/(\lambda p)) - \log(\tau_0))^+ + 2\right)$.

The expected number of migrations is expressed as the difference between logarithms of the ideal value $1/(\lambda p)$ and the initial value τ_0. If the initial migration interval is larger, $\tau_0 \geq 1/(\lambda p)$, the expected number of migrations is just 2.

This fact is reflected in the following Theorem. The upper bound on the communication effort only contains values $\log(1/(\lambda s_j))$ when the migration interval needs to be increased, i. e. $s_j < s_{j-1}$. For the special case where fitness levels get progressively harder, $s_1 \geq s_2 \geq \dots \geq s_{m-1}$, the bound simplifies significantly.

Theorem 23. *Given success probabilities $s_1, \dots, s_{m-1}$ as in Theorem 19, for Scheme B we have*

$$\mathrm{E}\left(T_B^{\text{par}}\right) \leq 3m + \frac{3}{2\lambda}\sum_{i=1}^{m-1}\frac{1}{s_i} \quad \textit{and}$$

$$\mathrm{E}\left(T_B^{\text{com}}\right) \leq \lambda(\lambda-1) \cdot \left(3m + \log\left(\frac{1}{\lambda s_1}\right) + \sum_{j=2}^{m-1}\left(\log\left(\frac{1}{\lambda s_j}\right) - \log\left(\frac{1}{\lambda s_{j-1}}\right)\right)^+\right).$$

For $s_1 \geq s_2 \geq \dots \geq s_{m-1}$ the latter simplifies to

$$\mathrm{E}\left(T_B^{\text{com}}\right) \leq \lambda(\lambda-1) \cdot \left(3m + \log\left(\frac{1}{\lambda s_{m-1}}\right)\right).$$

The upper bound for the expected parallel time is only by a factor of 3/2 larger than the upper bound for Scheme A. Hence, both upper bounds are asymptotically equal. In other words, the reduced communication in Scheme B does not worsen the running time, for problems where the upper bounds for Schemes A and B are asymptotically tight.

We give an informal argument to convey the intuition for this result. Assume that Scheme B has raised the migration interval from 1 to some large value $\tau^* = 2^i$ before an improvement is found. Then Scheme B has spent $1+2+4+\dots+2^i = 2^{i+1}-1$ generations leading up to this value. In the worst case, the new migration interval $\tau^*/2 = 2^{i-1}$ is too large: improvements are found easily, and then the algorithm has to idle for nearly $\tau^*/2$ generations before being able to communicate the current best fitness. The total time spent is around $3/2 \cdot (2^{i+1} - 1)$, whereas Scheme A in the same situation would spend $2^{i+1} - 1$ generations. So, in this setting Scheme B needs at most 3/2 as many generations as Scheme A.

A formal argument was given in [40] to derive an upper bound for the parallel time of Scheme B, which is 3/2 that for Scheme A. The bound on the expected communication effort follows from similar arguments and applying our refined Lemma 3. We refrain from giving a formal proof here as it can be obtained with straightforward modifications from the proof of Theorem 3 in [40].

7.5 Performance on Common Example Functions

The analytical frameworks for analysing fixed migration intervals and our two adaptive schemes can be applied by simply using lower bounds on success probabilities for improving the current fitness. We demonstrate this approach, and how to use the results from the previous sections to analyse the parallel time and the communication effort on common test problems.

We provide an analysis for the maximisation of the same pseudo-Boolean test functions investigated in [43]. For a search point $x \in \{0,1\}^n$, $x = x_1 \dots x_n$, we define

OneMax$(x) := \sum_{i=1}^{n} x_i$ as the number of ones in x, and LO$(x) := \sum_{i=1}^{n} \prod_{j=1}^{i} x_i$ as the number of leading ones in x. We also consider the class of unimodal functions taking d fitness values $f_1, \dots, f_d$. A function is called unimodal if every non-optimal search point has an Hamming neighbour with strictly larger fitness. Finally for $1 \leq k \leq n$ we consider

$$\text{Jump}_k := \begin{cases} k + \sum_{i=1}^{n} x_i, & \text{if } \sum_{i=1}^{n} x_i \leq n - k \text{ or } x = 1^n \ , \\ \sum_{i=1}^{n} (1 - x_i) & \text{otherwise} \ . \end{cases}$$

Where a "jump", a mutation flipping k specific bits at the same time, has to be made. The parameter k tunes the difficulty of this function.

7.5.1 Fitness partition and success probabilites

In order to apply Theorems 19, 22, 23, it is just necessary to define probability s_i to move from the fitness level A_i to a better one. Recall that s_i is a lower bound on the probability of one island finding a search point of strictly higher fitness, given that the population has current best fitness i.

For the simple (1+1) EA, these values are easy to derive:

- For OneMax, a search point with i ones has $n-i$ zeros. The probability of a mutation flipping only one of these zeros is $s_i \geq (n-i)/n \cdot (1 - 1/n)^{n-1} \geq (n-i)/(en)$.

- For LO, it is sufficient to flip the first 0-bit, which has probability $s_i \geq 1/n \cdot (1 - 1/n)^{n-1} \geq 1/(en)$.

- For unimodal functions the success probability on each fitness level is at least $s_i \geq 1/n \cdot (1 - 1/n)^{n-1} \geq 1/(en)$ as for any non-optimal point there is always a better Hamming neighbour.

- For Jump$_k$ with $k \geq 2$, it is possible to find an improvement to an individual having up to $n - k$ 1-bits just increasing the number of ones, thus the s_i with $i < n - k$ are equal to the ones for OneMax. A similar argument applies to levels $n - k < i < n$.

From an individual with $n-k$ 1-bits an improvement is found by generating as offspring a specific bitstring having Hamming distance k from the parent, which has probability $s_{n-k} \geq \left(\frac{1}{n}\right)^k \cdot \left(1-\frac{1}{n}\right)^{n-k} \geq \left(\frac{1}{n}\right)^k \cdot \left(1-\frac{1}{n}\right)^{n-1} \geq \frac{1}{en^k}$.

7.5.2 Fixed scheme

Given Theorem 19 it is possible to bound the parallel time and the communication effort for fixed migration intervals similarly to [43]. For example, for OneMax and the complete topology we get an expected parallel time of $\mathrm{E}\left(T^{\mathrm{par}}\right) = O\left(n\tau + \frac{n \log n}{\lambda}\right)$ and an expected communication effort of $\mathrm{E}\left(T^{\mathrm{com}}\right) = O\left(\lambda^2 n + \frac{\lambda n \log n}{\tau}\right)$.

For fixed λ, the value of τ yields a trade-off between upper bounds for the parallel time and that of the communication effort. In our example, for say $\lambda = O(1)$, we get $\mathrm{E}\left(T^{\mathrm{par}}\right) = O\left(n\tau + n \log n\right)$ and $\mathrm{E}\left(T^{\mathrm{com}}\right) = O\left(n + \frac{n \log n}{\tau}\right)$. We can notice how a large τ always minimises the bound for the communication effort, while a small one (i. e. $\tau = 1$) minimises the bound for the parallel time.

Given a fixed number of islands λ, we define the best τ as the largest τ that does not asymptotically increase the bound for the parallel time (compared to $\tau = 1$). This assures good scalability while minimising the communication effort. For the example proposed the best τ is $\tau = \Theta(\frac{\log n}{\lambda})$. This leads to $\mathrm{E}\left(T^{\mathrm{par}}\right) = O(\frac{n \log n}{\lambda})$ and $\mathrm{E}\left(T^{\mathrm{com}}\right) = O(\frac{|E|}{\tau} \cdot \mathrm{E}\left(T^{\mathrm{par}}\right)) = O(\frac{\lambda^2}{\tau} \cdot \mathrm{E}\left(T^{\mathrm{par}}\right)) = O(\lambda^2 n)$. The results for other topologies and problems are summarised in Table 7.1. Notice that fixing τ to its best value is possible, provided that the number of island is small enough. Particularly for the example proposed the number of islands must be $\lambda = O(\log n)$ for the best τ to be defined ($\tau \geq 1$).

7.5.3 Adaptive scheme

In order to calculate the parallel time for the adaptive Scheme A we can refer to the results for the fixed scheme when $\tau = 1$, as shown in Theorem 20. For example, our Scheme A running on a complete topology solves OneMax in time $O(\frac{n \log n}{\lambda} + n)$. Scheme

B has asymptotically the same parallel time of Scheme A, as shown in Theorem 23.

We only consider values of λ that lead to a linear speedup as defined in [42]: the expected parallel time for λ islands is by a factor of λ smaller than the expected optimisation time of a single island, in an asymptotic sense. In this setting an island model thus makes the same expected number of function evaluations, compared to a single island. In the example proposed linear speedup is achieved for a number of islands up to $\lambda = O(\log n)$, in fact for a larger number of islands the upper bound on the parallel time would be $O(n)$ regardless of λ. The bounds on λ limit the best upper bound on the parallel time achievable with our analytical framework. Table 7.1 shows the bound on λ for different problems and topologies and the best achievable parallel time bound.

In order to calculate the communication effort we use Theorem 22 and 23 for Schemes A and B, respectively. We first get a general bound for every λ included in the linear speedup bound, and then we calculate it for the maximum value of λ, thus providing the communication effort for the value of λ leading to the best achievable parallel time. Table 7.1 shows all results derived in this manner. In the following we provide an example of this calculation.

Example: Communication effort of Scheme A for LO

We provide details on how to calculate the expected communication effort for Scheme A using Theorem 22, choosing LO as an example. Calculations for other test functions are similar. The purpose is to illustrate how we derived the results in Table 7.1.

In the following, λ is restricted to the cases leading to linear speedup as stated below and in Table 7.1. The calculations often use that $\log(a + b + c) \leq \log(3 \max(a, b, c))$.

- For the complete topology ($\lambda = O(n)$)

$$\begin{aligned}E(T_A^{\text{com}}) &\le \lambda(\lambda-1)\sum_{j=0}^{n-1} \log(2+\frac{en}{\lambda})\\ &\le \lambda(\lambda-1)\sum_{j=0}^{n-1}\left(\log\left(\frac{2en}{\lambda}\right)\right) = O\left(\lambda^2 n \log\left(\frac{n}{\lambda}\right)\right).\end{aligned}$$

If we set $\lambda = \Theta(n)$,

$$E(T_A^{\text{com}}) \le O\left(n^2 \sum_{j=0}^{n-1} \log\left(2+\frac{en}{n}\right)\right) = O(n^3).$$

- For ring ($\lambda = O(n^{1/2})$)

$$\begin{aligned}E(T_A^{\text{com}}) &= \lambda \sum_{i=0}^{n-1} \log\left(\frac{1}{s_i^{1/2}} + \frac{1}{\lambda s_i} + \lambda\right)\\ &\le \lambda \sum_{i=0}^{n-1} \log\left((en)^{1/2} + \frac{en}{\lambda} + \lambda\right)\\ &\le \lambda \sum_{i=0}^{n-1} \log\left(\frac{3en}{\lambda}\right)\\ &= O\left(\lambda n \log \frac{n}{\lambda}\right).\end{aligned}$$

If we set $\lambda = \Theta(n^{1/2})$ we obtain

$$\begin{aligned}E(T_A^{\text{com}}) &= O\left(n^{1/2} \sum_{i=1}^{n-1} \log((en)^{1/2} + en^{1/2} + n^{1/2})\right)\\ &= O(n^{3/2}\log n).\end{aligned}$$

- For the grid ($\lambda = O(n^{2/3})$) we get

$$
\begin{aligned}
E(T_A^{\mathrm{com}}) &= 4\lambda \sum_{i=1}^{n-1} \log\left(\frac{1}{s_i^{1/3}} + \frac{1}{\lambda s_i} + 2\sqrt{\lambda}\right) \\
&= O\left(\lambda n \log\left(\frac{n}{\lambda}\right)\right).
\end{aligned}
$$

If we set $\lambda = \Theta(n^{2/3})$

$$
\begin{aligned}
E(T_A^{\mathrm{com}}) &= O\left(n^{2/3} \sum_{i=1}^{n-1} \log\left((en)^{1/3} + en^{1/3} + 2n^{2/6}\right)\right) \\
&= O(n^{5/3} \log n).
\end{aligned}
$$

- For the hypercube $\left(\lambda = O\left(\frac{n}{\log n}\right)\right)$

$$
\begin{aligned}
E(T_A^{\mathrm{com}}) &= \lambda(\log \lambda) \sum_{i=1}^{n-1} \log\left(\log\left(\frac{1}{s_i}\right) + \frac{1}{\lambda s_i} + \log \lambda\right) \\
&\leq \lambda(\log \lambda) \sum_{i=1}^{m-1} \log\left(\log(en) + \frac{en}{\lambda} + \log \lambda\right) \\
&= O\left(\lambda(\log \lambda) n \log\left(\frac{n}{\lambda}\right)\right).
\end{aligned}
$$

If we set $\lambda = \Theta\left(\frac{n}{\log n}\right)$ we get

$$
\begin{aligned}
E(T_A^{\mathrm{com}}) &= O\Big(\frac{n}{\log n} \log\left(\frac{n}{\log n}\right) \sum_{i=1}^{m-1} \Big(\log(\log(en) + e\log n + \log\left(\frac{n}{\log n}\right)\Big)\Big) \\
&= O\left(\frac{n}{\log n} \log\left(\frac{n}{\log n}\right) n \log\log n\right) \\
&= O(n^2 \log\log n).
\end{aligned}
$$

Table 7.1: Overview of expected parallel optimisation times and expected communication efforts for island models with λ islands running a simple (1+1) EA on common example functions, derived using Theorems 19, 20, 21, and 23. As explained in Sections 7.5.2 and 7.5.3, the table shows restrictions on λ to yield linear speedups and fixed values for τ leading to the best upper bounds for the communication effort, while not increasing the parallel running time. For both parallel time and communication effort we show bounds for general λ in the realm of linear speedups, and the best parallel time $\mathrm{E}(T^{\mathrm{par}}_{\mathrm{best}})$ achieved by using the largest such λ, along with the corresponding communication effort $\mathrm{E}(T^{\mathrm{par}}_{\mathrm{best}})$ for the same λ. For Jump_k we assume $3 \leq k = O(n/(\log n))$.

Problem	Topology and Scheme	λ	$\mathrm{E}(T^{\mathrm{par}}) \rightsquigarrow \mathrm{E}(T^{\mathrm{par}}_{\mathrm{best}})$	$\mathrm{E}(T^{\mathrm{com}}) \rightsquigarrow \mathrm{E}(T^{\mathrm{com}}_{\mathrm{best}})$
OneMax	Complete, Scheme A	$O(\log n)$	$O\left(\frac{n}{\lambda}\log n\right) \rightsquigarrow O(n)$	$O(\lambda^2 n) \rightsquigarrow O(n\log^2(n))$
	Complete, Scheme B	$O(\log n)$	$O\left(\frac{n}{\lambda}\log n\right) \rightsquigarrow O(n)$	$O(\lambda^2 n) \rightsquigarrow O(n\log^2(n))$
	Complete, $\tau = \Theta\left(\frac{\log n}{\lambda}\right)$	$O(\log n)$	$O\left(\frac{n}{\lambda}\log n\right) \rightsquigarrow O(n)$	$O(\lambda^2 n) \rightsquigarrow O(n\log^2(n))$
	Ring, Scheme A	$O(\log n)$	$O\left(\frac{n}{\lambda}\log n\right) \rightsquigarrow O(n)$	$O(n\lambda\log\lambda) \rightsquigarrow O(n\log(n)\log\log(n))$
	Ring, $\tau = \Theta\left(\left(\frac{\log n}{\lambda}\right)^2\right)$	$O(\log n)$	$O\left(\frac{n}{\lambda}\log n\right) \rightsquigarrow O(n)$	$O\left(\frac{\lambda^2 n}{\log n}\right) \rightsquigarrow O(n\log(n))$
	Grid, Scheme A	$O(\log n)$	$O\left(\frac{n}{\lambda}\log n\right) \rightsquigarrow O(n)$	$O(n\lambda\log\lambda) \rightsquigarrow O(n\log(n)\log\log(n))$
	Grid, $\tau = \Theta\left(\left(\frac{\log n}{\lambda}\right)^{3/2}\right)$	$O(\log n)$	$O\left(\frac{n}{\lambda}\log n\right) \rightsquigarrow O(n)$	$O\left(\frac{\lambda^{3/2} n}{\log^{1/2}(n)}\right) \rightsquigarrow O(n\log(n))$
	Hypercube, Scheme A	$O(\log n)$	$O\left(\frac{n}{\lambda}\log n\right) \rightsquigarrow O(n)$	$O(n\lambda\log(\lambda)\log\log(\lambda))$ $\rightsquigarrow O(n\log(n)\log\log(n)\log\log\log(n))$
	Hypercube, $\tau = \Theta\left(\frac{\log n}{\lambda}\right)$	$O(\log n)$	$O\left(\frac{n}{\lambda}\log n\right) \rightsquigarrow O(n)$	$O(n\lambda\log\lambda) \rightsquigarrow O(n\log(n)\log\log(n))$
LO	Complete, Scheme A	$O(n)$	$O\left(\frac{n^2}{\lambda}\right) \rightsquigarrow O(n)$	$O\left(\lambda^2 n\log\left(\frac{n}{\lambda}\right)\right) \rightsquigarrow O(n^3)$
	Complete, Scheme B	$O(n)$	$O\left(\frac{n^2}{\lambda}\right) \rightsquigarrow O(n)$	$O\left(\lambda^2 n\right) \rightsquigarrow O(n^3)$
	Complete, $\tau = \Theta\left(\frac{n}{\lambda}\right)$	$O(n)$	$O\left(\frac{n^2}{\lambda}\right) \rightsquigarrow O(n)$	$O\left(\lambda^2 n\right) \rightsquigarrow O(n^3)$
	Ring, Scheme A	$O(n^{1/2})$	$O\left(\frac{n^2}{\lambda}\right) \rightsquigarrow O(n^{3/2})$	$O(\lambda n\log\left(\frac{n}{\lambda}\right)) \rightsquigarrow O(n^{3/2}\log n)$
	Ring, $\tau = \Theta\left(\frac{n}{\lambda^2}\right)$	$O(n^{1/2})$	$O\left(\frac{n^2}{\lambda}\right) \rightsquigarrow O(n^{3/2})$	$O(\lambda^2 n) \rightsquigarrow O(n^2)$
	Grid, Scheme A	$O(n^{2/3})$	$O\left(\frac{n^2}{\lambda}\right) \rightsquigarrow O(n^{4/3})$	$O(\lambda n\log\left(\frac{n}{\lambda}\right)) \rightsquigarrow O(n^{5/3}\log n)$
	Grid, $\tau = \Theta\left(\frac{n}{\lambda^{3/2}}\right)$	$O(n^{2/3})$	$O\left(\frac{n^2}{\lambda}\right) \rightsquigarrow O(n^{4/3})$	$O(n\lambda^{3/2}) \rightsquigarrow O(n^2)$
	Hypercube, Scheme A	$O\left(\frac{n}{\log n}\right)$	$O\left(\frac{n^2}{\lambda}\right) \rightsquigarrow O(n\log n)$	$O\left(\lambda\log(\lambda)n\log\left(\frac{n}{\lambda}\right)\right) \rightsquigarrow O(n^2\log\log n)$
	Hypercube, $\tau = \Theta\left(\frac{n}{\lambda\log n}\right)$	$O\left(\frac{n}{\log n}\right)$	$O\left(\frac{n^2}{\lambda}\right) \rightsquigarrow O(n\log n)$	$O(\lambda\log(\lambda)n\log(n)) \rightsquigarrow O(n^2\log n)$
unimodal f with d f-values	Complete, Scheme A	$O(n)$	$O\left(\frac{dn}{\lambda}\right) \rightsquigarrow O(d)$	$O(\lambda^2 d\log\left(\frac{n}{\lambda}\right)) \rightsquigarrow O(n^2 d)$
	Complete, Scheme B	$O(n)$	$O\left(\frac{dn}{\lambda}\right) \rightsquigarrow O(d)$	$O(\lambda^2 d) \rightsquigarrow O(n^2 d)$
	Complete, $\tau = \Theta\left(\frac{n}{\lambda}\right)$	$O(n)$	$O\left(\frac{dn}{\lambda}\right) \rightsquigarrow O(d)$	$O(\lambda^2 d) \rightsquigarrow O(n^2 d)$
	Ring, Scheme A	$O(n^{1/2})$	$O\left(\frac{dn}{\lambda}\right) \rightsquigarrow O(dn^{1/2})$	$O\left(\lambda d\log\left(\frac{n}{\lambda}\right)\right) \rightsquigarrow O(dn^{1/2}\log n)$
	Ring, $\tau = \Theta\left(\frac{n}{\lambda^2}\right)$	$O(n^{1/2})$	$O\left(\frac{dn}{\lambda}\right) \rightsquigarrow O(dn^{1/2})$	$O(\lambda^2 d) \rightsquigarrow O(dn)$
	Grid, Scheme A	$O(n^{2/3}))$	$O\left(\frac{dn}{\lambda}\right) \rightsquigarrow O(dn^{1/3})$	$O\left(\lambda d\log\left(\frac{n}{\lambda}\right)\right) \rightsquigarrow O(dn^{2/3}\log n)$
	Grid, $\tau = \Theta\left(\frac{n}{\lambda^{3/2}}\right)$	$O(n^{2/3}))$	$O\left(\frac{dn}{\lambda}\right) \rightsquigarrow O(dn^{1/3})$	$O(\lambda^{3/2} d) \rightsquigarrow O(dn)$
	Hypercube, Scheme A	$O\left(\frac{n}{\log n}\right)$	$O\left(\frac{dn}{\lambda}\right) \rightsquigarrow O(d\log n)$	$O\left(\lambda\log(\lambda)d\log\left(\frac{n}{\lambda}\right)\right) \rightsquigarrow O(dn\log\log n)$
	Hypercube, $\tau = \Theta\left(\frac{n}{\lambda\log n}\right)$	$O\left(\frac{n}{\log n}\right)$	$O\left(\frac{dn}{\lambda}\right) \rightsquigarrow O(d\log n)$	$O(\lambda\log(\lambda)d\log(n)) \rightsquigarrow O(dn\log n)$
Jump_k	Complete, Scheme A	$O(n^{k-1})$	$O\left(\frac{n^k}{\lambda}\right) \rightsquigarrow O(n)$	$O(\lambda^2 n) \rightsquigarrow O(n^{2k-1})$
	Complete, Scheme B	$O(n^{k-1})$	$O\left(\frac{n^k}{\lambda}\right) \rightsquigarrow O(n)$	$O(\lambda^2 n) \rightsquigarrow O(n^{2k-1})$
	Complete, $\tau = \Theta\left(\frac{n^{k-1}}{\lambda}\right)$	$O(n^{k-1})$	$O\left(\frac{n^k}{\lambda}\right) \rightsquigarrow O(n)$	$O(\lambda^2 n) \rightsquigarrow O(n^{2k-1})$
	Ring, Scheme A	$O(n^{k/2})$	$O\left(\frac{n^k}{\lambda}\right) \rightsquigarrow O(n^{k/2})$	$O(n\lambda\log\lambda) \rightsquigarrow O(kn^{k/2+1}\log n)$
	Ring, $\tau = \Theta\left(\frac{n^k}{\lambda^2}\right)$	$O(n^{k/2})$	$O\left(\frac{n^k}{\lambda}\right) \rightsquigarrow O(n^{k/2})$	$O(\lambda^2) \rightsquigarrow O(n^k)$
	Grid, Scheme A	$O\left(n^{2k/3}\right)$	$O\left(\frac{n^k}{\lambda}\right) \rightsquigarrow O(n^{k/3})$	$O(n\lambda\log\lambda) \rightsquigarrow O(kn^{2k/3+1}\log n)$
	Grid, $\tau = \Theta\left(\frac{n^k}{\lambda^{3/2}})\right)$	$O\left(n^{2k/3}\right)$	$O\left(\frac{n^k}{\lambda}\right) \rightsquigarrow O(n^{k/3})$	$O(\lambda^{3/2}) \rightsquigarrow O(n^k)$
	Hypercube, Scheme A	$O(n^{k-1})$	$O\left(\frac{n^k}{\lambda}\right) \rightsquigarrow O(n)$	$O(n\lambda\log(\lambda)\log\log(\lambda))$ $\rightsquigarrow O(kn^k\log(n)\log\log(n^{k-1}))$
	Hypercube, $\tau = \Theta\left(\frac{n^{k-1}}{\lambda}\right)$	$O(n^{k-1})$	$O\left(\frac{n^k}{\lambda}\right) \rightsquigarrow O(n)$	$O(n\lambda\log\lambda) \rightsquigarrow O(kn^k\log n)$

7.5.4 Evaluation of results

Recall that Table 7.1 only shows results for linear speedups, hence all parallel times are equal, but the range of λ values varies between topologies.

Table 7.2 compares upper bounds from Table 7.1 on the communication efforts for the best fixed value of τ against our adaptive schemes. For OneMax on all topologies the communication effort is by a small $O(\log \log n)$ term larger for the adaptive schemes, compared to the best fixed τ. The latter varies according to the topology: it is $\tau = ((\log n)/\lambda)^2$ for the ring, $\tau = ((\log n)/\lambda)^{3/2}$ for the grid, and $\tau = (\log n)/\lambda$ for the hypercube and the complete graph. So, the additional $O(\log \log n)$ factor is a small price to pay for the convenience of adapting τ automatically.

For LO and the general bounds for unimodal functions, Scheme A on the ring has a communication effort of $O(n^{3/2} \log n)$ compared to $O(n^2)$, and for the grid it is $O(n^{5/3} \log n)$ versus $O(n^2)$. These significant improvements show that decreasing the migration interval is an effective strategy in lowering the communication costs, without harming the parallel running time. For the hypercube the communication effort is lower by a factor of $O(\log(n)/\log \log n)$, whereas for the complete graph no differences are visible in the upper bounds.

For Jump there are no differences for the complete graph, while on the hypercube Scheme A is by a $O(\log \log(n^{k-1}))$ factor worse than the best fixed value. For rings and grids the adaptive scheme is better; the performance gap even grows with k and hence the difficulty of the function.

Comparing Schemes A and B, both achieve similar results. For LO we see an advantage of Scheme B over Scheme A: the general bound for the communication effort of Scheme A is $O(\lambda^2 n \log(n/\lambda))$, whereas that for Scheme B is $O(\lambda^2 n)$. This makes sense as the probability for finding an improvement in one generation is of order $\Theta(\lambda/n)$ for the considered λ, and the ideal value for the migration interval is in the region of $\Theta(n/\lambda)$. Scheme A needs to increase the migration interval around $\log(n/\lambda)$ times to get into this range, which is precisely the performance difference visible in our upper bounds. The

	OneMax	LeadingOnes	Unimodal	Jump$_k$
Complete	1	1	1	1
Ring	$\frac{1}{\log\log n}$	$\frac{n^{1/2}}{\log n}$	$\frac{n^{1/2}}{\log n}$	$\frac{n^{k/2-1}}{k\log n}$
Grid/Torus	$\frac{1}{\log\log n}$	$\frac{n^{1/3}}{\log n}$	$\frac{n^{1/3}}{\log n}$	$\frac{n^{k/3-1}}{k\log n}$
Hypercube	$\frac{1}{\log\log\log n}$	$\frac{\log n}{\log\log n}$	$\frac{\log n}{\log\log n}$	$\frac{1}{\log\log\left(n^{k-1}\right)}$

Table 7.2: Comparison of communication efforts: the table shows the ratio of upper bounds on the communication effort from the rightmost column of Table 7.1 for the best fixed choice of τ and the best adaptive Scheme based on bounds from Table 7.1. A value less than 1 indicates that the best fixed τ leads to better bounds, and a value larger than 1 indicates a decrease of the communication effort by the stated factor. In all cases λ was chosen as the largest possible value that guarentees a linear speedup according to the above-mentioned upper bounds.

difference disappears for $\lambda = \Theta(n)$.

The same argument also applies to the more general function class of unimodal functions.

7.6 Summary of the results and contributions

The main contribution of this chapter is to propose and analyse two schemes for adapting the migration interval in island models. The analysis provides upper bounds on the expected parallel time and the expected communication effort, based on probabilities of fitness improvements in single islands running elitist EAs. The results show that our schemes are able to decrease the upper bound on the communication effort without significantly compromising exploitation. For arbitrary topologies, we got upper bounds on the expected parallel time no larger than those for maximum exploitation, that is, migration in every generation.

Example applications to common example functions revealed that, in the realm of linear speedups and comparing against the best fixed choice of the migration interval, the expected communication effort was larger by a tiny $O(\log\log n)$ term for OneMax and similarly for the hypercube on Jump, but significantly smaller on LO, for a general analysis of unimodal functions, and for rings and grids on Jump. Both schemes are able to match, in terms of upper bounds, the performance of the best fixed migration interval up to doubly logarithmic factors, or they can perform significantly better by polynomial factors.

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

The double aim of the thesis was to advance the theoretical understanding of Parallel Evolutionary Algorithms and Genetic Programming and to produce new theory grounded efficient designs in these two fields. In this chapter we will recap how we have accomplished these two objectives.

8.1 Genetic Programming

Concerning genetic programming we have started analysing the original mutation operators for Geometric Semantic Genetic Programming proposed in [59].

For the Boolean domain we have found out that SGMB takes exponential time to find an expression minimizing the training error in the black box Boolean learning problem even when the size of the training set is small (Section 3.3.2). Inspired by this analysis we have designed 4 block mutation operators (FBM, VBM, FABM, MSBM), two of which (FBM and FABM) can, with high probability, find an expression fitting the training set of a random Boolean problem in polynomial time when the size of the training set is polynomial in the size of the input variables and when the size of the blocks is set accordingly (Section 3.3.4). This claim is proved and the good setting for the size of the blocks is provided.

Experimental results comparing the average-case runtime of the 4 block mutations have shown that MSBM seems, on the average case, much superior to the others. Experiments comparing GSGP with Cartesian GP give indications of the superiority of GSGP to CGP when the training set encompass all the possible input-output pairs. In this situation of very large training set forcing point mutation (SGMB) is superior to MSBM.

For classification trees, after realising as for the Boolean domain that the original operator proposed in [59] would take exponential time to learn a classification tree, we have proposed a new block mutation operator (FBM) that can find the optimum of a random problem in polynomial time with high probability. Consideration on the fact that the size of the classification trees evolved with FBM grows exponentially with the number of generations, inspired the design of IFBM which has the same runtime and ability of finding solution of FBM, but produces classification trees growing linearly in size with the number of generations (Section 4.3). We have also proposed MSBM: while IFBM can find the optimum in expected polynomial time, but in some cases it cannot find it, MSBM is always able to find the optimum even if, for some combinations of training sets and problem instances, it may require exponential time.

For the basis function regression domain we have realised that the operator in [59] induces on the output vector a non-isotropic Gaussian mutation which has never been analysed before (Section 5.2.2). We have then redesigned the mutation operator to induce a known isotropic Gaussian mutation on the output vector and used the theoretical results on (1+1)-ES on Sphere function to analyse the convergence time of the new operator (Section 5.3). This new operator requires to know the input values of the training points, which breaks the black-box constraints. We have then provided a real black-box method which uses a surrogate grid instead of the training input. Experiments have confirmed that the surrogate grid method works and that the new operator, inducing on the output vector an isotropic Gaussian mutation, outperforms the original operator proposed in [59] (Section 5.4).

So, for all three domains we have analysed the existing operators, providing rigorously proved runtime results which were not available before. We were then inspired by the analysis to improve the existing operators proposing novel ones and providing both theoretical and experimental analysis for them.

8.1.1 Future work

The size of individuals is an important issue in GSGP. In designing mutation operators we have always tried to keep the growth in size of the individuals at most asymptotically linear in the number of generations. If this is, on one hand, a remarkable accomplishment since it gives *guarantees* on the size of individuals (compared to traditional designs for which the bloat is not predictable), on the other hand it might be not good enough in practice. Moreover our analysis has not taken into consideration crossover, mainly due to the fact that the geometric semantic crossover operators available at the moment makes the size of the individuals to grow exponentially in the number of generations [59]. Approximated geometric semantic crossover operators, keeping the size of individuals low, have been proposed [72]. However since these operators don't behave as exact geometric operators their rigorous analyses is still an open problem.

Another important future work on GSGP is the study of the generalization ability. In Section 4.5 we have proposed a framework to measure the generalization ability of an EA. This can be used, in future research, to calculate the generalization ability of the operators proposed in this thesis and to guide the design towards operators guaranteeing a good generalization ability while keeping the runtime low.

8.2 Parallel Evolutionary Algorithms

Concerning Parallel Evolutionary algorithms we have started realizing that there is no rigorous analysis on how island models perform on NP-hard problems, or how they deal with multi-objective fitness functions. We have then produced an analysis of an homogeneous island model finding, on each island, a good approximation of a multi-objective formulation of the NP-hard problem SetCover (Section 6.2)

The analysis highlighted that this model needs a large number of individuals to be exchanged throughout the run. It then inspired the design of an heterogeneous island model in which each island optimises a different fitness function, thus taking care of a different portion of the search space. This model can guarantee the same upper bounds on the runtime while needing less communication effort (Section 6.3).

Another open problem of Parallel Evolutionary Algorithms is how to set the migration interval. We have then proposed and analysed two new adaptive schemes for the migration interval and we have rigorously analysed them on common test functions and for different topologies (Chapter 7). In the realm of linear speedups these schemes can guarantee the same upper bound on the runtime of the optimal choice of the migration interval just needing a communication effort larger by a tiny $O(\log \log n)$ term for OneMax and similarly for the hypercube on Jump, but significantly smaller on LO, for a general analysis of unimodal functions, and for rings and grids on Jump. The optimal fixed migration interval is difficult to calculate for general problems, so it is an important result to have an adaptive scheme (thus needing no tuning) which can perform equally good, double-logarithmically worse or even polynomially better than the best fixed choice, for the problem considered in this study and in case the upper bounds are tight.

8.2.1 Future work

The heterogeneous island model proposed for SetCover could be applied to other problems having a multi-objective representation. Particularly it could be easily applied to any 2-objectives problem in which one objective is any real function $f_1 : S \to \mathbb{R}$, where S is the search space, while the second objective is a function taking a limited number of values from an alphabet $f_2 : S \to \Sigma$, with $|\Sigma| \in \mathbb{N}$. Other combinatorial problems admit this representation (for example in VertexCover f_1 would be the cost of the cover, while f_2 would be the number of covered vertices) and future work can analyse how this heterogeneous island model can perform on them.

In both Chapter 6 and 7 our analyses have provided upper bounds for the expected running time and the expected communication effort. If this is, on one hand, an important step towards the understanding of parallel EAs and it gives guarantees on the expected time to find a good solution (optimal or with a given approximation ratio), on the other hand it does not allow to make a solid comparison between different approaches. Understanding whether these bounds are tight, by providing corresponding *lower bounds*, is thus a important open problem which future research should tackle. A promising direction is using black-box complexity [18, 44], which describes universal lower bounds on the expected (worst-case) running time of *every* black-box algorithm on a given class of functions. Recent advances towards a black-box complexity for parallel and distributed black-box algorithms have been made [5, 6], which include island models using mutation for variation.

Finally, real implementations of parallel algorithms have to take into consideration issues that are not considered in the theoretical models, like synchronization and limited bandwidth and cache. Understanding how the models proposed perform when deployed on real parallel architecture and how to implement them efficiently is thus another open direction for future research.

LIST OF REFERENCES

[1] Enrique Alba. Parallel evolutionary algorithms can achieve super-linear performance. *Information Processing Letters*, 82(1):7–13, 2002.

[2] Enrique Alba, Gabriel Luque, and Sergio Nesmachnow. Parallel metaheuristics: Recent advances and new trends. *International Transactions in Operational Research*, 20(1):1–48, 2013.

[3] Lourdes Araujo and Juan Julián Merelo Guervós. Diversity through multiculturality: Assessing migrant choice policies in an island model. In *IEEE Transactions Evolutionary Computation*, pages 456–469, 2011.

[4] Anne Auger and Benjamin Doerr. *Theory of Randomized Search Heuristics: Foundations and Recent Developments.* Series on theoretical computer science. World Scientific, 2011.

[5] Golnaz Badkobeh, Per Kristian Lehre, and Dirk Sudholt. Unbiased black-box complexity of parallel search. In *13th International Conference on Parallel Problem Solving from Nature (PPSN 2014)*, volume 8672 of *LNCS*, pages 892–901. Springer, 2014.

[6] Golnaz Badkobeh, Per Kristian Lehre, and Dirk Sudholt. Black-box complexity of parallel search with distributed populations. In *Proceedings of Foundations of Genetic Algorithms (FOGA 2015)*. ACM Press, 2015. To appear.

[7] Lawrence Beadle and Colin G. Johnson. Sematically driven crossover in genetic programming. In *Proceedings of IEEE WCCI '08*, pages 111–116, 2008.

[8] Lawrence Beadle and Colin G. Johnson. Semantic analysis of program initialisation in genetic programming. *Genetic Programming and Evolvable Machines*, 10(3):307–337, 2009.

[9] Lawrence Beadle and Colin G. Johnson. Semantically driven mutation in genetic programming. In *Proceedings of IEEE CEC '09*, pages 1336–1342, 2009.

[10] Markus F Brameier and Wolfgang Banzhaf. *Linear genetic programming*. Springer, 2006.

[11] Yesnier Bravo, Gabriel Luque, and Enrique Alba. Influence of the migration period in parallel distributed GAs for dynamic optimization. In *Proceedings of LION*, pages 343–348, 2012.

[12] Tianshi Chen, Jun He, Guoliang Chen, and Xin Yao. Choosing selection pressure for wide-gap problems. *Theoretical Computer Science*, 411(6):926–934, 2010.

[13] Tianshi Chen, Jun He, Guangzhong Sun, Guoliang Chen, and Xin Yao. A new approach for analyzing average time complexity of population-based evolutionary algorithms on unimodal problems. *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, 39(5):1092–1106, 2009.

[14] Tianshi Chen, Ke Tang, Guoliang Chen, and Xin Yao. A large population size can be unhelpful in evolutionary algorithms. *Theoretical Computer Science*, 436:54–70, 2012.

[15] Vaclav Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3):233–235, 1979.

[16] Benjamin Doerr, Anton V. Eremeev, Frank Neumann, Madeleine Theile, and Christian Thyssen. Evolutionary algorithms and dynamic programming. *Theoretical Computer Science*, 412(43):6020–6035, 2011.

[17] Stefan Droste, Thomas Jansen, and Ingo Wegener. A rigorous complexity analysis of the (1 + 1) evolutionary algorithm for separable functions with boolean inputs. *Evolutionary Compututation*, 6(2):185–196, June 1998.

[18] Stefan Droste, Thomas Jansen, and Ingo Wegener. Upper and lower bounds for randomized search heuristics in black-box optimization. *Theory of Computing Systems*, 39(4):525–544, 2006.

[19] Greg Durrett, Frank Neumann, and Una-May O'Reilly. Computational complexity analysis of simple genetic programming on two problems modeling isolated program semantics. In *Workshop on Foundations of Genetic Algorithms*, pages 69–80, 2011.

[20] Tobias Friedrich, Jun He, Nils Hebbinghaus, Frank Neumann, and Carsten Witt. Approximating covering problems by randomized search heuristics using multi-objective models. *Evolutionary Computation*, 18(4):617–633, 2010.

[21] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.

[22] Jun He, Tianshi Chen, and Xin Yao. On the easiest and hardest fitness functions. *IEEE Transactions on Evolutionary Computation*, DOI:10.1109/TEVC.2014.2318025, 2014.

[23] Jun He and Xin Yao. From an individual to a population: an analysis of the first hitting time of population-based evolutionary algorithms. *IEEE Transactions Evolutionary Computation*, 6(5):495–511, 2002.

[24] Holger H. Hoos. Programming by optimization. *Commun. ACM*, 55(2):70–80, February 2012.

[25] David Jackson. Phenotypic diversity in initial genetic programming populations. In *Proceedings of EuroGP 2010*, pages 98–109, 2010.

[26] Jens Jägersküpper. Algorithmic analysis of a basic evolutionary algorithm for continuous optimization. *Theoretical Computer Science*, 379(3):329–347, June 2007.

[27] Thomas Jansen. *Analyzing Evolutionary Algorithms - The Computer Science Perspective*. Natural Computing Series. Springer, 2013.

[28] Timo Kötzing, Frank Neumann, and Reto Spöhel. PAC learning and genetic programming. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, GECCO '11, pages 2091–2096, New York, NY, USA, 2011. ACM.

[29] Timo Kötzing, Dirk Sudholt, and Madeleine Theile. How crossover helps in pseudo-boolean optimization. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, GECCO '11, pages 989–996, New York, NY, USA, 2011. ACM.

[30] Timo Kötzing, Andrew M. Sutton, Frank Neumann, and Una-May O'Reilly. The max problem revisited: the importance of mutation in genetic programming. In *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference*, GECCO '12, pages 1333–1340, New York, NY, USA, 2012. ACM.

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

[32] Krzysztof Krawiec and Pawel Lichocki. Approximating geometric crossover in semantic space. In *Proceedings of GECCO '09*, pages 987–994, 2009.

[33] Krzysztof Krawiec and Tomasz Pawlak. Locally geometric semantic crossover. In *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference companion*, GECCO Companion '12, pages 1487–1488, New York, NY, USA, 2012. ACM.

[34] Krzysztof Krawiec and Tomasz Pawlak. Quantitative analysis of locally geometric semantic crossover. In Carlos A. Coello Coello, Vincenzo Cutello, Kalyanmoy Deb, Stephanie Forrest, Giuseppe Nicosia, and Mario Pavone, editors, *PPSN (1)*, volume 7491 of *Lecture Notes in Computer Science*, pages 397–406. Springer, 2012.

[35] Krzysztof Krawiec and Bartosz Wieloch. Analysis of semantic modularity for genetic programming. *Foundations of Computing and Decision Sciences*, 34(4):265–285, 2009.

[36] William B. Langdon, Tery Soule, Riccardo Poli, and James A. Foster. The evolution of size and shape. In Lee Spector, William B. Langdon, Una-May O'Reilly, and Peter J. Angeline, editors, *Advances in genetic programming*, pages 163–190. MIT Press, Cambridge, MA, USA, 1999.

[37] Jörg Lässig and Dirk Sudholt. The benefit of migration in parallel evolutionary algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2010)*, pages 1105–1112. ACM Press, 2010.

[38] Jörg Lässig and Dirk Sudholt. Experimental supplements to the theoretical analysis of migration in the island model. In *11th International Conference on Parallel Problem Solving from Nature (PPSN 2010)*, pages 224–233. Springer, 2010.

[39] Jörg Lässig and Dirk Sudholt. General scheme for analyzing running times of parallel evolutionary algorithms. In *Parallel Problem Solving from Nature (PPSN 2010)*, pages 234–243. Springer, 2010.

[40] Jörg Lässig and Dirk Sudholt. Adaptive population models for offspring populations and parallel evolutionary algorithms. In *Proceedings of the 11th Workshop on Foundations of Genetic Algorithms (FOGA 2011)*, pages 181–192. ACM Press, 2011.

[41] Jörg Lässig and Dirk Sudholt. Analysis of speedups in parallel evolutionary algorithms for combinatorial optimization. In *22nd International Symposium on Algorithms and Computation (ISAAC 2011)*, pages 405–414. Springer, 2011.

[42] Jörg Lässig and Dirk Sudholt. Design and analysis of migration in parallel evolutionary algorithms. *Soft Computing*, 17(7):1121–1144, 2013.

[43] Jörg Lässig and Dirk Sudholt. General upper bounds on the runtime of parallel evolutionary algorithms. *Evolutionary Computation*, 22(3):405–437, 2014.

[44] Per Kristian Lehre and Carsten Witt. Black-box search by unbiased variation. *Algorithmica*, 64(4):623–642, 2012.

[45] Per Kristian Lehre and Xin Yao. On the impact of mutation-selection balance on the runtime of evolutionary algorithms. *IEEE Transactions Evolutionary Computation*, 16(2):225–241, 2012.

[46] Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Thomas Stützle, and Mauro Birattari. The irace package, iterated race for automatic algorithm configuration. Technical Report TR/IRIDIA/2011-004, IRIDIA, Université Libre de Bruxelles, Belgium, 2011.

[47] Gabriel Luque and Enrique Alba. *Parallel Genetic Algorithms–Theory and Real World Applications*, volume 367 of *Studies in Computational Intelligence*. Springer, 2011.

[48] Andrea Mambrini and Dario Izzo. Pade: A parallel algorithm based on the MOEA/D framework and the island model. In *Parallel Problem Solving from Nature - PPSN XIII*, pages 711–720, 2014.

[49] Andrea Mambrini and Luca Manzoni. A comparison between Geometric Semantic GP and Cartesian GP for Boolean functions learning. In *Genetic and Evolutionary Computation Conference*, GECCO '14, pages 143–144, 2014.

[50] Andrea Mambrini, Luca Manzoni, and Alberto Moraglio. Theory-laden design of mutation-based geometric semantic genetic programming for learning classification trees. In *IEEE Congress on Evolutionary Computation*, pages 416–423. IEEE, 2013.

[51] Andrea Mambrini and Dirk Sudholt. Design and analysis of adaptive migration intervals in parallel evolutionary algorithms. In *Genetic and Evolutionary Computation Conference*, GECCO '14, pages 1047–1054, 2014.

[52] Andrea Mambrini, Dirk Sudholt, and Xin Yao. Homogeneous and heterogeneous island models for the set cover problem. In *12th international conference on Parallel Problem Solving from Nature*, PPSN'12, pages 11–20, 2012.

[53] James McDermott, David R. White, Sean Luke, Luca Manzoni, Mauro Castelli, Leonardo Vanneschi, Wojciech Jaskowski, Krzysztof Krawiec, Robin Harper, Kenneth De Jong, and Una-May O'Reilly. Genetic programming needs better benchmarks. In *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference*, GECCO '12, pages 791–798, New York, NY, USA, 2012. ACM.

[54] Julian F. Miller. An empirical study of the efficiency of learning boolean functions using a cartesian genetic programming approach. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1135–1142, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

[55] Julian F. Miller. *Cartesian genetic programming.* Springer, 2011.

[56] Julian F. Miller and Stephen L. Smith. Redundancy and computational efficiency in cartesian genetic programming. *IEEE Transactions on Evolutionary Computation*, 10(2):167–174, April 2006.

[57] Alberto Moraglio. *Towards a Geometric Unification of Evolutionary Algorithms.* PhD thesis, University of Essex, 2007.

[58] Alberto Moraglio, Krzysztof Krawiec, and Colin Johnson. Geometric semantic genetic programming. In *Workshop on Theory of Randomized Search Heuristics*, 2011.

[59] Alberto Moraglio, Krzysztof Krawiec, and Colin Johnson. Geometric semantic genetic programming. In *PPSN '12*, volume 7491 of *Lecture Notes in Computer Science*, pages 21–31. Springer, 2012.

[60] Alberto Moraglio and Andrea Mambrini. Runtime analysis of mutation-based geometric semantic genetic programming for basis functions regressions. In *Genetic and Evolutionary Computation Conference, GECCO'13*, pages 989–996. ACM, 2013.

[61] Alberto Moraglio, Andrea Mambrini, and Luca Manzoni. Runtime analysis of mutation-based geometric semantic genetic programming on boolean functions. In *Foundations of Genetic Algorithms XII, FOGA'13*, pages 119–132. ACM, 2013.

[62] Alberto Moraglio and Riccardo Poli. Topological interpretation of crossover. In *Genetic and Evolutionary Computation–GECCO 2004*, pages 1377–1388. Springer, 2004.

[63] Alberto Moraglio and Dirk Sudholt. Runtime analysis of convex evolutionary search. In *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference*, GECCO '12, pages 649–656, New York, NY, USA, 2012. ACM.

[64] Heinz Mühlenbein. How genetic algorithms really work: Mutation and hillclimbing. In Reinhard Männer and Bernard Manderick, editors, *PPSN*, pages 15–26. Elsevier, 1992.

[65] Nadia Nedjah, Luiza de Macedo Mourelle, and Enrique Alba. *Parallel Evolutionary Computations.* Springer, 2006.

[66] Frank Neumann, Pietro S. Oliveto, Gunter Rüdolph, and Dirk Sudholt. On the effectiveness of crossover for migration in parallel evolutionary algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2011)*, pages 1587–1594. ACM Press, 2011.

[67] Frank Neumann, Una-May O'Reilly, and Markus Wagner. *Computational Complexity Analysis of Genetic Programming - Initial Results and Future Directions.* Genetic and Evolutionary Computation. Springer, Ann Arbor, USA, 12-14 May 2011.

[68] Frank Neumann and Carsten Witt. *Bioinspired Computation in Combinatorial Optimization – Algorithms and Their Computational Complexity.* Springer, 2010.

[69] Pietro S. Oliveto, Jun He, and Xin Yao. Time complexity of evolutionary algorithms for combinatorial optimization: A decade of results. *International Journal of Automation and Computing*, 4(3):281–293, 2007.

[70] Karel Osorio, Enrique Alba, and Gabriel Luque. Using theory to self-tune migration periods in distributed genetic algorithms. In *IEEE Congress on Evolutionary Computation*, pages 2595–2601, 2013.

[71] Karel Osorio, Gabriel Luque, and Enrique Alba. Distributed evolutionary algorithms with adaptive migration period. In *Proceedings of ISDA*, pages 259–264, 2011.

[72] Tomasz P Pawlak. Combining semantically-effective and geometric crossover operators for genetic programming. In *Parallel Problem Solving from Nature–PPSN XIII*, pages 454–464. Springer, 2014.

[73] Riccardo Poli. Exact schema theory for genetic programming and variable-length genetic algorithms with one-point crossover, June 2001.

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

[75] Riccardo Poli and Nicholas Freitag McPhee. General schema theory for genetic programming with subtree-swapping crossover: part i. *Evolutionary Computation*, 11(1):53–66, March 2003.

[76] Riccardo Poli and Nicholas Freitag McPhee. General schema theory for genetic programming with subtree-swapping crossover: Part ii. *Evolutionary Computation*, 11(2):169–206, May 2003.

[77] Riccardo Poli and Nicholas Freitag McPhee. Parsimony pressure made easy: Solving the problem of bloat in gp. In Yossi Borenstein and Alberto Moraglio, editors, *Theory and Principled Methods for Designing Metaheuristics*, chapter 9. Springer, 2012.

[78] Riccardo Poli, Nicholas Freitag McPhee, and Jonathan E. Rowe. Exact schema theory and markov chain models for genetic programming and variable-length genetic algorithms with homologous crossover. *Genetic Programming and Evolvable Machines*, 5(1):31–70, March 2004.

[79] Günter Rudolph. How mutation and selection solve long path problems in polynomial expected time. *Evolutionary Computation*, 4(2):195–205, 1996.

[80] Günter Rudolph. Finite markov chain results in evolutionary computation: a tour d'horizon. *Fundam. Inf.*, 35(1-4):67–89, 1998.

[81] Günter Rudolph. Takeover time in parallel populations with migration. In B. Filipic and J. Silc, editors, *Proceedings of the Second International Conference on Bioinspired Optimization Methods and their Applications (BIOMA 2006)*, pages 63–72, 2006.

[82] Zbigniew Skolicki. *An Analysis of Island Models in Evolutionary Computation.* PhD thesis, George Mason University, Fairfax, VA, 2000.

[83] Zbigniew Skolicki and Kenneth De Jong. The influence of migration sizes and intervals on island models. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2005)*, pages 1295–1302. ACM, 2005.

[84] Terence Soule and James A. Foster. Removal bias: a new cause of code growth in tree based evolutionary programming. In *In 1998 IEEE International Conference on Evolutionary Computation*, pages 781–186. IEEE Press, 1998.

[85] Dirk Sudholt. Crossover speeds up building-block assembly. In *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference*, GECCO '12, pages 689–702, New York, NY, USA, 2012. ACM.

[86] Walter Alden Tackett. *Recombination, Selection, and the Genetic Construction of Computer Programs.* PhD thesis, University of Southern California, Department of Electrical Engineering Systems, USA, 1994.

[87] Marco Tomassini. *Spatially Structured Evolutionary Algorithms: Artificial Evolution in Space and Time.* Springer, 2005.

[88] Nguyen Quang Uy, Nqugen Xuan Hoai, Michael O'Neill, R.I. McKay, and Edgar Galván-López. Semantically-based crossover in genetic programming: Application to real-valued symbolic regression. *Genetic Programming and Evolvable Machines*, 12(2):91–119, 2011.

[89] James A. Walker and Julian F. Miller. The automatic acquisition, evolution and reuse of modules in cartesian genetic programming. *IEEE Transaction on Evolutionary Computation*, 12(4):397–417, August 2008.

[90] Ingo Wegener. *Methods for the analysis of evolutionary algorithms on pseudo-Boolean functions*, pages 349–369. Kluwer, 2002.

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

[92] Carsten Witt. Tight bounds on the optimization time of a randomized search heuristic on linear functions. *Combinatorics, Probability & Computing*, 22(2):294–318, 2013.

[93] Douglas A. Wolfe and Myles Hollander. *Nonparametric statistical methods*. John Wiley New York, 1973.

[94] David H. Wolpert and William G. Macready. No free lunch theorems for optimization. *IEEE Transaction on Evolutionary Computation*, 1(1):67–82, 1996.

[95] Xin Yao. Evolutionary artificial neural networks. *International Journal of Intelligent Systems*, 4:539–567, 1993.

[96] Xin Yao. An overview of evolutionary computation. *Chinese Journal of Advanced Software Research (Allerton)*, 3:12–29, 1996.