Università degli studi di Pavia
Dipartimento di Matematica
Corso di Laurea Magistrale in Matematica

UNIVERSITÀ
DI PAVIA

Policy Dinamiche per Parametri per il Problema LeadingOnes su Spazi di Stato Estesi
Dynamic Parameter Policies for LeadingOnes in Enhanced State Spaces

Tesi di Laurea Magistrale in Matematica

Relatore (Supervisor):
Prof. Stefano Gualandi
Correlatore (Co-Supervisor):
Prof. Carola Doerr

Tesi di Laurea di:
Gianluca Covini
Matricola 526045

Anno Accademico 2023-2024

Abstract

Black-box optimization is a framework for optimizing problems for which a model or function is not explicitly provided or is too complex to be used by direct analytical means. In this context, the use of genetic algorithms is an established way to solve complex real-world problems of different types. A reliable analysis of their performance can be done through the study of the number of evaluations of solution candidates. In recent years, many studies about performance analysis highlighted the role of parameters in evolutionary algorithm's performance, and proposed several solutions to adapt the parameters in order to enhance speed, including the use of reinforcement learning. The limits of this novel approach highlighted the necessity of a deeper understanding of algorithms behavior in new settings.

In this thesis, we present an overview of the state of the art in black-box optimization and parameter control, existing theoretical results, and our contributions. We developed and tested a methodology to exploit information from enhanced state spaces in dynamic policies for the specific case of RLS, a randomized local search algorithm with dynamic choice of the search radius. We investigated optimal or close-to-optimal control policies for this algorithm for different scenarios optimizing the LeadingOnes problem. This work offers a foundational example that could inspire broader development of new dynamic parameter policies in enhanced state spaces.

Abstract (in italiano)

L'ottimizzazione black-box è un framework per l'ottimizzazione di problemi in cui un modello o una funzione non sono esplicitamente forniti oppure sono troppo complessi per essere utilizzati con metodi analitici diretti. In questo contesto, l'uso degli algoritmi genetici rappresenta un approccio consolidato per risolvere problemi reali complessi di varia natura. Un'analisi affidabile delle loro prestazioni può essere condotta studiando il numero di valutazioni di possibili candidati soluzione. Negli ultimi anni, numerosi studi sull'analisi delle prestazioni hanno evidenziato il ruolo cruciale dei parametri nell'efficacia degli algoritmi evolutivi, proponendo diverse strategie per adattarli al fine di migliorare la velocità, tra cui l'uso del reinforcement learning. Tuttavia, i limiti di questo approccio innovativo hanno messo in luce la necessità di una comprensione più approfondita del comportamento degli algoritmi in nuovi contesti.

In questa tesi, presentiamo una panoramica dello stato dell'arte nell' ottimizzazione black-box e nel controllo dei parametri, discutendo i risultati teorici esistenti e i nostri contributi. Abbiamo sviluppato e testato una metodologia per sfruttare le informazioni provenienti da spazi di stato estesi nelle policy dinamiche, con particolare riferimento al caso dell'RLS, un algoritmo di ricerca locale randomizzato con scelta dinamica del raggio di ricerca. Abbiamo studiato politicy di controllo dei parametri ottimali o quasi ottimali per questo algoritmo in diversi scenari di ottimizzazione del problema LeadingOnes. Questo lavoro fornisce un esempio fondamentale che potrebbe ispirare un più ampio sviluppo di nuove policy dinamiche per parametri in spazi di stato estesi.

Acknowledgements

I would like to thank everyone who supported me in the preparation of this thesis and throughout the completion of my academic journey. First and foremost, my gratitude goes to my supervisors. Carola, whom I thank, first of all, for the trust she placed in me, then for always listening to my needs and interests, and finally for guiding me every step of the way during my thesis. Her example has been fundamental for me in entering the world of research, understanding its challenges, but also getting inspired by the enthusiasm and kindness she shows in her work.

I also thank Professor Gualandi for being a reference point during this final stage of my academic journey and over the past five years, always ready to offer suggestions and accommodate my requests. My interest in optimization and programming stems from his courses, which I remember as some of the most engaging of these five years.

I also thank other people who supported me during the writing of this thesis, in particular Denis, who helped me in the final stages of this work, as well as Martin and Nguyen for the interesting conversations that raised some of the research questions I attempted to address.

I am grateful to the LIP6 team, especially Fraçnois, Georgii, Maria Laura, Yihang, and Dimitri, for welcoming me and making me feel, even during individual work, like part of a team. I would like to thank the friends from the Maison de l'Italie, especially Viola, Lorenzo, Asja, and Aurora, for showing me that one can feel at home even far away. I also thank all the other friends I met or who came to visit me, filling that period with affection.

I am grateful to the professors at the University of Pavia because I now realize how fortunate I have been to follow a path that I would choose again, even in hindsight, and I believe part of the credit for the passion I have developed over these years belongs to them.

Lastly, I want to thank all the other people who stood by my side, especially my parents and my family, all the people I met at the Collegio, my classmates, and my lifelong friends.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Optimization plays a crucial role in modern life. The rise of machine learning and data-driven methodologies in various industries has raised the need for efficient methods to make optimal decisions. The mathematical framework addressing these growing demands is the field of optimization, which is currently experiencing rapid advancements as a result. This discipline lies at the intersection of mathematics and computer science, focusing on the development of efficient algorithms capable of solving complex optimization problems. These algorithms are often tailored to the specific formulation of the problems they address. However, many real-world challenges involve optimization tasks in which the objective lacks a closed-form mathematical expression, and information about the function to optimize is only available through costly evaluations. This scenario is the focus of *black-box optimization*, a field dedicated to designing algorithms that operate iteratively without access to the objective's closed form. Among the most prominent approaches in black-box optimization are *evolutionary algorithms*, which mimic the processes of mutation and selection found in nature to iteratively improve solutions.

The study of these types of algorithms is primarily empirical [37], but theoretical investigations have also yielded significant insights in the field. In particular, several mathematically grounded concepts and tools have been developed to analyze the performance of black-box algorithms on benchmark problems. Among these, *black-box complexity* [27] and *drift analysis* [58] have proven especially fruitful. These tools enable precise studies of benchmark algorithms, such as RLS and (1+1) EA, as well as benchmark problems, including LeadingOnes and OneMax. Research in this area has highlighted the strong dependence of algorithm performance on parameter choices. For instance, it has been shown that the control of the radius parameter in RLS solving LeadingOnes can significantly affect the algorithm's exact performance, particularly in terms of the asymptotic runtime constant [8].

Recently, a new framework for parameter control, known as *dynamic algorithm configuration (DAC)*, has been introduced. This framework aims to dynamically adjust algorithm parameters during runtime by leveraging information about the algorithm's current state. The typical DAC approach utilizes a reinforcement learning (RL) agent, where the agent's actions correspond to

selecting the algorithm's current configuration.

In [7], a DAC agent was applied to control the radius parameter of RLS on LeadingOnes. The study demonstrated promising results for this approach but also highlighted several challenges that required further attention. In particular, the reinforcement learning (RL) agent exhibited difficulties in generalizing its policy as the problem dimension increased. The paper suggested further exploration of the problem, including a potential direction inspired by promising preliminary results in [11]: incorporating additional information into the state space of the parameter policy to enhance algorithm performance. A deeper understanding of how different state space representations affect the RL agent's behavior could be a crucial step toward addressing these challenges.

In this thesis, we tackle this problem by developing new parameter policies for RLS on LeadingOnes and comparing them against state-of-the-art policies. Through experiments conducted in various settings, we analyze the successes and limitations of these approaches, shedding light on unresolved challenges and posing new questions for future research in the field. We studied the effect of including further state information in the parameter policy rather than only the fitness value in both mutation and selection mechanisms of the algorithm. By providing exact computation of the expected runtime for low dimensions and Monte Carlo approximations in high dimensions, we show that the algorithm performance significantly improves when new information is exploited both for mutation and for selection; when it is exploited only for mutation, the improvement seems not to be statistically significant. We highlighted the differences of the two approaches and raised questions for further development of these methods, in particular in terms of generalization in high dimensions.

The thesis is organized as follows. In Chapter 2, we present a comprehensive introduction to black-box optimization. We first outline the general framework and cite some applications, and then describe the main algorithms we will focus on. In the last sections of the chapter, we give a general theoretical overview of the topics of black-box optimization, drift analysis and parameter control. In Chapter 3, we present the main results of black-box complexity and parameter control, focusing on the algorithms RLS and $(1 + 1)$ EA on the problems LeadingOnes and OneMax. These results constitute the theoretical basis for our contributions. In Chapter 4, we present the problem we addressed and the way in which we developed new parameter policies for RLS on LeadingOnes considering enhanced state spaces. In Chapter 5, we provide the empirical results of the policies we developed in various settings and the comparison with the state-of-the-art. In the appendix chapters, we present more deeply some topics we mention in the thesis and provide selected proofs and more complete results.

Chapter 2

Black-box Optimization

In this chapter, we will introduce key concepts in black-box optimization and evolutionary computation, which form the foundation of this work. We begin with a general overview, followed by a more detailed exploration of topics related to performance analysis and parameter control.

2.1 Notation

In this thesis, we adopt the following notation. Let $\mathbb{E}[X]$ denote the expected value of the random variable X, and $\mathbb{P}(A)$ the probability of event A. We use $\Theta(g(n))$, $\Omega(g(n))$, and $\mathcal{O}(g(n))$ to describe the asymptotic complexity of functions for $n \to \infty$: $\Theta(g(n))$ represents a function bounded both above and below by $g(n)$ up to constant factors, $\Omega(g(n))$ represents a lower bound, and $\mathcal{O}(g(n))$ an upper bound. For binary sequences, 1^n0^m refers to a string composed of n ones followed by m zeros. Additionally, $d_H(x, y)$ denotes the Hamming distance between bit strings x and y, defined as the number of positions in which the corresponding bits differ. We use $\lfloor x \rfloor$ and $\lceil x \rceil$ for the floor and ceiling functions, respectively, and $\log(x)$ for the logarithm base e. The notation $\binom{n}{k}$ represents the binomial coefficient; we will also use the inline notation (n, k) for the binomial coefficient, where its meaning will be clear from the context. $|S|$ denotes the cardinality of a set S. Throughout, argmin and argmax indicate the values that minimize or maximize a function, respectively. For two integer numbers a and b ($b \geq a$) by $[a..b]$ we denote an integer interval, that is, all integer numbers that are at least a and at most b; if $b > 0$; by $[b]$ we indicate $[0..b]$. Additional notation will be introduced and explained when used.

2.2 Black-Box Optimization

2.2.1 Black-Box Optimization Setting

Optimization is a fundamental area in Mathematics and Computer Science focused on finding the best within a set of possible options: mathematically, it consists of finding the value of $x \in S$ that corresponds to the optimum (maximum or minimum) of a certain function f, called *objective function*, within

a set of possible solutions S, called *search space*. In general, the complexity of the problem depends on the complexity of f, e.g. how much it is rugged, noisy, whether it is differentiable, linear, or other specific properties of the function or of the feasible set of solutions. However, in many real-world applications, the function f is in fact unknown or too complex to be used in its explicit form. In cases like this, we assume that we can only rely on the evaluation of the function in specific points of the domain. This is the setting of *black-box optimization*, which will be the focus of this thesis.

The black-box optimization problem can be formulated as follows.

$$\text{find } x^* \in \operatorname*{argmin}_{x \in S} f(x) \tag{2.1}$$

where the function f is a *black-box function*, which means that the function works as an oracle, taking an input x and giving as output the value of $f(x)$. This implies that we cannot exploit analytical properties of the objective function in the optimization process: in particular, we cannot compute the derivatives of f and thus use gradient-based methods such as gradient descent or Newton's method to solve (2.1). For this reason, black-box optimization is a type of *derivative-free optimization*. Thus, the optimization process is only led by the query of evaluations of the function at points in the search space, as represented in Figure 2.1.

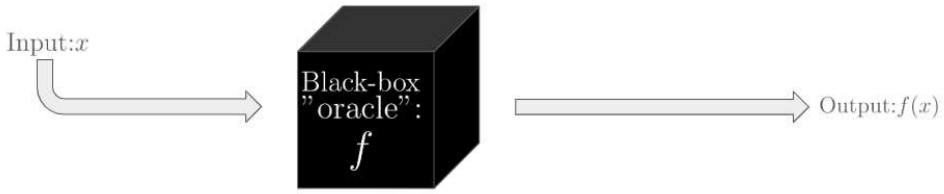

Figure 2.1: Basic mechanism of a black-box function

In the optimization problem (2.1), S is usually a subset of $\mathbb{R}^n$, where n is the dimension of the problem, and the function f is defined $S \longrightarrow \mathbb{R}$. This setting is quite general and allows us to address both discrete and continuous problems, upon an appropriate choice of S. In this thesis, we will focus primarily on discrete optimization where the search space S consists of finite elements. This choice is motivated by the broader theoretical landscape and challenges in discrete optimization. In discrete optimization, an optimal target state can often be identified before the optimization process begins. Additionally, in continuous optimization, gradient-based methods can sometimes be approximated even in a black-box setting. Theoretical insights into discrete black-box optimization have profound implications for algorithm design, particularly in problems involving combinatorial structures or complex constraints, and can also inspire the application of techniques in a continuous fashion.

2.2.2 Applications

Black-box optimization is one of the most general optimization frameworks, since it does not require any assumption on the objective function. In fact, it is particularly suitable for addressing many real-world problems where building a model is not feasible or too complex. In many of these cases, it is possible to gain information about the quantity we would like to optimize through evaluations of the objective function, which can consist in simulations or empirical measurements. However, black-box algorithms are generally less efficient than those that leverage the known structure of the objective function. Therefore, it is important to apply black-box methods only when the analytical form of the function cannot be effectively exploited.

Black-box optimization is also the framework that implicitly governs our everyday decision-making, where choices are based on past experiences rather than complex models of behavior. We can imagine a hypothetical *well-being function* as an objective black-box function that we would like to maximize and past experiences as evaluations of that function.

A classical, illustrative, yet playful example is the *Coffee Tasting Problem* [46], which consists of finding the optimal mixture of different types of coffee, to obtain a target taste. A coffee company generally would like to keep the taste of their mixture consistent from year to year, even if the quality of the coffee beans changes after each harvest. In this case, the objective function f represents the opinion of an expert who evaluates the taste of a mixture. Obviously, f is a black-box function: it cannot be determined analytically, but we can evaluate it with different tastings.

In general, black-box optimization algorithms are applied whenever:

- the analytical form of the function is unknown and we can evaluate the quality of a solution only through simulations (also referred, in this case, as *simulation-based optimization*);
- the analytical form of the function is too complex to be used in algorithms that involve gradients or integrals;
- there is no method known to be better for the particular problem.

Some examples involve:

- *Monte Carlo integration* [85]: this is one of the simplest examples of black-box problems. We can interpret it as a black-box, since it relies only on the evaluation of the target function f at specific points and not on the analytical form of it. In this case, the function we would like to study is too complex to be derived analytically, so we rely on evaluations at specific points to overcome the problem.
- *Machine learning*: many machine learning problems can be addressed through a black-box optimization approach. An example is clustering, one of the main problems in unsupervised learning. Having fixed the number K of clusters, we can define as objective function a clustering

metric $\mathcal{M}$ that measures the distance of the points of a cluster from its center and uses a black-box algorithm to optimize it [61].

- *Deep learning*: black-box optimization can play an important role in hyperparameter tuning of artificial neural networks or other models. In these cases, the relation between the choice of hyperparameters and the performance of the model is not defined by a specific function, but the quality of a particular configuration can be evaluated by the performance of the network over a specific benchmark.
- *Robotics*: some virtual robots of different shapes have been taught to walk upright through a simulation and optimization process that follows the black-box optimization framework: in this case, each evaluation of the function corresponds to a simulation [36].
- *Industry applications*: black-box optimization is involved in several industrial applications; whenever we want to optimize a quality too complex to be directly modeled, we can use simulations to evaluate the objective function; this includes a variety of problems in the fields of structural mechanics [75], wind turbines planning [4] and scheduling problems [70, 67].

2.3 Algorithms

2.3.1 Black-box Algorithms

Due to the nature of the framework, the modeling part plays a minor role in black-box optimization, therefore algorithms gain the central role. Black-box algorithms face the challenge of addressing a function that can be known only through evaluations at some points in the search space. They proceed iteratively, sampling the search space while looking for a better solution until the optimum is reached or some termination criteria are satisfied. The general structure is summarized in Algorithm 1 and in Figure 2.2.

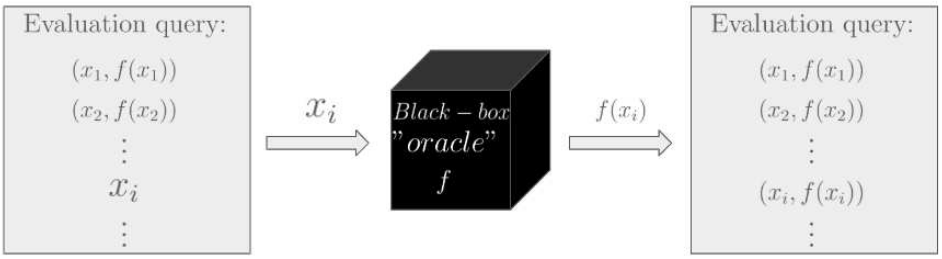

Figure 2.2: Basic mechanism of a black-box algorithm

It starts with a solution candidate x, usually drawn at random from the search space S, and evaluates the objective function f on it. Then, it iteratively generates new solution candidates based on some strategy and evaluates them on the objective function f, while keeping track of the best solution so far. When the termination criteria are met, the best solution is displayed as output.

Algorithm 1 General structure of a black-box algorithm

Input: Search space S, objective function f
Initialize a random solution $x \in S$
Set $best \leftarrow x$
Set $best\_value \leftarrow f(x)$
while termination criteria not met **do**
    Generate a new candidate solution $x' \in S$ (based on some strategy)
    Evaluate $f(x')$
    if $f(x') \leq best\_value$ **then**
        Set $best \leftarrow x'$
        Set $best\_value \leftarrow f(x')$
    end if
end while
Output: Best solution *best* and its value *best_value*

The algorithm structure is quite simple and general. The way new solution candidates are generated is what characterizes different black-box algorithms; even without making assumptions about the function f, strategies could be effectively adapted to address specific problems and several categorizations exist.

Optimization algorithms can be first divided into two classes: *heuristics* and *non-heuristics*. We refer as heuristics to algorithms that do not include a guarantee of success in finding the optimum: in general, these algorithms are easier to understand and implement, and sometimes this can lead to a more detailed analysis. Non-heuristic algorithms guarantee, by definition, convergence to the global optimum. In addition, the term *metaheuristics* is often used: according to [79], a metaheuristic is defined as *"a high-level, problem-independent algorithmic framework that provides a set of guidelines or strategies to develop heuristic optimization algorithms"*. Metaheuristics can also refer to problem-specific implementations of these general strategies. Many black-box optimization algorithms fall under the category of metaheuristics, as they provide flexible general-purpose frameworks that can be adapted to a wide range of situations by developing problem-specific heuristics within the overarching framework.

A further class division of optimization algorithms is between *deterministic* and *randomized*: the difference consists of whether the new solution candidate is generated through a deterministic process or involving randomness. A very simple deterministic algorithm is *exhaustive search* [3]: it can be applied if the search space S is discrete and finite. It consists of evaluating the objective function on each element in order to find the optimum. Usually, random algorithms, even though they cannot guarantee consistency from one run to another, are able to broadly explore the search space with a limited cost and help avoid

local minima. For this reason, they have become the standard in black-box optimization. We can cite the most common ones:

- *Random sampling*: random sampling is the randomized version of exhaustive search. In its simplest version, it consists of sampling uniformly at random elements from the search space S and evaluating them to find the optimum [82]. Finest versions can be used, leveraging a quasi-random sampling, in order to exploit some specific information about the distribution of elements in the search space [69, 83].

- *Surrogate-based optimization algorithms*: in the surrogate-based algorithms we overcome the complexity given by a black-box function f by replacing the "real" black-box function with a *surrogate* $\hat{f}$, i.e. an approximation, based on some points previously sampled [33]. In these cases, $\hat{f}$ can be chosen in many ways, such as linear regression, polynomial interpolation, or an artificial neural network.

- *Local Search*: local search is the basic of many heuristic methods. The idea is that the algorithm starts with a solution candidate and then looks for a better point in the neighborhood of the candidate solution. If found, the new point becomes the center of the new search neighborhood [37]. Variations of the algorithm have been developed to take advantage of different definitions of the neighborhood. Local search can find solutions quite rapidly; however, it has the drawback of being likely to be stuck in a local optimum. In order to prevent this, various techniques have been introduced; stochastic variations of local search allow us to accept a worse solution with a certain probability in order to let the algorithm explore more widely the search space and avoid local optima. For example, in the Metropolis algorithm, a worse solution candidate is accepted with a probability that decreases exponentially as the distance from the optimum value increases [26, 62]. A generalized version, Simulated Annealing, reduces this probability also over time, under the control of a parameter called temperature [53].

- *Evolutionary Algorithms*: evolutionary algorithms are among the most widely used and studied black-box algorithms. They are inspired by the biological theory of natural selection and are quite similar to local search but do not limit the exploration to a neighborhood of the best solution candidate. Incorporating the ability to exploit information from different sampled points, they build a nonuniform probability distribution on the search space that is then used to generate new solution candidates. Since they will be the main topic of our analysis we will describe them more in detail in the following section.

2.3.2 Evolutionary Algorithms

As we said in the previous section, evolutionary algorithms are among the most popular black-box algorithms, and their study constitutes the field of evolutionary computation. They were developed in the second half of the 20^{th}

century [76] with the aim of developing automated problem solvers taking inspiration from the *natural problem solver* of species evolution. The fundamental metaphor of evolutionary computing is indeed to apply natural evolution to a particular style of problem solving: that of trial-and-error. The development of evolutionary algorithms answered the challenge of providing some robust algorithms, working in an acceptable time, for a wide range of problems. From their introduction, they have been widely applied in various areas, including university class timetabling [9, 68], industrial design optimization [52], and machine learning [61, 34].

Evolutionary algorithms are a metaheuristic which provides a general framework to design heuristics. The common underlying structure is connected to the biological inspiration at the base of evolutionary algorithms. The key idea comes from natural evolution: according to evolutionary theory, individuals interact in an environment with a scarcity of resources; only those with a higher *fitness* (identified as the *quality of survive*) can reproduce and pass to their offspring their characteristics that determine a high fitness. Out of metaphor, we consider a function f we would like to maximize (note that maximization and minimization are equivalent problems). We start by initializing some solution candidates (individuals) and evaluating them using our fitness function f. If the population has more than one individual, we can select them based on their fitness. We apply a *mutation* operator which generates new solutions candidates (children) from the previous (parents) and, eventually, apply a *crossover* operator which mixes characteristics of the new individuals in order to augment variety. In the end, a *selection* operator decides which individuals are the best and eliminates the others before a new cycle starts. A schematic description is provided in Algorithm 2.

Algorithm 2 General structure of an evolutionary algorithm

1: **Input:** Population size N, selection size k, fitness function f
2: **Initialize:** Generate an initial population P of N individuals at random
3: **while** termination condition not met **do**
4: **Selection:** Select k individuals from P based on their fitness using a selection method
5: **Crossover:** Generate offspring by recombining pairs of selected individuals
6: **Mutation:** Mutate some characteristics of the offspring
7: **Evaluation:** Evaluate the fitness of the offspring using the fitness function f
8: **Replacement:** Select individuals to form the new population from both parents and offspring
9: **end while**
10: **Output:** Best individual(s) found

Evolutionary algorithms are iterative by nature, meaning they progressively refine solutions over multiple generations. The choice of stopping criteria is crucial because it determines when the algorithm halts. Common stopping criteria include:

- The reach of the optimum, if feasible and if the problem has a known optimal fitness level;
- The reach of an optimal solution within a precision $\varepsilon > 0$;
- A fixed number of iterations;
- The drop of population diversity under a given threshold.

The path to lower fitness is guided by mutation and crossover, called *variation operators*, which ensure differentiation in the population and thus exploration of the search space. The other leading mechanism is *selection* which acts to increase the mean quality of solutions in the population. These operators work randomly; thus evolutionary algorithms can be classified as randomized heuristics, enabling a broader exploration of the search space and making them effective at avoiding local optima. Within the class, they are then distinguished on the basis of:

- representation of individuals;
- variation operators;
- selection mechanisms.

With *representation of individuals* we intend how potential candidate solutions are mathematically represented with reference to the problem we want to solve. This choice is anything but trivial: it is often one of the most difficult parts in designing an evolution algorithm and influences how variation and selection operators are built. The individual representation indeed defines the different classes of evolutionary algorithms. In particular, individuals are represented as:

- strings over a finite alphabet in *genetic algorithms*;
- real-valued vectors in *evolution strategies*;
- finite state machines in *classical evolutionary programming*;
- trees in *genetic programming*.

2.3.3 Genetic Algorithms

Genetic algorithms have a particular importance: they are among the most widely known evolutionary algorithms, and their simple structure also serves well our aim to provide strong theoretical foundation for our analysis. Their first use in optimization can be traced back to the seminal works of Goldberg [38] and De Jong [14] where the *Simple Genetic Algorithm* (SGA) was defined. SGA uses binary representation, i.e. individuals are represented as bit-strings, selection proportional to the fitness, low probability of mutation and a emphasis on genetically inspired recombination.

Genetic algorithms then developed widely from that first example; however, binary representation still characterizes a large part of the class. Bit-strings are used in many practical cases where the objective function is pseudo-boolean. For example, in binary decision problems, such as the knapsack problem. Moreover, genetic algorithms, even in their simplest forms, such as SGA and its variants, are still extensively studied to provide important theoretical insight and practical inspiration into the behavior of evolutionary processes in combinatorial search spaces, as well as for benchmarking new algorithms.

While dealing with binary representation, we can build different algorithms based on different choices of variation operators. In particular, mutation works by changing the value from 0 to 1 and from 1 to 0 of every single bit with a certain probability p: the choices to change bit values can be *independent* from one another, i.e. each bit value is changed by flipping a coin, or can be *dependent*, namely if we fix the number of bits to be flipped. In general, the mutation operators for the bit strings are *unbiased*, which means that they do not take into account the index or the value of the single bit when deciding to flip it.

In our analysis, we are interested in the study of an efficient choice of the mutation operator, so we will not consider recombination. However, we will give a brief description of this technique for the sake of completeness. For binary representation, we have three standard forms of recombination:

- *One-Point Crossover* [47]: it consists of creating two children by splitting the genomes of two parents at a randomly chosen position;
- *n-Point Crossover*: it generalizes One-Point Crossover by dividing the parents into multiple segments and taking alternating segments from each parent to create a child;
- *Uniform Crossover* [81]: it consists of randomly choosing, for each bit, which parent it should be inherited from when creating a new child.

We will close this section by presenting some of the most important evolutionary algorithms that we will discuss later. A rather simple but theoretically interesting algorithm is *(1+1) Randomized Local Search* ((1+1) RLS), or simply Randomized Local Search (RLS). As the name suggests, it can also be seen as a local search algorithm; however, the iterative structure and bit-string individual representation allow us to consider it in the category of genetic algorithms. It operates on a population of a single individual and does not involve crossover operators. The neighborhood is defined as a ball in *Hamming distance* [41]. The Hamming distance between two strings is an important concept in information theory and is defined as the number of string positions at which the corresponding symbols are different, or the number of substitutions necessary to obtain one string from another. In case of bit strings, it can be expressed by the formula (2.2).

$$d_H(x,y) = \sum_{i=1}^{n} \mid x_i - y_i \mid, \quad \forall x, y \in \{0,1\}^n \tag{2.2}$$

For example, two strings with one different bit value have distance 1. The radius of the neighborhood is defined by a parameter k, which is simply called *radius* and controls how many bits the algorithm flips at each step. The choice of which bit to flip, fixed k, is then random. Our work focuses mainly on the control of the radius parameter k.

The complete pseudocode of RLS algorithm is summarized in Algorithm 3.

Algorithm 3 (1+1) Randomized Local Search (RLS)

Input: Fitness function f, bit-string length n
Initialize: Generate a random solution $x \in \{0,1\}^n$
while termination criteria are not met **do**
    Choose the radius k
    Create $x' \leftarrow x$ by flipping k randomly chosen bits in x
    if $f(x') \leq f(x)$ **then**
        $x \leftarrow x'$
    end if
end while
Output: Best solution x found

We will sometimes also refer to $(\mu+\lambda)$ Evolutionary Algorithm, $(\mu+\lambda)$ EA, or (μ, λ) Evolutionary Algorithm, (μ, λ) EA. Both are genetic algorithms (but can also be adapted as evolution strategies) where μ and λ are two integers that denote, respectively, the population size and the offspring size, selected based on some chosen operator. In $(\mu + \lambda)$ EA both parents and offspring are combined to create a pool of $\mu+\lambda$ individuals, while in (μ, λ) EA parents do not compete to form the next generation; thus, the best μ individuals are selected from the λ offspring. Mutation is carried out differently from RLS, flipping each bit independently with probability p. If $\mu = \lambda = 1$, the algorithms do not have crossover, selection, or replacement. Algorithm 4 and Algorithm 5 summarize $(\mu + \lambda)$ EA and (μ, λ) EA respectively.

We note that RLS and $(1 + 1)$ EA are very similar, with the only difference being that the bit flips in RLS are not independent since the total number k is fixed. We can write $(1 + 1)$ EA as RLS, by adding a step in which k is drawn independently as $k \sim \text{Bin}(n, p)$.

Evolutionary algorithms have also achieved great results in the treatment of continuous problems. We will not discuss the topic in this thesis, but we presented in Appendix A some of the most used and known evolutionary algorithms for continuous optimization.

2.4 Black-box Complexity

2.4.1 Motivations

One of the main research paths in the study of black-box algorithms and, in particular, of evolutionary algorithms is *complexity analysis*. Intuitively, we intend, as complexity *how fast* a class of algorithms can solve specific problem instances; later in this section, we will give a formal definition. Of course, this

Algorithm 4 $(\mu + \lambda)$ Evolutionary Algorithm

Input: Population size μ, offspring size λ, fitness function f, bit-string length n
Initialize: Generate a population $P = \{x_1, x_2, \ldots, x_\mu\}$ of μ random solutions, each $x_i \in \{0,1\}^n$
while termination criteria are not met **do**
    Generate Offspring:
        Create an offspring population $P' = \{x'_1, x'_2, \ldots, x'_\lambda\}$ by mutating λ individuals randomly chosen from P
    for each $x'_i \in P'$ **do**
        Mutate x'_i by flipping each bit independently with mutation probability p
    end for
    Combine populations: Create a combined population $P \cup P'$
    Selection: Select the best μ individuals from $P \cup P'$ based on fitness f
end while
Output: Best individual(s) found

Algorithm 5 (μ, λ) Evolutionary Algorithm

Input: Population size μ, offspring size λ, fitness function f, maximum iterations T, bit-string length n
Initialize: Generate a population $P = \{x_1, x_2, \ldots, x_\mu\}$ of μ random solutions, each $x_i \in \{0,1\}^n$
while termination criteria are not met **do**
    Generate Offspring:
        Create an offspring population $P' = \{x'_1, x'_2, \ldots, x'_\lambda\}$ by mutating λ individuals randomly chosen from P
    for each $x'_i \in P'$ **do**
        Mutate x'_i by flipping each bit independently with mutation probability p
    end for
    Selection: Select the best μ individuals from P' based on fitness f
    Replace: Set $P \leftarrow \{x'_1, x'_2, \ldots, x'_\mu\}$ (the best μ offspring)
end while
Output: Best individual(s) found

type of study is crucial to understanding when an algorithm works well on a certain problem and to helping to choose the suitable heuristic for the instance we would like to solve. The general nature of evolutionary heuristics and the No Free Lunch Theorem [87], which proved that no black-box algorithm can outperform random walk when averaged over *all* problems, make it impossible to determine an algorithm that outperforms all the others on any problem instance; therefore, the algorithm choice should be tailored to the problem. Furthermore, the study of algorithm complexity has been strictly connected to design of heuristics, in the way that performance insights lead the building of efficient algorithms to satisfy the theoretical benchmarks on specific problems. In particular, this occurs with two different mechanisms. On one hand, complexity studies could give us an idea of how well we have understood a black-box problem — if there is a large gap between complexity and state-of-the-art performance, this could mean that further research to improve existing algorithm can be done; on the other hand, complexity measures are a good way to benchmark and validate algorithmic choices: comparing complexity restricted to a class of problems, with unrestricted performances can give precious insights on which features should be included in efficient algorithms.

A comprehensive review of black-box complexity can be found in [27], which we mainly followed for this section. A tutorial is also available at [16].

In white-box complexity, we assume that the algorithm has a full access to the problem data: we know what operations the algorithm has to do, so performance analysis mainly consists of counting the number of steps until the algorithm outputs a solution. In the black-box setting, on the other hand, algorithms can learn about the problem only through evaluations of potential solution candidates, and thus a rigorous development of the topic was slower: a new field called *black-box complexity* has been systematically studied only after 2010. Despite the young age, this kind of studies already gave many insights on the development of new efficient algorithms and now provide a wide theory to build on when designing new heuristics.

In our study we will focus on discrete optimization problems and, in particular, pseudo-boolean functions, i.e. functions $f : \{0,1\}^n \longrightarrow \mathbb{R}$. The analysis will also regard random algorithms since they are a more complex and therefore less understood class of algorithms. It is important to note also that deterministic algorithms are a simple case of randomized algorithms; thus any lower bound found on randomized black-box complexity applies also to any deterministic algorithm.

2.4.2 Unrestricted Black-box Complexity

Black-box complexity is a feature of a problem (or a class of problems) and is referred to an algorithm or a class of algorithms trying to solve that problem. The most general black-box complexity model is the *unrestricted black-box* model [30] since it does not make any assumption about the class of algorithms it addresses. The only assumption it makes is that the algorithms do not have any information about the problem other than the fact that the objective function f comes from some function class $\mathbb{F} \subseteq \mathbb{R}^S$. $\mathbb{F}$ represents the objectives

of the class of problems we would like to study. The unrestricted black-box algorithms proceed by evaluating, at each iteration, a solution candidate x, generated according to some probability distribution D, chosen from the query of previous evaluations. A blueprint of unrestricted black-box algorithms can be found in Algorithm 6. It is important to note that the choice of the distribution D can be critical: the algorithm could take a significant amount of time to determine this distribution. Sometimes it is referred to as *time-restricted model*, if we consider algorithms where the distribution $D^{(t)}$ can be decided in a polynomial number of algebraic steps.

Algorithm 6 General structure of unrestricted randomized black-box algorithms

1: **Initialization:**
2: Sample $x(0)$ according to some probability distribution $D^{(0)}$ over S and query $f(x(0))$.
3: **Optimization:**
4: **for** $t = 1, 2, 3, \ldots$ **do**
5: Depending on the previously evaluated points

$$(x(0), f(x(0))), \ldots, (x(t-1), f(x(t-1))),$$

choose a probability distribution $D(t)$ over S and sample $x(t)$ according to $D^{(t)}$.
6: Query $f(x(t))$.
7: **end for**

Note that in the algorithm we are considering just one solution candidate is sampled at each iteration, but the definition of unrestricted black-box complexity naturally extends to population-based heuristics: that case corresponds to sampling and querying more points in parallel, ignoring the information from previous iterations.

We also note that Algorithm 6 does not have termination criteria: since it is necessary for our definition of black-box complexity, we assume in this study that the algorithms in analysis can effectively reach the optimum in finite time; therefore, the first evaluation of an optimal solution is taken as stopping criteria.

To define the unrestricted black-box complexity, we consider an algorithm A and a function $f : S \to \mathbb{R}$. We indicate as $T(A, f) \in \mathbb{R} \cup \{\inf\}$ the number of function evaluations the algorithm A does until it evaluates an optimal point for f, and we call it the *running time of A for f*. If we consider a random algorithm A, we notice that $T(A, f)$ is a random variable, depending on the choices made by the algorithm A. Therefore, we can build a performance measure considering its expected value $\mathbb{E}[T(A, f)]$. The formal definition is the following.

Definition 1. Given a black-box algorithm A and a class of functions $\mathbb{F} = \{f : S \to \mathbb{R}\}$, we define the *$A$-black-box complexity of $\mathbb{F}$* as

$$\mathbb{E}[T(A, \mathbb{F})] := \sup_{f \in \mathbb{F}} \mathbb{E}[T(A, f)]$$

Given a class of algorithms $\mathcal{A}$, we define the *$\mathcal{A}$-black-box complexity of* $\mathbb{F}$ as

$$\mathbb{E}[T(\mathcal{A}, \mathbb{F})] := \inf_{A \in \mathcal{A}} \mathbb{E}[T(A, \mathbb{F})]$$

Intuitively, the A-black-box complexity of $\mathbb{F}$ is the worst-case expected running time of A among the functions of $\mathbb{F}$, while the $\mathcal{A}$-black-box complexity of $\mathbb{F}$ is the best complexity among the algorithms in $\mathcal{A}$, with reference to the problem instances in $\mathbb{F}$. This definition makes common sense since we assume that the only thing we know a priori is that f is in a class $\mathbb{F}$, so we benchmark on the worst case in the class. However, since we are free to choose the algorithm, we can choose the best algorithm A among the class $\mathcal{A}$. We then talk about *unrestricted black-box complexity* if the class of algorithms $\mathcal{A}$ is unrestricted.

Other metrics in addition to the average optimization time were defined in order to model the quality of a solution after a certain time [71, 50, 23]; however, the average optimization time remains the predominant indicator.

Directly from the definition, we can state that:

Proposition 2. *Given $\mathbb{F} \subset \mathbb{R}^S$. For every collection $\mathcal{A}'$ of black-box optimization algorithms for $\mathbb{F}$, the $\mathcal{A}'$-black-box complexity of $\mathbb{F}$ is at least as large as its unrestricted one.*

The proposition is immediate following from the properties of the inf. Nevertheless, the property is particularly relevant since we will often restrict the complexity analysis to subsets of algorithms.

We also remark that it is not without reason that we give the definition using a class of problem instances, rather than a single objective function. In fact, choosing $\mathbb{F} = \{f\}$, the infimum is realized by the algorithm that queries the optimum immediately; therefore, the unrestricted complexity is always 1. It is analogous if we consider a class of functions $\mathbb{F}$ with all the same optimum. More generally, if $\mathbb{F}$ is a class of functions and $X \subseteq S$ is such that for all $f \in \mathbb{F}$ there exists at least one point $x \in X$, such that $x \in \operatorname{argmax}_S\{f\}$, the unrestricted complexity of $\mathbb{F}$ is $(|X| + 1)/2$. A same argument can lead us to assert that for every finite set $\mathbb{F}$ of functions, the unrestricted black-box complexity is bounded from above by $(|\mathbb{F}| + 1)/2$.

2.4.3 Bounds on Black-box Complexity

Given Proposition 2, we are particularly interested in providing lower bounds for unrestricted black-box complexity. In fact, a lower bound for the unrestricted complexity still holds despite the class of algorithms that we choose. The most widely used tool for this is the *minimax principle* of Yao [88]. The theorem uses the following definition:

Definition 3. A *deterministic black-box algorithm* is a randomized black-box algorithm like the Algorithm 6, where at each time t the probability distribution $D^{(t)}$ is a mass distribution of points.

Intuitively, a deterministic black-box algorithm corresponds to a decision tree.

We can now state the following from [65].

Theorem 4 (Yao's minimax principle)**.** *Let Π be a problem with a finite set $\mathcal{I}$ of input instances (of fixed size) that allows a finite set $\mathcal{A}$ of deterministic algorithms. Let p be a probability distribution over $\mathcal{I}$ and q be a probability distribution over $\mathcal{A}$. Then,*

$$\min_{A \in \mathcal{A}} \mathbb{E}[T(I_p, A)] \leq \max_{I \in \mathcal{I}} \mathbb{E}[T(I, A_q)]$$

where I_p denotes a random input chosen from $\mathcal{I}$ according to p, A_q a random algorithm chosen from $\mathcal{A}$ according to q, and $T(I, A)$ denotes the running time of algorithm A on input I.

The principle is particularly powerful: taking into account the fact that randomized algorithms can be written as convex combinations of deterministic algorithms, it affirms that we can bound the expected running time of any randomized algorithm by the *fastest* deterministic algorithm A, evaluated on a random instance, drawn from a probability distribution p. Since it is usually easier to evaluate the complexity of a deterministic algorithm, the principle becomes particularly useful, as the following corollary suggests; however, the principle does not still hold if we consider a restricted class of algorithms.

Corollary 5 (Simple information-theoretic lower bound [30])**.** *Let S be a finite set. Let $\mathbb{F} \subseteq \mathbb{R}^S$ be such that for every $s \in S$ there exists a function $f_s \in \mathbb{F}$ for which the cardinality of $f_s(S) := \{f_s(x) | x \in S\}$ is bounded by k and for which s is a unique optimum. The unrestricted black-box complexity of $\mathbb{F}$ is at least $\lceil \log_k(|S|) \rceil - 1$.*

The previous corollary states that if we have a class of functions $\mathbb{F}$, such as for every possible point s the search space S contains a problem instance f_s for which s is the unique optimum and the value of the objective is bounded by k, then the black-box complexity of $\mathbb{F}$ is bounded from below by a logarithm of the dimension of the search space S.

An intuitive sketch of the proof of corollary 5 is the following: if we want to optimize over f_s, we need to find s among $|S|$ possibilities. Considering binary encoding, this means that we need approximately $\log_2(|S|)$ bits of information. One query of a search point gives us $\log_2(k)$ bits of information, since it gives us one out of k possible f_s outcomes. Therefore, the number of queries needed to find s is $\log_2(|S|)/\log_2(k) = \log_k(|S|)$. This argument also shows the limit of the corollary, since it suggests that the bound is well provided if, at each query, each of the k values for f_s are equally probable and therefore the information given by the evaluation is exactly $\log_2(k)$. However, normally the information provided is less since new point queries are usually nearer to points already queried. As the optimization process proceeds, the function often produces values that are close to the current best-so-far solution more frequently than values that are very different, reducing the amount of information gained per query. Therefore, the optimization process will usually require much more time than the lower bound provided.

Upper bounds are also important in analysis of black-box complexity: in fact, a small upper bound for a class of problems $\mathbb{F}$ shows that there exists an

efficient algorithm that solves every problem instance $f \in \mathbb{F}$ efficiently and can thus provide valuable insights about possible improvements in the state-of-the-art.

The simplest upper bound is the expected performance of random sampling without repetitions, namely

Theorem 6. *For every finite set S and every class $\mathbb{F} \subseteq \mathbb{R}^S$, the unrestricted black-box complexity of $\mathbb{F}$ is at most $(|S|+1)/2$.*

Rather than being trivial, this upper bound gains value with reference to the No Free Lunch Theorem [87], which states that no optimization algorithm is universally better than random sampling on all possible problems.

2.4.4 Memory-restricted Black-box Complexity

The unrestricted black-box complexity is often too generous with heavy problem-tailored algorithms, giving a biased complexity measure compared to typical black-box. One of the reasons is the fact that we are considering heuristics that can store the whole iteration history, which is often a too strong assumption. To solve this problem, we can restrict the length of queries to a value μ and thus obtain *memory-restricted complexity model*. We refer to $(\mu + \lambda)$ *memory-restricted black-box model* when we consider a model that saves in memory μ values and queries λ points at each step. In Chapter 3, we will delve deeper in two examples of $(1+1)$ *memory-restricted black-box model*, where only one point is queried and one is saved at each round.

The strong limits of the unrestricted black-box model started to emerge in [30]. Many bounds derived in this setting are artificially low: that is also clear from our discussion on the simple information-theoretic lower bound (Theorem 5). After 2006, thus, the study of unrestricted black-box complexity seemed to rapidly exhaust its power. However, in 2010, the situation changed with the introduction of *unbiased black-box complexity* in [56]. The idea was to restrict the class of algorithms taken into account in a natural way that covers the most widely used ones.

The proposed setting considers pseudo-boolean functions, i.e. $\mathbb{F} \subseteq \{f : \{0,1\}^n \longrightarrow \mathbb{R}\}$, and assumes that all solution candidates must be sampled from distributions which are unbiased, which means that they do not discriminate between bit positions or bit entries.

We give the following definition, where the concept of k-arity is used, i.e. the idea that information is gained from k queries.

Definition 7. Let $k \in \mathbb{N}$. A *k-ary unbiased distribution* $(D(\cdot|y^{(1)}, \dots, y^{(k)}))_{y^{(1)},\dots,y^{(k)}}$ is a family of probability distributions over $\{0,1\}^n$ such that for all inputs $y^{(1)}, \dots, y^{(k)} \in \{0,1\}^n$ the following two conditions hold.

(i) $\forall x, z \in \{0,1\}^n : D(x|y^{(1)}, \dots, y^{(k)}) = D(x \oplus z|y^{(1)} \oplus z, \dots, y^{(k)} \oplus z)$,

(ii) $\forall x \in \{0,1\}^n \; \forall \sigma \in S_n : D(x|y^{(1)}, \dots, y^{(k)}) = D(\sigma(x)|\sigma(y^{(1)}), \dots, \sigma(y^{(k)}))$

We refer to the first condition as $\oplus$-*invariance*, which indicates the invariance with respect to the xor operator, and to the second as *permutation invariance*. The two conditions formalize the idea of unbiasedness.

A variation operator creating an offspring by sampling from a k-ary unbiased distribution is called a *k-ary unbiased variation operator*.

It is immediate to note that the 0-ary unbiased distribution is trivial and corresponds to the uniform distribution over $\{0,1\}^n$. The 1-ary operators, instead, are of particular importance for the study of black-box algorithms and are referred to as *unary unbiased operators* or, in the context of evolutionary computation, as *mutation operators*. Examples of unary unbiased operators are the standard bit mutation, used in (μ, λ) EA and $(\mu + \lambda)$ EA, and the random bit flip used by RLS, both described in Section 2.3.2. Also Simulated Annealing and Metropolis algorithm, described in Appendix A, are unary unbiased black-box models.

An important result states that all unary unbiased variation operators are similar in type. It follows from the definition [20, 24].

Definition 8. Let $n \in \mathbb{N}$ and $r \in [0..n] := \{0, 1, 2, \ldots, n\}$. For every $x \in \{0,1\}^n$, let $flip_r$ be the variation operator that creates an offspring y from x by selecting r positions $i_1, \ldots, i_r$ in $[n] := [0..n]$ uniformly at random (without replacement), setting $y_i := 1 - x_i$, for $i \in \{i_1, \ldots, i_r\}$, and copying $y_i := x_i$, for all other bit positions $i \in [n] \setminus \{i_1, \ldots, i_r\}$.

The characterization of unary unbiased variation operators is as follows.

Proposition 9. *For every unary unbiased variation operator $(p(\cdot|x))_{x\in\{0,1\}^n}$ there exists a family of probability distributions $(r_{p,x})_{x\in\{0,1\}^n}$ on $[0..n]$ such that for all $x, y \in \{0,1\}^n$ the probability $p(y|x)$ that $(p(\cdot|x))_{x\in\{0,1\}^n}$ samples y from x equals the probability that the routine first samples a random number r from $(r_{p,x})_{x\in\{0,1\}^n}$ and then obtains y by applying $flip_r$ to x. On the other hand, each of these families of distributions $(r_{p,x})_{x\in\{0,1\}^n}$ in $[0..n]$ induces a unary unbiased variation operator.*

The k-ary unbiased black-box algorithm is a black-box algorithm that generates offspring from an k-ary unbiased distribution. In general, it follows the structure of Algorithm 7.

Algorithm 7 Blueprint of a k-ary unbiased black-box algorithm

1: **Initialization:** Sample $x(0) \in \{0,1\}^n$ uniformly at random and query $f(x(0))$
2: **Optimization:** For $t = 1, 2, 3, \ldots$ do
3: Depending on $(f(x(0)), \ldots, f(x(t-1)))$, choose up to k indices $i_1, \ldots, i_k \in [0..t-1]$ and a k-ary unbiased distribution $D(\cdot \mid y^{(1)}, \ldots, y^{(k)})$ where $y^{(1)}, \ldots, y^{(k)} \in \{0,1\}^n$
4: Sample $x(t)$ according to $D(\cdot \mid x^{(i_1)}, \ldots, x^{(i_k)})$ and query $f(x(t))$

Note also that for each $k \leq l$, a k-ary unbiased algorithm is also l-ary unbiased, since we can choose the same indexes more than once.

The unbiased black-box models solve the problems of the unrestricted model with individual problem instances $\mathbb{F} = \{f\}$ discussed in Section 2.4.2. Since an algorithm that queries the optimum in the first step is no longer considered in this class, the black-box complexity of a single instance is not trivial. This idea is also at the base of the following useful theorem [56].

Theorem 10. *Let $f : \{0,1\}^n \longrightarrow \mathbb{R}$ be a function that has a single global optimum. The unary unbiased black-box complexity of f is $\Omega(n \log n)$.*

The proof of the theorem uses multiplicative drift analysis, which is explained in Section 2.5.4 and is provided in [56].

One of the problems in black-box optimization is to avoid getting stuck in local optima. A strategy developed is *global sampling*: the underlying idea is to sample with positive probability also solution candidates far away from the current population. This also models hill-climber algorithms, such as the already cited $(\mu+\lambda)$ EA and RLS. Since these are the algorithms we will mainly study, it is useful to also present the concept of elitist black-box complexity, which covers algorithms of this type. In general we define as elitist black-box algorithm every algorithm which follows the structure of Algorithm 8.

Algorithm 8 $(\mu + \lambda)$ elitist black-box algorithm

1: **Initialize:** Choose a probability distribution $D^{(0)}$ over S; sample from it $x^{(1)}, \ldots, x^{(\mu)} \in S$ and query $f(x^{(1)}), \ldots, f(x^{(\mu)})$
2: Set $X \leftarrow \{(x^{(1)}, f(x^{(1)})), \ldots, (x^{(\mu)}, f(x^{(\mu)}))\}$
3: **for** $t = 1, 2, 3, \ldots$ **do**
4: Depending only on the multiset X, choose a probability distribution $D^{(t)}$ over S
5: Sample $y^{(1)}, \ldots, y^{(\lambda)} \in S$ and query $f(y^{(1)}), \ldots, f(y^{(\lambda)})$
6: Set $X \leftarrow X \cup \{(y^{(1)}, f(y^{(1)})), \ldots, (y^{(\lambda)}, f(y^{(\lambda)}))\}$
7: **for** $i = 1, \ldots, \lambda$ **do**
8: Select $(x, f(x)) \in X$ and update $X \leftarrow X \setminus \{(x, f(x))\}$
9: **end for**
10: **end for**

Black-box elitist heuristics are particularly useful, since the optimization process can strongly benefit from the exploration of points in the search space with a lower fitness than the current best solution candidates. However, these benefits come along with a higher theoretical complexity, since Yao's minimax principle is no longer applicable. Elitist algorithms, in fact, cannot be written as convex combinations of deterministic algorithms [28] and thus different strategies must be followed.

2.5 Drift Analysis

2.5.1 Basic Definitions

A review of tools for analysis of evolutionary algorithms would not be complete without a section on drift analysis. This concept is somehow complementary

to black-box complexity: it aims to provide a framework to derive information on the expected time of a randomized algorithm through the study of how the algorithm moves toward the optimum. As black-box optimization tools we described in the previous section, drift analysis gives important insights on the performance of black-box algorithms and on the development of new algorithms. The framework was first developed in [39] and progressively gained popularity. A complete introduction is given in [58], which we will follow in this section.

The drift analysis framework is quite general and can be seen as a tool to analyze discrete stochastic processes $(X_t)_{t=0,1,2,\dots}$. While applying it to algorithms analysis, we follow these three steps:

1. We identify a quantity X_t, called *potential* (or also *drift function* or *distance function*). X_t aims to measure the progress the algorithm has made after t steps and constitute the stochastic process to analyze;
2. For any X_t, we understand the nature of the random variable $X_t - X_{t+1}$, the one-step change in the distance;
3. We translate the data from step 2 into information about the runtime T of the algorithm, measured in number of steps until the algorithm has achieved its goal.

Step 1 is often the less trivial, since it requires some insight in the problem to well define a distance function. In an evolutionary algorithm, a rather natural choice is usually the fitness of the best individual in the current population, but one could also find more suitable functions to model the distance from the optimum. Step 2, on the other hand, is generally more technical, while Step 3 is the one where drift analysis comes in.

In the general setup, we consider a non-negative discrete stochastic process $(X_t)_{t=0,1,2,\dots}$ with a finite state space $S \subseteq \mathbb{R}_{\geq 0}$, such that $0 \in S$. The choice of a finite search space is fine with our setting, which involves algorithms with bit-string representation. We will call *stopping time of 0* of the process the smallest time t such that $X_t = 0$. Note that we are considering a process where the target value is 0: in our evolutionary algorithms setting, this is coherent to the choice of X_t as the distance from the optimum rather than the fitness. This also explains why, in Step 2 of the described framework, we study $X_t - X_{t+1}$ rather than $X_{t+1} - X_t$.

We then call *drift*, the quantity $\delta_t(s) := \mathbb{E}[X_t - X_{t+1} | X_t = s]$, for a certain state $s \in S$ search space. The purpose of drift analysis is to provide tools to derive bounds on $\mathbb{E}[T]$, given bounds on drift for every $s \in S$. Note also that we will not consider the case $\mathrm{P}[X_t = s] = 0$, in which the drift is not defined, as it is not of practical interest. There are, however, ways to derive drift theorems which are well defined with reference to the cases of $\mathrm{P}[X_t = s] = 0$ [57].

With reference to black-box complexity, it is important to notice the definition of *optimization time* used in drift analysis. In this setting, time is referred to the index of the random variable considered: therefore, it does not necessarily coincide with the number of function evaluations; however, that could often be the case if the potential is a function of the evaluations made in the iterative algorithmic process.

2.5.2 Additive Drift

The first notion of drift analysis is the additional drift, where the different in expectation of $X_{t+1} - X_t$ is an additive constant. The main result, first presented in [45, 44], is the following.

Theorem 11 (Additive Drift Theorem)**.** *Let $(X_t)_{t=0,1,2,\dots}$ be a sequence of non-negative random variables with a finite state space $S \subseteq \mathbb{R}_{\geq 0}$ such that $0 \in S$. Let $T := \inf\{t \geq 0 | X_t = 0\}$.*

(i) If there exists $\delta > 0$ such that for all $s \in S \setminus \{0\}$ and for all $t \geq 0$,

$$\Delta_t(s) := \mathbb{E}[X_t - X_{t+1} | X_t = s] \geq \delta,$$

then

$$\mathbb{E}[T] \leq \frac{\mathbb{E}[X_0]}{\delta}.$$

(ii) If there exists $\delta > 0$ such that for all $s \in S \setminus \{0\}$ and for all $t \geq 0$,

$$\Delta_t(s) := \mathbb{E}[X_t - X_{t+1} | X_t = s] \leq \delta,$$

then

$$\mathbb{E}[T] \geq \frac{\mathbb{E}[X_0]}{\delta}.$$

The theorem gives an upper bound for the expected time if we can give a constant lower bound for the drift at each step and at each point different from the optimum, while it gives a lower bound if we can provide a constant upper bound for the drift. The proof of the theorem is provided in Appendix B.1.

2.5.3 Variable Drift

A problem of the additive drift theorem is that it requires us to find a potential function that has a constant that bounds the drift: this can eventually be rather hard. A useful theorem that overcomes this problem is *variable drift theorem* [51, 77]:

Theorem 12 (Variable Drift Theorem)**.** *Let $(X_t)_{t \geq 0}$ be a sequence of non-negative random variables with a finite state space $S \subseteq \mathbb{R}_{\geq 0}$ such that $0 \in S$. Let $s_{\min} := \min(S \setminus \{0\})$, let $T := \inf\{t \geq 0 \mid X_t = 0\}$, and for $t \geq 0$ and $s \in S$, let $\Delta_t(s) := \mathbb{E}[X_t - X_{t+1} \mid X_t = s]$. If there exists an increasing function $h : \mathbb{R}_+ \to \mathbb{R}_+$ such that for all $s \in S \setminus \{0\}$ and all $t \geq 0$,*

$$\Delta_t(s) \geq h(s),$$

then

$$\mathbb{E}[T] \leq \frac{s_{\min}}{h(s_{\min})} + \mathbb{E}\left[\int_{X_0}^{s_{\min}} \frac{1}{h(\sigma)}\, d\sigma\right],$$

where the expectation in the latter term is over the random choice of X_0.

The variable drift theorem provides us with a more general framework to determine bounds on the expected time. It requires the drift to be bounded from the low by an increasing function: this is a reasonable assumption, since we can expect that as the algorithm progresses towards $s = 0$ and $h(s)$ becomes smaller, also the drift $\delta_t(s)$ becomes smaller. Given this hypothesis, the expected time is then bounded by a term which depends on h and thus on how fast you progress. The proof of the theorem is provided in Appendix B.1.

2.5.4 Multiplicative Drift

A very important case of variable drift is *multiplicative drift*. It corresponds to the case where the drift is proportional to the potential, i.e. $h(s) = \delta s$ for some δ. It was introduced in [22, 21] and is widely used in the analysis of evolutionary algorithms. It is stated as follows.

Theorem 13 (Multiplicative Drift Theorem)**.** *Let $(X_t)_{t\geq 0}$ be a sequence of non-negative random variables with a finite state space $S \subseteq \mathbb{R}_{\geq 0}$ such that $0 \in S$. Let $s_{\min} := \min(S \setminus \{0\})$, let $T := \inf\{t \geq 0 \mid X_t = 0\}$, and for $t \geq 0$ and $s \in S$, let $\Delta_t(s) := \mathbb{E}[X_t - X_{t+1} \mid X_t = s]$. Suppose that there exists $\delta > 0$ such that for all $s \in S \setminus \{0\}$ and all $t \geq 0$, the drift is*

$$\Delta_t(s) \geq \delta s.$$

Then

$$\mathbb{E}[T] \leq 1 + \frac{\mathbb{E}[\log(X_0/s_{\min})]}{\delta}.$$

As we can see, the derived bound on $\mathbb{E}[T]$ is simpler than the one provided by the general variable drift theorem. In this case, the runtime is controlled by a term that is inversely proportional to the value of δ. The theorem follows from Theorem 12 with $h(s) = \delta s$.

Some results on lower bounds drift theorems have also been developed: in particular, we state the multiplicative drift theorem for lower bounds, from [86, 55] since it is used in the proof of Corollary 5.

Theorem 14 (Multiplicative Drift Theorem, lower bound)**.** *Let $(X_t)_{t\geq 0}$ be a sequence of non-negative random variables with a finite state space $S \subseteq \mathbb{R}_{\geq 0}$ such that $0 \in S$, and with associated filtration $\mathcal{F}_t$. Let $s_{\min} := \min(S \setminus \{0\})$, and let $T := \inf\{t \geq 0 \mid X_t = 0\}$. Suppose there are two constants $0 < \beta, \delta \leq 1$ such that for all $s \in S \setminus \{0\}$ and all $t \geq 0$ the following conditions hold:*

(i) $X_{t+1} \leq X_t$;

(ii) $\Pr[X_t - X_{t+1} \geq \beta X_t \mid \mathcal{F}_t, X_t = s] \leq \frac{\beta\delta}{1+\log(s/s_{\min})}$;

(iii) $\mathbb{E}[X_t - X_{t+1} \mid \mathcal{F}_t, X_t = s] \leq \delta s$.

Then,

$$\mathbb{E}[T] \geq \frac{1-\beta}{1+\beta} \cdot \frac{1 + \mathbb{E}[\log(X_0/s_{\min})]}{\delta}.$$

As we can see, this lower bound theorem is less intuitive and more complex also to verify.

2.6 Parameter Control

Evolutionary algorithms and many black-box iterative heuristics are parametrized algorithms. That means that the search behavior depends on the choice of parameters. The study of black-box complexity and drift analysis, and more generally of algorithm performance, is strongly connected to the study of parameter influence on algorithms behavior: both in terms of benchmarking parameter choices and in terms of designing new algorithms. In fact, in many cases, the practical motivation behind complexity studies is to provide insights about how an algorithm could be improved by a new parameter choice rule. That is also the case of our study.

2.6.1 Parameter Dependence

The so-referred *parameter selection problem*, known also *algorithm configuration*, has been strongly studied: a recent summary can be found at [19] and we will follow it for this section. Parameter dependence is a particular relevant task also because of its difficulties which arise for several reasons:

- *Complexity of performance prediction*: predict how the performance of an algorithm depends on the chosen parameter values is a very challenging problem. The tools exposed in the previous sections are useful to give theoretical insights about how different algorithmic choices can influence performances, but are not always easily applicable.
- *Problem- and instance-dependence*: since no globally good parameter choice exist, suitable parameter values can differ between different problems or even instances of the same problem.
- *State-dependence*: the best parameter values can change during the optimization process.

To overcome these challenges, two main approaches have been adopted:

- *Parameter tuning*: this process aims to identify parameters which are globally suitable for the optimization process, trying to address the problem- and instance-dependence. In empirical work, this means conducting a series of initial experiments to identify good parameter choices that can be used in the algorithm execution, while in theoretical work this usually consists of the choice of parameters that minimize performance bounds. However, this approach usually provides a static policy and thus does not help in addressing the state-dependence challenge.
- *Parameter Control*: although more challenging, parameter control addresses the problem of state-dependence, using a non-static choice of parameters. The optimal configuration is adjusted *on the fly*, taking advantage of the evolution of the optimization process.

This taxonomy has been commonly adopted in the field from the work [31].

It is intuitively to understand that parameter control is a much more promising approach and has become in the last years the leading solution for black-box optimization problems. In [19] the following classification of the techniques adopted to address a dynamic parameter choice is proposed:

- *State-dependent parameter control*: those are mechanisms that depend only on the current state of the search process, e.g. the current best fitness or the current population, and do not take into account the history of the algorithm. Examples of this are time-dependent parameter policies, as the one used in simulated annealing, or fitness-dependent policies, widespread in the class of evolutionary computing.

- *Success-based parameter control*: those are mechanisms that change the parameters from one iteration to the next. The current parameter is determined by the success of the previous configuration. These are mechanisms that take into account the previous iteration for the choice of the parameters of the following steps.

- *Learning-inspired parameter control*: those schemes are similar to the previous ones but tend to exploit a longer history than just one iteration; several of these approaches come from machine learning or reinforcement learning.

- *Endogenous (or self-adaptive) parameter control*: this category involves all the cases where parameters are encoded within the individuals of the population and evolve alongside the solution candidates. This is a particularly elegant approach, since there is no external or global policy but the parameters are carried by the individuals. In evolutionary computation, this means that the parameters evolve within the iterative process together with the individuals. As expected, this approach is particularly difficult to design and therefore not well understood in empirical application or in theoretical works.

- *Hyper-heuristics*: this is a higher-level optimization framework designed to automatically generate or select heuristics to solve a specific problem and thus automatize the entire optimization process, involving also the choice of the suitable low-level heuristic.

Since results on theory of parameter control are more algorithm and problem-specific, we will delve deeper into the topic in the following chapters when our analysis will take into account more specific example.

2.6.2 Dynamic Algorithm Configuration

Recently, the parameter selection problem has gained particular importance within the raise of AI, both because machine learning problems subsume the use of parameter-based optimization algorithm [48, 72], and because the rise

of machine learning suggested innovative approaches to address the problem of selecting the best parameter through learning-inspired mechanisms [6].

In particular, this approach gave birth to a new framework called *Dynamic Algorithm Configuration*. It was first proposed in [6] and then gained popularity in the following years [7, 13]. In [1] a complete review of the state-of-the-art and a formal definition of the field can be found and it is illustrated in this section.

The *Algorithm Configuration* problem can be stated as the process of determining a policy to set algorithm parameters with the aim of maximizing performance across a problem instance distribution. In the static setting, the process corresponds to the already named parameter tuning: the parameter configuration is fixed prior to execution. We have the following setting: a target algorithm A, with parameters $p_1, p_2, \ldots, p_k$ that we would like to configure. This consists of finding a suitable value in the *configuration space* $\Theta \subseteq \Theta_1 \times \Theta_2 \times \cdots \times \Theta_k$ which minimizes a given cost metric c. We also consider $i \in I$ the instances of algorithm A and $\mathcal{D}$, a distribution of the instances of the target problem.

In the classical, or *per-distribution (AC)*, approach we would like to minimize the cost metric over all possible instances, along the distribution $\mathcal{D}$. We therefore obtain the following optimization problem.

Definition 15. Given $\langle A, \Theta, \mathcal{D}, c \rangle$:

- A target algorithm A with configuration space Θ.
- A distribution $\mathcal{D}$ over the target problem instances with domain I.
- A cost metric $c : \Theta \times I \to \mathbb{R}$ that assesses the cost of using A with $\theta \in \Theta$ on $i \in I$.

Find a $\theta^* \in \arg\min_{\theta \in \Theta} \mathbb{E}_{i \sim D} [c(\theta, i)]$.

Note that in this definition, $A, \mathcal{D}$ and c are usually not given in closed form: c is itself normally a black-box procedure that evaluates A over a specific configuration of parameters θ.

A better result can be obtained with *per-instance algorithm configuration (PIAC)*, which consists in making the choice of θ depending on the specific problem instance $i \in I$. It is defined by the following optimization problem.

Definition 16. Given $\langle A, \Theta, D, \Psi, c \rangle$:

- A target algorithm A with configuration space Θ.
- A distribution D over target problem instances with domain I.
- A space of per-instance configuration policies $\psi \in \Psi$ with $\psi : I \to \Theta$ that choose a configuration $\theta \in \Theta$ for each instance $i \in I$.
- A cost metric $c : \Psi \times I \to \mathbb{R}$ assessing the cost of using A with $\psi \in \Psi$ on $i \in I$.

Find a $\psi^* \in \arg\min_{\psi \in \Psi} \mathbb{E}_{i \sim D} [c(\psi, i)]$.

We note that AC is a special case of PIAC when the configuration policies are chosen constant with respect to the instances, i.e. when $\Psi = \{\psi | \psi(i) = \psi(i'), \forall i, i' \in I\}$.

In Dynamic Algorithm Configuration (DAC), the aim is to optimally vary $\theta \in \Theta$ while executing A. The definition of DAC as an optimization problem analogously as how we defined AC is then the following.

Definition 17. Given $\langle A, \Theta, D, \Pi, c \rangle$:

- A step-wise reconfigurable target algorithm A with configuration space Θ.
- A distribution D over target problem instances with domain I.
- A space of dynamic configuration policies $\pi \in \Pi$ with $\pi : S \times I \to \Theta$ that choose a configuration $\theta \in \Theta$ for each instance $i \in I$ and state $s \in S$ of A.
- A cost metric $c : \Pi \times I \to \mathbb{R}$ assessing the cost of using $\pi \in \Pi$ on $i \in I$.

Find a $\pi^* \in \arg\min_{\pi \in \Pi} \mathbb{E}_{i \sim D} [c(\pi, i)]$.

Note that DAC is a further generalization of PIAC where the elements in Ψ also depend on the state $s \in \mathcal{S}$, i.e. $\Psi \subseteq \{\pi | \pi(i, s) = \pi(i, s'), \forall s, s' \in S, \forall i \in I\}$

In order for it to be possible to update the parameters on-the-fly, the algorithm must be of a specific iterative form that can be summarized in Algorithm 9. Essentially, the algorithm must proceed iteratively, and at each iteration there must be a point where the parameter configuration θ could be updated. It is evident that an evolutionary algorithm satisfies this scheme.

Algorithm 9 Step-wise execution of a dynamically configured target algorithm A

```
Input: Dynamic configuration policy π ∈ Π; target problem instance i ∈ I
s ← init(i)
while ¬is_final(s, i) do
    θ ← π(s, i)
    s ← step(s, i, θ)
end while
return s
Output: Solution for i found by A (following π)
```

DAC problems have often been addressed manually: dynamic parameter policies have been determined by humans and not in an automatic and data-driven fashion. However, recent developments in the field have seen the emerging of two different approaches: DAC by Reinforcement Learning (RL) and DAC by optimization.

In Reinforcement Learning [80], an agent interacts in an environment trying to optimize some kind of feedback. The mathematical framework is the Markov

Decision Process (MDP) and is explained more thoroughly in the Appendix C. The RL agent takes actions $a \in A$, observes a transition T from the current state $s \in S$ of the environment to $T(s, a) \in S$ and receives a reward $R(s, a) \in \mathbb{R}$. The aim of the RL agent is to maximize the expected future reward. In the DAC setting, T and R are in the form of a black-box. In particular, in this case the environment consists of an algorithm $\mathcal{A}$ solving an instance $i \in I$. The current state of the environment is $s = s(i, s')$ depends on the state of the algorithm s' and the initial instance i, which depends on a distribution $\mathcal{D}$. The action of the RL agent corresponds to the choice of a parameter configuration $\theta \in \Theta$. The reward corresponds to a performance cost of the step-wise execution of the algorithm.

Several approaches have been tried for the development of a DAC RL agent [54, 74, 5]. Modern solutions have been adopted recently, among them in particular the use of deep reinforcement learning [78, 7]. To overcome the fact that RL methods are not instance-aware and do not choose their starting state, in [6] it has been proposed to model DAC as a *contextual MDP (cMDP)*, i.e. collection of MDPs $\mathcal{M}(i)$, one for each instance, with a common action space A and a state space S, but instance-based transition functions T_i and reward functions R_i. The contextual RL agent can directly exploit specific knowledge about the instance we are considering.

DAC by optimization, on the other hand, formulates the configuration problem as non-sequential optimization problem. Given a search space Π for the policy π, the aim is to solve the following optimization problem:

$$\text{find } \pi^* \in \operatorname*{argmin}_{\pi \in \Pi} \mathbb{E}_{i \sim \mathcal{D}}[c(\pi, i)]$$

The different methods can then be developed depending on the representation of Π and the use of information from the objective function $f(\pi) = \mathbb{E}_{i \sim \mathcal{D}}[c(\pi, i)]$.

Chapter 3

Complexity Results for LeadingOnes and OneMax

In the theory of black-box randomized heuristics, it is common opinion to consider certain problems and algorithms as fundamental benchmarks for theoretical studies. In particular, OneMax is often referred to as the *Drosophila* of combinatorial problems, and it is widely used to benchmark evolutionary algorithms. The classic version takes as objective function f the number of ones in a bit string x, i.e. $\sum_{i=1}^{n} x_i$. Since, as we observed in section 2.4, single-instances problems lead to trivial black-box unrestricted complexity, we can generalize the OneMax problem as in [27]:

Definition 18. For all $n \in \mathbb{N}$ and all $z \in \{0,1\}^n$, let

$$\mathrm{OM}_z : \{0,1\}^n \to [n], \quad x \mapsto \mathrm{OM}_z(x) = |\{i \in [n] \mid x_i = z_i\}|,$$

be the function that assigns to each length-n bit string x the number of bits in which x and z agree. Being the unique optimum of OM_z, the string z is called its *target string*.

We refer to $\textsc{OneMax}_n$, or simply $\textsc{OneMax} := \{\mathrm{OM}_z \mid z \in \{0,1\}^n\}$ as the set of all (generalized) OneMax functions.

It is easy to note that the choice of $z = (1, 1, \ldots, 1)$ corresponds to the classic problem described before which we will denote by OM. It is also easy to see that for every $z \in \{0,1\}^n$ the fitness landscape of OM_z is isomorphic to that of OM.

If OneMax is the Drosophila of combinatorial problems, then we can define LeadingOnes as the lab mouse, used for benchmark algorithms but dealing with a more nuanced fitness landscape. It is probably the second-most studied function in the field and it complements OneMax in defining a problem where bits are no longer evaluated independently from their position. In fact, the fitness function of the classic instance corresponds to maximizing the longest prefix of ones in a bit string, i.e. $\mathrm{LO}(x) := \max\{i \in [0..n] | \forall j \in [i] : x_j = 1\}$. Also in this case, we can write a generalization of the problem to deal with a non-trivial analysis [27]:

Definition 19. For all $n \in \mathbb{N}$ and $z \in \{0,1\}^n$, let

$$\mathrm{LO}_z : \{0,1\}^n \to [n], \quad x \mapsto \max\{i \in [0..n] \mid \forall j \in [i] : x_j = z_j\},$$

be the length of the maximal joint prefix of x and z. Let $\textsc{LeadingOnes}^*_n := \{\mathrm{LO}_z \mid z \in \{0,1\}^n\}$.

For $z \in \{0,1\}^n$ and $\sigma \in S_n$ permutation of n elemnts, let

$$\mathrm{LO}_{z,\sigma} : \{0,1\}^n \to [n], \quad x \mapsto \max\{i \in [0..n] \mid \forall j \in [i] : x_{\sigma(j)} = z_{\sigma(j)}\},$$

be the maximal joint prefix of x and z with respect to σ. The set $\textsc{LeadingOnes}_n$ is the collection of all such functions; i.e.,

$$\textsc{LeadingOnes}_n := \{\mathrm{LO}_{z,\sigma} \mid z \in \{0,1\}^n, \sigma \in S_n\}.$$

Our study will thus refer to OneMax and LeadingOnes as benchmark for parameter control in black-box randomized heuristics.

Continuing with our metaphor, we can say that, in terms of algorithms, (1+1)EA is without doubt the corresponding *Drosophila* of black-box randomized heuristics, while RLS, the lab rat of evolutionary computation, with a more nuanced mutation operator which does not treat each bit independently of the position. We already described the two heuristics in Section 2.3.2.

In this chapter, therefore, we will provide the main results on complexity and parameter control for RLS and (1+1) EA on LeadingOnes and OneMax, with a particular focus on RLS on LeadingOnes, which is the most complex and interesting case.

3.1 OneMax and LeadingOnes Complexity

3.1.1 Unrestricted Complexity

We can start studying the most general black-box complexity model: the unrestricted one. It is clear from theorem 6, that the unrestricted black-box complexity of OneMax is at most $(2^n+1)/2$, however far better bounds have been derived. The unrestricted complexity of OneMax has indeed been studied since the 1960s, and therefore before the notion of black-box computation came out. It has been studied firstly by Erdős and Rényi [32] related to a question about *coin-weighing problems*. They were the first to prove a result that we can translate as the unrestricted black-box complexity of OneMax; several similar bounds have been proved in the following years and then again in the early 2000s. In particular, we can trace from [32, 60, 59, 12] the following theorem.

Theorem 20 (Unrestricted black-box complexity of OneMax)**.** *The unrestricted black-box complexity of OneMax is at least* $(1-o(1))n/\log_2(n)$ *and at most* $(1+o(1))2n/\log_2(n)$*. Thus,* $\Theta(n/\log n)$*.*

The proof of the lower bound follows Yao's minimax principle, applied to $\textsc{OneMax}_n$ with uniform distribution. The argument is similar to the one used

to intuitively prove the corollary 5: with each function evaluation, we obtain roughly $\log_2(n+1)$ bits of information as value of the objective, i.e. a number in between 0 and n. Since we need n bits of information to obtain the optimal string, we have to perform at least $n/\log_2(n+1)$ iterations to get to the optimum.

It is interesting to give a sketch also of the upper bound proof; in [32], it is proven that if you sample $\mathcal{O}(n/\log n)$ bit strings uniformly at random from the solution space $0,1^n$, you can find the target string with high probability. Therefore, an asymptotically optimal algorithm for OneMax consists in randomly sampling $\mathcal{O}(n/\log n)$ bit strings independently and uniformly [25].

The unrestricted black-box complexity is well understood also for LeadingOnes and its generalizations. In particular, for LeadingOnes^*_n we have the following result from [30]:

Theorem 21. *The unrestricted black-box complexity of the set* LeadingOnes^*_n *is* $n/2 \pm o(n)$.

The result is quite intuitive: we can consider an algorithm that, knowing that the first k bits are correct, proceeds by flipping the $(k+1)-th$ bit and queries the new string. It continues this process of flipping and querying until it finds the optimal solution. On average, this requires around $n/2$ queries because, for a random bit string, roughly half of the bits are expected to be correct at random.

We also have a result for LeadingOnes_n, taken from [2]:

Theorem 22. *The unrestricted black-box complexity of* LeadingOnes_n *is* $\Theta(n \log\log n)$.

The proof is quite complex: for the lower bound, it uses Yao's minimax principle applied to the uniform distribution over the instances $\text{LO}_{z,\sigma}$ with a uniform distribution over the instances $z_{\sigma(i)} = (i \mod 2), i = 1,\ldots,n$. Once σ is determined, we can reduce the problem to the case of Theorem 22, so we need around $n/2$ evaluations on average to find the optimum. Therefore, complexity is determined by the determination of the target permutation σ and the expected number of evaluations to find it is determined by drift analysis.

3.1.2 Memory-Restricted Complexity

In our analysis, it is also important to talk about the $(1+1)$ memory-restricted black-box complexity model. In fact, RLS and $(1+1)$ EA are part of this class since they exploit only the best solution so far and, therefore, have memory of queries only of length 1.

A complexity result exists for OneMax and is stated in [17]:

Theorem 23. *The* $(1+1)$ *memory-restricted black-box complexity of* OneMax *is* $\Theta(n/\log n)$.

The idea behind the theorem is to divide the string into $\sqrt{n}$ substrings of length $\sqrt{n}$ and optimize one substring at a time through the random sampling strategy from [32] we have already mentioned. Then it is proven that this requires $\mathcal{O}(n/\log n)$ evaluations.

It is important to note that the complexity is the same as the unrestricted one, which means that for solving OneMax it is not really asymptotically beneficial in terms of complexity to have more memory than 1 evaluation; this further motivates our choice of studying $(1+1)$ EA and RLS.

3.1.3 Unary Unbiased Black-box Complexity

Another important complexity result for our study is the unary unbiased, since this bound is attained by both RLS and $(1+1)$ EA. For OneMax, the Theorem 10 can be applied and thus it is immediate to state that the unary unbiased complexity of OneMax is greater than $\Omega(n \log n)$. This bound is better derived in [20] where the following theorem is stated:

Theorem 24. *The unary unbiased black-box complexity of OneMax is* $n \log(n) - cn \pm o(n)$ *for a constant* c *between 0.2539 and 0.2665.*

Also for LeadingOnes, Theorem 10 gives a lower bound for unary unbiased black-box complexity of $\Omega(n \log n)$. However, problem-specific arguments lead to a better bound [56]:

Theorem 25. *The unary unbiased black-box complexity of LeadingOnes is* $\Theta(n^2)$.

The theorem is proved through additive drift. It is defined as a potential function a map from the time t to the largest number of initial ones and initial zeros in the query $x^{(1)}, \ldots, x^{(t)}$ and shown the fact that a potential of $k > n/2$ cannot increase in one iteration by more than an additive term $4/(k+1)$, in expectation.

We note that both these complexity bounds give us important information because they are higher than the unrestricted ones, thus we can estimate better the performance time for RLS and $(1+1)$ EA that attain to the class of unary unbiased algorithms.

3.1.4 Elitist Black-box Complexity

Lastly, it is important to talk about the results on elitist black-box complexity, since this notion includes all hill-climbing algorithms, such as RLS and $(1+1)$ EA.

However, for OneMax, the problem of (1+1) elitist black-box complexity remains unsolved. The nearest result gives complexity results with a certain probability and is stated in [29]:

Theorem 26. *For every constant* $0 < \varepsilon < 1$*, there exists a (1+1) elitist black-box algorithm that finds the optimum of any OneMax instance in time* $\Omega(n)$ *with probability at least* $1 - \varepsilon$*, and this running time is asymptotically optimal.*

For LeadingOnes, the $(1+1)$ elitist complexity has been shown in [28].

Theorem 27. *The* $(1+1)$ *elitist black-box complexity of LeadingOnes is* $\Theta(n^2)$.

The proof cannot take advantage of Yao's minimax principle directly since elitist algorithms can not be written as a convex combination of deterministic algorithms, as already discussed. Nevertheless, extending the class of elitist algorithms to a superset in which every randomized algorithm can be expressed as a probability distribution over deterministic heuristics, one can still find use of Yao's principle. The lower bound for this superset trivially extends to all elitist black-box algorithms.

It is interesting to make some last considerations on this case: for LeadingOnes, the complexity is the same as the unary unbiased one that we can thus take as reference for RLS and (1+1) EA; for OneMax, the (hypothetical) complexity of $\Theta(n)$ would be lower than $\Theta(n \log n)$, which will thus be the reference for RLS and $(1 + 1)$ EA.

3.1.5 Drift Analysis for OneMax and LeadingOnes

A more comprehensive overview of OneMax and LeadingOnes and of the behavior of RLS and $(1 + 1)$ EA can be completed with the drift analysis tools presented in Section 2.5. This type of analysis is complementary to black-box complexity and is indeed part of the proof of some complexity bounds. However, a more detailed explanation can give interesting insights on how the algorithms work beyond the simple complexity result, since drift analysis, as we already noted, studies how the algorithm proceeds to understand the execution time of an algorithm.

Several interesting results exist for $(1 + 1)$ EA and RLS on OneMax and LeadingOnes that make use of drift analysis: the first deals with (1+1)EA on linear pseudo-boolean objective functions, i.e. $f : \{0, 1\}^n \longrightarrow \mathbb{R}, x \mapsto \sum_{i=1}^{n} w_i x_i$, where w_i are constant. It is easy to see that, taking $w_i = 1 \forall i$, f corresponds to the OneMax function. We consider $(1 + 1)$EA with constant mutation rate $p = 1/n$, and choose as potential function the fitness, $X_t = f(x^{(t)})$. This yields to the lower bound for multiplicative drift

$$\Delta_t(s) \geq \Omega(\frac{s}{n})$$

since $(1 + 1)$EA has a constant probability $1/n$ of flipping exactly one bit at each of the s iterations.

Therefore, from the multiplicative drift theorem (Theorem 13) we have the following.

Theorem 28. *There exists a constant $\delta > 0$, such that*

$$\mathbb{E}[T] \leq \mathcal{O}\left(\frac{1 + \mathbb{E}[\log(X_0)/w_n]}{\delta}\right)$$

which turns out to be $\mathcal{O}(n \log n)$ for OneMax substituting $w_n = 1$, which is tight as we already observed in the previous section.

It is important to note that T, the random variable that denotes time, effectively represents the number of evaluations and, thus, the number of iterations of the algorithm. Therefore, it corresponds to the time measurement we take for considerations about complexity.

Also, a result for RLS on OneMax that makes use of drift analysis exists. To prove it, we exploit the fact that one round of one-bit RLS corresponds to a *coupon collector process (CCP)*. The coupon collector process is a stochastic process: the setting is that we have n types of coupons and a collector wants to get at least one coupon of each type. However, the coupons are sold in opaque wrappings, so the collector does not know which type of coupon it will receive. We assume that each coupon has the same frequency $1/n$. Through drift analysis, we answer the question of how many coupons does one have to buy to possess one for each type. The analogy with RLS is clear: having the coupon i out of n corresponds to having a 1 in position i of a bit string of length n.

We denote as X_t the number of coupons after t purchases, thus the OneMax fitness in our analogy. For $X_t = s$, the probability of obtaining a new type with the next purchase is s/n. In this case, X_t decreases by one and thus the drift is $\Delta_t(s) = s/n$. We can thus apply the variable drift theorem with $s_{min} = 1$ and $h(s) = s/n$ and obtain the following.

Theorem 29. *Let T be the number of RLS evaluations with radius 1 needed to solve OneMax, then*

$$\mathbb{E}[T] \leq n \log n + n$$

Multiplicative drift can also be applied to provide a lower bound for RLS on OneMax. We take the same X_t as in the CCP; since it decreases by at most 1, we choose $\xi(s) := s - 1$ and $h(s) := (s + 1)/n$. Therefore, we obtain

Theorem 30. *A lower bound for the expected optimization time of OneMax using RLS with radius 1 is*

$$\begin{aligned}
\mathbb{E}[T] &\geq \frac{s_{\min}}{h(s_{\min})} + \mathbb{E}[\int_{s_{\min}}^{X_0} \frac{1}{h(\sigma)} d\sigma] = \\
&= \frac{1}{2/n} + \mathbb{E}[\int_{1}^{X_0} \frac{n}{\sigma + 1} d\sigma] = \\
&= \frac{n}{2} + n \cdot \mathbb{E}[\log(X_0 + 1) - \log(2)] \geq \\
&\geq n \log n - \mathcal{O}(n)
\end{aligned}$$

As we can see, drift analysis gives us insight into how the value of $\Theta(n \log n)$ complexity is obtained for OneMax. It also says us that RLS and $(1 + 1)$ EA are optimal representative of the class of unary unbiased black-box algorithms.

The last result which makes use of drift analysis is the upper bound of the optimization time of RLS on LeadingOnes. It is an application of additive drift: we take as potential the fitness distance from the optimum $X_t = n - f(x^{(t)})$. In this case, we can apply the additive drift and obtain the following result.

Theorem 31. *Let T denote the number of RLS evaluations with radius 1 to*

optimize LeadingOnes*. It follows that*

$$\frac{(n-1)n}{2} \leq \mathbb{E}[T] \leq n^2$$

The proof is more elaborated and consists in bounding the drift and then applying the additive drift theorem. The key observation consists in calling $\mathcal{E}$ the event *improvement of the* $k+1$*-th bit* and write the drift as

$$\begin{aligned}\Delta_t(k) &= \mathbb{E}[X_t - X_{t+1} | X_t = k] = \\ &= \mathrm{P}[\mathcal{E}]\mathbb{E}[X_t - X_{t+1} | X_t = k, \mathcal{E}] = \\ &= \frac{1}{n}\mathbb{E}[X_t - X_{t+1} | X_t = k, \mathcal{E}]\end{aligned}$$

and then bound $\mathbb{E}[X_t - X_{t+1} | X_t = k, \mathcal{E}]$ in between 1 and 2.

Also in the case of RLS for LeadingOnes, drift analysis allows us to better understand the complexity bounds. We observe indeed why the complexity of LeadingOnesis $\Theta(n^2)$ for unary unbiased algorithms and that RLS is an optimal representative of the class.

3.2 Parameter Control for RLS and $(1+1)$ EA

3.2.1 Motivations

In the previous section, we discussed theoretical results for RLS and $(1+1)$ EA on OneMax and LeadingOnes. The results we stated are referred to black-box complexity and drift analysis and generally give good asymptotic results, involving Θ and big-Oh bounds. However, we are now interested in seeing how and in which sense an optimal configuration of the algorithm could impact on performances; we already observed through drift analysis that RLS and $(1+1)$ EA are asymptotically optimal for OneMax and LeadingOnes, as they realize the complexity bounds. We summarize the asymptotic results in Table 3.1.

Algorithm	**Problem**	**Asymptotic bound**
$(1+1)$ EA	OneMax	$\Theta(n \log n)$
RLS	OneMax	$\Theta(n \log n)$
$(1+1)$ EA	LeadingOnes	$\Theta(n^2)$
RLS	LeadingOnes	$\Theta(n^2)$

Table 3.1: Summary of asymptotic results

Building on this, we are interested in studying the influence of parameter control and algorithmic choices. In particular, since we have tight asymptotic bounds, it is interesting to analyze if better asymptotic constants can be found and if different algorithmic choices could improve the behavior towards the solution, e.g. speeding up the earliest phases of the optimization process. This is also the opportunity to understand in concrete cases the influence of parameter choice we have already discussed in Section 2.6.

So far the results obtained are referred to a standard use of static parameters, in particular the standard choices are 1 for the radius of RLS and $1/n$ as mutation rate for $(1+1)$ EA. For RLS the choice to fix the radius at 1 is natural: a different static parameter will make the algorithm get stuck when encountering a solution of fitness $n-1$; flipping exactly one bit at a time, on the other hand, guarantees convergence regardless of the starting individual.

For $(1+1)$ EA, on the other hand, the choice of $1/n$ seems less motivated: it is indeed the choice analogous to running RLS with radius 1. With radius 1, in fact, each bit has a $1/n$ probability of being flipped at each RLS iteration. However, any choice of mutation rate $p \in (0,1)$ would guarantee convergence for LeadingOnes and for OneMax, so a better parameter p may exist.

To provide a motivated choice of parameters, we follow some theoretical insights: a precise study of dependence of the expected runtime from the parameters can be exploited to find the best configuration through an optimization process. In this section, we present the main results for the control of the radius of RLS and the mutation rate of $(1+1)$ EA on LeadingOnes and OneMax.

3.2.2 Static Parameter Policies for RLS and $(1+1)$ EA

An exact expectation runtime for $(1+1)$ EA optimizing OneMax, is not actually known. However, in [35], a rather precise study of the expected optimization time with mutation rate $p = c/n$ was carried out, providing some tight results. The expected runtime can be summarized in the following:

Theorem 32. *The runtime estimate of* $(1+1)$ *EA with mutation rate* $p = c/n$*, with* $c \in (0,n)$*, is*

$$\mathbb{E}[T] = \frac{e^c}{c} n \ln(n) \pm \mathcal{O}(n)$$

The proof is particularly elaborated and requires deep tools from probability theory, and is provided it in [86].

However, the result is of direct importance because it allows us to find the optimal static mutation rate for $(1+1)$ EA on OneMax; by optimizing e^c/c for c, we easily find that the minimum expected time is given by $c = 1$, which indeed motivates the standard static choice of $p = 1/n$. A simpler proof with a detailed analysis of the bound for the expected time in the case $c = 1$ has been provided in [49] and restates the tight result we reported in Theorem 24.

The study of RLS on OneMax can indeed lead to an exact result, as stated in [18] and [15]. This result makes use of the fact, already observed in Section 3.1.5, that RLS on OneMax with radius 1 is equivalent to a coupon collector process in which each type of coupon is initially present with probability 1/2. The runtime of the coupon collector process with $n-k$ coupon out of n initially present is distributed as $\sum_{i=1}^{k} \mathrm{Geom}(i/n)$, where all random variables are intended to be independent. We then call X the random variable which indicates the number of initial ones; it is easy to observe that it follows a binomial distribution $\mathrm{Bin}(n, 1/2)$. The exact distribution of T is therefore the following:

Theorem 33. *Let T be the number of iterations taken by RLS with radius 1 to sample the optimum of the* OneMax *function. Then*

$$T \sim \sum_{i=0}^{n-1} \mathbf{1}_{\{X \leq i\}} \cdot Geom(\frac{n-i}{n})$$

where $X \sim Bin(n, 1/2)$ and X and the geometric distributions are assumed to be independent.

By calculating the expected value of the previous expression, we obtain that the expected runtime is

$$\mathbb{E}[T] = \sum_{k=0}^{n-1} \frac{1}{n-k} = n \sum_{k=1}^{n} \frac{1}{k} =: nH_n$$

where H_n, the n-th harmonic number. H_n can be approximated asymptotically by $\log(n) + \gamma + \mathcal{O}(1/n)$, where γ is the Euler-Mascheroni constant, and in coherence with the asymptotic result of OneMax complexity of $\Theta(n \log n)$.

Some interesting results easier to interpret are obtained from the precise study of expected runtime on LeadingOnes. For (1+1) EA on LeadingOnes an exact analysis is performed in [8]. The paper is particularly important, since it contains indeed the first precise analysis which does not rely on asymptotic bounds. The result for static parameters is the following.

Theorem 34. *The expected optimization time of* (1+1) *EA with fixed mutation rate p for* LeadingOnes *is*

$$\mathbb{E}[T] = \frac{1}{2p^2}[(1-p)^{1-n} - (1-p)]$$

The theorem is derived in Appendix B.2.

The result is particularly powerful, as it allows us to determine the optimal parameter choice through optimization: we just need to find the value of p which minimizes $\mathbb{E}[T]$. Therefore, we calculate the derivative of $\mathbb{E}[T]$ with respect to p and pose it equal to zero:

$$\frac{d\mathbb{E}}{dp}(p) = -\frac{1}{p^3}[(1-p)^{1-n} - (1-p)] + \frac{1}{2p^2}[-(1-n)(1-p)^{-n} + 1] = 0 \quad (3.1)$$

We cannot solve this equation analytically, but we can derive numerically that the optimal parameter is $p \simeq 1.5936/n$ which leads to an optimization time of $\mathbb{E}[T] \simeq 0.77201n^2$, for n sufficiently large. In this way, we can also compute the expected runtime when choosing $p = 1/n$, which corresponds to $\mathbb{E}[T] \simeq 0.85914n^2$. This means that the standard choice is not optimal and that even with a static parameter choice, we can obtain a performance improvement of 16.1%.

The previous result is straightforward and effective. However, a more general analysis was conducted in [15]: it provides exact results for all (1+1) al-

gorithms on LeadingOnes function. It is particularly useful because it holds not only for $(1+1)$ EA but also for RLS:

Theorem 35. *Let T be the first time a (1+1) algorithm generates the optimum of the LeadingOnes function. Then*

$$T \sim \sum_{i=0}^{n-1} X_i \cdot Geom(q_i)$$

where $X_0, \ldots, X_{n-1}$ are uniformly distributed binary random variables, the X_i and $Geom(q_i)$ are mutually independent, and for all $i \in [0..n-1]$ we denote by q_i the probability that the mutation operator generates from a search points of fitness exactly i a strictly better search point.

Consequently,

$$\mathbb{E}[T] = \frac{1}{2} \sum_{i=0}^{n-1} \frac{1}{q_i}$$

where we read $1/p_i = \infty$ if $p_i = 0$.

The key idea of the proof is to observe that the waiting time for an improvement of a solution of fitness i corresponds in the time to flip the $i+1$-th bit and is thus modeled by a geometric distribution with parameter the probability q_i of flipping the $i+1$-th bit. The arguments proceeds then by induction on the distance $n-i$ from the optimum. Details are provided in Appendix B.2.

For $(1+1)$ EA we obtain again the previous result: in this case, if p is the mutation rate the value of q_i corresponds to the probability of flipping more zeros than ones: it depends on p and is $q_i(p) = (1-p)^i p$ which yields to an expected time of $\mathbb{E}[T] = 1/2 \cdot p^2((1-p)^{1-n} - (1-p))$. For the standard choice $p = 1/n$, this is $\mathbb{E}[T] \simeq 0.86n^2$, for $p \simeq 1.59/n$, $\mathbb{E}[T] \simeq 0.77n^2$, which correspond to the values previously obtained.

For RLS, on the other hand, with the standard parameter choice $r = 1$, the value of q_i id $1/n$ and thus $\mathbb{E}[T] = 0.5n^2$.

3.2.3 Dynamic Parameter Policies for RLS and $(1+1)$ EA

However, it is intuitive to notice that both (1+1) EA and RLS will benefit from a dynamic choice of parameters. In particular, this could be beneficial in the early stages of the optimization, since the algorithm would take advantage of a more "risky" attitude to better explore the search space and only later exploit a small neighborhood of the best solution so far. Studies conducted so-far associate the control of this behavior with the distance from the optimum measured in fitness terms: the idea behind is that as we approach the optimal solution, we would not like to look for the solution *too far* from the best candidate so far. Theoretical-inspired policies have thus been developed that suggest different parameter choices depending on the fitness value of the current solution candidate.

For $(1+1)$ EA on OneMax the optimal adaptive mutation rate can be directly derived following a result similar to Theorem 34, also reported in [8].

The execution time is now written as a function of n parameters $p_1, \ldots, p_n$, each of which represents the mutation rate used by the algorithm dependent on the fitness of the current individual x, i.e. if x has fitness $f(x)$, the algorithm will operate on the solution with mutation rate $p_{f(x)}$. Following a similar argument used to find the optimal fixed mutation rate in 3.1, we obtain the following.

Theorem 36. *Let $f(x)$ be the fitness of the current individual, then the optimal mutation rate is*

$$p_{f(x)} = \frac{1}{f(x)+1}$$

Through these choices, we obtain almost matching upper and lower bounds for the expected optimization time:

Theorem 37. *Let $p_{f(x)} = 1/(f(x)+1)$, the the expected optimization time of the adaptive $(1+1)$ EA is bounded as follows:*

$$\frac{e}{4}n^2 - \frac{e}{4}n \leq \mathbb{E}[T] \leq \frac{e}{4}n^2 + \frac{e}{4}n$$

Therefore, we obtain a better result than the one obtained with a fixed mutation rate, with an improvement of approximately 12.0% to the optimal fixed mutation rate $p \simeq 1.59/n$ and of approximately 20.9% to the standard mutation rate $p = 1/n$.

A dynamic policy can also be beneficial for the radius r_i of RLS on LeadingOnes. In this case, however, we do not rely on a precise analysis of the expected runtime with fitness-dependent mutation. There is still a good approach to determine an optimal fitness-based policy. It follows the idea that an optimal algorithm behavior is to iteratively maximize the fitness. This translates into trying to choose the parameters that maximize the probability of a strict fitness improvement of the candidate solution.

The probability of obtaining a strictly better solution by flipping k random bits in a search point of fitness i can be calculated directly and results in the following.

$$q(n,k,i) = \frac{k(n-i-1)\cdot\cdots\cdot(n-i-k+1)}{n(n-1)\cdot\cdots\cdot(n-k+1)}$$

The value is easy to study and, in particular, to maximize with respect of k. The key observation is that

$$q(n,k,i) \leq q(n,k+1,i) \text{ if and only if } i \leq (n-k)/(k+1)$$

Therefore the optimal fitness-dependent policy, i.e. the one that chooses k with respect to the fitness i maximizing q_i, is

$$k(n,i) := \lfloor n/(i+1) \rfloor \tag{3.2}$$

bits when the current fitness is i. This results in an expected runtime of $0.39n^2$, which corresponds to a 22% improvement of the choice of fixed parameters.

It is important to note also that this policy is monotonically decreasing, i.e. the higher the fitness, the fewer the optimal number of bits to be flipped.

This gives us insight on the behavior of the algorithm in solving the optimization problem: as expected, an optimal policy requires looking further from the current individual if it is not a good solution candidate.

3.2.4 Exact Runtime Analysis for OneMax

For OneMax, as well, a study of the optimal dynamic parameter choices has been conducted for both RLS and $(1+1)$ EA. The first approach follows the same idea that an optimal algorithm tries to maximize fitness improvement at each step. This led to the construction of a *so-called* drift-maximizer parameter policy in [20]. The main idea is to choose as radius $k_{\text{drift}}(n,l)$ a value that maximizes the expected drift, defined in terms of the fitness value, for the specific current fitness l. Mathematically, the expression to be maximized is the following:

$$\begin{aligned}\mathbb{E}[\Delta(n,l,k)] &:= \mathbb{E}[\max\{\mathrm{OM}(y)-\mathrm{OM}(x),0\}|\mathrm{OM}(x)=l, y \leftarrow \mathrm{flip}_k(x)] \\ &= \sum_{i=l+1}^{l+k} (i-l)\,\mathrm{P}[\mathrm{OM}(y)=i|\mathrm{OM}(x)=l, y \leftarrow \mathrm{flip}_k(x)] \\ &= \sum_{i=\lceil k/2\rceil}^{k} \frac{\binom{n-l}{i}\binom{l}{k-i}(2i-k)}{\binom{n}{k}}\end{aligned} \tag{3.3}$$

In the last line, the fact that flipping i of the previously incorrect $n-l$ bits implies that we flip $k-i$ of the previously correct l bits is used, resulting in a fitness increase of $i-(k-i)=2i-k$. This event occurs with probability $(\binom{n-l}{i}\binom{l}{k-i}(2i-k))/\binom{n}{k}$, since there are $\binom{n-l}{i}$ ways of choosing i previously incorrect bits, $\binom{l}{k-i}$ ways of choosing $k-i$ previously correct bits, and $\binom{n}{k}$ ways of choosing pairwise different bit positions.

Therefore, we choose as radius $k_{\text{drift}} \in \operatorname{argmax}_k \mathbb{E}[\Delta(l,k)]$. For RLS on OneMax this algorithm has been proved to be *near-optimal*, i.e. to exceed the unary unbiased black-box complexity of at most $\mathcal{O}(n^{2/3})\log^9 n)$.

It is straightforward to see that an analogous approach could be built for $(1+1)$ EA on OneMax, choosing the mutation rate $p_{\text{drift}}(n,l)$ which maximizes the following

$$
\begin{aligned}
&\mathbb{E}[\Delta(n,l,p)] := \\
&= \mathbb{E}[\max\{\mathrm{OM}(y) - \mathrm{OM}(x), 0\} | \mathrm{OM}(x) = l, y \leftarrow \mathrm{flip}_k(x), k \sim \mathrm{Bin}(n,p)] \\
&= \sum_{i=l+1}^{n} (i-l)\, \mathrm{P}[\mathrm{OM}(y) = i | \mathrm{OM}(x) = l, y \leftarrow \mathrm{flip}_k(x), k \sim \mathrm{Bin}(n,p)] \\
&= \sum_{k=1}^{n} \mathrm{Bin}(n,p)(k) \mathbb{E}[\Delta(n,l,k)] \\
&= \sum_{k=1}^{n} \left(\binom{n}{k} p^k (1-p)^{n-k} \sum_{i=\lceil k/2 \rceil}^{k} \frac{\binom{n-l}{i}\binom{l}{k-i}(2i-k)}{\binom{n}{k}} \right)
\end{aligned}
$$

However, although this approach seems promising and, in fact, obtains good results, it is still *near optimal*. This led to the development of an exact optimal approach in [10]. The idea is that the best-possible algorithm is the one that minimizes the expected remaining optimization time. The optimal remaining time is then defined iteratively backward. For RLS we first of all observe that if the fitness is $n-1$, the optimal radius k is 1, since it is also the only feasible choice. The expected time to the optimum is then n, since we have a probability of $1/n$ to flip the last bit and reach the optimum and a probability of $(n-1)/n$ to stay in state $n-1$.

Then, we calculate the optimal expected runtime from any possible starting fitness value with the following recursive formula.

$$
\mathbb{E}[T^{(k)}_{\mathrm{opt}}(n,l)] := 1 + \sum_{i=l+1}^{n-1} \mathrm{P}[\mathrm{OM}(y) = i | \mathrm{OM}(x) = l, y \leftarrow \mathrm{flip}_k(x)] \mathbb{E}[T_{\mathrm{opt}}(n,i)]
$$

We then simply set $k_{\mathrm{opt}}(n,l) = \mathrm{argmin}_k\, \mathbb{E}[T^{(k)}_{\mathrm{opt}}(n,l)]$.

In [10] it is shown that the optimal k is effectively different from the *drift-maximizer* k, even if the performance improvement is not asymptotically notable.

The analogous approach can also be used to study the optimal mutation rate $p_{\mathrm{opt}}(n,l)$ for $(1+1)$ EA. The expression to be minimized is, in this case, the following:

$$
\begin{aligned}
&\mathbb{E}[T^{(p)}_{\mathrm{opt}}(n,l)] = \\
&= 1 + \mathbb{P}[\mathrm{OM}(y) \leq l | \mathrm{OM}(x) = l, y \leftarrow \mathrm{flip}_k(x), k \sim \mathrm{Bin}(n,p)] \mathbb{E}[T^{(p)}_{\mathrm{opt}}(n,l)] + \\
&+ \sum_{i=l+1}^{n} (i-l) \mathbb{P}[\mathrm{OM}(y) = i | \mathrm{OM}(x) = l, y \leftarrow \mathrm{flip}_k(x), k \sim \mathrm{Bin}(n,p)] \mathbb{E}[T_{\mathrm{opt}}(n,i)]
\end{aligned}
$$

and for $l = n-1$, we take $p_{\mathrm{opt}}(n,l) = 1/n$, $\mathbb{E}[T^{(1/n)}_{\mathrm{opt}}(n,n-1)] = n$.

The formula seems heavy but the computation is actually straightforward, using again the transition probabilities as in (3.3).

3.2.5 DAC on LeadingOnes

As we saw, the study of parameter selection is quite well understood for benchmark problems, in particular for RLS on LeadingOnes, which, among the ones presented, corresponds also to the more nuanced case, incorporating dependence in the choice of bit to flip and also position-dependence in the fitness function. It is therefore useful to build on these theoretical results a study of modern approaches to the parameter problem. This is the reason why the radius choice for RLS in LeadingOnes has been used as an important benchmark for the study of DAC techniques in [7]. The work is particularly relevant for two results: it studies the behavior of the optimal policy of RLS on LeadingOnes for restricted portfolios of possible radii, and then applies a DAC agent to the radius selection in RLS solving LeadingOnes for both the full and restricted portfolio.

The idea behind the study of different restricted portfolios is to understand the influence of different parameter possibilities in the learning process of the RL agent, in particular the dimension of the portfolio. This type of study gives interesting insights also in terms of algorithm design, since in many practical problems the full portfolio of parameter choices is not well known and therefore it is useful to understand what a good choice of the portfolio would be.

The baseline of the work in [7] is the optimal policy for the full portfolio $\mathcal{K} = [0..n]$ already presented in (3.2). The optimal policy for restricted portfolios $\mathcal{K}$ is derived consequently as

$$\pi_{\text{opt}}^{(\mathcal{K})}(i) = \underset{r \in \{r_i^{\text{sup}}, r_i^{\text{inf}}\}}{\text{argmax}} \; q(r, i)$$

where $r_i^{\text{sup}} = \max\{r \in \mathcal{K} | r < \pi_{\text{opt}}(i)\}$ and $r_i^{\text{inf}} = \min\{r \in \mathcal{K} | r > \pi_{\text{opt}}(i)\}$, and $q(r, i)$ is the probability of strict improvement of fitness if the current fitness is i and the radius chosen is r. The determination of the highest value for the probability of strict improvement $q(\cdot, i)$ follows from the observation that the function $q(\cdot, i)$ is concave: it has a single maximum. Therefore, to find the value of $r \in \mathcal{K}$ that realizes argmax, we simply have to order $\mathcal{K}$ and find the highest value before $q(\cdot, i)$ begins to decrease.

Once the value k of the dimension of the portfolios is fixed, in the paper the following is studied:

- `powers_of_2`: $\{2^i | 2^i \leq n \wedge i \in [0..k-1]\}$
- `initial_segment`: $[1..k]$
- `evenly_spread`: $\{i \cdot \lfloor n/k \rfloor + 1 | i \in [0..k-1]\}$
- `optimal`: portfolio with the lowest expected time among all k-subsets of n that contain the search radius 1. This portfolio is determined by a brute-force approach by trying all possible subsets.

We note that initial studies on optimal policies for restricted portfolios are conducted thoroughly: the first insights suggest that `initial_segment` and

`powers_of_2` are the similar to the `optimal` portfolio, which matches results for full portfolio even for small k. `powers_of_2` seems then to obtain better results than `initial_segment`. This suggests that it is good to have more than one small radius to choose among, but having also some bigger radii is more beneficial than having more small radii.

In the paper, these results are used as a benchmark for an DAC agent. The DAC framework has already been presented as a contextual Markov decision process (cMDP) in Section 2.6.2. In this case a RL agent is used which operates in an offline training phase: the state space corresponds to the possible fitness values, the action phase is the chosen portfolio $\mathcal{K}$. The transition and reward functions remain as black-box; in particular, the reward at time t is modeled as $r_t = f(x_t) - f(x_{t-1}) - 1$, where x_t is the current individual at time t and $f(x_t) =: s_t$ its fitness.

The DAC agent adopted for the problem is $\mathcal{Q}$-learning [84], which is a reinforcement learning paradigm in which the goal is to learn the $\mathcal{Q}$-function $\mathcal{Q} : S \times \mathcal{A} \longrightarrow \mathcal{R}$, which maps a state-action pair to the cumulative function reward that is received after playing an action a in state s. Given a state s_t and an action a_t, the $\mathcal{Q}$-value $\mathcal{Q}(s_t, a_t)$ can be updated using temporal differences (TD) as

$$\mathcal{Q}(s_t, a_t) \leftarrow \mathcal{Q}(s_t, a_t) + \alpha((r_t + \gamma \max \mathcal{Q}(s_{t+1}, \cdot)) - \mathcal{Q}(s_t, a_t))$$

where α is the learning rate and γ is the discount factor.

The reward-maximizing policy can then be defined only by using the learned $\mathcal{Q}$-function as $\pi(s) = \text{argmax}_{a \in \mathcal{A}} \mathcal{Q}(,\cdot)$. Typically, for better exploration, ε-greedy approach is used, where ε gives the probability that an action a_t is replaced with a randomly sampled one.

Furthermore, in [63] is proposed to model the $\mathcal{Q}$-function using a neural network. This approach was shown to work very well, in particular if combined with the idea in [43] to use two copies of the neural network, one used to select maximizing actions and the other to predict the value, in order to improve stability. The result is the so-called double deep $\mathcal{Q}$ network, which was used in the paper [7] as DAC agent for RLS on LeadingOnes.

The results are analyzed in detail in the paper: we will sum up the main ones. Three experiments are carried out: in the first one, the RL agent tries to optimize LeadingOnes for $n = 50$. The result is very good: the DDQN is able to reach the performance of the optimal policy in all cases tested, and the learned policies are quite similar to the optimal ones. Secondly, the impact of portfolio size on RL's learning behaviors is further investigated for $n = 50$ and $n = 100$. It seems that the larger portfolio dimension k, the worse the learning behavior of the DDQN. For $n = 100$, in particular, it is observed that performance measures decreased rapidly as k increased, becoming no longer competitive with the optimal ones. In the third experiment, this phenomenon of poor generalization is studied more in depth, training DDQN agents on problems of dimension $n \in \{150, 200\}$. Even with small portfolios, the DAC agent seems not to work well, failing to get close to the optimal policy.

The results are shown in terms of *hitting ratio*, which indicates the fre-

quency the agent reaches the expected optimal performance within 0.25% of its standard deviation during the training process. We provide the plots from the paper in Figure 3.1. In the first plot, hitting ratio for dimension $n = 50$ and $n = 100$ for the `evenly_spread` portfolio of various sizes k are shown. In the second plot, it is represented the number of total hittings, i.e. the number of times the agent reaches the expected optimal performance, during the entire training process in various RL runs. The plots show both the dependence on the dimension and on the portfolio size. In particular, it is clear that the RL agent does not generalize well when n increases, not being able to reach the optimal policy a satisfactory number of times when $n = 150$ and $n = 200$.

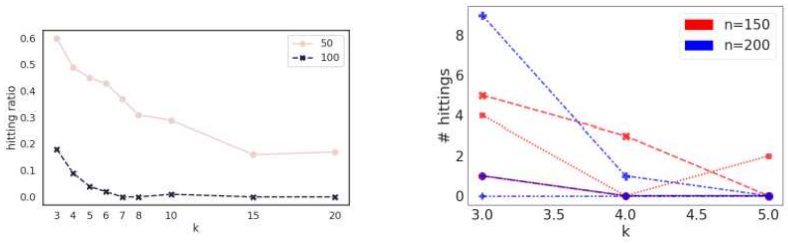

Figure 3.1: Hitting ratio and number of hitting points for the DDQN agent in various dimensions

The conclusion of the paper suggests that the promising results for the lower dimension should lead to the development of a fruitful stream of research on dialogue between parameter control and dynamic algorithm configuration, starting with studies that could help to overcome the problem of poor generalization of the RL agent. In particular, a better understanding of the problem and new parameter control solutions may lead to interesting outcomes.

Chapter 4

Dynamic Policies on Enhanced State Spaces

As we observed from the results in [7], the RL agent still fails to gain good performance in high dimensions. A possible research direction, also suggested by the authors of the paper in the conclusion, is to try to incorporate more information into the RLS policy than just fitness. A study of this approach will be particularly useful also to understand how to treat more general cases; for example, to build a good configuration of DDQN also for more complex algorithms of practical concerns for which theoretical guarantees are not given. Moreover, this research direction seems promising, as a first tentative study was conducted in the paper *Can OneMax help optimizing LeadingOnes using the EA+RL method?* [11]. In this article, the authors suggested that OneMax could help optimize LeadingOnes: one of the strategies adopted is to use a random agent which in each state, i.e. for each LeadingOnes fitness value, selects as action the objective function, choosing LeadingOnes objective with probability q, and OneMax with probability $1 - q$. The selected objective is then used to determine the reward after applying RLS with a standard radius of 1. In the paper, both theoretical and empirical analyses are conducted. The result is promising, since the expected average running time for LeadingOnes using the random agent is $n^2q/(1 + q)$. In particular, selecting $q = 1/2$, which still optimizes LeadingOnes in at least half of the cases, the expected runtime is $n^2/3$, which can improve the expected runtime with radius 1 of $0.5n^2$ by a factor of 1.5.

The main original result of this thesis is, in fact, the answer to the question *Can OneMax help optimizing LeadingOnes?* We studied different theoretical-inspired RLS configurations, modifying both the parameter policy and the selection process, trying to incorporate information from OneMax fitness in solving LeadingOnes. The information from OneMax fitness is taken into account in each round of the algorithm together with LeadingOnes fitness, and not alternatively to it with some probability, as in [11]. Furthermore, inspired by the approach followed to integrate OneMax fitness into RLS solving LeadingOnes, we also studied the most general setting, taking an n-dimensional state space where to each single bit-string x corresponds a parameter configuration.

4.1 Optimal Policy for Lexicographic Selection

4.1.1 Optimal Policy for Lexicographic RLS in Two-dimensional State Space

Incorporating more information from the current search point in RLS iterations brings along a further layer of complexity rather than the cases studied in the previous sections. In fact, if we consider other metrics in addition to LeadingOnes fitness, the progression of the metric is no longer monotonic. When we take into account only the LeadingOnes fitness to determine the policy state, the space $\mathcal{S}^{(1)}$ we consider is a one-dimensional space. Thus, we can naturally define on it a total ordering relation based on LeadingOnes fitness. Furthermore, we are sure that when the algorithm moves to a new state, it moves monotonically, always reaching a state corresponding to higher fitness.

In a two-dimensional space $\mathcal{S}^{(2)}$, this no longer holds, since we no longer have a natural total ordering relation. In particular, if we take into account the fitness of both LeadingOnes and OneMax, this makes an analysis of the exact distribution of the runtime more complex. This is even more difficult if we consider a n-dimensional state space $S^{(n)}$ where at each bit-string x corresponds a different state.

Introducing a two-dimensional state space $S^{(2)}$ based on LeadingOnes and OneMax fitness makes the progression of the algorithm no longer monotone as we intended before: the algorithm, as it is designed, includes the rule that it is always better to move to a higher LeadingOnes fitness state, but does not have any rule regarding OneMax fitness. This because we do not know the answer to questions like: is it better a state, e.g. $(\mathrm{LO}(x) = 3, \mathrm{OM}(x) = 5)$ or $(\mathrm{LO}(x) = 3, \mathrm{OM}(x) = 3)$ to solve LeadingOnes when $n = 7$? It is not obvious that we should make on OneMax fitness the same assumption made on LeadingOnes fitness that it is always preferable to be in a state with a higher value of fitness.

In any case, studying better this situation may be interesting, since it means to intervene not only on the radius policy but also on the selection mechanism. The already good results on LeadingOnes obtained by controlling the optimal policy in [15] suggest that controlling a further element of the algorithm, the selection operator, may be the key to further notable improvement. For this reason, we will first conduct our analysis under the assumption that, when optimizing LeadingOnes, given the same LeadingOnes fitness, a state with higher OneMax fitness is preferable; as we will note along the analysis, this assumption is also motivated from a theoretical point of view and will help us to simplify the analysis of the optimal policy.

The assumption is integrated in the algorithm through the selection operator: in particular, as before, we accept a new solution y, mutated from x, if y improves the LeadingOnes fitness of x, regardless OneMax fitness, but now, when y has the same LeadingOnes fitness of x, we accept the mutation only if y improves the OneMax fitness of x. The theoretical motivation behind this choice can be seen as the fact that introducing this rule also means adding a total ordering relation in the two-dimensional state space, based on a

lexicographic rule on LeadingOnes and OneMax fitness. In particular, if we consider as states the couples $(\mathrm{LO}(x), \mathrm{OM}(x))$ of LeadingOnes and OneMax fitness, the state space is a lattice of $(n+1)(n+2)/2$ points with the following relation.

$$(l_1, m_1) > (l_2, m_2) \text{ if and only if } (l_1 > l_2) \text{ or } (l_1 = l_2 \text{ and } m_1 > m_2)$$

where the first value represents the LeadingOnes fitness, and the second the OneMax fitness.

As one can expect, the lexicographic ordering extremely simplifies the analysis. In particular, in this way we have a projection of the two-dimensional search lattice into a one-dimensional space, with the algorithm moving towards a growing direction of the ordering. We will later empirically verify that the assumption also leads to a performance improvement.

In order to build the optimal policy, we would like to study, as done in the examples from the previous chapter, the transition probabilities in the state space; the search direction guaranteed by the lexicographic ordering makes it easier to study the possible transitions. The idea is to exploit this fact by calculating the transition probabilities backwards, i.e. from a lexicographically higher to a lower state: since the algorithm can only move to a lexicographically higher state, we will have that the only possible transitions will be to states already processed.

We analyze one state at a time: the analysis of the target state (n, n) is trivial, since it will not move to any other state and the expected runtime is 0. We then study the state $(l = n-1, m = n-1)$, which corresponds to the bit-string $1^{n-1}0$ with 1 in every position and a 0 in the last one; in this case it is obvious that the only transition possibilities are to stay in $(n-1, n-1)$ or to move to the optimal state (n, n). Thus, the only possible radius is 1. We therefore define

$$k_{opt}(n-1, n-1) = 1 \tag{4.1}$$

where we use the notation $k_{opt}(l, m)$ to indicate the optimal radius, i.e. the one that minimizes the expected runtime, in the state with $\mathrm{LO}(x) = l$ and $\mathrm{OM}(x) = m$. With $T^{(k)}_{opt}(l, m)$ we will denote the runtime with radius choice k for state (l, m) and optimal radius for the other states. We will denote as $T_{opt}(l, m)$ the optimal runtime $T^{(k_{opt})}_{opt}(l, m)$.

It is obvious, given the radius equal to 1, that the transition probability from $(n-1, n-1)$ to (n, n) is $1/n$, which is the probability of flipping the last bit.

In this case, it is also easy to compute the expected time to the optimum. In fact, for the transition considered — from $(n-1, n-1)$ to (n, n) — the OneMax fitness is not really taken into account since it has to be the same as the LeadingOnes fitness. Consequently, we have a situation similar to the one described in section 3.2 with reference to the paper [15]. In the paper it is shown that the time to a strict LeadingOnes improvement follows a geometric

distribution $Geom(q_i)$, where q_i is the probability of strict improvement: in this case, a strict improvement corresponds to a transition to the target state. The runtime to the optimum from $(n-1, n-1)$ is therefore $T_{opt}(n-1, n-1) \sim Geom(1/n)$ and the expected runtime is

$$\mathbb{E}[T_{opt}(n-1, n-1)] = n \tag{4.2}$$

Since we fixed the optimal values of a point, we can proceed recursively backward to determine the optimal expected runtime and the optimal policy for each state. In particular, we readapted the idea proposed in [10] to find the optimal policy for OneMax that we presented in Section 3.2.

We proceed one state at a time: after the state $(n-1, n-1)$, moving lexicographically backward, we analyze the state $(n-2, n-1)$, which corresponds to having all the ones in the bit-string and a 0 in position $n-1$, or $1^{n-1}01$. In this case, the possibilities are to take the radius $k = 1$ and move directly to the optimal state or to take $k = 2$ and move to the state $(n-1, n-1)$ analyzed before. Defining the optimal radius policy corresponds to determine which of these two radius options guarantees a lowest runtime starting from $(n-2, n-1)$. To do this, we calculate the expected runtime in the two cases, namely $\mathbb{E}[T^{(1)}_{opt}(l, m)]$ and $\mathbb{E}[T^{(2)}_{opt}(l, m)]$ in the notation introduced before.

For radius choice $k = 1$, the transition probabilities are similar to the case $(n-1, n-1)$, because the only possible transition is to the target space (n, n), which happens with a probability of $1/n$. The time to the optimum is again distributed as a geometric random variable with parameter $1/n$, therefore the expected time to the optimum is n.

For $k = 2$, the only feasible transition is to the state $(n-1, n-1)$ when we flip the last two bits, which happens with probability $1/(n, 2)$. The expected runtime from $(n-1, n-1)$ to the optimum is then n. Therefore, it is clear, even without a precise analysis, that the expected runtime $\mathbb{E}[T^{(2)}_{opt}(l, m)]$ is greater than n and that $k_{opt}(n-2, n-1) = 1$.

We can formalize and generalize the argument made for this case, by explicitly writing down the runtime expectation from a certain starting state (l, m), $\mathbb{E}[T^{(k)}_{opt}(l, m)]$.

Proposition 38. *Let (l, m) be the state corresponding to $LO(x) = l$ and $OM(x) = m$. Suppose that we have already determined for every (l', m') such that $(l', m') > (l, m)$ with lexicographic ordering the optimal radius choice $k_{opt}(l', m')$, i.e. the one which minimizes the expected runtime. Then, given a radius choice k, the expected runtime to the optimum from (l, m) choosing a radius k for (l, m) and the optimal radius for the other states is*

$$\begin{aligned}\mathbb{E}[T^{(k)}_{opt}(l,m)] &= 1+ \\ &+ \mathbb{P}(LO(y)=l, OM(y)=m | LO(x)=l, OM(x)=m, y \leftarrow variate(x)) \cdot \mathbb{E}[T^{(k)}_{opt}(l,m)] + \\ &+ \sum_{\substack{\lambda=l \\ (\lambda,\mu)\neq(l,m)}}^{n-1} \sum_{\mu=\lambda}^{n-1} \mathbb{P}(LO(y)=\lambda, OM(y)=\mu | LO(x)=l, OM(x)=m, y \leftarrow variate(x)) \cdot \\ &\cdot \mathbb{E}[T_{opt}(\lambda,\mu)]\end{aligned} \tag{4.3}$$

For $variate(x)$, we intend that y is the effect of a complete iteration of the RLS algorithm from x, with mutation and then lexicographic selection. y is equal to x every time the selection is not accepted. From now on, we will use the notation $\mathbb{P}((\lambda,\mu)|(l,m)) := \mathbb{P}(\mathrm{LO}(y)=\lambda, \mathrm{OM}(y)=\mu | \mathrm{LO}(x)=l, \mathrm{OM}(x)=m, y \leftarrow \mathrm{variate}(x))$.

Proof. The formula is a direct consequence of the law of total probabilities, with the initial $+1$ which takes into account the evaluation of the target state (n,n). □

As we can see from the formula (4.3), when we calculate the expected runtime $\mathbb{E}[T^{(k)}_{opt}(l,m)]$ from a given starting state (l,m), the term $\mathbb{E}[T^{(k)}_{opt}(l,m)]$ appears both on the right- and on the left-hand side; we can therefore reformulate as follows (4.3), explicitly writing the unknown term.

$$\mathbb{E}[T^{(k)}_{opt}(l,m)] = \frac{1 + \sum_{\lambda=l+1}^{n-1} \sum_{\mu=\lambda}^{n-1} \mathbb{P}((\lambda,\mu)|(l,m)) \cdot \mathbb{E}[T_{opt}(\lambda,\mu)]}{1 + \mathbb{P}((l,m)|(l,m))} \tag{4.4}$$

It consists simply in taking the right-hand side of (4.3) with the sum starting from $\lambda = l+1$ and $\mu = \max\{m+1, \lambda\}$ and dividing it by $1 + \mathbb{P}((l,m)|(l,m))$. Note that $\mathbb{P}((l,m)|(l,m))$ is the probability that the mutation is not accepted, which happens when $y \leftarrow mutate(x)$ is such that $\mathrm{LO}(y) < \mathrm{LO}(x)$ or $\mathrm{LO}(y) = \mathrm{LO}(x)$ and $\mathrm{OM}(y) < \mathrm{OM}(x)$.

From Proposition 38, we can then find the optimal expected runtime for RLS to solve LeadingOnes by starting from a random bit-string.

Theorem 39. *Assume that $x^{(0)}$ is an initial random bit-string of length n selected uniformly at random among the bit-strings of length n. The optimal runtime of RLS to solve the LeadingOnes problem of dimension n starting from $x^{(0)}$ is given by the formula*

$$\mathbb{E}[T_{opt}] = 1 + \sum_{l=0}^{n-1} \sum_{m=l}^{n-1} \mathbb{P}(LO(x^{(0)})=l, OM(x^{(0)})=m) \mathbb{E}[T_{opt}(l,m)] \tag{4.5}$$

Also in this case the proof is a trivial application of the law of total probabilities.

Note that $\mathbb{P}(\mathrm{LO}(x^{(0)}) = l, \mathrm{OM}(x^{(0)}) = m)$ is the probability of the starting state to be (l, m), which is not uniform but must take in account the fact that certain states are more probable since the number of bit-strings in the state are more. For example, if $n = 4$, $\mathbb{P}(x^{(0)} = (3, 3)) = 1/16$, while $\mathbb{P}(x^{(0)} = (1, 2)) = 1/8$.

Given the formula (4.3), to determine the expected runtime $\mathbb{E}[T^{(k)}_{opt}(l, m)]$ from the state (l, m) we can therefore derive the optimal policy as

$$k_{opt}(l, m) = \operatorname*{argmin}_{k \in [n-l]} \mathbb{E}[T^{(k)}_{opt}(l, m)] \tag{4.6}$$

It is important to note that this straightforward approach leads, by definition, to the optimal policy. In fact, if we define as optimal policy the one which minimizes the expected runtime, this approach finds it directly. The efficacy of this approach was shown also in [10] for the optimal radius policy for RLS on OneMax. In the paper, it is shown that maximizing the drift is indeed not the optimal approach by noting the differences with the *runtime minimizer* we also adopted.

Algorithmically, the optimal radius policy is determined by the procedure described in Algorithm 10. The idea is to fix the state (l, m), apply the formula (4.3) computing the expected runtime for each value of $k \in [n - l]$ and then to take the k which leads to the lowest runtime as optimal.

Algorithm 10 Determination of the optimal lexicographic policy

Input: Dimension n
$K[n-1, n-1] \leftarrow 1$
$T[n-1, n-1] \leftarrow n$
for $l = 1, \ldots, n-2$ **do**
  for $m = l+1, \ldots, n-2$ **do**
    for $\lambda = l, \ldots, n-1$ **do**
      for $\mu = m, \ldots, n-1$ **do** calculate $\mathbb{P}^{(k)}((\lambda, \mu)|(l, m))$
      end for
      $T^{(k)}[l, m] \leftarrow (1 + \sum_{\lambda=l+1}^{n-1} \sum_{\mu=m+1}^{n-1} \mathbb{P}^{(k)}((\lambda, \mu)|(l, m)))/(1 + \mathbb{P}^{(k)}((l, m)|(l, m)))$
      $K[l, m] \leftarrow \operatorname{argmin}_k T^{(k)}(l, m)$
      $T[l, m] \leftarrow \min_k T^{(k)}(l, m)$
    end for
  end for
end for

Note that the output of the algorithm are two matrices: K and T, whose (l, m)-th entry represents, respectively, the optimal radius and the optimal expected runtime of and from the state (l, m). One of the advantages of using a two-dimensional state space is indeed the possibility to use matrices as data structures, which makes the computation more efficient and gives also the possibility to visualize the policy used and the expected runtime, as we will show in the following sections.

Algorithm 10 still does not specify how to compute the transition probabilities $\mathbb{P}((\lambda,\mu)|(l,m)))$. The question is far from being trivial and is indeed one of the biggest challenges in this method: one may hope to derive distributional information on the state space that helps to determine the transition probabilities. This is indeed the approach that was followed in some of the results stated before, in particular to determine the transition probabilities for OneMax in [10] it is simply observed that moving to a different OneMax fitness state consists of flipping a certain number of ones and zeros. In particular, it is simple to observe that

$$P(\mathrm{OM}(y) = m_2 \mid \mathrm{OM}(x) = m_1) = \sum_{i=\max(0,m_1-m_2)}^{\min(m_1,k)} \frac{\binom{m_1}{i} \cdot \binom{n-m_1}{k-i}}{\binom{n}{k}} \tag{4.7}$$

However, LeadingOnes introduces a position dependence that makes the computation of these probabilities much more difficult. Indeed, in previous work, the transition probability from a LeadingOnes state to another is never computed directly: the only probability computed is the probability of strict improvement in [15], which takes into account only the probability of flipping the $(\mathrm{LO}(x)+1)$-th bit. The fact that the distribution of ones in the tail remains unknown makes it impractical to determine a formula for the transition probabilities. If we consider the one-dimensional state space based only on LeadingOnes fitness, the problems are given by cases like, e.g., $n = 4, k = 1, \mathrm{LO}(x) = 1$. In the example, the information on LeadingOnes fitness does not allow us to discriminate between $x = 1010$ or $x = 1001$ for which the transition probability to $y = 1110$ is, respectively, $1/4$ or 0. It is easy to see from the same example that the situation does not change by introducing also information about OneMax fitness. Therefore, in our two-dimensional state space, it is still impractical to find a formula to compute the transition probabilities.

The approach implemented to determine the transition probabilities thus consists in directly counting the possible cases and divide it by the total cases. The procedure for doing it is summarized in Algorithm 11.

Algorithm 11 Computation of transition probabilities

1: **Input:** Starting state (l,m); arrival state (λ,μ); dimension n; radius k
2: $\mathbb{P}^{(k)}((\lambda,\mu)|(l,m)) \leftarrow 0$
3: **for** $x \in (l,m)$ **do**
4: **for** $y \in (\lambda,\mu)$ **do**
5: **if** $|x-y| = k$ **then** $\mathbb{P}^{(k)}((\lambda,\mu)|(l,m)) \leftarrow \mathbb{P}^{(k)}((\lambda,\mu)|(l,m)) + 1/(n,k)$
6: **end if**
7: **end for**
8: **end for**

The idea of the procedure is to *collapse* different bit-cases x into the same state.

However, as we can see, it is computationally demanding to calculate the transition probabilities in this way, since the algorithm needs to look at every possible starting and arrival state, and for each state at very corresponding bit-string. It is easy to see that the number of operations required is exponential, since only the number of possible starting bit-strings is 2^n. Considering also the possible arrival states, we have $1 + 2 + \cdots + 2^n = 2^{n-1} \cdot (2^n + 1)$. If we consider that this operation must be repeated for every possible k, we obtain a number of operations that is $\mathcal{O}(n^2 4^n)$.

On the other hand, it does not seem promising to follow a solution similar to the one adopted for the one-dimensional state space in [15], which models directly the distribution of the runtime based on the probability of strict improvement of LeadingOnes fitness from a starting state. In fact, taking into account only the probability of strict improvement does not take advantage of OneMax fitness because it means looking only at the probability to flip the $(\mathrm{LO}(x) + 1)$-th bit, while OneMax fitness gives information on the number of ones in positions from $(\mathrm{LO}(x) + 2)$ to n. This will therefore result in the same policy of the one-dimensional state space.

One could try to look at the probability of strict lexicographic improvement. However, even in this case, trying simply to maximize this probability does not seem promising to obtain an optimal policy. Similarly to what observed in [15], we can state that the time for improvement is distributed as a geometric distribution with parameter $q_{i,j}$, where i and j are the LeadingOnes and OneMax fitness, respectively. In the lexicographic two-dimensional space, the probability of strict improvement from state (i, j) with radius r is

$$q(r,i,j) = \begin{cases} \frac{r}{n} \prod_{t=1}^{r-1} \frac{n-i-t}{n-t}, & \text{if } \frac{r}{2} > n-j-1 \\ \frac{r}{n} \prod_{t=1}^{r-1} \frac{n-i-t}{n-t} + \prod_{t=1}^{r} \frac{n-i-t}{n-t+1} \left(\sum_{k=\lceil r/2 \rceil}^{n-j-1} \frac{\binom{n-j-1}{k}\binom{j-1}{r-k}}{\binom{n-i-1}{r}} \right), & \text{otherwise} \end{cases}$$

However, the function $q(r, i, j)$ has no immediate concavity properties as $q(r, i)$, the probability of improvement when considering only LeadingOnes fitness. We tried to plot some values of $q(r, i, j)$ fixing (i, j) to understand the behavior of the function. The plots are in Figure 4.1. Fixed n, i and j, we represented in blue the function $q(r, i, j)$ with respect of the radius choice r, and in green the function $\frac{r}{n} \prod_{t=1}^{r-1} \frac{n-i-t}{n-t}$. The red line represents the value $r = 2(n - j - 1)$. Obviously, the chosen radius would be r that maximizes $q(r, i, j)$.

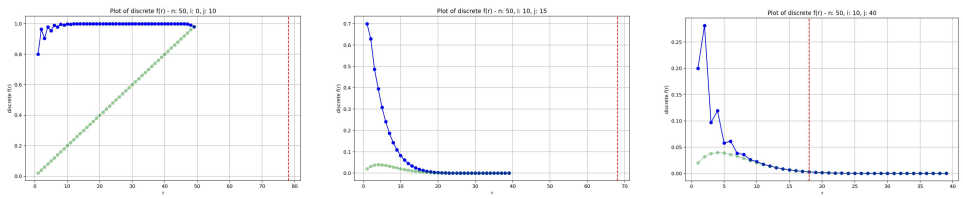

Figure 4.1: Plots of $q(r, i, j)$ with respect to r.

All simulations present a similar behavior. As we can also see from the figure, in this approach, the policy obtained maximizing $q(r, i, j)$ with respect to r is too conservative and is not really able to adapt to the current state. The main problem of this approach is that a strict improvement in the lexicographic sense it does not always represent a notable gain, e.g. it could be a transition from 0^n to $0^{n-1}1$, and it does not exploit the real advantage of the information on OneMax, which is the possibility of aiming for bigger updates in terms of LeadingOnes fitness if we have a higher number of ones in the tail, e.g. if we are in $(0, n-1)$ we could aim for the optimum (n, n) rather than a strict lexicographic improvement which could consist in going to $(1, 0)$.

Therefore, we decided to focus on a deeper analysis of the policy developed in (4.6).

4.1.2 Optimal Policy for Lexicographic RLS in n-dimensional State Space

The challenges concerning the computation of transition probabilities explained in the previous section suggested us to extend the state space to dimension n, i.e. to consider each individual bit-string as a distinct state. In this way, we can overcome the problem of the computation of transition probabilities since we no longer have the collapse of different bit-strings into the same state. In particular, given a radius k, the transition probabilities are now easily computed as

$$\mathbb{P}(y \mid x, y \leftarrow x) = \begin{cases} \frac{1}{\binom{n}{k}}, & \text{if } d_H(x, y) = k \\ 0, & \text{otherwise} \end{cases} \tag{4.8}$$

where we indicate with $d_H(x, y)$ the *Hamming distance* of x from y, which, in our case, corresponds to the number of different bits in the two strings.

The idea of using an n-dimensional state space is not only driven by the need to simplify the computation of transition probabilities but is deeply rooted in the core motivations of our research. Our analysis aims to understand the performance gains achieved by leveraging additional state information, therefore constructing a policy in the full n-dimensional state space pushes this exploration to its theoretical limit, as it utilizes all available information from the current bit-string. Consequently, the results obtained serve as an essential benchmark for dynamic parameter policies in enhanced state spaces.

To build the optimal policy for the general state space, we note that we can readapt the argument from Section 4.1.1.

We now use the notation $k_{opt}(x)$ to indicate the optimal radius for the bit-string x. We will analogously denote as $T_{opt}(x)$ the optimal runtime, starting from the bit-string x. With $T_{opt}^{(k)}(x)$ we refer to the runtime with parameter choice radius k for state x and optimal radius for the other states.

The formulas (4.3) and (4.6) for the optimal policy can be readapted in the n-dimensional state space case in a natural way, analogously to what was done before.

$$\mathbb{E}[T^{(k)}_{opt}(x)] = 1 + \mathbb{P}(x|x) \cdot \mathbb{E}[T^{(k)}_{opt}(x)] + \\ + \sum_{y:y>x} \mathbb{P}(y|x, y \leftarrow \text{variate}(x)) \cdot \mathbb{E}[T_{opt}(y)] \tag{4.9}$$

Note that in this case, the probabilities can be directly computed by looking at x and y and applying (4.8). Note also that the sum is only on the states y, with $y \geq x$, where the relation is intended with respect to the lexicographical ordering. In fact, given the lexicographic selection, the only possible transitions are from a state y lexicographically greater than x.

In order to simplify the computation of the expected execution time (4.9), we decided to accept new candidate solutions y, if and only if y brings a strict improvement to x, i.e. $y > x$ lexicographically. This little change makes the computation far easier because it excludes a transition between two states lexicographically equivalent, i.e. with same LeadingOnes and OneMax fitness. Therefore, the formula (4.9) includes only one unknown expectation, $\mathbb{E}[T^{(k)}_{opt}(x)]$ and makes it possible a explicit solution. This choice is also coherent with the lexicographic assumption, which suggest that the best moving direction for the algorithm is to states with higher LeadingOnes or same LeadingOnes and higher OneMax fitness.

The optimal policy for every single bit-string is then derived as

$$k_{opt}(x) = \underset{k \in [n-\text{LO}(x)]}{\text{argmin}} \mathbb{E}[T^{(k)}_{opt}(x)] \tag{4.10}$$

And the expected total time,

$$\mathbb{E}[T_{opt}] = 1 + \sum_{x^{(0)} \text{ bit-string}} \mathbb{P}(x^{(0)}) \mathbb{E}[T_{opt}(x^{(0)})] \tag{4.11}$$

where in this case the sum is over all possible bit strings and the probability of initial bit-string $\mathbb{P}(x^{(0)})$ is uniform, i.e. $\mathbb{P}(x^{(0)}) = 1/2^n$ for every possible value of $x^{(0)}$.

The arguments and the implementation insights can then be easily adapted from the previous section. We note that in this case the computation of the probabilities no longer requires to count every case by enumeration, but the derivation of the optimal policy is still quite computationally costly. In fact, to find k_{opt} the algorithm must try all possible values of k, which are $n - \text{LO}(x)$, for all 2^n states. Thus the cost still remains exponential in the dimension. Moreover, there is no longer the possibility to exploit in the implementation a matrix data structure.

The policy proposed in this section is particularly important, since it deals with the most general setting where all the information from the bit-string is exploited to build the optimal policy. Therefore the one provided is the best and most general policy possible for RLS on LeadingOnes in the lexicographic setting, and thus provides us with a very important benchmark for this notable problem.

4.2 Optimal Policy for Standard Selection

Even if we widely motivated the lexicographic assumption, it is indeed interesting to study the optimal policy for the algorithm in the classical setting, without modifying the selection mechanism. In this way, it is possible to isolate the effect of parameter selection policy in a well-studied setting. The obtained results will also be more easily interpretable and comparable with the previous studied from literature. For this reason, we will strictly adhere to the setting studied in the literature, where a new solution y generated from x is accepted if and only if $\mathrm{LO}(y) \geq \mathrm{LO}(x)$.

In this setting, we will focus on the two-dimensional state space: the computation is more demanding with the standard selection and is therefore beneficial to limit the number of states. Moreover, the results for lexicographic selection that we present in Section 5.1.1 highlight that taking information from the whole bit-string does not significantly improve the expected runtime with respect to when we consider LeadingOnes and OneMax fitness. Lastly, policies on the two-dimensional space are easier to interpret since they can be visualized as matrices.

We will therefore build a near-to-optimal policy for standard selection in the two-dimensional state space. In particular, we will follow the argument in the section above, adapting it to the new setting, where we accept a mutation y from x if and only if $\mathrm{LO}(y) \geq \mathrm{LO}(x)$.

The argument is similar to what was done before; we start considering the two-dimensional search space. As before, the expected runtime is

$$\begin{aligned}\mathbb{E}[T^{(k)}_{opt}(l,m)] = 1 + \mathbb{P}((l,m) \mid (l,m)) \cdot \mathbb{E}[T^{(k)}_{opt}(l,m)] + \\ + \sum_{\substack{\lambda=l,\\ (\lambda,\mu)\neq(l,m)}}^{n-1} \sum_{\mu=\lambda}^{n-1} \mathbb{P}((\lambda,\mu) \mid (l,m)) \cdot \mathbb{E}[T_{opt}(\lambda,\mu)]\end{aligned} \tag{4.12}$$

The only difference in the formula from (4.3) is in the starting index of the sum over the OneMax fitness values. In particular, we no longer have the guarantee that states reachable from (l,m) are the ones lexicographically bigger; thus we do not have the guarantee that they are the ones already processed and for which we already know $\mathbb{E}[T_{opt}(\lambda,\mu)]$. In particular, if we proceed backward as before, we have already processed all nodes with strictly higher LeadingOnes fitness but, for every value of LeadingOnes fitness l, we have as unknown $\mathbb{E}[T^{(k)}_{opt}(l,m)]$ for $m = l,\dots,n-1$. The result is that equation (4.12) leads to a linear system of $n-l+1$ equations in $n-l+1$ unknown. To write it in standard form $Ax = b$, we move to the left-hand side the terms involving the states with LeadingOnes fitness of l, whose expected runtime is unknown.

$$\begin{aligned}\mathbb{E}[T^{(k)}_{opt}(l,m)] &= \sum_{\mu=l}^{n-1} \mathbb{P}((l,\mu)|(l,m))\mathbb{E}[T_{opt}(l,\mu)] = \\ &= 1 + \sum_{\lambda=l+1}^{n-1}\sum_{\mu=\lambda}^{n-1} \mathbb{P}((\lambda,\mu)|(l,m))\mathbb{E}[T_{opt}(\lambda,\mu)], \quad \forall m = l,\dots,n-1\end{aligned} \tag{4.13}$$

Or equivalently

$$\begin{aligned}&(1-\mathbb{P}^{(k)}((l,m) \mid (l,m)))\mathbb{E}[T^{(k)}_{opt}(l,m)] - \sum_{\substack{\mu=l \\ \mu\neq m}}^{n-1} \mathbb{P}((l,\mu)|(l,m)) \cdot \mathbb{E}[T_{opt}(l,\mu)] = \\ &= 1 + \sum_{\lambda=l+1}^{n-1}\sum_{\mu=\lambda}^{n-1} \mathbb{P}((\lambda,\mu)|(l,m))\mathbb{E}[T_{opt}(\lambda,\mu)], \quad \forall m = l,\dots,n-1\end{aligned} \tag{4.14}$$

We can write the linear system in matrix form $Ax = b$ as follows.

$$x = \begin{bmatrix} \mathbb{E}[T^{(k)}_{opt}(l,l)] \\ \mathbb{E}[T^{(k)}_{opt}(l,l+1)] \\ \vdots \\ \mathbb{E}[T^{(k)}_{opt}(l,n-1)] \end{bmatrix} \quad b = \begin{bmatrix} 1 + \sum_{\lambda=l+1}^{n-1}\sum_{\mu=\lambda}^{n-1} \mathbb{P}((\lambda,\mu)|(l,l))\mathbb{E}[T_{opt}(\lambda,\mu)] \\ 1 + \sum_{\lambda=l+1}^{n-1}\sum_{\mu=\lambda}^{n-1} \mathbb{P}((\lambda,\mu)|(l,l+1))\mathbb{E}[T_{opt}(\lambda,\mu)] \\ \vdots \\ 1 + \sum_{\lambda=l+1}^{n-1}\sum_{\mu=\lambda}^{n-1} \mathbb{P}((\lambda,\mu)|(l,n-1))\mathbb{E}[T_{opt}(\lambda,\mu)] \end{bmatrix}$$

$$A = \begin{bmatrix} (1-\mathbb{P}^{(k)}((l,l) \mid (l,l))) & \mathbb{P}^{(k)}((l,l+1) \mid (l,l)) & \cdots & \mathbb{P}^{(k)}((l,n-1) \mid (l,l)) \\ \mathbb{P}^{(k)}((l,l) \mid (l,l+1)) & (1-\mathbb{P}^{(k)}((l,l+1) \mid (l,l+1))) & \cdots & \mathbb{P}^{(k)}((l,n-1) \mid (l,l+1)) \\ \vdots & \vdots & \ddots & \vdots \\ \mathbb{P}^{(k)}((l,l) \mid (l,n-1)) & \mathbb{P}^{(k)}((l,l+1) \mid (l,n-1)) & \cdots & (1-\mathbb{P}^{(k)}((l,n-1) \mid (l,n-1))) \end{bmatrix}$$

To address this system, we examine its solvability. In particular, we would like to prove that the matrix A is non-singular, in that case it means that it is invertible and thus $x = A^{-1}b$, i.e. the solution always exist unique.

We note that for each value of LeadingOnes fitness $l < n$ and for each value of OneMax fitness μ and m, the probability $\mathbb{P}^{(k)}((\lambda,\mu) \mid (l,m))$ is strictly greater than 0 for at least one value of $\lambda > l$. This means that $\sum_{\mu=l}^{n-1} \mathbb{P}^{(k)}((l,\mu) \mid (l,m)) < 1$ for each $m = l,\dots,n-1$. In particular, it follows that

$$\sum_{\substack{\mu=l \\ \mu\neq m}}^{n-1} \mathbb{P}^{(k)}((l,\mu) \mid (l,m)) < 1 - \mathbb{P}^{(k)}((l,m) \mid (l,m)), \quad \forall m = l,\dots,n-1 \tag{4.15}$$

This means that the matrix A is diagonally dominant and thus non-singular.

We can therefore solve the system for each k and thus find the value of k that minimizes the expected runtime for each state.

We must note that in this case, the procedure to obtain the optimal policy

requires one to compute the expected runtime for all possible combinations of k, which means that, in solving the system above, we should choose a vector k with a different radius value for each OneMax fitness and then choose the best combination. However, this would require us to try an exponential number of combinations, namely $(n-l)^{n-l-1}$. So, instead of doing that, we decided to develop an approximation by fixing the same value of k for all states (l,m) with $m = l, \dots, n-1$ with the same LeadingOnes fitness l and use it to compute the corresponding $\mathbb{E}[T^{(k)}_{opt}(l,m)]$ for every $k = n-l, \dots, n$. We then choose $k_{opt} = \text{argmin}_k \, \mathbb{E}[T^{(k)}_{opt}(l,m)]$. The policy we propose for this setting is therefore a *quasi-optimal* policy, build upond the standard selection. In the following section, we propose another solution when the algorithm does not have access to OneMax fitness during selection.

4.2.1 Strict Standard Selection

As emerges before, the near-optimal policy is computationally quite demanding: empirical results highlighted that it is not possible to compute the policy for more than very low dimensions. For this reason, we also propose a slight variance of the setting, keeping the requirement of using only LeadingOnes fitness in selection but also simplifying the computation. In this setting, that we will call *strict standard*, we accept a new candidate y from x if and only if $\text{LO}(y) > \text{LO}(x)$. The difference from the discussion in previous section is that we no longer accept a mutation if $\text{LO}(y) = \text{LO}(x)$: this choice, even if is not popular in literature, should not heavily affect the behavior of the algorithm. In fact, the idea of accepting movements to bit-strings with the same LeadingOnes fitness is mainly motivated by the preference over a wider exploration of the search space rather than possibility to move nearer to the optimum. Therefore, we can assume that the optimization process does not change significantly in this setting while hugely simplifying the policy computation. We also note that a similar assumption was made in the lexicographic setting to simplify the notation when we decide not to accept transitions from y to x with same LeadingOnes and OneMax fitness.

Under the strict standard selection, the computation is far easier and we can follow the argument from Section 4.1.1. In particular, formula (4.3) still holds with different transition probabilities. In particular, $\mathbb{P}((l,m) \mid (l,m))$ corresponds only to the probability of not accepting a transition and $\mathbb{P}((l,\mu) \mid (l,m))$ is 0 for every $\mu \neq m$. However, these probabilities can still be computed with the direct counting method we used before.

The same case can be adapted to build a policy for standard strict selection in the n-dimensional space but we decided to focus on the two-dimensional space and leave analysis of the n-dimensional space for further research.

Chapter 5

Empirical Results on Enhanced State Spaces

In this chapter, we report the exact and simulated results of the policy proposed in the previous chapter for the different settings described.

Implementation details

To produce the results presented in this chapter, we implemented the procedures we described and will discuss in Python, making wide use of the libraries `numpy` and `matplotlib`. The calculations carried out were quite heavy, so we adopted some strategies to optimize our code. To optimize the computation of fitness values for the LeadingOnes and OneMax functions, we employed Python's `lru_cache` decorator. This caching mechanism stores previously computed results, significantly reducing redundant calculations when the same input is evaluated multiple times. Specifically, for the LeadingOnes function, the cumulative product of the input vector is cached to efficiently determine the longest prefix of ones. Similarly, for the OneMax function, the sum of the input vector is cached to avoid recalculation. This optimization is particularly beneficial in iterative processes where these fitness functions are frequently queried.

To further enhance computational efficiency, we use Python's `multiprocessing` library to parallelize computationally intensive tasks, such as evaluating fitness values and updating state probabilities. The design of the algorithm decomposes iterations on fitness levels, mutation operations, and distances into independent tasks that are distributed among multiple processes. Specifically, the function that calculates expected runtimes for each state is parallelized by mapping its iterations across multiple worker processes.

However, in general, the computation of the exact runtime scaled exponentially in the problem dimension n.

We run our simulations on the MCMeSU cluster of Sorbonne Université in the standard computing server, which provided 32 cores (AMD EPYC Milan) and 256GB DDR4 memory. All codes were run for 24 hours.

The code is provided for reproducibility at the present repository.

5.1 Lexicographic Selection

5.1.1 Low-dimension

We start the discussion of empirical results by looking at a simple case that explicitly represents the graph structure for $n = 4$. We represent the graph in Figure 5.1 considering as nodes the states in the two-dimensional space. Note that for low n ($n < 8$), we do not have differences in our policy developed between for the two-dimensional state space and the n-dimensional. In the figure, we represent the states (l, m) as nodes; we also indicate inside the nodes the optimal k in square brackets and the expected optimal runtime. As edges we represented the possible transitions with the associated probability reported.

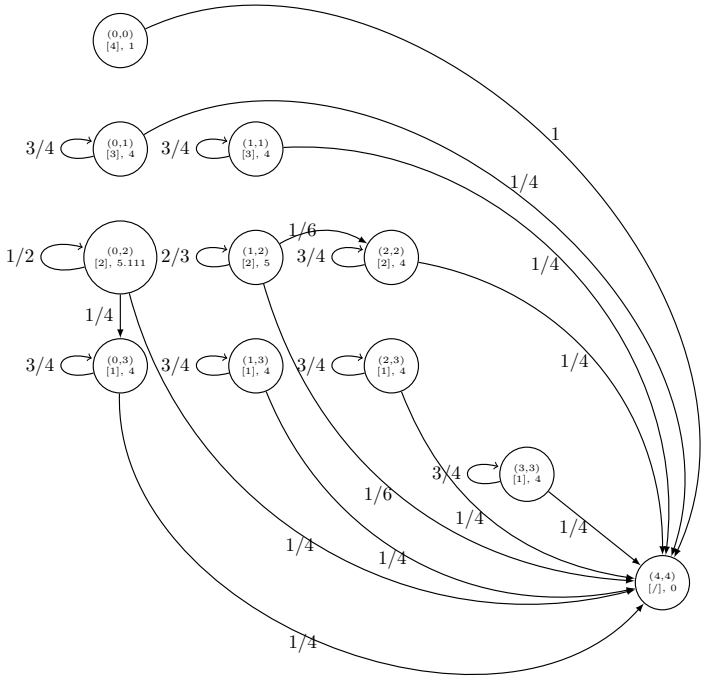

Figure 5.1: Graph structure for $n = 4$.

The graph gives us some interesting insights on how the algorithm moves in the search space under the lexicographic hypothesis. Even if the dimension is low, it is already possible to appreciate some differences between states with the same LeadingOnes fitness but different OneMax. In general, it seems to be very advantageous for a states to have a direct transition to the target. This is the case for *external* states, since the expected time is not greater than n. All states for $n = 4$, with the exception of $(0, 2)$ and $(1, 2)$ have an optimal expected runtime of n (or lower in $(0, 0)$). Therefore, these two cases are the

most interesting to analyze: in both states, the optimal radius is 2, which means that the algorithm wants strongly to preserve the ones in the string, and therefore, even if the LeadingOnes fitness is low, it does not flip too many bits. It is also interesting to note how the incorporation of OneMax information influences the optimal policy: a clear example is the states with LeadingOnes fitness equal to 0. If OneMax fitness is also equal to 0, the optimal strategy is obviously to flip n bits, while if it is equal to $n-1$, it makes sense to flip fewer bits, even only 1.

We already observed that there is no difference in the optimal policy with full information (n-dimensional state space). From our computations the first differences highlighted by our computation begin for dimension $n = 8$, where we have that the string 01^40^3 is associated with an optimal k of 5 and 0^41^4 is associated with an optimal k of 4, even if both strings have LeadingOnes fitness of 0 and OneMax of 4. The optimal k associated with the state $(0, 4)$ in the 2-dimensional state space is 5. Other differences are, e.g., when $n = 10$, a radius of $k = 6$ is selected for 0^61^4, $k = 8$ for 0101^30^3, and $k = 7$ for $(0, 4)$ in two-dimensional state space.

It seems that the algorithm in general could have a slightly more conservative behavior when the ones are at the end of the tail. However, this different behavior may be limited to some specific cases and not be relevant overall.

To validate our considerations and study the algorithm behavior, we calculated exactly the results for the lexicographic selection using the formula (4.5) and the analogous for state space of dimension 1, 2 and n. We stated in Table 5.1 the expected runtime of LeadingOnes in the growing dimension n, using the optimal policies for three different state spaces. For results in column LO we used the optimal policy obtained considering information on OneMax fitness available only for selection and not for mutation adapting formula (4.6).

Dim (n)	LO	(LO, OM)	x
2	1.500	1.250	1.250
3	3.125	2.375	2.375
4	5.500	4.375	4.375
5	7.857	6.491	6.491
6	11.511	8.946	8.946
7	14.197	11.549	11.549
8	18.748	14.368	14.365
9	22.031	17.248	17.248
10	27.234	20.289	20.287
11	30.337	23.393	23.393
12	37.156	26.63	26.629
13	40.306	29.914	29.914
14	46.758	33.282	33.279
15	50.941	36.747	36.743
16	61.910	41.237	40.236

Table 5.1: Expected time for lexicographic selection

The results are very interesting and deserve some comments. We first notice that in the lexicographic context we have a notable gain by also exploiting information from OneMax fitness in mutation. The improvement is indeed dependent on the policy state and not only on the lexicographic selection: adding more information in the parameter policy seems fruitful in this case. However, adding the information of the whole string x seems to improve the algorithm only by a factor that is not greater than 10^{-3}. This phenomenon confirms what we observed before: the differences in the optimal policies are limited in cases and values, especially for low dimensions. However, we do not expect big changes for higher n and therefore preferred to analyze in detail the optimal policy for the two-dimensional state space. This is also motivated by the fact that we can take advantage of the two-dimensional space to represent optimal policies as heatmaps. Some of them are provided in Figure 5.2.

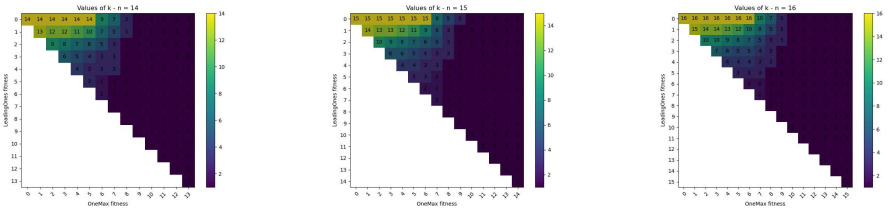

Figure 5.2: Heatmaps of optimal policies for lexicographic selection

First of all, we visually note the strong gain we have by adding the OneMax dimension. In particular, if $n = 16$ we have that for LeadingOnes fitness of 0, the algorithm will always choose a radius of 15 when policy information consists only in LeadingOnes fitness, while adding information about OneMax makes the policy far more nuanced, taking into account the strong differences from, e.g., 0^n and 01^{n-1}. In particular, it is clear that from 0^n we could jump instantly to the target, while for 01^{n-1}, the best choice is $k = 1$ and wait to flip the first bit.

In general, it is easy to see that for higher LeadingOnes fitness it is normal for the algorithm to be more conservative and flip only 1 bit: in these cases, the one-dimensional policy models the optimal behavior well. However, the main advantages of incorporating more information in our policy are given in the earliest stages, with a lower LeadingOnes fitness, where the uncertainty in the distribution of bits in the tail of the string is higher and the algorithm takes advantage of a more risky behavior in lower OneMax situations and more conservative when the OneMax fitness is higher.

We can also note some patterns in the heatmaps: first of all, we note that there is a large region in the right-bottom, that is, with both high LeadingOnes and OneMax fitness, for which the optimal policy is equal to 1. This is quite intuitive, since, with lexicographic selection the algorithm tries to move in the two-dimensional space to the lower-right, this means that in that region we are closer to the optimum, and thus the algorithm trade-off between conserving the ones found so far and flipping the remaining zeros tends to the conserva-

tion once the algorithm has reached a higher fitness, either LeadingOnes or OneMax.

Another pattern is monotonicity; we observe in the optimal policy:

- LeadingOnes monotonicity: growing LeadingOnes fitness, the optimal radius becomes smaller;
- OneMax monotonicity: growing OneMax fitness, the optimal radius becomes smaller;
- Lexicographic monotonicity: for lexicographically higher states, the corresponding radius is smaller, i.e. the algorithms always takes a non-increasing radius in the optimization process.

It is obvious that the lexicographic monotonicity is a natural consequence of the other two. The others reflect the monotonicity in the optimal policies we have for RLS on fitness-based policy for LeadingOnes and OneMax, shown in [15] and [20], respectively. The idea is quite intuitive, since it represents that the algorithm becomes more conservative while we are nearer to the fitness.

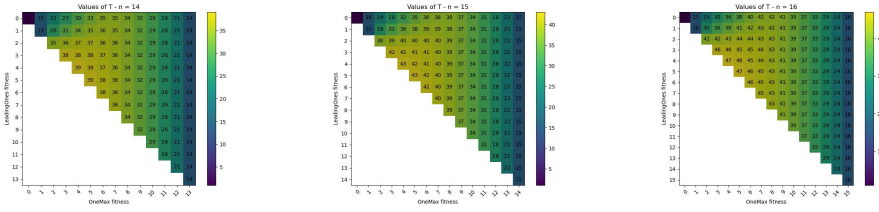

Figure 5.3: Heatmaps of optimal expected runtime for lexicographic selection

We can also make some considerations on the expected runtime heatmaps. We note that the higher times are obtained for states with either LeadingOnes and OneMax fitness not too high. In these states, the algorithm cannot take advantage of the risky behavior of lexicographical small states, neither of the good positioning of states near the solution. In general, this suggests what would be the good strategy for the algorithm that would be trying to move away from those states with low fitness.

It is also interesting to note that for a sufficiently large OneMax fitness, it seems that the LeadingOnes fitness no longer influences neither the optimal policy, which becomes always equal to 1, nor the expected runtime. This further confirms that our algorithm design is efficient and motivated in particular in the first stages of the optimization process, when we can expect the fitness to be lower.

5.1.2 High-dimension

As we see in Table 5.1, the exact analysis of the expected runtime is possible only for quite low dimensions since, as we observed when presenting the algorithm used, the number of operations is exponential in the dimension n. To overcome

the problem of having results only for low dimensions, we can conduct a Monte Carlo simulation, running for a certain number N of times the RLS algorithm with a chosen policy, and count for each iteration the number of iterations. The empirical mean and variance obtained from the number of evaluations for each simulation are consistent estimators of the real mean and variance of the runtime. For more details, refer to the Appendix D. The empirical estimation of the standard deviation gives us information that is not considered in the exact computation of the expected runtime. It is indeed relevant for the algorithm, since it gives us a measure of how much the runtime of the algorithm can change. In particular, a reinforcement learning agent such as the one used in [7] can benefit strongly from a lower standard deviation: it is thus useful to see how integrating more information in the policy design can also affect this measure.

The problem of this approach is that to conduct a simulation we need to know before the policy we wish to apply. This is actually possible for the one-dimensional state space, for which we can take the policy from the formula expressed in [7] that corresponds to the state-of-the-art policy of RLS on LeadingOnes. We would like to compare the results with a two-dimensional policy. Since, from results in low-dimension, we observed that the n-dimensional state space seems not to provide great performance improvement and the information of policies on it are limited, we will focus only on the two-dimensional state space.

Therefore, we developed an *heuristic* policy defined on the two-dimensional state space. Since there seemed to be some quite evident patterns in the two-dimensional optimal policy calculated exactly for low dimensions, we tried to reproduce these patterns and tested the goodness of the heuristic policy on the expected runtime for the low dimensions for which we calculated the exact policy.

To build the heuristic policy, we observed that for LeadingOnes fitness of 0 and low OneMax fitness, the optimal policy was always to flip n bits. We then noticed that the state $(n-1, n-1)$ always has, in the cases we computed, an optimal radius of $k_e = n-1$. Lastly, we noted that states with LeadingOnes fitness of $n-1$ and high OneMax fitness had rapidly the value of 1 as optimal radius. We then filled the policy matrix with some uniformity using the monotonicity hypothesis.

Formally, we build the policy matrix K_e as follows:

$$K_e(n,i) := \begin{cases} n & \text{for } i = \lfloor \frac{n-1}{2} \rfloor, \dots, n, \\ \text{uniformly spaced from } n-1 \text{ to } 1 & \text{for } i = \lfloor \frac{n-1}{2} \rfloor, \dots, n - \lfloor \frac{n+1}{3} \rfloor, \\ 1 & \text{for } i > n - \lfloor \frac{n+1}{3} \rfloor, \end{cases}$$

$$K_e(n-1,j) := \begin{cases} n-1 & \text{for } j = n-1, \\ \text{uniformly spaced from } n-1 \text{ to } 1 & \text{for } j = n - \lfloor \frac{n+1}{3} \rfloor, \dots, n-1, \\ 1 & \text{for all other } j, \end{cases}$$

$K_e(l,i)$:=uniformly spaced from values$[l-2]$ to 1 for $l = 2,\ldots,n - \lfloor\frac{n+1}{3}\rfloor - 2$

and $i = 0,\ldots,n - \lfloor\frac{n+1}{3}\rfloor$

where:

$$\text{values} := \text{a linearly spaced sequence from } n-2 \text{ to } 2,$$

and the starting value for each row l in the range $2 \leq l \leq n - \lfloor\frac{n+1}{3}\rfloor - 2$ is determined by values$[l-2]$, i.e. the $l-2$-th element of the vector *values*.

It is more intuitive to show some heatmaps of the policy developed, compared also with the optimal policies we computed exactly. We provided some of them in Figure 5.4.

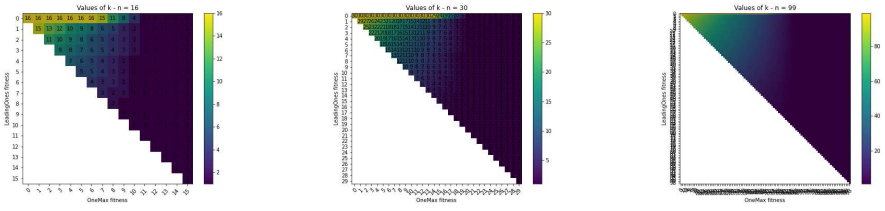

Figure 5.4: Heatmaps of heuristic policy

First of all, we compared the expected runtime calculated exactly for the heuristic policy on low dimensions with the results we already computed and summarized it in Table 5.2.

Dim (n)	LO	(LO, OM)	x	heuristic
2	1.5	1.25	1.25	1.25
3	3.125	2.375	2.375	2.375
4	5.5	4.375	4.375	4.594
5	7.857	6.491	6.491	6.668
6	11.511	8.946	8.946	9.125
7	14.197	11.549	11.549	11.79
8	18.748	14.368	14.365	14.535
9	22.031	17.248	17.248	17.582
10	27.234	20.289	20.287	20.634
11	30.337	23.393	23.393	23.635
12	37.156	26.63	26.629	27.037

Table 5.2: Expected time for lexicographic selection with the heuristic policy

As we see from the table, the heuristic policy seems to work very well, guaranteeing values similar to the ones of the optimal exact policy with two-dimensional state and far better than the one-dimensional results. In particular,

the expected runtime seems to proceed in a higher dimension with a similar order of magnitude and thus can be used as a good approximation also for higher dimensions.

We used therefore the heuristic policy to approximate the algorithm behavior in high dimensions, comparing it with the fitness-based policy. Selected results of the expected runtime along with the corresponding standard deviations are stated in Table 5.3. The complete results are reported in Appendix E.

n	LO	Heuristic	Improvement
10	27.929 ± 15.160	20.369 ± 11.564	27.08%
20	83.052 ± 35.401	56.817 ± 24.935	31.58%
50	365.279 ± 113.261	189.796 ± 70.485	48.06%
70	686.021 ± 201.404	322.379 ± 107.782	52.99%
99	1484.031 ± 357.983	616.595 ± 161.838	58.47%

Table 5.3: Simulated results for lexicographic selection and large problem dimensions

In the table, we simulated the results up to dimension $n = 99$ due to computational limits. The results deserve several considerations. The first one concerns the expected runtime: as we can see, exploiting more information in the policy, as is done by the heuristic policy, leads to a notable advantage in the expected runtime. In the third column, we report the percentage advantage of using the heuristic instead of the optimal fitness-based policy. We see that the relative advantage is not only consistently high - and higher than the standard deviation - but also grows growing the dimension.

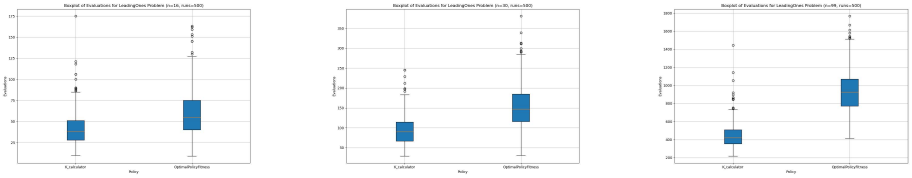

Figure 5.5: Boxplots for simulations of lexicographic RLS

We note, also observing the boxplots reported in Figure 5.5, the standard deviation is consistently lower when using the heuristic policy than when using the fitness-based one. However, this effect is not observed in the relative standard deviation (the standard deviation over the value of the empirical mean) reported in columns four and five of the Table E.1. In this case, the values are similar, with the relative value for the one-dimensional policy being consistently smaller. However, we notice that the relative standard deviation decreases for both policies as n grows. In general it seems that for similar values of expected runtime the corresponding standard deviation is similar: obviously this is noted by looking at different dimensions in order to take in account similar expected runtime values for the two policies.

In Table 5.4, we computed the ratio of the expected runtime obtained with the fitness-based and heuristic policy as in Table 5.3 divided by n^2 and $n \log n$, since, as we saw in previous chapters, we knew that the unary unbiased complexity of RLS on LeadingOnes is $\Theta(n^2)$ and on OneMax is $\Theta(n \log n)$. From Table 5.4, it is clear that our solution is much faster than the state-of-the-art for the RLS on the LeadingOnes problem. In particular, it is interesting to note that for both policies the value of this ratio becomes monotonically smaller as n grows: this means that the algorithm is faster than $\Theta(n^2)$, thus the unary unbiased complexity for RLS on LeadingOnes no longer applies with the lexicographic selection. However, it is interesting to note the normalization by $n \log n$. In this case, the ratio increases, growing n for the fitness-based policy, but remains almost stable for the heuristic. Therefore, the fitness-based policy is slower than $\Theta(n \log n)$, the unary unbiased complexity of OneMax, but the heuristic seems comparable with this bound. It is indeed interesting to note that we have a significant improvement in the runtime dependent on the policy selected, even in terms of asymptotic rate.

n	LO	Heuristic	$\frac{\text{LO}}{n^2}$	$\frac{\text{Heuristic}}{n^2}$	$\frac{\text{LO}}{n \log n}$	$\frac{\text{Heuristic}}{n \log n}$
10	27.929	20.369	0.2793	0.2037	4.0301	2.9413
20	83.052	56.817	0.2076	0.1420	5.9931	4.1037
50	365.279	189.796	0.1461	0.0759	7.5483	3.9244
70	686.021	322.379	0.1400	0.0658	7.8452	3.6870
99	1484.031	616.595	0.1514	0.0630	8.3445	3.4677

Table 5.4: Simulated results for lexicographic selection and large problem dimensions with normalized columns

A possible objection that follows the observation that the heuristic policy has a runtime comparable with the OneMax complexity is that the algorithm with lexicographic selection simply tries to maximize OneMax regardless of LeadingOnes fitness. However, we can easily disprove this objection by simulating the algorithm using the radius from the optimal policy for OneMax solving LeadingOnes with lexicographic selection. The results are stated in Table 5.5.

We see from the results that even for small dimensions, the OneMax optimal policy does not lead to an execution runtime near to our heuristic policy. Even if it is effectively faster than the policy based solely on LeadingOnes fitness, it is closer to it than how close it is to the policy that considers both LeadingOnes and OneMax for mutation.

Moreover, it is clear from looking at the optimal two-dimensional policies that it also differentiates states with the same OneMax fitness but different LeadingOnes. Therefore, we can conclude that in the lexicographic setting, the algorithm works faster than the standard setting regardless of the state space chosen and that our heuristic policy, which is based on a two-dimensional state space, improves both the optimal policy based solely on LeadingOnes fitness and the one based only on OneMax fitness.

However, the fact that the unary unbiased complexity bounds for Leadin-

Dim (n)	LO	H.	OM
2	1.521 ± 1.519	1.735 ± 1.653	1.285 ± 1.46
3	3.079 ± 2.849	2.397 ± 2.565	2.443 ± 2.554
4	5.424 ± 4.370	4.435 ± 4.157	4.744 ± 3.947
5	8.054 ± 5.631	6.477 ± 4.810	6.543 ± 5.311
6	11.027 ± 7.388	9.137 ± 6.276	9.228 ± 7.615
7	13.751 ± 8.729	11.402 ± 7.302	13.107 ± 11.509
8	18.476 ± 11.109	14.413 ± 9.253	16.781 ± 18.901
9	21.882 ± 12.419	18.415 ± 10.473	19.915 ± 18.067
10	27.929 ± 15.160	20.369 ± 11.564	30.152 ± 69.686
11	30.574 ± 14.944	23.619 ± 12.446	27.066 ± 29.701
12	36.457 ± 18.309	27.685 ± 14.463	31.036 ± 29.386
13	39.183 ± 18.929	31.704 ± 16.737	37.151 ± 36.638

Table 5.5: Simulation for lexicographic selection with OM optimal policy

gOnes no longer hold makes our result less aligned with the current literature on LeadingOnes. So, it is also interesting to study the results in the standard selection setting, which will allow us to better understand what influences and under which conditions the algorithm behavior.

5.2 Standard Selection

In this section, we will present results from the standard selections setting. To focus on comparison with previous literature results, we decided to analyze the algorithm behavior without strict selection and use the *near-optimal* policy we developed in Section 4.2. We present at the end of the section an overview of the results in the strict standard setting but we will leave a thorough analysis for future research.

5.2.1 Low-dimension

We already observed in the previous chapter that an approximation of the optimal policy can be done for the two-dimensional state space, but it is computationally very costly since it requires one to solve a linear system of $n-l-1$ equations for each LeadingOnes fitness l. The exact computation is therefore possible only for very low dimensions; in this case we managed to do it only for $n \leq 7$. We report the results in Table 5.6.

First of all, we note that, as expected, the expected runtime are much larger than the ones obtained with lexicographic selection. So, the first conclusion we draw is that changing the selection mechanism has a strong influence on the expected runtime and a first improvement of the algorithm is on the selection level.

Furthermore, even from these few results it seems that incorporating also OneMax information in the mutation policy improves the expected runtime of the algorithm by a certain percentage advantage. An improvement was ex-

Dim (n)	LO	(LO, OM)	% Improvement	$\frac{\text{LO}}{n^2}$	$\frac{\text{(LO, OM)}}{n^2}$
2	1.5	1.125	25.0%	0.3750	0.2813
3	3.5	2.375	32.14%	0.3889	0.2639
4	6.0	4.448	25.87%	0.3750	0.2780
5	9.667	7.463	22.79%	0.3867	0.2985
6	13.667	11.488	15.96%	0.3796	0.3191
7	18.875	16.435	12.94%	0.3837	0.3354

Table 5.6: Expected time for standard selection

pected, since a two-dimensional state space differentiates cases that effectively require very different strategies, such as states $(0, 0)$ and $(0, n-1)$. However, it is difficult to estimate the impact of additional information only from these low-dimension values and extend conclusions on higher dimensions. As we saw, in addition, the normalization of the two-dimensional policy by n^2 provided in column 6 of Table 5.6 may grow as n becomes bigger, with decreasing improvements of the two-dimensional policy. This is plausible considering the fact that, growing n, the region where it is optimal to flip only 1 bit becomes larger, and thus the information given by the OneMax fitness becomes less relevant.

We can delve deeper in the policy adopted to better understand the behavior of the algorithm. We provide some heatmaps in Figure 5.6

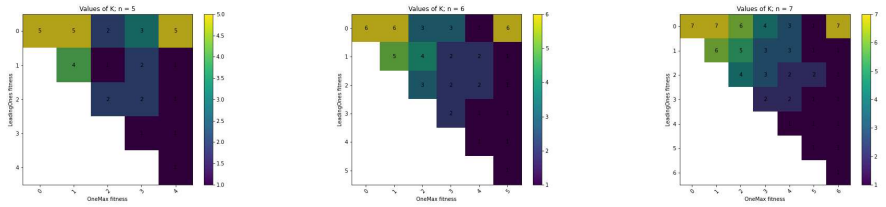

Figure 5.6: Heatmaps of two-dimensional policies for standard selection

The first thing we note is that in the standard selection framework we have less clearer patterns. In particular, we no longer have monotonic policies, neither in LeadingOnes, nor in OneMax fitness. We then have some unexpected patterns, such as the optimal radius of n for states of $(0, n-1)$ and not clear values in the *central* states. However, in general, some of the behavior seems similar, with a prevalence of ones on the lower right of the policy and of high values (such as n) on the upper left. In general, the possibility for the algorithm to lower its OneMax fitness to move to a faster state makes it more difficult to model.

We can also study the expected runtime heatmaps to have ideas of possible good strategies of the algorithm. We provide some of them in Figure 5.7.

In this case, the patterns seem similar to the lexicographic case, with a generally high execution runtime for states with medium values of LeadingOnes and OneMax fitness. This suggests us that an optimal strategy will try to move away as possible from this region and instead move on the border states,

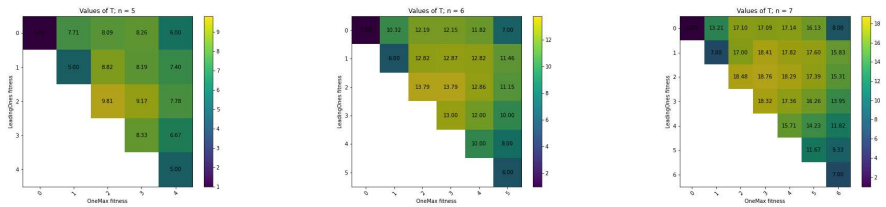

Figure 5.7: Heatmaps of expected runtime for standard selection and two-dimensional policy

i.e. the ones with or very low LeadingOnes or very high OneMax.

5.2.2 High-dimension

Similarly to the lexicographic setting, we tried to obtain results for high dimensions through Monte Carlo simulations with a predefined policy. For the one-dimensional state space, as before, we already knew the optimal policy, which is the one in the paper [7] and we applied it. For the 2-dimensional state space we tried to apply the heuristic developed in the previous section. Selected results are stated in Table 5.7, while the complete table can be found in Table E.2 in Appendix E.

n	LO	Heuristic	Improvement
10	38.878 ± 23.354	34.238 ± 19.830	13.57%
20	159.806 ± 63.532	151.438 ± 61.158	5.53%
50	970.620 ± 254.967	960.984 ± 234.036	1.00%
73	2080.514 ± 437.115	2077.114 ± 462.841	0.16%

Table 5.7: Simulated standard results

We note that the results of the heuristic are not satisfactory: it guarantees in general better results than the one-dimensional optimal policy, since it is generally faster than the state-of-the-art. However, its improvement is not statistically relevant since it is too small compared to the standard deviation. The development of a better heuristic policy could be a research direction to improve the state-of-the-art on the standard problem; however, as we saw, the patterns of the exact policies we developed in Figure 5.6 do not give clear insights. However, computation of a higher dimension using more computational power may be helpful to gain more insight into the problem.

We may conclude, at the end of this section, that we developed three levels of improvement on the state-of-the-art of the problem: one related to mutation, one to selection, and one to mutation and selection combined. The first was not statistically relevant for high dimensions and still needs further study to verify whether asymptotically we can effectively expect an improvement. However, the change in selection and the two-dimensional policy under the lexicographic assumption provide a clear and relevant improvement in the state-of-the-art,

suggesting that when selection and mutation work together, exploiting both more information, the improvement is notable. This insight could be extremely useful for studying control policies of other, more practical algorithms on real-world problems.

5.2.3 Strict Standard Selection

For the sake of completeness, we present in this section results on the strict standard setting, which provide some further interesting insights.

The strict selection assumption allows to significantly simplify the computation and therefore to compute results for dimensions $n > 7$ and, in particular, for dimensions up to $n = 16$ bridging the gap with the lexicographic setting. The results obtained are presented in Table 5.8, compared with the expected runtime for fitness-based policy obtained from literature without strict selection ($0.39n^2$).

Dim (n)	LO	(LO, OM)	Improvement
2	1.560	1.250	19.87%
3	3.510	2.375	32.34%
4	6.240	4.375	29.89%
5	9.750	7.630	21.69%
6	14.040	10.780	23.21%
7	19.110	14.783	22.61%
8	24.960	19.629	21.41%
9	31.590	25.530	19.20%
10	39.000	31.737	18.61%
11	47.190	38.575	18.24%
12	56.160	46.423	17.31%
13	65.910	55.265	16.15%
14	76.440	64.194	16.00%
15	87.750	74.006	15.66%
16	99.840	84.919	14.96%

Table 5.8: Expected time for strict standard selection

The results are very interesting since they suggest a consistent improvement in exploitation of further information: the improvement seems to be in particular in low dimensions and to reduce while n grows bigger. Even with low data this phenomenon seems plausible since with growing n the percentage of states for which the optimal radius is 1 becomes bigger and therefore the influence of exploiting more information reduces. It is not possible, however, to establish whether this improvement becomes 0 for $n \to \infty$ or if the policy developed still guarantees improvement asymptotically.

We also present some of the obtained policies in Figure 5.8.

It is interesting to note that we find very similar patterns we had in the lexicographic setting, with also monotonicity of the policy. For this reason, we did not develop a new heuristic for this setting and decided not to focus on

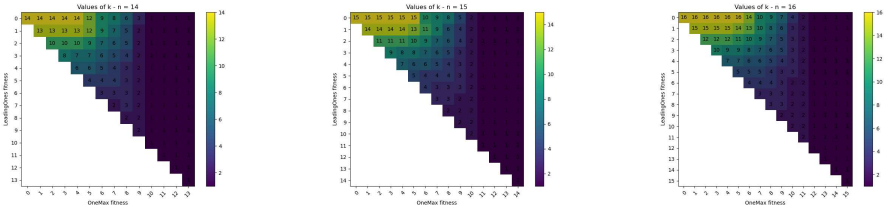

Figure 5.8: Heatmaps of optimal policy for strict standard selection

further simulations.

The strict selection mechanist, however, highlighted an interesting behavior and we aim to delve deeper in the questions raised in this section in future research.

5.2.4 Limited Portfolio

In this last section, we provide the results of computation of the expected runtime for limited portfolio of search radii. We choose to provide computational results for the setting we focus on the most, which is the lexicographic with two-dimensional state space. Taking inspiration from the paper [7] in which limited portfolios are used to train the RL-agent, we computed the exact runtime for the following portfolios.

- `powers_of_two`: $\{2^i \mid 2^i \leq n\}$;
- `initial_segment` with 3 elements: $[1..3]$;
- `evenly_spread` with 3 elements: $\{i \cdot \lfloor n/3 \rfloor + 1 \mid i \in [0..2]\}$.

Selected results are in Table 5.9.

Dim (n)	LO	(LO, OM)	`powers_of_two`	`initial_segment`	`evenly_spread`
2	1.5	1.25	1.25	1.25	1.75
3	3.125	2.375	2.75	2.375	2.375
4	5.5	4.375	4.625	4.687	4.687
5	7.857	6.491	6.87	7.087	7.087
6	11.511	8.946	9.537	9.684	9.261
7	14.197	11.549	12.205	12.471	12.037
8	18.748	14.368	14.574	15.306	14.906
9	22.031	17.248	17.589	18.318	17.693
10	27.234	20.289	20.683	21.413	20.81
11	30.337	23.393	23.908	24.6	24.028
12	37.156	26.63	27.203	27.903	27.1
13	40.306	29.914	30.58	31.247	30.482
14	46.758	33.282	34.024	34.694	33.938
15	50.941	36.747	37.53	38.214	37.329
16	58.558	40.237	40.469	41.772	40.9

Table 5.9: Expected time for lexicographic selection with different portfolios

In the limited portfolio setting, the computation is made easier by the fact that we have less values of k to try to find the optimal. However, this operation was performed in parallel in our code; therefore, we do not have a notable improvement and we managed to compute, also in this case, the results for dimensions up to $n = 16$. In these low dimensions, however, the results regarding expected runtime are not particularly interesting since they are all very similar to the two-dimensional optimal policy runtime. It is more interesting, therefore, to visualize through heatmaps some of the optimal policies for limited portfolios. Selected policies are provided in Figure 5.9.

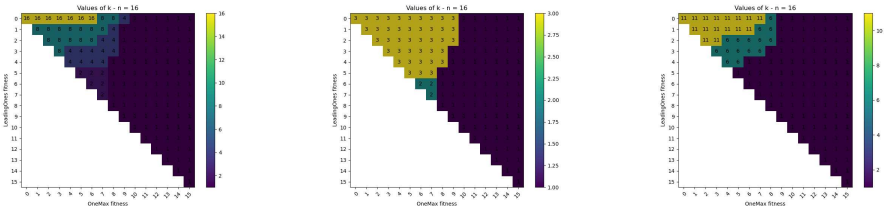

Figure 5.9: Heatmaps of optimal policy for lexicographic selection and limited portfolio

We can note that the optimal policy exploits the available portfolio the most if it has a wider range of possible values: in fact, the `initial segment` policy seems the one that exploits the least the portfolio, using almost only the 1 and 3. In general, it confirms the hypothesis made in [7], where it is said that having some larger search radii is more beneficial than covering exclusively small search radii. However, it seems better to have more than one single small search radius. Further studies on the limited portfolio setting could build upon advancements in the development of optimal dynamic policies.

Chapter 6

Conclusion

Black-box optimization plays an important role in many real-life applications; the design and benchmarking of efficient algorithms is a fruitful research direction with many open problems, since a more theoretical approach to the field is not yet developed enough [79].

In this thesis, we addressed the general problem of parameter control in black-box optimization algorithms, following the idea that a good parameter policy could strongly affect algorithms performance and that theory-based benchmarking could inspire design choices of new algorithms. In the first part of the thesis, we provided a wide review of the existing literature: in Chapter 2 we presented motivations around black-box optimization in general and genetic algorithms in particular. We then introduced the concept of black-box complexity, as a way to model the best runtime for black-box algorithms on classes of problems, showing how different choices of algorithm class can lead to different complexity concepts. We then delved into drift analysis, a set of tools to provide runtime bounds on black-box algorithms, and we presented the general problem of *algorithm configuration*. In Chapter 3, we then discussed the application of these techniques on two important benchmark algorithms, RLS and $(1+1)$ EA, solving two equally important benchmark problems, LeadingOnes and OneMax. We presented complexity results and how the algorithms could benefit from parameter control policies. In this chapter, we also discussed the results from [7] where for the first time an RL agent was used to control the radius parameter in RLS on LeadingOnes.

In the paper, the RL agent showed some generalization limits that suggested the need for a better understanding of the best parameter policy in the problem discussed. Our contributions mainly concerned this point. We aimed to enhance parameter control policies in enhanced state spaces that included more state information. In particular, for the first time, we studied the behavior of RLS on LeadingOnes considering state spaces that also included information on OneMax fitness (two-dimensional state space) and information on the whole bit-string (n-dimensional state space), inspired by some promising results in [11].

We presented our contributions in Chapter 4, where we focused on two settings: first we considered a lexicographic selection, where the algorithms accept

a new solution if it improves LeadingOnes fitness or if it maintains LeadingOnes fitness but improves OneMax, and then we took into account the standard selection based solely on LeadingOnes fitness. In the lexicographic setting, we developed exact optimal policies on the two-dimensional state space and on the n-dimensional space. Since the computation of the optimal policy was computationally heavy, we also developed a *heuristic* policy that we tested in higher dimensions. In the standard setting, we developed an approximated policy only for the two-dimensional state space. We tested all our policies on LeadingOnes and compared them with the state-of-the-art, developed in [15], which was based on a one-dimensional state space and only took into account LeadingOnes fitness.

In Chapter 5, we presented empirical results. In the lexicographic setting, our policies outperform the state-of-the-art. We noted improvement on two levels: first, adding the lexicographic selection but maintaining a one-dimensional state space policy, then a further improvement — both in terms of expected runtime and standard deviation — is gained through a two-dimensional state space policy. However, the n-dimensional policy did not seem to improve significantly the two-dimensional one. In the standard setting, on the other hand, considering also information on OneMax fitness seemed to lead to a minor improvement and it is not clear if this improvement can be statistically significative asymptotically.

Our results highlighted the importance of further research on the problem: the study of enhanced state spaces for RLS policy on LeadingOnes has just begun and this thesis highlighted several questions. First, a further study of the n-dimensional state space policy is required, both in the lexicographic and in the standard setting. Moreover, a heuristic policy that captures more patterns of the optimal policy could further improve the results obtained, in particular in the standard setting where the unclear behavior of our approximated policy did not allow us to develop a specific heuristic. Furthermore, other state spaces can be studied, for example delving deeper in the limited portfolio setting.

Given our motivation raised by [7], our results can be used to benchmark the behavior of an RL agent for the dynamic algorithm configuration of RLS on LeadingOnes, testing it on the enhanced state spaces we considered and comparing them with our benchmark.

This initial exploration of enhanced state spaces in parameter control opens the door for further research on various algorithms with greater practical relevance. For instance, future work could focus on refining dynamic algorithm configuration policies for the RLS variant of the $(1 + (\lambda, \lambda))$ genetic algorithm [13] and extending these techniques to CMA-ES, a sophisticated genetic algorithm widely used in real-world applications. Investigating different state spaces has the potential to significantly advance the field of dynamic algorithm configuration [1], a promising research direction that leverages reinforcement learning to address parameter control challenges.

Beyond evolutionary algorithms, the impact of improved parameter control extends to the broader field of Automated Machine Learning (AutoML), where building learning-inspired control policies is still a major challenge. AutoML aims to automate the design and tuning of machine learning models, reducing

the need for human expertise and extensive trial-and-error experimentation. Developing adaptive parameter control methods that adjust hyperparameters dynamically, without requiring exhaustive pre-tuning, could make AutoML systems more efficient, scalable, and accessible. This is particularly relevant in deep learning, where hyperparameters such as learning rates, batch sizes, and regularization terms can significantly influence model performance. By integrating dynamic algorithm configuration into AutoML frameworks, future research could help bridge the gap between theoretical optimization techniques and real-world machine learning applications, making model training more adaptive and resource-efficient.

These advancements have the potential to make optimization techniques more applicable across diverse domains, from evolutionary computation to deep learning and beyond. As parameter control methods continue to evolve, they could play a crucial role in enhancing automated decision-making systems, reducing computational costs, and improving the generalization capabilities of machine learning models.

Bibliography

[1] Adriaensen, Steven et al. *Automated Dynamic Algorithm Configuration*. 2022.

[2] Afshani, Peyman et al. The Query Complexity of Finding a Hidden Permutation. In: *Space-Efficient Data Structures, Streams, and Algorithms: Papers in Honor of J. Ian Munro on the Occasion of His 66th Birthday*. Springer Berlin Heidelberg, 2013, pp. 1–11.

[3] Audet, Charles and Hare, Warren. *Derivative-Free and Blackbox Optimization*. Springer Cham, 2017.

[4] Barlas, T. et al. Surrogate-based aeroelastic design optimization of tip extensions on a modern 10 MW wind turbine. *Wind Energy Science*. 2021.

[5] Battiti, Roberto and Campigotto, Paolo. An Investigation of Reinforcement Learning for Reactive Search Optimization. *Autonomous Search*. 2012.

[6] Biedenkapp, André et al. Dynamic Algorithm Configuration: Foundation of a New Meta-Algorithmic Framework. In: *European Conference on Artificial Intelligence*. 2020.

[7] Biedenkapp, André et al. Theory-inspired parameter control benchmarks for dynamic algorithm configuration. In: *Proceedings of the Genetic and Evolutionary Computation Conference* GECCO '22. 2022.

[8] Böttcher, Süntje, Doerr, Benjamin, and Neumann, Frank. Optimal fixed and adaptive mutation rates for the leadingones problem. In: *Proceedings of the 11th International Conference on Parallel Problem Solving from Nature: Part I* PPSN'10. 2010.

[9] Burke, Edmund K., Newall, James P., and Weare, Rupert F. Initialization strategies and diversity in evolutionary timetabling. *Evol. Comput.* 1998.

[10] Buskulic, Nathan and Doerr, Carola. Maximizing Drift Is Not Optimal for Solving OneMax. *Evolutionary Computation*. 2021.

[11] Buzdalov, Maxim and Buzdalova, Arina. Can OneMax help optimizing LeadingOnes using the EA+RL method? In: *2015 IEEE Congress on Evolutionary Computation (CEC)*. 2015.

[12] Cantor, David Geoffrey and Mills, W. H. Determination of a Subset from Certain Combinatorial Properties. *Canadian Journal of Mathematics*. 1966.

[13] Chen, Deyao et al. Using Automated Algorithm Configuration for Parameter Control. In: *Proceedings of the 17th ACM/SIGEVO Conference on Foundations of Genetic Algorithms* FOGA '23. 2023.

[14] De Jong, Kenneth Alan. An analysis of the behavior of a class of genetic adaptive systems. AAI7609381. PhD thesis. USA, 1975.

[15] Doerr, Benjamin. Analyzing randomized search heuristics via stochastic domination. *Theoretical Computer Science*. 2019.

[16] Doerr, Benjamin and Doerr, Carola. Black-box complexity: from complexity theory to playing mastermind. In: *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation* GECCO Comp '14. 2014.

[17] Doerr, Benjamin and Doerr, Carola. Playing Mastermind With Constant-Size Memory. *Theory of Computing Systems*. 2011.

[18] Doerr, Benjamin and Doerr, Carola. The impact of random initialization on the runtime of randomized search heuristics. In: *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation* GECCO '14. 2014.

[19] Doerr, Benjamin and Doerr, Carola. Theory of Parameter Control for Discrete Black-Box Optimization: Provable Performance Gains Through Dynamic Parameter Choices. In: *Theory of Evolutionary Computation: Recent Developments in Discrete Optimization*. Springer International Publishing, 2020, pp. 271–321.

[20] Doerr, Benjamin, Doerr, Carola, and Yang, Jing. Optimal parameter choices via precise black-box analysis. *Theoretical Computer Science*. 2020.

[21] Doerr, Benjamin and Goldberg, Leslie. Adaptive Drift Analysis. *Algorithmica*. 2011.

[22] Doerr, Benjamin, Johannsen, Daniel, and Winzen, Carola. Multiplicative Drift Analysis. *Algorithmica*. 2012.

[23] Doerr, Benjamin et al. A method to derive fixed budget results from expected optimisation times. In: *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation* GECCO '13. 2013.

[24] Doerr, Benjamin et al. *Black-Box Complexities of Combinatorial Problems*. 2011.

[25] Doerr, Benjamin et al. Faster black-box algorithms through higher arity operators. In: *Proc. of FOGA (Foundations of genetic algorithms)*. 2011.

[26] Doerr, Benjamin et al. How Well Does the Metropolis Algorithm Cope With Local Optima? In: *Proceedings of the Genetic and Evolutionary Computation Conference* GECCO '23. 2023.

[27] Doerr, Carola. *Complexity Theory for Discrete Black-Box Optimization Heuristics*. 2018.

[28] Doerr, Carola and Lengler, Johannes. Introducing Elitist Black-Box Models: When Does Elitist Behavior Weaken the Performance of Evolutionary Algorithms? *Evolutionary Computation*. 2017.

[29] Doerr, Carola and Lengler, Johannes. OneMax in Black-Box Models with Several Restrictions. In: *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation* GECCO '15. 2015.

[30] Droste, Stefan, Jansen, Thomas, and Wegener, Ingo. Upper and Lower Bounds for Randomized Search Heuristics in Black-Box Optimization. *Theory of Computing Systems*. 2006.

[31] Eiben, A., Hinterding, Robert, and Michalewicz, Zbigniew. Parameter control in evolutionary algorithms. *IEEE Trans. Evolutionary Computation*. 1999.

[32] Erdos, Paul and Rényi, Alfréd. *On two problems of information theory*. 1963.

[33] Forrester, Alexander I.J. and Keane, Andy J. Recent advances in surrogate-based optimization. *Progress in Aerospace Sciences*. 2009.

[34] Freitas, Alex A. *Data Mining and Knowledge Discovery with Evolutionary Algorithms*. Berlin, Heidelberg: Springer-Verlag, 2002.

[35] Garnier, Josselin, Kallel, Leila, and Schoenauer, Marc. Rigorous Hitting Times for Binary Mutations. *Evolutionary Computation*. 1999.

[36] Geijtenbeek, Thomas, Panne, Michiel van de, and Stappen, A. Frank van der. Flexible muscle-based locomotion for bipedal creatures. *ACM Trans. Graph.* 2013.

[37] Gendreau, Michel and Potvin, Jean-Yves. *Handbook of Metaheuristics*. Vol. 146. 2010.

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

[39] Hajek, Bruce. Hitting-Time and Occupation-Time Bounds Implied by Drift Analysis with Applications. *Advances in Applied Probability*. 1982.

[40] Hallak, Assaf, Castro, Dotan, and Mannor, Shie. Contextual Markov Decision Processes. 2015.

[41] Hamming, R. W. Error detecting and error correcting codes. *The Bell System Technical Journal*. 1950.

[42] Hansen, Nikolaus. *The CMA Evolution Strategy: A Tutorial*. 2023.

[43] Hasselt, Hado van, Guez, Arthur, and Silver, David. Deep reinforcement learning with double Q-Learning. In: *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence* AAAI'16. 2016.

[44] He, Jun and Yao, Xin. A study of drift analysis for estimating computation time of evolutionary algorithms. *Natural Computing*. 2004.

[45] He, Jun and Yao, Xin. Drift analysis and average time complexity of evolutionary algorithms. *Artificial Intelligence*. 2001.

[46] Herdy, Michael. Evolution strategies with subjective selection. In: *Parallel Problem Solving from Nature — PPSN IV*. Ed. by Hans-Michael Voigt et al. 1996.

[47] Holland, John H. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. The MIT Press, 1992.

[48] Hutter, Frank, Hoos, Holger H., and Leyton-Brown, Kevin. Automated Configuration of Mixed Integer Programming Solvers. In: *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*. Ed. by Andrea Lodi, Michela Milano, and Paolo Toth. 2010.

[49] Hwang, Hsien-Kuei et al. Probabilistic Analysis of the (1+1)-Evolutionary Algorithm. *Evolutionary Computation*. 2018.

[50] Jansen, Thomas and Zarges, Christine. Performance analysis of randomised search heuristics operating with a fixed budget. *Theoretical Computer Science*. 2014.

[51] Johannsen, Daniel. Random combinatorial structures and randomized search heuristics. PhD thesis. 2010.

[52] Keane, Andy and Brown, S. The design of a satellite boom with enhanced vibration performance using genetic algorithm techniques. *Journal of The Acoustical Society of America - J ACOUST SOC AMER*. 1996.

[53] Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P. Optimization by Simulated Annealing. *Science*. 1983.

[54] Lagoudakis, Michail and Littman, Michael. *Algorithm Selection using Reinforcement Learning*. 2004.

[55] Lehre, Per and Witt, Carsten. General Drift Analysis with Tail Bounds. 2013.

[56] Lehre, Per Kristian and Witt, Carsten. Black-Box Search by Unbiased Variation. *Algorithmica*. 2012.

[57] Lengler, J. and Steger, A. Drift Analysis and Evolutionary Algorithms Revisited. *Combinatorics, Probability and Computing*. 2018.

[58] Lengler, Johannes. Drift Analysis. In: *Theory of Evolutionary Computation: Recent Developments in Discrete Optimization*. Springer International Publishing, 2020, pp. 89–131.

[59] Lindström, Bernt. On a Combinatorial Problem in Number Theory. *Canadian Mathematical Bulletin*. 1965.

[60] Lindström, Bernt. On a combinatory detection problem I. *Mathematical Institute of the Hungarian Academy of Science*. 1964.

[61] Maulik, Ujjwal and Bandyopadhyay, Sanghamitra. Genetic algorithm-based clustering technique. *Pattern Recognition*. 2000.

[62] Metropolis, Nicholas et al. Equation of State Calculations by Fast Computing Machines. *The Journal of Chemical Physics*. 1953.

[63] Mnih, Volodymyr et al. Human-level control through deep reinforcement learning. *Nature*. 2015.

[64] Moiola, Andrea. *Dispense del corso di* Modellistica Numerica. 2024.

[65] Motwani, Rajeev and Raghavan, Prabhakar. *Randomized Algorithms*. Cambridge University Press, 1995.

[66] Nocedal, Jorge and Wright, Stephen. *Numerical Optimization*. Springer New York, NY, 2006.

[67] Ozdamar, Linet. A Genetic Algorithm Approach to a General Category Project Scheduling Problem. *IEEE Transactions on Systems, Man, and Cybernetics, Part C*. 1999.

[68] Paechter, Ben et al. Timetabling the Classes of an Entire University with an Evolutionary Algorithm. *Lecture Notes in Computer Science*. 2000.

[69] Pérez, Carlos et al. Quasi-Random Sampling Importance Resampling. *Communication in Statistics- Simulation and Computation*. 2005.

[70] Pezzella, F., Morganti, G., and Ciaschetti, G. A genetic algorithm for the Flexible Job-shop Scheduling Problem. *Computers Operations Research*. 2008.

[71] Pinto, Eduardo Carvalho and Doerr, Carola. *Towards a More Practice-Aware Runtime Analysis of Evolutionary Algorithms*. 2018.

[72] Probst, Philipp, Bischl, Bernd, and Boulesteix, Anne-Laure. *Tunability: Importance of Hyperparameters of Machine Learning Algorithms*. 2018.

[73] Puterman, Martin L. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. 1st. USA: John Wiley & Sons, Inc., 1994.

[74] R, Ex, Pettinger, James, and Everson, Richard. *Controlling Genetic Algorithms with Reinforcement Learning*. 2003.

[75] Raponi, Elena et al. Kriging-assisted topology optimization of crash structures. *Computer Methods in Applied Mechanics and Engineering*. 2019.

[76] Reynolds, Robert. Evolutionary Computation: The Fossil Record. David Fogel, IEEE PRESS, 1998, ISBN 0-7803-3481-7. *Biosystems*. 2000.

[77] Rowe, Jonathan E. and Sudholt, Dirk. The choice of the offspring population size in the (1,) evolutionary algorithm. *Theoretical Computer Science*. 2014.

[78] Sharma, Mudita et al. *Deep Reinforcement Learning Based Parameter Control in Differential Evolution*. 2019.

[79] Sörensen, Kenneth and Glover, Fred. Metaheuristics. In: 2013, pp. 960–970.

[80] Sutton, Richard S. and Barto, Andrew G. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018.

[81] Syswerda, Gilbert. Uniform Crossover in Genetic Algorithms. In: 1989.

[82] Tillé, Yves. Sampling Algorithms. In: *International Encyclopedia of Statistical Science*. Ed. by Miodrag Lovric. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 1273–1274.

[83] Vitter, Jeffrey S. Random sampling with a reservoir. *ACM Trans. Math. Softw.* 1985.

[84] Watkins, C. J. C. H. Learning from Delayed Rewards. PhD thesis. King's College, Oxford, 1989.

[85] Weinzierl, Stefan. *Introduction to Monte Carlo methods*. 2000.

[86] Witt, Carsten. Tight Bounds on the Optimization Time of a Randomized Search Heuristic on Linear Functions. *Combinatorics, Probability and Computing*. 2013.

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

[88] Yao, Andrew Chi-Chin. Probabilistic computations: Toward a unified measure of complexity. In: *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. 1977.

Appendix A

Evolutionary Algorithms for Continuous Optimization

Evolutionary algorithms are also widely used to address continuous optimization problems, namely optimization problems in the form of (A.1) where the search space S is continuous.

$$x^* = \underset{x \in S}{\operatorname{argmin}} f(x) \tag{A.1}$$

The study of the field conducted to the development some of the most popular evolutionary algorithms. Since we referred from time to time to them in our analysis, we present here three among the most continuous evolutionary algorithms: Metropolis Algorithm, Simulated Annealing (SA) and Covariance Matrix Adaptation Evolutionary Strategy (CMA-ES).

Metropolis algorithm [62] can also be considered a local search algorithm: it looks iteratively for solutions through random offspring generation and selection. New candidates are always accepted if they improve the fitness of the current best solution, but are also accepted with a probability exponentially decreasing with distance from the current best fitness even if they do not improve the current best. The structure of the algorithm is summarized in Algorithm 12.

Simulated annealing [53, 66] is a more elaborate version of the metropolis algorithm. In this case, the optimization process takes advantage of a dynamic choice of the acceptance probability. To do so, a parameter t, called *temperature*, is introduced which controls the acceptance of lower fitness values. In particular, the parameter is chosen to follow a *time-dependent policy*, reducing the probability over time (measured in iterations), in order to explore the search space broadly in the first phases to avoid local optima and then focus more on exploiting the best solution so far to reach a global optimum. The algorithm is also one of the most well-known examples of advantages given by a dynamic parameter policy. In Algorithm 13 the simulation annealing scheme is summarized.

Lastly, CMA-ES [42] is a very popular evolution strategy that was developed in the last few years. The algorithm is quite complex and elaborated; the fundamental idea is to generate an offspring of λ individuals from a multivari-

Algorithm 12 General scheme of Metropolis algorithm

Initialize the current point $x = x_{\text{in}}$
Initialize the best solution found so far: $x_{\text{best}} = x$ and $y_{\text{best}} = f(x)$
while stopping criterion not met **do**
    for a fixed number of iterations **do**
        Generate a new candidate point x' randomly from the proposal distribution
        Evaluate $y' = f(x')$
        if $y' < f(x)$ **then**
            Accept the new point: $x = x'$
        else
            Compute the acceptance probability $p = e^{-(y'-f(x))}$
            Draw a random number $u \sim U(0, 1)$
            if $u \leq p$ **then**
                Accept x', i.e., $x = x'$
            end if
        end if
        if $f(x) < y_{\text{best}}$ **then**
            Update the best solution: $x_{\text{best}} = x$ and $y_{\text{best}} = f(x)$
        end if
    end for
end while
return x_{best} and y_{best}

Algorithm 13 General scheme of simulated annealing

Initialize the current point $x = x_{\text{in}}$ and temperature $t = t_{\text{in}}$
Initialize the best solution found so far: $x_{\text{best}} = x$ and $y_{\text{best}} = f(x)$
while stopping criterion not met **do**
    for a fixed number of iterations **do**
        Generate a new candidate point x' randomly
        Evaluate $y' = f(x')$
        if $y' < f(x)$ **then**
            Accept the new point: $x = x'$
        else
            Compute the acceptance probability $p = e^{-(y'-f(x))/t}$
            Draw a random number $u \sim U(0, 1)$
            if $u \leq p$ **then**
                Accept x', i.e., $x = x'$
            end if
        end if
        if $f(x) < y_{\text{best}}$ **then**
            Update the best solution: $x_{\text{best}} = x$ and $y_{\text{best}} = f(x)$
        end if
    end for
    Decrease the temperature t according to the annealing schedule (e.g., $t = \alpha t$, where $\alpha < 1$)
end while
return x_{best} and y_{best}

ate normal, where the mean and the covariance matrix are updated based on the information gained in previous iterations through weighted selection mechanisms. We will not delve into details, but we reported the general structure of the algorithm in Algorithm 14.

Algorithm 14 General structure of CMA-ES

Set parameters λ, w_i for $i = 1, \dots, \mu$, c_σ, d_σ, c_c, c_1, and c_μ according to Table 1
Initialize evolution paths $p_\sigma = 0$, $p_c = 0$, covariance matrix $C = I$, and generation counter $g = 0$
Choose distribution mean $m \in \mathbb{R}^n$ and step-size $\sigma \in \mathbb{R}_{>0}$ based on the problem
while termination criterion not met **do**
    $g \leftarrow g + 1$ ▷ Increment generation counter
    for $k = 1, \dots, \lambda$ **do** ▷ Sample new population of search points
        $z_k \sim \mathcal{N}(0, I)$
        $y_k = BDz_k \sim \mathcal{N}(0, C)$
        $x_k = m + \sigma y_k \sim \mathcal{N}(m, \sigma^2 C)$
    end for
    Selection and Recombination:
    $\hat{y}_w = \sum_{i=1}^{\mu} w_i y_i$ where $\sum_{i=1}^{\mu} w_i = 1$ and $w_i > 0$ for $i = 1, \dots, \mu$
    $m \leftarrow m + c_m \sigma \hat{y}_w$ ▷ Update mean
    Step-size Control:
    $p_\sigma \leftarrow (1 - c_\sigma) p_\sigma + \sqrt{c_\sigma (2 - c_\sigma) \mu_{\text{eff}}} C^{-1/2} \hat{y}_w$
    $\sigma \leftarrow \sigma \cdot \exp\left(\frac{c_\sigma}{d_\sigma} \left(\|p_\sigma\| / \mathbb{E}\|\mathcal{N}(0, I)\| - 1\right)\right)$ ▷ Adapt step-size
    Covariance Matrix Adaptation:
    $p_c \leftarrow (1 - c_c) p_c + \sqrt{c_c (2 - c_c) \mu_{\text{eff}}} \hat{y}_w$
    $w_i^{\text{rank}} \leftarrow w_i \cdot \left(1 \text{ if } w_i \geq 0 \text{ else } n / \|C^{-1/2} y_i\|^2\right)$ ▷ Rank-one update weights
    $C \leftarrow (1 + c_1 h_\sigma - c_1 - c_\mu \sum w_i^{\text{rank}}) C + c_1 p_c p_c^T + c_\mu \sum_{i=1}^{\lambda} w_i^{\text{rank}} y_i y_i^T$ ▷ Covariance matrix adaptation
end while
return m and σ ▷ Final mean and step-size as solution

Appendix B

Selected Proofs

B.1 Proofs from Section 2.5

Proof of Theorem 11.

Proof. (i) Since we are only interested in T, we can assume that $X_t = 0$ for every $T \geq 0$ and therefore $X_t > 0$ if and only if $T > t$.

We rewrite the condition on the drift as

$$\mathbb{E}[X_{t+1} \mid X_t = s] \leq \mathbb{E}[X_t \mid X_t = s] - \delta, \quad \forall s \in \mathcal{S} \setminus \{0\}$$

For the previous observation, we can write

$$\mathbb{E}[X_{t+1} \mid T > t] \leq \mathbb{E}[X_t \mid T > t] - \delta$$

By the law of total probabilities, we have that

$$\begin{aligned} \mathbb{E}[X_t] &= \mathbb{P}(T > t)\mathbb{E}[X_t \mid T > t] + \mathbb{P}(T \leq t)\mathbb{E}[X_t \mid T > t] \\ &= \mathbb{P}(T > t)\mathbb{E}[X_t \mid T > t] \end{aligned}$$

From analogous argument and integrating the previous formula,

$$\begin{aligned} \mathbb{E}[X_{t+1}] &= \mathbb{P}(T > t)\mathbb{E}[X_{t+1} \mid T > t] \\ &\leq \mathbb{P}(T > t)(\mathbb{E}[X_t \mid T > t] - \delta) \\ &= \mathbb{E}[X_t] - \delta\mathbb{P}(T > t) \end{aligned}$$

In particular,

$$\delta\mathbb{P}(T > t) \leq \mathbb{E}[X_t] - \mathbb{E}[X_{t+1}]$$

The claim follows then by considering the sum of the series of all possible values of t.

$$\delta\mathbb{E}[T] = \lim_{\tau\to\infty}\sum_{t=0}^{\tau}\delta\mathbb{P}(T>t) \le \lim_{\tau\to\infty}\sum_{t=0}^{\tau}(\mathbb{E}[X_t]-\mathbb{E}[X_{t+1}]) = \lim_{\tau\to\infty}(\mathbb{E}[X_0]-\mathbb{E}[X_{\tau+1}]) \le \mathbb{E}[X_0] \tag{B.1}$$

(*ii*) All the equations with the exception of (B.1) hold in reverse. (B.1) becomes

$$\delta\mathbb{E}[T] = \lim_{\tau\to\infty}\sum_{t=0}^{\tau}\delta\mathbb{P}(T>t) \ge \lim_{\tau\to\infty}(\mathbb{E}[X_0]-\mathbb{E}[X_{\tau+1}])$$

If $\mathbb{P}(T>\tau)$ does not converge to 0, then $\mathbb{E}[T]=\sum_{t=0}^{\infty}\mathbb{P}(T>t)=\infty$ and (*ii*) holds trivially. Otherwise, $\mathbb{P}(T>t)\to 0$ and $\lim_{\tau\to\infty}\mathbb{E}[X_{\tau+1}]=0$. □

Proof of Theorem 12.

Proof. The main idea is to rescale the process X_t by the function

$$g(s) := \begin{cases} \frac{s_{\min}}{h(s_{\min})} + \int_{s_{\min}}^{s}\frac{1}{h(\sigma)}d\sigma, & s \ge s_{\min} \\ \frac{s}{h(s_{\min})}, & 0 \le s \le s_{\min} \end{cases}$$

Since h is increasing, the integral is well-defined. We also note that g is strictly increasing. We claim that for all $s \in \mathcal{S}\setminus\{0\}$ and all $r \ge 0$,

$$g(s) - g(r) \ge \frac{s-r}{h(s)}$$

To prove it we consider three cases: first, if $s \ge r \ge s_{\min}$,

$$g(s)-g(r) = \int_r^s \frac{1}{h(\sigma)}d\sigma \ge \int_r^s \frac{1}{h(s)}d\sigma = \frac{s-r}{h(s)}$$

If $r \ge s \ge s_{\min}$, then

$$g(r)-g(s) = \int_s^r \frac{1}{h(\sigma)}d\sigma \le \int_s^r \frac{1}{h(s)}d\sigma = \frac{r-s}{h(s)}$$

The third case is $s \ge s_{\min} > r \ge 0$:

$$g(s)-g(r) = \frac{s_{\min}}{h(s_{\min})} + \int_{s_{\min}}^{s}\frac{1}{h(\sigma)}d\sigma - \frac{r}{h(s_{\min})} \ge \frac{s_{\min}-r}{h(s_{\min})} + \frac{s-s_{\min}}{h(s)} \ge \frac{s-r}{h(s)}$$

We now consider the stochastic process $(Y_t)_{t\ge 0}$ where $Y_t := g(X_t)$. For all $s \in \mathcal{S}\setminus\{0\}$,

$$\mathbb{E}[Y_t - Y_{t+1} \mid Y_t = g(s)] = \mathbb{E}[g(X_t)-g(X_{t+1}) \mid g(X_t)=g(s)] \ge \mathbb{E}[\frac{X_t - X_{t+1}}{h(X_t)} \mid X_t = s] = \frac{\Delta_t(s)}{h(s)} \ge 1$$

The claim then follows the additive drift applied to the process $(Y_t)_{t\geq 0}$. □

B.2 Proofs from Section 3.2

Proof of Theorem 34.

Proof. First, we denote by A_i the time required to find an improvement given that the initial solution has fitness $n-i$.

We observe that A_i follows a geometric distribution of parameter the probability of improvement $\mathbb{P}[\mathrm{LO}(y) > \mathrm{LO}(x)]$. Therefore $\mathbb{P}[f(y) > f(x)] = (1-p_{n-i})^{n-i}p_{n-i}$ and the expected time is

$$\mathbb{E}[A_i] = \frac{1}{\mathbb{P}[f(y) > f(x)]} = \frac{1}{(1-p_{n-i})^{n-i}p_{n-i}} \tag{B.2}$$

We then observe, using the law of total distributions and the assumption that those in the tail are uniformly distributed, that

$$\mathbb{E}[T] = \frac{1}{2}\sum_{i=1}^{n}\mathbb{E}[A_i] \tag{B.3}$$

From the combination of (B.2) and (B.3) and some simple algebra, it follows the claim. □

Proof of Theorem 35.

Proof. We start assuming a uniform distribution of ones in the tail, which means that x_j, for $j \in [i+2..n]$, are random variables i.i.d. uniformly distributed in $\{0,1\}$.

We then denote with T_i^0 for every $i \in [0..n]$ the runtime of the algorithm when starting with a random search point of fitness exactly i. With T_i^{rand} we then denote the runtime starting with a search point of fitness at least i; in this case also x_{i+1} is a random variable.

It is immediate to observe that $T_n^0 = T_n^{\mathrm{rand}}$. We then exploit the fact that

$$T_i^0 = \mathrm{Geom}(q_i) + T_{i+1}^{\mathrm{rand}}, \quad \forall i < n$$

that follows the observation that the waiting time for flipping the $i+1$-th bit follows a geometric distribution with parameter the probability q_i of flipping the $i+1$-th bit during an iteration of the algorithm.

It follows

$$\begin{aligned} T_i^{\mathrm{rand}} &= X_i T_i^0 + (1-X_i)T_{i+1}^{\mathrm{rand}} \\ &= X_i(\mathrm{Geom}(q_i) + T_{i+1}^{\mathrm{rand}}) + (1-X_i)T_{i+1}^{\mathrm{rand}} \\ &= X_i\mathrm{Geom}(q_i) + T_{i+1}^{\mathrm{rand}} \end{aligned}$$

where X_i is a uniform binary random variable independent from any other randomness of the other distributions.

By induction and observing that $T = T_0^{\mathrm{rand}}$ it follows the claim. □

Appendix C

Markov Decision Processes

Markov decision processes (MDPs) are classical formalizations of sequential decision making, where action influences not just the immediate reward but also subsequent states and actions, and thus future rewards. They are often used to model sequential decision making, in particular in the context of reinforcement learning (RL). They are also the framework used for dynamic algorithm configuration (DAC) presented in Section 2.6.2. Reviews of the topic can be found in [80, 73].

Formally, we can give the following definition.

Definition 40. A *Markov decision process (MDP)* is a tuple $(\mathcal{S}, \mathcal{A}, p(y \mid x, a), r(x), \pi_0)$, where $\mathcal{S}$ is the state space, $\mathcal{A}$ is the action space, $p(y \mid x, a)$ is the transition probability $(y, x \in \mathcal{S}, a \in \mathcal{A})$, $r(x)$ is a reward function and π_0 is the initial state distribution.

Informally, the setting considered is that of the interaction of an agent with an environment. The interaction occurs in time steps $t \in \mathcal{N}$. The initial state $s_0 \in \mathcal{S}$ is determined by the initial state distribution π_0. The environment in each time step is described by the state $s_t \in \mathcal{S}$, which determines the choice of the action a_t, the reward in the next step $r_{t+1} \in \mathbb{R}$, and the new state s_{t+1} is reached following the probability of transition $p(\cdot \mid s_t, a_t)$.

The MDP and agent together thereby give rise to a sequence or *trajectory* that begins like this:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \ldots$$

In a finite MDP, the sets of states, actions, and rewards ($\mathcal{S}, \mathcal{A}$, and $\mathcal{R}$) all have a finite number of elements. In this case, the random variables R_t and S_t have well-defined discrete probability distributions dependent only on the preceding state and action.

In a Markov decision process, the probabilities given by p completely characterize the dynamics of the environment. That is, the probability of each possible value for s_t and r_t depends on the state and action immediately preceding, s_{t-1} and a_{t-1}, and given them, not at all on previous states and actions. This is the *Markov property* of MDPs.

In general, we seek to maximize the expected return, where the return, denoted G_t, is defined as some specific function of the reward sequence, with a discount factor γ that describes how much we value immediate rewards over long-term gains:

$$G_t := R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{T} \gamma^k R_{t+k+1},$$

where $\gamma \in [0, 1]$ and $T \in \mathcal{N} \cup \{\infty\}$.

Obviously,

$$G_t := R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = R_{t+1} + \gamma \left(R_{t+2} + \gamma R_{t+3} + \cdots \right) = R_{t+1} + \gamma G_{t+1}.$$

Solving MDPs usually involves estimating *value functions* that estimate how good it is for the agent to be in a given state defined in terms of expected return. Of course, the rewards the agent can expect to receive in the future depend on the actions it will take. Consequently, value functions are defined with respect to particular ways of acting, called *policies*.

Formally, a policy is a mapping from states to probabilities of selecting each possible action. If the agent follows the policy π at time t, then $\pi(a \mid s)$ is the probability that $A_t = a$ if $S_t = s$.

The value function of a state s under a policy π, denoted $v_\pi(s)$, is the expected return when starting in s and following π thereafter. For MDPs, we can define v_π as:

$$v_\pi(s) := \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s \right], \quad \forall s \in \mathcal{S}, \tag{3.12}$$

The function v_π is called *state-value function* for policy π.

Similarly, we define the value of taking action a in state s under a policy π, denoted $q_\pi(s, a)$, as the expected return starting from s, taking the action a, and thereafter following policy π:

$$q_\pi(s, a) := \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a \right]. \tag{3.13}$$

We call q_π the *action-value function* for policy π.

The value functions v_π and q_π can be estimated from experience, for example through Monte Carlo methods (see Appendix D) because they involve averaging over many random samples of actual returns.

A fundamental property of value functions is that they satisfy a recursive relationship. For any policy π and any state s, the following consistency condition holds:

$$\begin{aligned} v_\pi(s) &:= \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\ &= \sum_a \pi(a \mid s) \sum_{s', r} p(s', r \mid s, a) \left[r + \gamma v_\pi(s') \right], \quad \forall s \in \mathcal{S}. \end{aligned} \tag{3.14}$$

This is the *Bellman equation* for v_π.

For finite MDPs, we define an *optimal policy* π^* as one that achieves the highest expected return for all states.

In Section 2.6.2, we cited *contextual Markov decision processes (cMDPs)*. They have been introduced in [40] and, informally, can be defined as a collection of MDPs that share the same state and action spaces. We give the definition.

Definition 41. A *contextual Markov decision process (cMDP)* is a tuple $(\mathcal{C}, \mathcal{S}, \mathcal{A}, M(c))$ where:

- $\mathcal{C}$ is the context space,
- $\mathcal{S}$ and $\mathcal{A}$ are the state and action spaces, respectively,
- M is a function mapping any context $c \in \mathcal{C}$ to an MDP $M(c) = (\mathcal{S}, \mathcal{A}, p_c(y \mid x, a), r_c(x), \pi_c^0)$.

The simplest scenario in a cMDP setting occurs when the context is observable. In this setting, the problem reduces to correctly generalizing the model from the context. If the observable context c is finite, where $|\mathcal{C}| = K$, then without further assumption, one can simply learn K different models. Otherwise, other strategies may be applied.

Appendix D

Monte Carlo Methods

We usually refer to *Monte Carlo methods* whenever we derive a certain quantity through a simulation process. We referred to it in Chapter 5, when we derived high-dimensional results of expected runtime and standard deviation through simulation of the algorithm and we will now give a better formalization following [85]. Formally, Monte Carlo methods can be defined as a simple way to estimate the value of an integral from a set of samples. We consider thus the problem of approximating

$$I = \int_{\Omega} f(x) d\mu(x). \tag{D.1}$$

Taking μ as a probability distribution in the domain Ω, we can write as I many relevant quantities in statistics, such as the expectation or the standard deviation.

The easiest way to proceed is to approximate the integral in (D.1) using a sample average approximation. Suppose that we have a set of samples $\{x_i\}_{m=1}^{M}$ drawn according to the probability described by μ, we can approximate the integral I with the sum.

$$\hat{I} = \frac{1}{M} \sum_{m=1}^{M} f(x_m)$$

We note that the implementation of this process can easily take advantage of parallelization in the calculation of $f(x_m)$ for each m, as we did in our simulations.

The theoretical guarantee behind the consistency of $\hat{I}$ is the law of large numbers, which guarantees that $\hat{I} \to I$ ad $M \to \infty$. Using the central limit theorem, we can even study the convergence rate. We take the proof from [64].

The Central Limit Theorem states that, given a sequence of i.i.d. random variables $(X_m)_{m\in\mathbb{N}}$ with mean μ and variance $\sigma^2 < \infty$, the following holds:

$$Z_M := \frac{\sqrt{M}}{\sigma} \left(\frac{1}{M} \sum_{m=1}^{M} X_m - \mu \right) \xrightarrow{D} \mathcal{N}(0, 1).$$

The symbol $\xrightarrow{D}$ means that the sequence of random variables $(Z_M)_{M\in\mathbb{N}}$

converges in distribution to a normal random variable with mean zero and variance 1. In this case, convergence "in distribution" means that the cumulative functions (or distribution functions) $F_{Z_M}(x) := P(Z_M \leq x)$ converge pointwise in $\mathbb{R}$ to that of the normal variable $\Phi(x) := \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{1}{2}t^2}\, dt$.

This implies that for any $c > 0$, the following hold:

$$\lim_{M\to\infty} P(Z_M \leq c) = \Phi(c)$$

and therefore

$$\lim_{M\to\infty} P(|Z_M| \leq c) = \Phi(c) - \Phi(-c) = 2\Phi(c) - 1.$$

Thus, the probability that the Monte Carlo estimator (the empirical mean)

$$X_M := \frac{1}{M} \sum_{m=1}^{M} X_m$$

differs from the desired value μ by less than $\frac{\sqrt{c}\sigma}{\sqrt{M}}$, in the limit $M \to \infty$, depends only on c.

For example, choosing $c = 3$, the probability that the Monte Carlo estimator X_M has an error less than $\frac{3\sigma}{\sqrt{M}}$ is approximately $2\Phi(3) - 1 \approx 0.9973$. It follows that the Monte Carlo method converges at a rate $\frac{1}{\sqrt{M}}$, often written informally as $|X_M - \mu| \sim \frac{\sigma}{\sqrt{M}}$.

Appendix E

Complete Results

In this Appendix, we report the complete simulation results described in Chapter 5.

In Table E.1, we provided the complete results for the simulation of the optimal fitness-based policy and the heuristics with lexicographic selection. The first column indicates the problem dimension; the second and third report the estimated runtime expectation and standard deviation using the fitness-based and the heuristic policy, respectively; the fourth indicates the percentage advantage; the fifth and the sixth, the relative standard deviation for the two policies.

n	LO	H.	Adv.	Rel SD LO	Rel SD h.
2	1.521 ± 1.519	1.735 ± 1.653	-12.45%	0.998	0.952
3	3.079 ± 2.849	2.397 ± 2.565	22.16%	0.924	1.070
4	5.424 ± 4.370	4.435 ± 4.157	18.23%	0.806	0.937
5	8.054 ± 5.631	6.477 ± 4.810	19.56%	0.699	0.743
6	11.027 ± 7.388	9.137 ± 6.276	17.15%	0.670	0.687
7	13.751 ± 8.729	11.402 ± 7.302	17.06%	0.634	0.640
8	18.476 ± 11.109	14.413 ± 9.253	21.98%	0.601	0.642
9	21.882 ± 12.419	18.415 ± 10.473	15.84%	0.568	0.569
10	27.929 ± 15.160	20.369 ± 11.564	27.08%	0.543	0.568
11	30.574 ± 14.944	23.619 ± 12.446	22.73%	0.489	0.527
12	36.457 ± 18.309	27.685 ± 14.463	24.05%	0.502	0.522
13	39.183 ± 18.929	31.704 ± 16.737	19.10%	0.483	0.528
14	46.835 ± 22.381	33.810 ± 16.276	27.83%	0.478	0.482
15	52.032 ± 23.872	37.208 ± 18.122	28.47%	0.459	0.487
16	57.457 ± 26.569	40.549 ± 18.181	29.41%	0.462	0.448
17	61.977 ± 27.859	45.181 ± 21.168	27.12%	0.449	0.469
18	69.339 ± 29.535	48.380 ± 21.923	30.25%	0.426	0.453
19	74.388 ± 31.414	53.886 ± 23.645	27.57%	0.422	0.439
20	83.052 ± 35.401	56.817 ± 24.935	31.58%	0.426	0.439
21	88.691 ± 36.442	59.768 ± 26.665	32.59%	0.411	0.446
22	95.302 ± 37.489	63.291 ± 26.779	33.58%	0.394	0.423
23	100.346 ± 38.826	67.683 ± 28.609	32.54%	0.387	0.423

24	108.795 ± 40.219	72.138 ± 29.395	33.67%	0.369	0.408
25	112.672 ± 40.352	76.243 ± 32.626	32.33%	0.358	0.428
26	124.253 ± 47.718	79.950 ± 33.704	35.63%	0.384	0.422
27	128.409 ± 46.414	86.065 ± 33.028	32.97%	0.362	0.384
28	140.431 ± 51.147	89.395 ± 36.086	36.35%	0.364	0.404
29	141.650 ± 48.752	93.281 ± 36.266	34.15%	0.344	0.389
30	156.440 ± 57.204	97.100 ± 38.187	37.92%	0.366	0.393
31	157.080 ± 57.249	101.878 ± 38.332	35.14%	0.364	0.376
32	174.734 ± 58.622	107.705 ± 40.617	38.36%	0.336	0.377
33	174.486 ± 60.132	108.068 ± 39.889	38.06%	0.345	0.369
34	184.784 ± 60.764	114.528 ± 42.385	38.02%	0.329	0.370
35	189.289 ± 63.387	119.655 ± 47.764	36.79%	0.335	0.399
36	198.642 ± 64.546	124.477 ± 46.496	39.53%	0.314	0.364
37	206.226 ± 68.097	131.135 ± 46.604	36.41%	0.330	0.355
38	226.386 ± 75.399	131.394 ± 48.877	41.97%	0.333	0.372
39	229.472 ± 76.926	136.728 ± 50.120	40.41%	0.335	0.367
40	247.874 ± 78.601	142.537 ± 51.631	42.49%	0.317	0.362
41	253.898 ± 81.027	143.097 ± 55.524	43.65%	0.319	0.388
42	269.894 ± 86.111	149.254 ± 57.001	44.71%	0.319	0.382
43	276.784 ± 88.894	153.651 ± 57.818	44.47%	0.321	0.376
44	284.146 ± 88.953	156.014 ± 59.811	45.10%	0.313	0.383
45	292.925 ± 93.407	164.274 ± 61.643	43.42%	0.319	0.375
46	313.325 ± 98.978	168.542 ± 63.163	46.20%	0.316	0.375
47	316.871 ± 100.616	171.923 ± 65.781	45.73%	0.318	0.383
48	341.467 ± 108.826	181.289 ± 66.109	46.91%	0.319	0.364
49	350.737 ± 110.615	185.174 ± 70.050	47.20%	0.316	0.378
50	365.279 ± 113.261	189.796 ± 70.485	48.06%	0.310	0.371
51	375.247 ± 117.238	191.904 ± 71.795	48.90%	0.312	0.374
52	390.667 ± 120.894	199.386 ± 73.285	48.98%	0.309	0.368
53	399.279 ± 125.013	208.874 ± 73.365	47.68%	0.313	0.351
54	418.741 ± 133.688	214.174 ± 79.111	49.00%	0.319	0.369
55	422.569 ± 130.423	217.804 ± 77.857	48.44%	0.309	0.357
56	439.817 ± 135.901	227.416 ± 82.550	48.29%	0.309	0.363
57	448.724 ± 134.249	235.257 ± 82.344	47.35%	0.299	0.350
58	478.070 ± 142.799	241.436 ± 86.061	49.45%	0.299	0.356
59	489.768 ± 147.194	245.529 ± 87.676	49.82%	0.301	0.357
60	497.972 ± 148.056	249.367 ± 87.830	49.90%	0.297	0.352
61	508.935 ± 150.723	258.354 ± 90.569	49.24%	0.296	0.351
62	530.522 ± 160.356	266.594 ± 92.727	49.79%	0.302	0.348
63	552.918 ± 170.515	272.634 ± 95.887	50.35%	0.308	0.352
64	548.249 ± 157.014	275.498 ± 93.911	49.67%	0.286	0.341
65	573.929 ± 173.709	283.913 ± 99.019	50.53%	0.303	0.349
66	600.781 ± 177.708	295.673 ± 98.462	50.80%	0.296	0.333
67	622.059 ± 183.716	301.529 ± 101.056	51.54%	0.295	0.335

68	632.316 ± 183.328	310.225 ± 103.255	50.97%	0.290	0.333
69	658.671 ± 192.938	317.198 ± 105.193	51.58%	0.293	0.332
70	686.021 ± 201.404	322.379 ± 107.782	52.99%	0.294	0.334
71	691.160 ± 204.654	330.573 ± 109.750	52.18%	0.296	0.332
72	730.918 ± 216.754	343.061 ± 114.235	53.05%	0.297	0.333
73	739.098 ± 209.201	348.078 ± 112.268	52.90%	0.283	0.323
74	769.792 ± 225.377	362.014 ± 113.681	53.02%	0.293	0.314
75	793.235 ± 230.551	368.101 ± 117.044	53.60%	0.291	0.318
76	799.644 ± 230.308	376.487 ± 118.391	52.60%	0.288	0.315
77	830.993 ± 235.029	384.601 ± 121.068	53.73%	0.283	0.315
78	857.703 ± 241.768	394.934 ± 122.296	54.00%	0.282	0.310
79	884.473 ± 247.100	405.277 ± 125.830	54.16%	0.279	0.311
80	913.492 ± 258.417	409.831 ± 126.823	55.03%	0.283	0.310
81	930.277 ± 253.452	419.451 ± 128.335	54.91%	0.272	0.306
82	964.273 ± 262.197	431.721 ± 130.199	55.23%	0.272	0.302
83	982.580 ± 265.156	441.317 ± 132.255	55.09%	0.270	0.300
84	1007.287 ± 269.950	447.661 ± 133.727	55.57%	0.268	0.299
85	1035.882 ± 276.669	457.912 ± 135.113	55.78%	0.267	0.295
86	1072.063 ± 286.797	471.109 ± 137.508	56.05%	0.267	0.292
87	1075.034 ± 286.262	477.635 ± 138.618	55.56%	0.266	0.290
88	1121.810 ± 292.071	492.453 ± 143.356	56.11%	0.260	0.291
89	1155.908 ± 297.655	499.779 ± 145.225	56.76%	0.257	0.291
90	1195.506 ± 308.327	512.781 ± 144.698	57.13%	0.258	0.282
91	1212.222 ± 311.790	518.417 ± 149.401	57.22%	0.257	0.288
92	1243.608 ± 314.378	532.831 ± 150.272	57.14%	0.253	0.282
93	1266.778 ± 318.859	541.303 ± 151.651	57.26%	0.252	0.280
94	1301.315 ± 324.150	549.702 ± 154.222	57.76%	0.249	0.281
95	1341.020 ± 332.507	563.537 ± 155.153	58.00%	0.248	0.275
96	1380.477 ± 340.808	578.265 ± 157.106	58.12%	0.247	0.272
97	1398.453 ± 342.208	589.471 ± 160.173	57.86%	0.245	0.272
98	1448.055 ± 357.629	604.158 ± 159.772	58.28%	0.247	0.264
99	1484.031 ± 357.983	616.595 ± 161.838	58.47%	0.241	0.263

Table E.1: Simulation results for lexicographic selection

In Table E.2, we provided the corresponding results for standard selection.

n	LO	H.	Adv.	Rel SD LO	Rel SD h.
2	1.540 ± 1.548	1.984 ± 2.114	-22.38%	1.005	1.065
3	3.354 ± 3.234	2.564 ± 2.629	30.80%	0.964	1.026
4	5.984 ± 5.110	5.222 ± 5.483	14.60%	0.854	1.050
5	9.934 ± 7.493	8.008 ± 6.524	24.06%	0.754	0.814
6	13.968 ± 9.445	11.922 ± 9.734	17.12%	0.676	0.817
7	18.122 ± 11.562	16.270 ± 11.785	11.39%	0.638	0.724
8	24.878 ± 16.563	22.444 ± 15.201	10.84%	0.666	0.677

n	LO	H.	Adv.	Rel SD LO	Rel SD h.
9	30.476 ± 18.156	27.874 ± 17.174	9.33%	0.596	0.616
10	38.878 ± 23.354	34.238 ± 19.830	13.57%	0.600	0.579
11	47.232 ± 24.559	45.948 ± 24.219	2.79%	0.520	0.527
12	54.064 ± 28.766	51.584 ± 27.595	4.80%	0.532	0.535
13	66.516 ± 34.984	61.540 ± 29.190	8.08%	0.526	0.474
14	76.644 ± 38.734	73.636 ± 33.568	4.09%	0.505	0.456
15	87.720 ± 42.997	85.900 ± 38.781	2.12%	0.490	0.451
16	100.468 ± 43.384	90.538 ± 39.881	10.97%	0.432	0.440
17	111.104 ± 48.380	107.988 ± 47.683	2.89%	0.435	0.442
18	124.712 ± 54.559	117.618 ± 49.745	6.03%	0.437	0.423
19	142.262 ± 59.733	130.030 ± 53.195	9.39%	0.420	0.409
20	159.806 ± 63.532	151.438 ± 61.158	5.53%	0.398	0.404
21	170.142 ± 68.026	165.094 ± 59.392	3.06%	0.400	0.360
22	180.898 ± 70.501	174.248 ± 65.548	3.82%	0.390	0.376
23	205.540 ± 78.145	197.738 ± 71.211	3.94%	0.380	0.360
24	225.634 ± 85.848	215.244 ± 79.943	4.82%	0.381	0.372
25	237.016 ± 90.579	228.238 ± 77.480	3.84%	0.382	0.339
26	255.660 ± 89.167	249.706 ± 91.585	2.39%	0.349	0.367
27	280.474 ± 97.709	279.278 ± 96.741	0.43%	0.348	0.347
28	302.822 ± 103.388	287.494 ± 93.560	5.33%	0.341	0.325
29	333.348 ± 109.443	316.956 ± 103.010	5.17%	0.328	0.325
30	344.132 ± 114.888	347.866 ± 102.573	-1.07%	0.334	0.295
31	377.698 ± 125.013	358.148 ± 109.138	5.45%	0.331	0.305
32	396.354 ± 126.355	396.740 ± 113.515	-0.10%	0.319	0.286
33	424.156 ± 132.659	409.274 ± 124.055	3.64%	0.313	0.303
34	452.094 ± 155.913	418.974 ± 131.209	7.90%	0.345	0.313
35	466.338 ± 149.266	460.110 ± 131.712	1.35%	0.320	0.286
36	501.080 ± 151.340	481.282 ± 145.533	4.11%	0.302	0.302
37	524.030 ± 162.983	507.542 ± 134.957	3.25%	0.311	0.266
38	558.194 ± 157.231	548.504 ± 148.992	1.77%	0.282	0.272
39	594.958 ± 172.625	566.120 ± 151.610	5.10%	0.290	0.268
40	634.470 ± 178.173	599.824 ± 165.272	5.78%	0.281	0.276
41	638.622 ± 183.742	634.850 ± 171.318	0.59%	0.288	0.270
42	676.268 ± 188.897	675.190 ± 181.503	0.16%	0.279	0.269
43	723.706 ± 200.559	700.890 ± 184.341	3.25%	0.277	0.263
44	768.130 ± 204.703	736.860 ± 183.127	4.24%	0.266	0.248
45	784.818 ± 228.262	751.108 ± 204.712	4.49%	0.291	0.273
46	814.578 ± 227.979	800.156 ± 211.897	1.80%	0.280	0.265
47	863.886 ± 236.384	857.376 ± 214.573	0.76%	0.274	0.250
48	892.408 ± 240.167	875.020 ± 222.680	1.99%	0.269	0.255
49	933.238 ± 233.251	903.270 ± 231.230	3.32%	0.250	0.256
50	970.620 ± 254.967	960.984 ± 234.036	1.00%	0.263	0.243
51	1024.944 ± 263.181	998.240 ± 243.115	2.68%	0.257	0.244
52	1041.664 ± 254.046	1028.558 ± 251.407	1.27%	0.244	0.244

n	LO	H.	Adv.	Rel SD LO	Rel SD h.
53	1084.788 ± 291.656	1072.108 ± 260.001	1.18%	0.269	0.243
54	1155.574 ± 289.462	1130.680 ± 278.674	2.20%	0.251	0.246
55	1177.862 ± 276.582	1156.378 ± 255.059	1.86%	0.235	0.221
56	1210.000 ± 282.884	1177.922 ± 277.622	2.72%	0.234	0.236
57	1264.870 ± 300.372	1220.366 ± 272.436	3.65%	0.238	0.223
58	1306.196 ± 296.951	1291.686 ± 295.628	1.12%	0.227	0.229
59	1356.370 ± 319.237	1332.690 ± 295.024	1.78%	0.235	0.221
60	1406.206 ± 331.049	1376.612 ± 329.040	2.15%	0.235	0.239
61	1426.620 ± 333.408	1410.034 ± 337.567	1.18%	0.234	0.239
62	1530.988 ± 367.547	1465.888 ± 318.839	4.44%	0.240	0.217
63	1525.422 ± 338.358	1537.746 ± 329.047	-0.80%	0.222	0.214
64	1601.854 ± 359.003	1552.214 ± 345.653	3.20%	0.224	0.223
65	1640.820 ± 379.043	1610.160 ± 356.001	1.90%	0.231	0.221
66	1676.506 ± 382.343	1665.882 ± 383.157	0.64%	0.228	0.230
67	1731.672 ± 398.312	1729.554 ± 366.457	0.12%	0.230	0.212
68	1795.718 ± 396.543	1784.460 ± 377.619	0.63%	0.221	0.212
69	1871.666 ± 416.838	1847.394 ± 402.277	1.31%	0.223	0.218
70	1886.518 ± 405.619	1866.500 ± 416.876	1.07%	0.215	0.223
71	1950.272 ± 425.311	1961.024 ± 428.716	-0.55%	0.218	0.219
72	2031.002 ± 420.325	2016.996 ± 434.788	0.69%	0.207	0.216
73	2080.514 ± 437.115	2077.114 ± 462.841	0.16%	0.210	0.223

Table E.2: Simulation results for standard selection