On the Runtime Dynamics of the Univariate Marginal Distribution Algorithm on Jump Functions

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Václav Hasenöhrl

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

Dr. Andrew Sutton

May 2018

Acknowledgements

I would like to express my appreciation to Dr. Andrew Sutton for his support and patience as my advisor. His critiques and advice were valuable assets while working on this thesis. My thanks should also be extended to the faculty and staff of the department of Computer Science at University of Minnesota Duluth. Together they provided a daily support and guidance that made for a great learning environment. I would also like to thank Dr. Pete Willemsen and Dr. Bruce Peckham for serving on my committee. I would like to give special thanks to my family and closest friends. I am grateful for their constant support and encouragement.

Abstract

Solving jump functions by using traditional evolutionary algorithms (EAs) seems to be a challenging task. Mutation only EAs have a hard time flipping the right number of bits to generate the optimum. To optimize a jump function, an algorithm must be able to execute an initial hill-climbing phase, after which a point across a large gap must be generated. We study a family of EAs called estimation of distribution algorithms (EDAs) which works differently than standard EAs. In EDAs, we do not store the actual bitstrings, but rather a probability distribution that is initially uniform and should evolve to a model that always generates the global optimum.

We study an EDA called Univariate Marginal Distribution Algorithm (UMDA) and analyze it on jump functions with gap k. We show experimental work on runtimes and probability of succeeding to solve the jump function for different values of k. We take an innovative approach and modify the UMDA by turning off selection. For this new algorithm we present a formal analyses in which, if certain conditions are met, we prove an upper bound on generating the optimum all 1s bistring. Lastly, we compare our results with a different EDA called the compact Genetic Algorithm (cGA) analyzing the jump function. We mention pros and cons of both algorithms under different scenarios.

Contents

List of Tables

List of Figures

1 Introduction

Evolutionary Algorithms (EAs), sometimes also called Genetic Algorithms (GAs), have been used both in practice and theory as optimization tools for decades. EAs represent a robust method for optimizing as they do not require a lot of knowledge about the underlying fitness function that is being optimized. In this thesis, we focus on analyses of special class of evolutionary algorithms called Estimation of Distribution Algorithms (EDAs). While the traditional EAs store actual individuals from the given state space and use mutation or crossover to generate new offspring, EDAs store a probability vector that is used to generate offspring instead. This gives EDAs various advantages over the standard EAs. We discuss this more in detail in Chapter 2.

We analyze an estimation of distribution algorithm called the Univariate Marginal Distribution Algorithm (UMDA) and compare it to another EDA called the Compact Genetic Algorithm (cGA). The difference between these two algorithms is in their update policy of the probability vector. The fitness function we use for our analysis is called Jump. The class of jump functions was introduced in the context of analyzing situations where crossover could be beneficial. Jump functions are specific because they exhibit a gap of length k in the fitness values. They are divided into two phases. First, there is an initial hillclimbing phase in which the algorithm needs to climb towards the gap. This phase is the same as in other very frequently used fitness function called OneMax. After that the algorithm needs to jump across a 'valley' of points by flipping the right k bits to generate the global optimum. This becomes very hard for traditional EAs as they usually tend to fail to flip multiple bits per iteration. However, given the nature of EDAs, flipping multiple bits at once is not an issues. If the individual marginal probabilities of the probability vector remain high enough and tightly concentrated around their mean value we can easily generate the optimum if given reasonable amount of time.

We present both theoretical and experimental analysis and posit an upper bound for the UMDA (Chapter 4) on the Jump fitness function. We mention a result about the cGA analysis on Jump from a recent paper [1] in Chapter 3. From our comparison between the cGA and the UMDA we draw a conclusion and suggest what algorithm performs better under what circumstances. We also introduce a new innovative updating policy through a modified version of the UMDA. We call this new algorithm the Selection Free Univariate Marginal Distribution Algorithm. What is different about this algorithm is that we turn off selection after a certain number of steps to preserve diversity throughout the run. There are other diversity-preserving mechanisms, i.e. *deterministic crowding* or *fitness sharing* but we use this one, as it is easy to extend the cGA analysis. This leads to an upgrade in maximum gap lengths for which the UMDA is able to solve the Jump fitness function.

2 Background

2.1 Evolutionary and Genetic Algorithms

Before we jump into the actual topic of this work, we would like to spend some time on a general background of Evolutionary Algorithms (EAs). As the name suggests, evolutionary algorithms have their roots in natural biological evolution. Through an evolving process we can model and study a certain problem. In biology, this approach could be used to study the actual evolution itself. However, in computer science the focus is different. Computer scientists treat an evolutionary algorithm as a random process which can be applied to a specific problem to find a solution. As in biology, the evolution favors the fittest, in computer science EAs are used to search for improvement. Specifically, even though evolutionary algorithms have many applications, the most common usage is for optimization. The origins of EAs are dated to the middle of 20th century. Since then, a lot of different algorithms have emerged. For the purposes of this chapter we only provide basic preliminaries involving simpler algorithms. This should make it easy to understand the important concepts of evolutionary algorithms.

The interest to use evolutionary algorithms as a tool for optimization stems from their overall simplicity and robustness. If we decide to optimize a problem (function) with a problem-tailored algorithm, we need to exploit the properties of this function and make all kinds of explicit assumption before hand. With this approach, we might obtain a perfect solution to the problem but at the cost of the algorithm used being complex and possibly hard to understand. Moreover, sometimes domain-specific knowledge is costly and unavailable, or the quality of a potential solution is only available after an expensive simulation. However, this is not true about evolutionary algorithms. With EAs we do not need to know a lot about the underlying function being optimized. Normally, the only thing required is to be able to first, represent the domain of a function and second, evaluate such function. This area of optimization is called black-box optimization. Unlike with problem specific algorithms, EAs are usually easy to implement, but might not produce the best or even any solution at all. When we say solution, we mean an optimum (maximum or minimum) regarding an optimization problem. What we expect is a good solution within a reasonable amount of time. As mentioned before, there have been many different algorithms introduced so far. One usually creates a new algorithm with some interesting idea in mind and only after that tries to find problems which could be solved with this new approach. This implies that the research of evolutionary algorithms is empirical. Nonetheless, this does not mean that EAs are not studied theoretically. Their practical applications justify the needs of theoretical work after all as well as the fact that their working principles are not well-understood. However, the theoretical work can sometimes be very challenging.

This being said, researchers tend to use simple frameworks which make it easier to analyze algorithms. The reason for this is that these frameworks help to tease out the general working principles of EAs without bogged down in messy details. We normally work in a space of binary strings (bitstrings), where the bitstrings of length n are vertices of the *unit hypercube* $\{0,1\}^n$. We will use the set $\{0,1\}^n$ for our formal analysis as well. We might also refer to bitstrings as *individuals.* Another important thing to mention are the underlying functions used for analyses. We use the notation from biology and call these *fitness functions.* As natural evolution favors the fittest individuals, optimizing a fitness function is almost the same with one difference - biology does not require any underlying knowledge to optimize fitness. In black-box optimization we traditionally maximize or minimize a fitness function and the optimum then can be interpreted as the fittest individual found. There is a variety of fitness function classes, i.e. linear functions, unimodal functions, functions of unitation, etc. For the purposes of this thesis we work with the functions of unitation. See the following definition.

Definition 2.1. A function $f : \{0,1\}^n \to \mathbb{R}$ is a *function of unitation* when $f(x)$ depends only on the number of ones in the bitstring x and is non-negative.

We denote the number of ones in a bitstring x by $|x|$ and the ith bit by $x[i]$. Not surprisingly, there are several different well-known functions of unitation. We will mention two examples - $\textsc{OneMax}(x)$ and $\textsc{Jump}_k(x)$. The first mentioned function is the simplest function which is used for both theoretical and empricial analyses. It is defined as the number of ones in a bitstring x. See Definition 2.2 and Figure 2.1.

Definition 2.2. Let function $f : \{0,1\}^n \to \mathbb{R}$ be defined as

$$f(x) = \sum_{i=1}^{n} x[i] = |x|$$

for all $x \in \{0,1\}^n$. Then, f is called $\textsc{OneMax}$fitness function.

The focus of this work is the $\textsc{Jump}_k(x)$ fitness function, described in Definition 2.3. The $\textsc{Jump}_k$ fitness function is characterized by a gap of length k. We denote the set of individuals which belong to this gap by $\mathcal{G}$. In other words, individual $x \in \mathcal{G}$ if $n - k < |x| < n$. Notice that there is an initial gradient stage where the algorithm needs to climb up as it does in the $\textsc{OneMax}$and then it needs to flip exactly k bits to jump over a potentially large gap. This is normally hard for many black-box

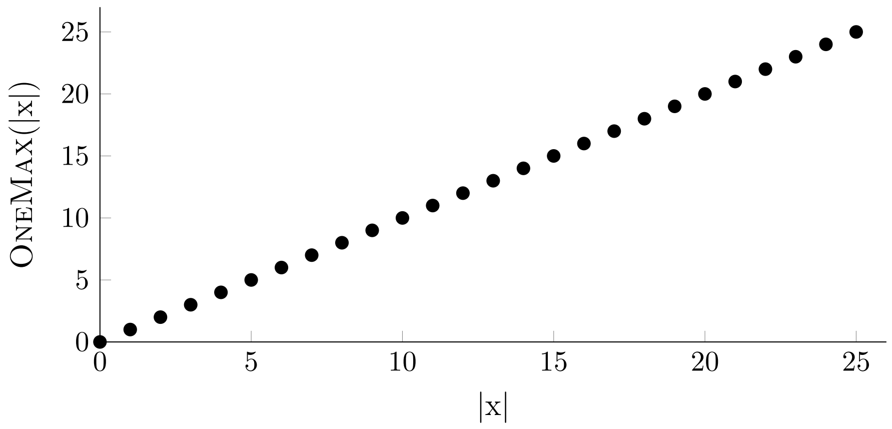


Figure 2.1: The OneMax: $\{0,1\}^n \to \mathbb{R}$ fitness function for $n = 25$.

optimization algorithms, especially as the gap length gets bigger. See Figure 2.2 for an example.

Definition 2.3. Let function $f : \{0,1\}^n \to \mathbb{R}$ be defined as

$$f(x) = \begin{cases} k + |x| & \text{if } |x| \leq n - k \text{ or } |x| = n, \\ n - |x| & \text{otherwise} \end{cases}$$

for all $x \in \{0,1\}^n$. We call f the Jump_k fitness function parameterized by k.

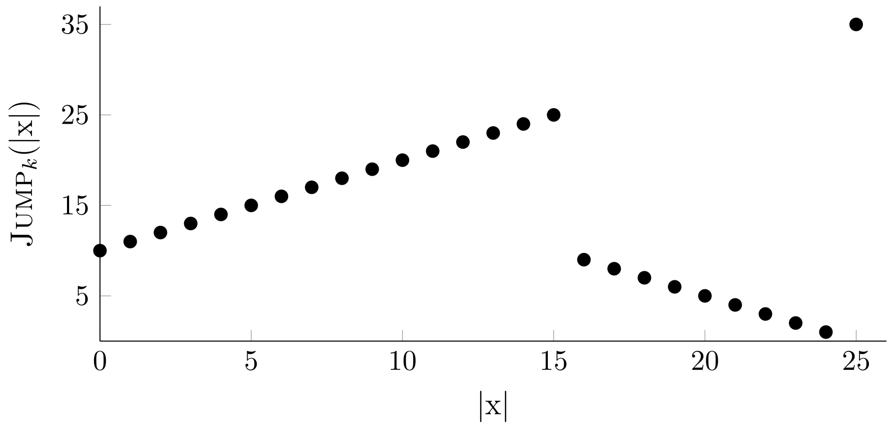


Figure 2.2: The Jump_{10}: $\{0,1\}^n \to \mathbb{R}$ fitness function for $n = 25$ and $k = 10$.

Jump functions were originally introduced as benchmarks on which recombinant evolutionary algorithms can provably outperform those that use mutation alone. Specifically, whether crossover even did anything provably useful. Standard EAs mix mutation and crossover to achieve the initial climb and then crossing the gap. Note that features as mutation and crossover will be discussed in Section 2.1.1 further in this chapter. We will also mention additional preliminaries of the Jump_k fitness function in Section 2.3. Before we start talking about some specific evolutionary algorithms, we want to provide the reader with some ideas behind EAs. Let us consider the following example.

Example 2.1. Imagine that the given fitness function $g : \{0,1\}^{10} \to \mathbb{R}$ is OneMaxand that we would like to maximize this function. This means that the solution for this problem is an all 1s binary string of length 10 yielding the maximum of $g(x) = 10$. The solution for this example is rather straightforward, but how can we find it using EAs? As is typical of many evolutionary algorithms, we initialize a bitstring (or a population of bitstrings) uniformly at random (u.a.r.). We will do the same and start with a random bitstring x generated u.a.r. Then, we will do the following. Choose a position in a bitstring x uniformly at random and flip a bit in that position to create a new bitstring y. After that, if $f(y) \geq f(x)$ let $x \leftarrow y$ otherwise discard y. We repeat the process until the solution is generated. Given the properties of OneMax, we are guaranteed to generate the optimal binary string eventually.

In Example 2.1, we showed how to maximize the OneMaxfitness function with a basic approach. The algorithm we used is a "Hello World" of EAs called *Random Local Search* or *Stochastic Hill-climber* and is described in Algorithm 1. Even though the example we present is simple, it brings up several important points we want to make. One is the terminology - the bitstrings x and y are called the *parent* and the *offspring*, respectively. While generating new offspring we denote the tth generation by $x^{(t)}$ or $y^{(t)}$. The initial population size is normally denoted by μ while the number of offspring generated is λ. Second, the operation we used to create new offspring is called *mutation* and the algorithm's *termination criterion* was finding the solution (maximum). We will discuss these modules in the section 2.1.1. Lastly, how do we measure the performance of an evolutionary algorithm? One way is to run the algorithm and time it. This does not account for implementation or system details and does not tell us how an algorithm is likely to generalize. Another was is to describe the number of calls to the fitness function as a random variable and analyze it. For our purposes, we count the total number of the fitness function evaluations. This is a standard approach because in practice, the fitness function evaluation is usually the most costly operation.

Algorithm 1: The Random Local Search (Hill-climber)

```
1 Choose $x \in \{0,1\}^n$ u.a.r.;
2 while termination criterion not met do
3     Create $y$ by flipping exactly one bit in $x$ u.a.r.;
4     if $f(y) \geq f(x)$ then $x \leftarrow y$;
```

In the Hill-climber algorithm one cannot generate an offspring that differs in more than one position than its parent. The difference in fitness between two bitstrings is formalized by the following definition.

Definition 2.4. Let $x, y \in \{0,1\}^n$. We denote the ith bit in x, y as $x[i]$ and $y[i]$, respectively. Then,

$$H(x,y) = \sum_{i=1}^{n} (x[i] + y[i] - 2x[i]y[i])$$

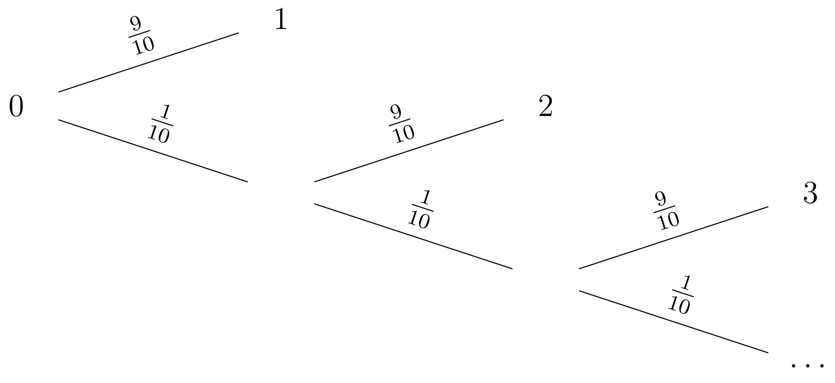
is called the *Hamming distance*. Moreover, the set

$$N(x) = \{x' \in \{0,1\}^n \mid H(x,x') = k\}$$

is called the *k-Hamming neighborhood*. If $k = 1$ then we call that the *direct Hamming neighborhood*.

This definition might be useful in later sections when we perform an analysis of certain algorithms. Before we move onto a description of the modules of EAs we show an additional example.

Example 2.2. Consider the same setup as in Example 2.1 with one difference. Instead of initializing the bitstring x u.a.r., we set x to be the all 0s bitstring. The goal is to compute how many generations it takes on average to generate the solution - all 1s binary string. In this case, the number of generations is also equal to the number of fitness function evaluations. For $t = 0$ we can flip any bit in x to generate y and we are guaranteed that $f(y) > f(x)$. However, for $t > 0$ the situation is different. In case we choose to flip a bit i such that $x[i] = 1$, we discard offspring y because $f(y) < f(x)$. The following probability tree



shows the situation when there is exactly one position j in x such that $x[j] = 1$. We have $\frac{9}{10}$ chance to flip a bit i such that $i \neq j$ and $\frac{1}{10}$ for $i = j$. The nodes denote the number of generations needed to flip a bit in x so that the new offspring y has larger fitness value. The tree keeps growing infinitely in the described manner. We denote the stochastic process described in the diagram by a random variable X_a, where $a = 9$. The index a represents the number of bits remaining to flip. Then,

$$\mathbb{E}[X_9] = 1\frac{9}{10} + 2\frac{1}{10}\frac{9}{10} + 3\frac{1}{10}\frac{1}{10}\frac{9}{10} + \dots$$
$$\mathbb{E}[X_9] = \sum_{i=1}^{\infty} i\frac{9}{10}\left(\frac{1}{10}\right)^{i-1}$$
$$\mathbb{E}[X_9] = \frac{10}{9}.$$

This means that it takes $\frac{10}{9}$ generations on average to flip a bit which was not flipped at $t = 0$. We can generalize the formula to

$$\mathbb{E}[X_a] = \sum_{i=1}^{\infty} i\frac{a}{10}\frac{10-a}{10}^{i-1}$$

where a is the number of all the remaining bits which need to be flipped. If we consider a general bitstring of length n and sum over all bits (all positions must be flipped from zero to generate the maximum) for $i = 1, \dots, n$ we obtain the following formula

$$\mathbb{E}[X] = \sum_{j=1}^{n} (\mathbb{E}[X_j]) = \sum_{j=1}^{n} \left(\sum_{i=1}^{\infty} j\frac{i}{n}\left(\frac{n-j}{n}\right)^{i-1} \right) = \sum_{j=1}^{n} \left(\sum_{i=1}^{\infty} j\frac{i(n-j)^{i-1}}{n^i} \right) = nH_n$$

where X denotes the random variable of the time it takes to flip all bits from 0 to 1 using the Hill-climber algorithm. H_n is called the nth harmonic number [2]. From here, we can conclude that the expected running time of the simple Random local search is $\mathcal{O}(nH_n)$ since there is only $\mathcal{O}(1)$ calls to the fitness function in each step.

Example 2.2 shows an analysis of the Hill-climber algorithm. Normally, the analyses of different algorithms might be more difficult, but we can use this as a building block for more complex tasks. Note that the analysis presented in Example 2.2 mimics the approach of analyzing the Coupon Collector's problem [3]. The Coupon collector's problem is an example where we need to collect all coupons to win a contest. Specifically, imagine an urn from which coupons are being drawn. There is exactly one coupon of each kind and every time you draw a coupon you put it back in the urn. The question the problem asks is the following: How many times do you have to put your hand in the urn to draw a coupon so that you see every kind of coupons at least once? If we denote this random process by Y then the expected value is exactly $\mathbb{E}[Y] = nH_n$, the same expectation as the process discussed in Example 2.2. The value of $\mathbb{E}[Y]$ can also be expressed as

$$\mathbb{E}[Y] = nH_n = n \log n + \gamma n + \frac{1}{2} + \mathcal{O}\left(\frac{1}{n}\right),$$

where $\gamma \approx 0.5772$ is the Euler-Mascheroni constant [4].

2.1.1 Modules of Evolutionary Algorithms

One of the most important features of evolutionary algorithms is how offspring are generated. In every generation, we might create one or more offspring. In other words, we compute a sequence $P_1, P_2, \ldots, P_t$ of multisets (*populations*) of bitstrings. In order to do that, we use the following modules: *selection*, *mutation* and *crossover*. We provide brief description of these modules but one can find more details in [5].

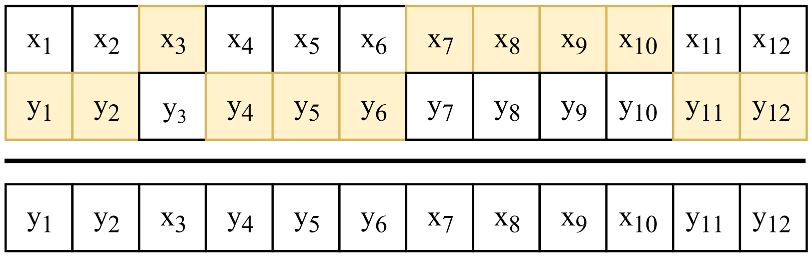
The first module we start describing is *selection*. The selection mechanism appears twice in offspring generation - for reproduction and for replacement. The selection for replacement is usually influenced by the fact that we are trying to maximize the fitness functions. Therefore, we replace the parent(s) with the fittest offspring. If there are ties in fitness we resolve them by preferring the offspring over the parents and then, if there are unresolved ties among the offspring, we break them uniformly at random. When it comes to the selection for reproduction there are several approaches, i.e. *uniform selection*, *fitness proportional selection*, *tournament selection*, *roulette selection*, *stochastic universal sampling*, etc. [6, 7, 5]. Uniform selection is the simplest type of selection and it selects an individual bitstring uniformly at random. Uniform selection is widely used in evolutionary algorithms for its simplicity. Since the goal of an EA is to maximize a given fitness function f, one can prefer different type of selection - fitness proportional selection. In this case, an individual bitstring x is selected from the population P with probability

$$\frac{f(x)}{\sum_{i \in P} f(i)}.$$

An obvious drawback of fitness proportional selection is that if the differences between the fitness values are large, then the fittest individuals will be highly favored. This might lead to almost deterministic behavior. On the other hand, if the fitness values are similar, fitness proportional selection will behave almost as uniform selection. There are variants of the fitness proportional selection, for more details refer to [5].

After we select the parent bitstring we can start modifying it. The classical operation used to produce offspring in most EAs is *mutation*. We only mention mutation operators for the search space $\{0,1\}^n$. First, *standard bit mutation* involves creating an offspring y by flipping each bit in an individual x independently with probability p_m (the parameter p_m is called the *mutation probability*). The most common mutation probability is $p_m = 1/n$. There is both experimental and theoretical support for this. On linear functions, using any $p_m = \frac{c}{n}$ does not affect asymptotic runtime [8], but $c = 1$ seems to be optimal on linear functions [9] when considering exact leading terms (referring to a fitness function called Leading Ones). Moreover, this choice of $p_m = \frac{1}{n}$ leads to an average Hamming distance between x and y to be $H(x,y) = 1$. Generally, we favor small changes in the parent bitstring x because otherwise the mutation can be too aggressive and more likely to produce less fit offspring. We can choose $p_m \in (0, 1/2]$ where if $p_m = 1/2$, then the offspring y is generated uniformly at random. Another version of mutation is *b-bit mutation* where we create an offspring y from a parent x by flipping exactly $b \in \{1, 2, \ldots, n\}$ bits. The bits are normally chosen u.a.r. As in the previous case, since we prefer small changes in x, b is usually small.

The last module of evolutionary algorithms is *crossover*. From a biological point of view, crossover is an analogy to an offspring sharing genes from both parents. Normally, we use two parent individuals but there are some crossover operators which utilize more parents. One example of crossover is *uniform crossover* - each bit is copied from one of the parents and the choice between parents is made uniformly at random. The second crossover operator we mention is *k-point crossover*. In this case, we select k different bit chunks where one chunk is copied from one of the parents while the next chunk is copied from the other parent. See an example for $n = 12$ and $k = 5$ in Figure 2.3. Note that the described crossover methods create a single offspring. If



x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12
y1 y2 y3 y4 y5 y6 y7 y8 y9 y10 y11 y12
y1 y2 x3 y4 y5 y6 x7 x8 x9 x10 y11 y12

Figure 2.3: Example of a k-point crossover for $n = 12$ and $k = 5$ generating one offspring.

we wanted to produce two offspring instead of one, we would use the unused chunks (bits) to produce the additional offspring. When crossover is applied, the offspring's bits should match with both parents at all positions where the parents are equal. It might be desired not to use crossover for every generation. In such case, we define the probability p_c with which we perform crossover. The crossover probability p_c is normally a large constant value, i.e. $p_c >= 0.5$.

Now, when we know how to produce offspring in evolutionary algorithms we must establish the stopping point of an algorithm. Such stopping point is called the *termination criterion*. Normally, we distinguish between *fixed termination criteria*, *adaptive termination criteria* and *no termination criteria*. For fixed termination criteria, we usually have a predefined number of generations or the fitness function evaluations to be the stopping point. Adaptive termination criteria is more flexible. A typical example is when certain predefined fitness value is found. This value is typically the maximum, which is the solution of an optimization problem. We will use this kind of termination criterion throughout this thesis and will be mostly interested in what is the expected time before the optimum is generated (refer to Definition 2.5). Lastly, no termination criteria means that the algorithm runs forever. This approach is generally only theoretical.

Definition 2.5. The *expected running time* $E(T)$ of an algorithm $\mathcal{A}$ on a fitness function f is defined as expectation of the random variable that counts the number of function evaluations performed during the run of the algorithm $\mathcal{A}$.

2.1.2 Examples of Evolutionary Algorithms

The previous section summarized the modules used in EAs to produce offspring. Therefore, we can now move on to presenting some specific examples of evolutionary algorithms. The following table shows the notation used in the algorithms. A canonical example of an evolutionary algorithm is the $(\mu + \lambda)$ EA. In this algorithm, we keep a population P of μ individuals. In each generation, we create λ offspring using the standard bit mutation with mutation probability $p_m = 1/n$. Then, we choose

n	Dimension of the search space $\{0,1\}^n$
P	Parent population
μ	Parent population size
λ	Offspring population size
p_c	Crossover probability
p_m	Mutation probability

Table 2.1: Parameters notations used in the described evolutionary algorithms

the new generation to be the μ fittest individuals among both the offspring and the previous parent generation. We break ties uniformly at random and by preferring the offspring. This approach is called the *truncation selection* [10]. The pseudocode is shown in Algorithm 2.

Algorithm 2: The $(\mu + \lambda)$ EA

```
1  $t \leftarrow 0$;
2  $P_t \leftarrow \mu$ elements of $\{0,1\}^n$ u.a.r.;
3  while termination criterion not met do
4      $P' \leftarrow \emptyset$;
5      for $i \in \{1, \ldots, \lambda\}$ do
6          Select $x \in P_t$ u.a.r;
7          Create $y$ by flipping each bit of $x$ independently with probability $1/n$;
8          $P' \leftarrow P' \cup y$;
9      Sort all individuals in $P_t \cup P'$ breaking ties by preferring offspring and
        breaking still unresolved ties u.a.r.;
10     $P_{t+1} \leftarrow$ first $\mu$ individuals from $P_t \cup P'$;
11     $t \leftarrow t + 1$;
```

The simplest and most straightforward version of the $(\mu + \lambda)$ EA is the $(1 + 1)$ EA. In this case, we only have one individual bitstring x in the population P and in every generation we only create one offspring y. If $f(y) \geq f(x)$, then y replaces x. Plenty of research has been done using this algorithm [11, 12] and there are known results for several fitness functions. For example, the expected running time for any linear function (OneMaxis also a linear function) is $\Theta(n \log n)$ while for Jump$_k$ it is $\Theta(n^k + n \log n)$ where $k \in \{1, 2, \ldots n\}$. In general, the $(1 + 1)$ EA optimizes any arbitrary fitness function in at most n^n time. A modification to the $(\mu + \lambda)$ EA is the $(\mu + \lambda)$ GA (GA stands for genetic algorithm). The difference between these two algorithms is that the GA also involves crossover. A simplified version of the $(\mu + \lambda)$ GA can be found in Algorithm 3. Other versions of this genetic algorithm include the crossover probability p_c. In such case, we perform crossover with the probability p_c on two parents x_1 and x_2 to produce an individual y and with probability $1 - p_c$, y is selected from the population without the crossover. In both cases, the parents can be chosen either using uniform selection or fitness proportional selection.

Algorithm 3: The $(\mu + \lambda)$ GA

```
1  $t \leftarrow 0$;
2  $P_t \leftarrow \mu$ elements of $\{0,1\}^n$ u.a.r.;
3  **while** *termination criterion not met* **do**
4      $P' \leftarrow \emptyset$;
5      **for** $i \in \{1, \ldots, \lambda\}$ **do**
6          Select $x_1, x_2 \in P_t$ u.a.r;
7          Create $y$ by uniform crossover between $x_1, x_2$;
8          Flip each bit of $y$ independently with probability $1/n$;
9          $P' \leftarrow P' \cup y$;
10     Sort all individuals in $P_t \cup P'$ breaking ties by preferring offspring and
         breaking still unresolved ties u.a.r.;
11     $P_{t+1} \leftarrow$ first $\mu$ individuals from $P_t \cup P'$;
12     $t \leftarrow t + 1$;
```

The Algorithm 3 is also a classical example of evolutionary algorithm which have been widely studied on many different fitness functions [13]. This paper shows that the $(\mu + \lambda)$ GA with uniform crossover and standard bit mutation is at least twice as fast as every evolutionary algorithm that only uses standard bit mutations on the OneMaxfitness function.

We also present a more advanced algorithm called the $(1 + (\lambda, \lambda))$ GA [14]. Refer to Algorithm 4 for the pseudocode. The $(1 + (\lambda, \lambda))$ GA starts by initializing one

Algorithm 4: The $(1 + (\lambda, \lambda))$ GA

```
1  $t \leftarrow 0$;
2  Choose $x \in \{0,1\}^n$ u.a.r.;
3  while termination criterion not met do
4      Mutation phase: Sample $\ell$ from $\mathcal{B}(n,p)$;
5      $P_x \leftarrow \lambda$ elements of $\text{mut}_\ell(x_t)$;
6      Choose $x' \in P_x$ s.t. $\forall v \in P_x : f(x') \geq f(v)$ u.a.r;
7      Crossover phase:
8      $P_y \leftarrow \lambda$ elements of $\text{cross}_c(x, x')$;
9      Choose $y \in P_y$ s.t. $\forall v \in P_y : f(y) \geq f(v)$ u.a.r;
10     Selection step: if $f(y) \geq f(x)$ then $x \leftarrow y$;
11     $t \leftarrow t + 1$;
```

bitstring x uniformly at random. Then, by sampling a random number l from the Binomial distribution $\mathcal{B}(n,p)$, we create λ offspring using a mutation operator mut_l. We choose the fittest individual x' from the λ offspring we created with the mutation (breaking ties u.a.r.) After that, we create λ offspring by applying crossover operator $cross_c(x, x')$. Again, we choose an individual y with the highest fitness value from the λ offspring we created with the crossover (breaking ties u.a.r.) Lastly, if $f(y) \geq f(x)$ then we replace x with y. For more details on the $(1 + (\lambda, \lambda))$ GA, the mutation operator mut_l and the crossover operator $cross_c$ refer to [14]. The Algorithm 4 was very innovative and presented a ground-breaking result. It was proven to optimize OneMaxin runtime $\mathcal{O}(n\sqrt{\log n})$. The traditional bound for black-box optimization algorithms using only unbiased unary operators (operators that modify only one individual) is $\Omega(n \log n)$. This bound is broken because $(1 + \lambda, \lambda)$ GA uses non-unary operations. In general, the $(1 + (\lambda, \lambda))$ GA yields an optimization time of $\mathcal{O}(\frac{1}{\lambda} n \log n + \lambda n)$, which is minimized by $\lambda = \sqrt{\log n}$, leading to the runtime mentioned above.

As mentioned above, we evaluate our algorithms based on the number of fitness function evaluations because it is, in practice, usually the most costly operation.

Let us denote this number by T. With this being said, we would like to look at the performance of the three mentioned algorithms. Do they outperform a basic algorithm such as the simple Hill-climber (Algorithm 1)? This is a very common question in evolutionary algorithm research.

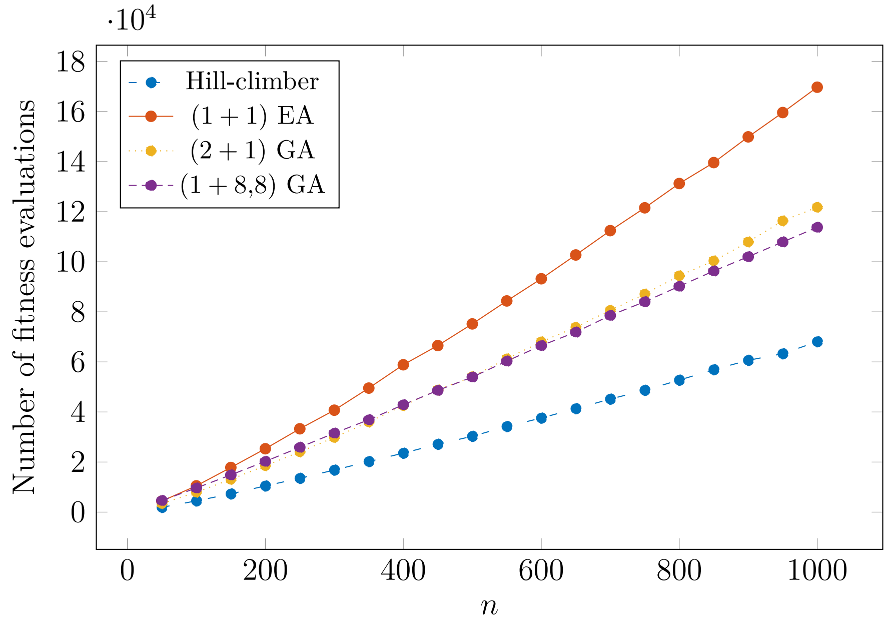


Figure 2.4: Number of fitness function evaluations to find the solution for different bitstring length n for the Hill-climber (Alg 1), the $(1+1)$ EA (Alg 2), the $(2+1)$ GA (Alg 3) and the $(1+\lambda,\lambda)$ GA (Alg 4). We show the mean values of 1000 trials on the OneMaxfitness function.

In the graph above (Fig 2.4), we compare the average number of fitness function evaluations on 1000 trials to solve OneMaxfor the presented algorithms. The parameters chosen for the algorithms were the following. Mutation probability $p_m = 1/n$ for the $(1+1)$ EA and the $(2+1)$ GA. Moreover, for the $(2+1)$ GA, we always performed crossover as described in Algorithm 3. For the $(1+8,8)$ GA, we set $k = \lambda$, $p = k/n$ (mutation probability) and $c = 1/k$ where $\lambda = 8$. Note that for the $(1+8,8)$ GA, we follow the notation from the paper that introduced this algorithm [14]. We can see that the Hill-climber outperforms the other algorithms for smaller values of n. However, for large values of n, the $(1+\lambda,\lambda)$ GA becomes faster given that $\mathcal{O}(n\sqrt{\log n}) < \mathcal{O}(nH_n)$.

2.2 Estimation of Distribution Algorithms

In this part, we move on to the preliminaries of the main topic of this work. In the previous section, we described the traditional evolutionary algorithms that use selection, mutation and crossover. There is a special class of EAs that use different optimization technique called *Estimation of Distribution Algorithms* [15] (EDAs). Sometimes, they are also called *Probabilistic model-building genetic algorithms* (PMBGAs) or *Iterated density estimation evolutionary algorithms* (IDEAs). Recently, EDAs have been applied to many optimization problems (see the references on p. 899 in [15]). In EDAs, we do not store the actual bitstrings, but rather a probability distribution that is initially uniform and should evolve to a model that with reasonable probability generates the global optimum. For each position i, $i \in \{1,\ldots,n\}$ in a bitstring we have a probability, p_i, which tells us how likely it is that a hypothetical bitstring would have a 1 in that position. The probabilities p_i are called *frequencies* or *marginal probabilities*. In each generation, we modify the distribution in a certain way that differs for different algorithms. We will mention two EDAs - *Compact Genetic Algorithm* (cGA) [16] and *Univariate Marginal Distribution Algorithm* (UMDA) [17]. In the following sections we describe the two mentioned EDAs and compare their runtimes to some of the previously mentioned EAs.

2.2.1 Compact Genetic Algorithm

In each generation, the cGA generates two parents x and y of length n from the current version of the probability distribution $p_t = (p_{1,t}, p_{2,t}, \ldots, p_{n,t})$ and then it compares the fitness of both x and y. If $f(x) < f(y)$ then swap x and y. Lastly, for every bit $i \in \{1, \dots, n\}$ if $x_i > y_i$ we set $p_{i,t+1} = p_{i,t} + \frac{1}{K}$ and if $x_i < y_i$ then $p_{i,t+1} = p_{i,t} - \frac{1}{K}$. If $x_i = y_i$ then $p_{i,t}$ remains unchanged so it always moves the probability distribution to increase the likelihood of generating the winner. See the pseudocode of the algorithm in Algorithm 5. Note that the parameter K has a significant impact on the runtime of the cGA. If K is large then the frequencies updates are very small. Therefore, the optimization time can be slow due to that and one might need to try different values of K for different problems to achieve a satisfying performance.

Algorithm 5: The cGA

```
$t \leftarrow 0$;
$p_{1,t} \leftarrow p_{2,t} \leftarrow \dots \leftarrow p_{n,t} \leftarrow 1/2$;
while termination criterion not met do
    for $i \in \{1, \dots, n\}$ do
        $x_i \leftarrow 1$ with probability $p_{i,t}$;
        $x_i \leftarrow 0$ with probability $1 - p_{i,t}$;
    for $i \in \{1, \dots, n\}$ do
        $y_i \leftarrow 1$ with probability $p_{i,t}$;
        $y_i \leftarrow 0$ with probability $1 - p_{i,t}$;
    if $f(x) < f(y)$ then swap $x$ and $y$;
    for $i \in \{1, \dots, n\}$ do
        if $x_i > y_i$ then $p_{i,t+1} \leftarrow \min\{p_{i,t} + 1/K, 1 - 1/n\}$;
        if $x_i < y_i$ then $p_{i,t+1} \leftarrow \max\{p_{i,t} - 1/K, 1/n\}$;
        if $x_i = y_i$ then $p_{i,t+1} \leftarrow p_{i,t}$;
    $t \leftarrow t + 1$;
```

The first rigorous analysis of the cGA [18] presented a lower bound on the running time of the cGA over all fitness functions $f : \{0, 1\}^n \to \mathbb{R}$ and an upper bound for all linear functions over $\{0, 1\}^n$. It was shown that the expected optimization time on ONEMAXis $\mathcal{O}(\sqrt{n}K)$ where $K = n^{1/2+\epsilon}$ and $\epsilon > 0$. However, the probability of such runtime is only at least $\frac{1}{2}$. The cGA was also analyzed in the context of noisy optimization. Noisy optimization means that there is a stochastic factor added to the fitness function. Usually, Gaussian noise with mean 0 is used which yields the fitness to be $f + \mathcal{N}(0, \sigma)$. The authors of [19] proved that the cGA scales *gracefully* with noise, meaning that any polynomial increase in the variance of an additive Gaussian variate only results in a corresponding polynomial slow down in optimization time.

2.2.2 Univariate Marginal Distribution Algorithm

A series of recent papers have supplied upper and lower bounds on the UMDA algorithm optimizing OneMax[20, 21, 22, 23]. In 2017, the author of [20] showed that the expected runtime of UMDA on OneMaxis $\mathcal{O}(\mu\sqrt{n})$ if $\mu \geq c\sqrt{n}\log n$ for a constant $c > 0$ and $\lambda = (1 + \Theta(1))\mu$. This is better that the previously best known upper bound $\mathcal{O}(n \log n \log\log n)$ with $\lambda = \mathcal{O}(\log n)$ by [21] from 2015. The univariate marginal distribution algorithm works differently than the cGA. See the pseudocode of the UMDA in Algorithm 6.

The UMDA samples λ parents from the current probability distribution $p_t = (p_{1,t}, p_{2,t}, \ldots, p_{n,t})$. Then, it chooses the μ fittest individuals breaking ties uniformly at random and updates each p_i using the following formula

$$p_{i,t+1} = \frac{\sum_{j=1}^{\mu} x_i^{(j)}}{\mu}.$$

In other words, $p_{i,t+1}$ is a relative occurrence of 1s in bit position i among the μ chosen individuals. Note that such an update, unlike for the cGA, allows the jumps of the frequencies p_i to be large, i.e. from $\frac{\mu-1}{\mu}$ to $\frac{1}{\mu}$ or vice versa. This means that the UMDA can, in general, exhibit faster optimization time than the cGA (Fig 2.5. This is especially true for large values of λ and μ chosen correspondingly. We can demonstrate this behavior on a simple example.

Algorithm 6: The UMDA

```
1  $t \leftarrow 0$;
2  $p_{1,t} \leftarrow p_{2,t} \leftarrow \ldots \leftarrow p_{n,t} \leftarrow 1/2$;
3  while termination criterion not met do
4      $P_t \leftarrow \emptyset$;
5      for $j \in \{1, \ldots, \lambda\}$ do
6          for $i \in \{1, \ldots, n\}$ do
7              $x_i^{(j)} \leftarrow 1$ with probability $p_{i,t}$;
8              $x_i^{(j)} \leftarrow 0$ with probability $1 - p_{i,t}$;
9          $P_t \leftarrow P_t \cup \{x^{(j)}\}$;
10     Sort all individuals in $P_t$ in descending order breaking ties u.a.r.;
11     for $i \in \{1, \ldots, n\}$ do
12         $r = \frac{\sum_{j=1}^{\mu} x_i^{(j)}}{\mu}$;
13         if $r < 1/n$ then $p_{i,t+1} \leftarrow 1/n$;
14         else $p_{i,t+1} \leftarrow r$;
15         if $r > 1 - 1/n$ then $p_{i,t+1} \leftarrow 1 - 1/n$;
16         else $p_{i,t+1} \leftarrow r$;
17     $t \leftarrow t + 1$;
```

Example 2.3. Consider the UMDA optimizing the OneMaxfitness function with bitstring length $n = 100$. Let the marginal probabilities $p_{i,1} = \frac{1}{2}$ for each $i \in \{1, \ldots, n\}$. This example is supposed to demonstrate how fast can the marginal probabilities change in just one generation.

Part a. Let $\lambda = 400$ and $\mu = 100$. We show (Fig 2.6) the distribution of fitness values among the λ generated individuals. The red part of the bar plot depicts the μ fittest individuals chosen to update the marginal probabilities. See the distribution of the marginal probabilities along with their mean value (0.5679) and standard deviation in Figure 2.7. We can say that on average, the change in the frequencies for this generation is $|0.5 - 0.5679| = 0.0679$. This is almost 7 times faster update compared to the cGA with $K = \frac{1}{n}$. It is not uncommon to choose K to be even smaller. If $K = \frac{1}{n^2}$ then the update would be 700 times faster.

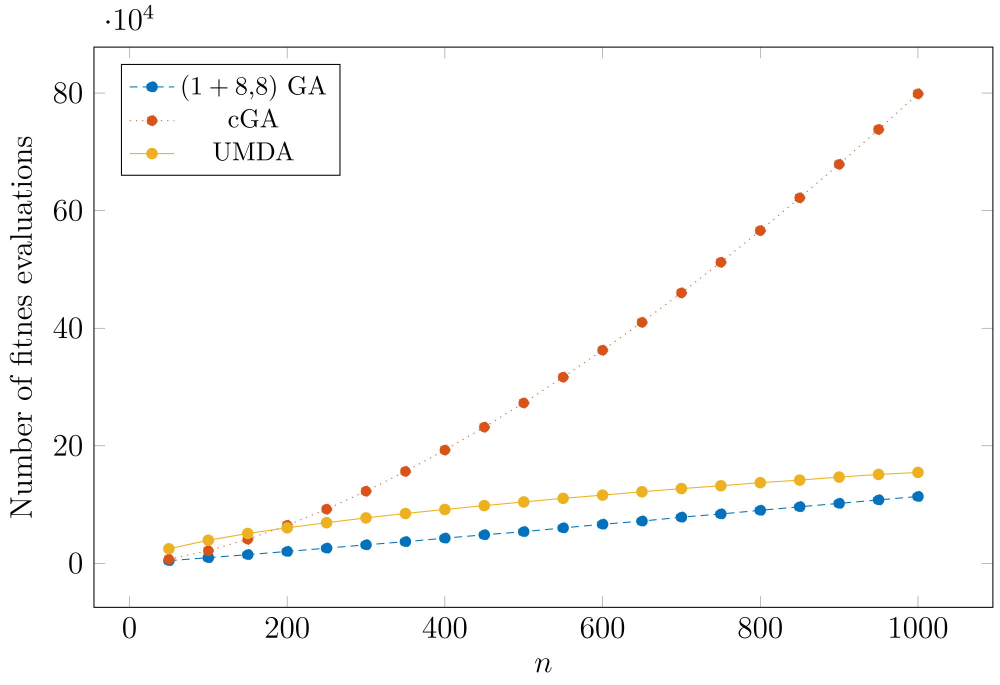


Figure 2.5: Number of fitness function evaluations to find the solution for different bitstring length n for the EDA algorithms - the cGA (Alg 5), the UMDA (Alg 6), and one of the EAs - the $(1+\lambda, \lambda)$ GA (Alg 4). We show the mean values of 1000 trials. For this experiment, we set $K = n$ for the cGA and $\lambda = 400, \mu = 100$ for the UMDA.

Part b. Let $\lambda = 4000$ and $\mu = 100$. We increase λ 10 times compared to the previous part to demonstrate (Fig 2.8) how the distribution of fitness values changes and also to show that we are more likely to generate even fitter individuals leading to even bigger jumps in the marginal probabilities. In this case, we obtain mean value of 0.6153 (Fig 2.9) yielding average frequency change $|0.5 - 0.6201| = 0.1201$. This is 12 times faster update than the cGA with $K = \frac{1}{n}$.

In general, as λ approaches ∞ the distribution of fitness values gets smoother and if we choose μ to be small, we can update the frequencies even faster. We ran an experiment with $\lambda = 5 \cdot 10^6$ and $\mu = 100$ yielding an average change in marginal probabilities of $|0.5 - 0.7107| = 0.2107$. However, choosing λ too large might not be desired. Increasing λ can lead to faster frequencies updates but it also means more

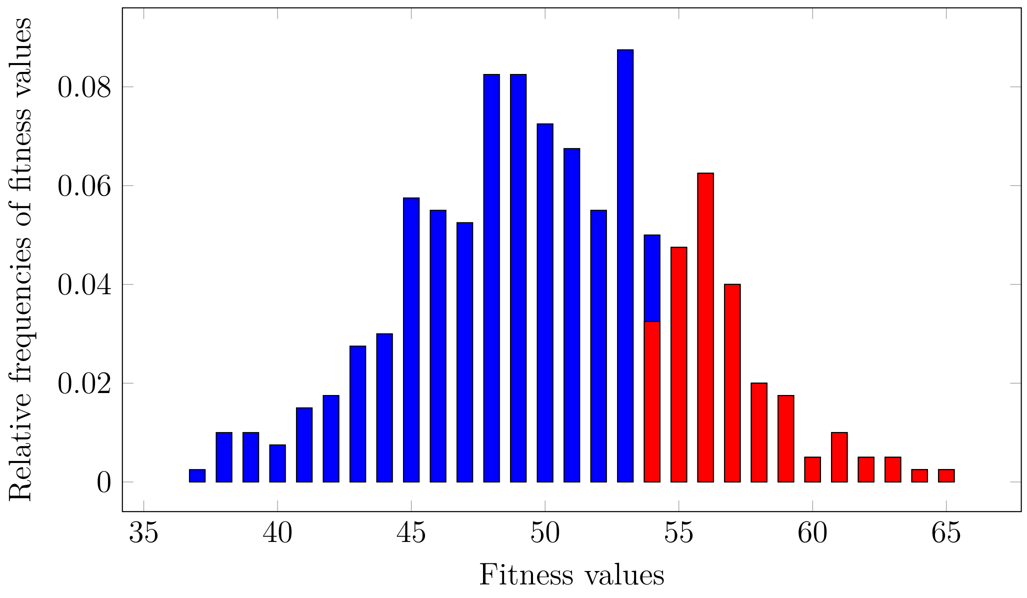


Figure 2.6: The relative frequencies of fitness values among the $\lambda = 400$ individuals generated. Red part denotes the $\mu = 100$ fittest individuals chosen to update the marginal probabilities.

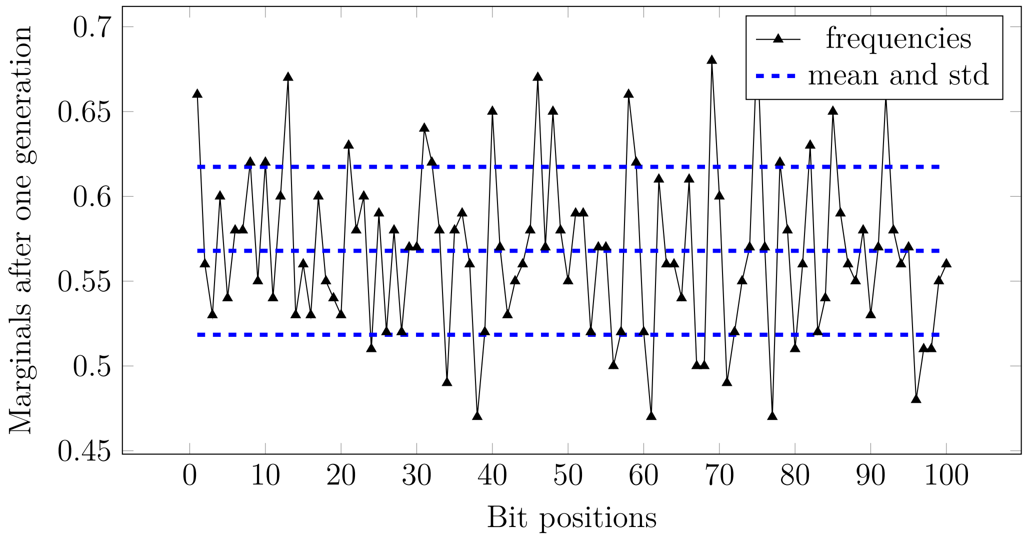


Figure 2.7: The marginal probabilities after one generation starting from the uniform distribution on the OneMaxfitness function. We set $n = 100$, $\lambda = 4000$ and $\mu = 100$ for this experiment.

fitness function evaluations in every generation and that is not desired. Thus, it is important to choose λ carefully, so that it maximizes frequencies updates as much as

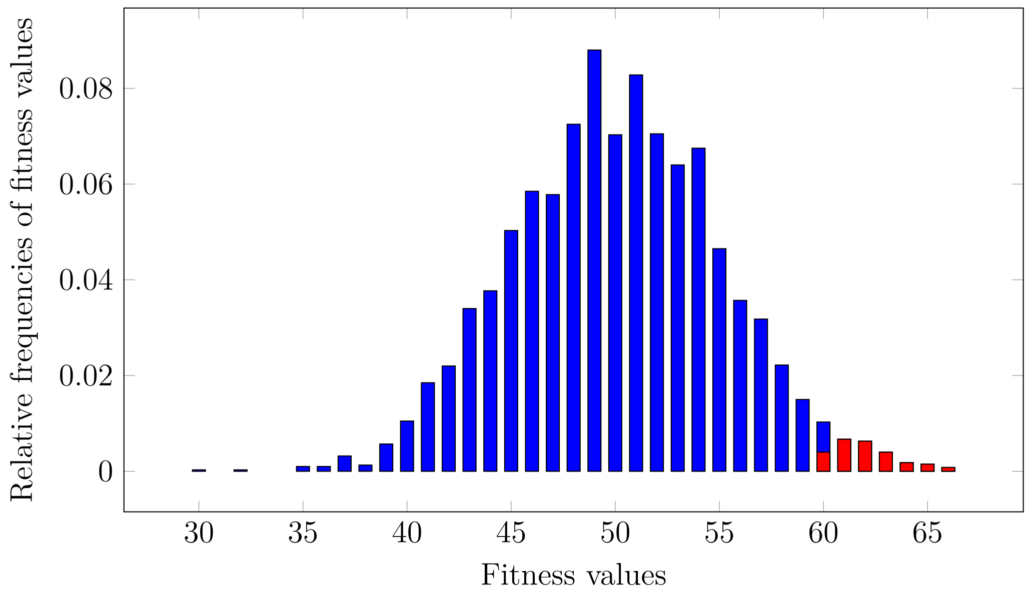


Figure 2.8: The relative frequencies of fitness values among the $\lambda = 4000$ individuals generated. Red part denotes the $\mu = 100$ fittest individuals chosen to update the marginal probabilities.

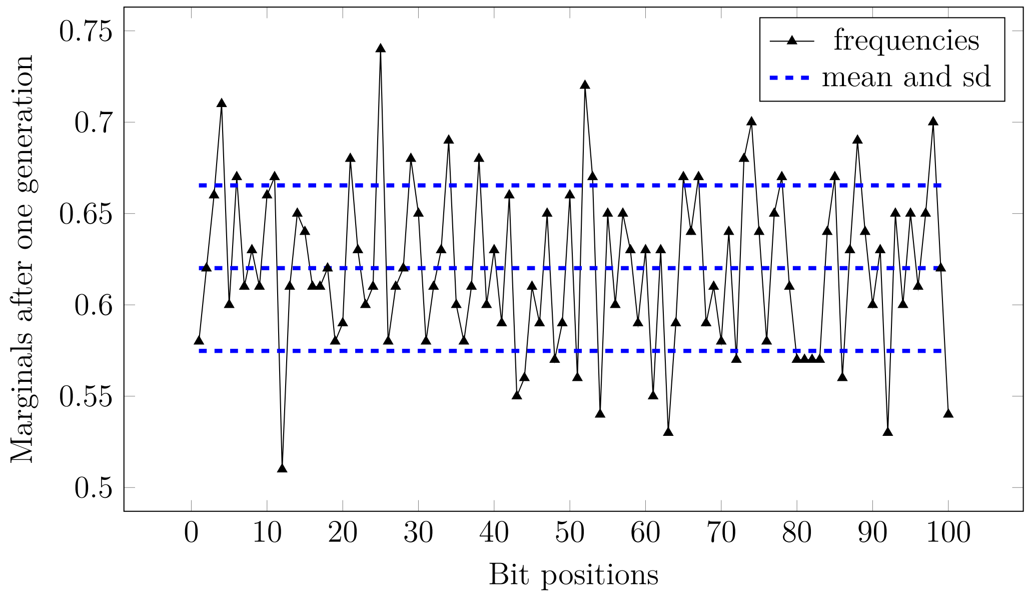


Figure 2.9: The marginal probabilities after one generation starting from the uniform distribution on the OneMaxfitness function. We set $n = 100$, $\lambda = 4000$ and $\mu = 100$ for this experiment.

possible, but does not extensively slow the whole algorithm down overall.

One drawback of the mentioned EDAs is a premature convergence of the marginal probabilities. This can occur when $p_{i,t} \in \{0, 1\}$, but the target solution does not have the corresponding bit in the i-th position. Should this event occur, the cGA or UMDA fail to converge in finite time. A natural way to prevent this is by restricting the marginal probabilities to an interval $[1/n, 1-1/n]$ to ensure there is always a nonzero probability of producing either a 1 or a 0 in position i for all times t. This idea comes from the field of *Ant Colony Optimization* [24]. We follow this procedure and impose this restriction in lines 12 and 13 of Algorithm 5 and 13 and 15 of Algorithm 6, respectively.

We finish this section with the following remark which tells us what is the probability of generating the optimum at generation t.

Remark 2.1. The probability of generating the optimum for both the OneMaxand the Jump$_k$ fitness functions using either the cGA or the UMDA during tth generation is

$$\prod_{i=1}^{n} p_{i,t}. \tag{2.1}$$

Proof. We know that every bit $i \in \{1, \ldots, n\}$ during the tth generation is 1 with probability $p_{i,t}$. Since all bits are independent, the probability of generating the all 1s bitstring is $p_{1,t}p_{2,t} \ldots p_{n,t} = \prod_{i=1}^{n} p_{i,t}$. □

2.3 The Jump fitness function

The main focus of this work is to analyze the two estimation of distribution algorithms on the Jump$_k$ fitness function. We have recently presented an article analyzing the cGA [1]. Optimizing the Jump$_k$ fitness function seems to be especially difficult for some of the traditional EAs. For example, the Hill-climber is not even be able to solve the function - it fails to cross the gap as it can only generate offspring with Hamming distance equal to one from its parent. The other mentioned algorithms can solve the $\textsc{Jump}_k$ fitness function, but it is required that they flip exactly the right k bits when they reach the gap of length k. For large values of k it becomes very unlikely. For example, the $(\mu + 1)$ EA, unlike the Hill-climber algorithm, can reach any point in the search space in a single step. However, the probability decreases rapidly with increasing Hamming distance from the original individual. In fact, it requires $\Omega(n^k)$ steps to solve the $\textsc{Jump}_k$ which can be decreased to $\mathcal{O}(\mu n^2 k^3 + \frac{4^k}{p_c})$ by performing an additional crossover operation with probability $p_c = \mathcal{O}(\frac{1}{kn})$ [1]. The difficulty of crossing large gaps stays true for EDAs as well, but as long as the probability distribution stays compact, we have a bigger chance of generating the solution. This will be discussed in detail in the later chapters.

Definition 2.6. Given a particular jump function $\textsc{Jump}_k$, the set $\mathcal{G}= \{x \in \{0,1\}^n : n - k < |x| < n\}$ is called *gap* points.

Remark 2.2. For any two individuals $x, y \in \{0,1\}^n$, if $x, y \notin \mathcal{G}$, then $|x| \geq |y| \iff \textsc{Jump}_k(x) \geq \textsc{Jump}_k(y)$. Otherwise $|x| \geq |y| \iff \textsc{Jump}_k(x) \leq \textsc{Jump}_k(y)$.

Remark 2.2 says an important fact about the μ individuals that are being chosen through the selection process for the UMDA. As long as λ is large enough and μ is relatively small compared to λ we do not, with high probability, ever accept gap individuals to update the marginal probabilities. In other words, there is always enough 'outside of the gap' individuals. However, in case we ever select a gap individual then it is most likely a bitsring with fitness close the 'edge' value $(n - k)$ due the result of Remark 2.2.

Definition 2.7. Given a product distribution $p_t = (p_{1,t}, p_{2,t}, \ldots, p_{n,t})$, we define the *gap probability* as

$$P_{\mathcal{G}}(t) = \Pr(x \in \mathcal{G} \mid x \sim p_t),$$

i.e., the probability that a bitstring drawn from the product distribution p_t is a gap point.

Membership in $\mathcal{G}$ is uniquely determined by Hamming distance. The Hamming distance of a bitstring drawn from p_t follows the Poisson-Binomial distribution, and so the precise gap probability $P_{\mathcal{G}}$ on the $\textsc{Jump}_k$ can be written down as a function of p_t as follows.

$$P_{\mathcal{G}}(t) = \sum_{l=n-k+1}^{n} \sum_{A \in F_l} \prod_{a \in A} p_{i.t} \prod_{j \in A^c} (1 - p_{j,t}), \tag{2.2}$$

where F_l is the set of all cardinality-l subsets of $\{1, 2, \ldots, n\}$ and $A^c = \{1, 2, \ldots, n\} \setminus A$. The expression 2.2 is rather hard to work with but later on, we will use it to formulate a drift of the cGA.

The last part of this section is dedicated to experiments on the $\textsc{Jump}_k$ fitness function. First, we show runtimes (Fig 2.10 of the algorithms mentioned in the section 2.1. We omit the Hill-climber algorithm (1) because it cannot solve the $\textsc{Jump}_k$ for any value of $k \geq 1$. Note that we are able to generate the all 1s bitstring only for very small gap length k. It is also important to say that the bitstring length n does also influence how far the evolutionary algorithms can jump. EAs need to flip the correct k bits when they reach the gap and the probability to do that increases with n. As we will see later, this is not the case for EDAs. Estimation of distribution algorithms can jump further as n increases. In the experiment depicted in Figure 2.10, the algorithms finished and found the solution in 10^7 fitness function evaluations only for gap length $k \in \{2, 3\}$. For $k \in \{4, 5\}$ the algorithms failed to generate the optimum for some cases in the maximum given time. For $k >= 6$ it becomes very unlikely that any of the EAs finds the solution. The ratios of how many runs finished in the maximum allocated time is depicted in Figure 2.11. Additionally, refer to Table 2.2 for the standard deviations of the runs. For clarity, we chose not to show the standard deviations in the graph. Overall, we can see that the standard evolutionary algorithms using mutation and crossover only are not very efficient in solving the JUMP$_k$ function.

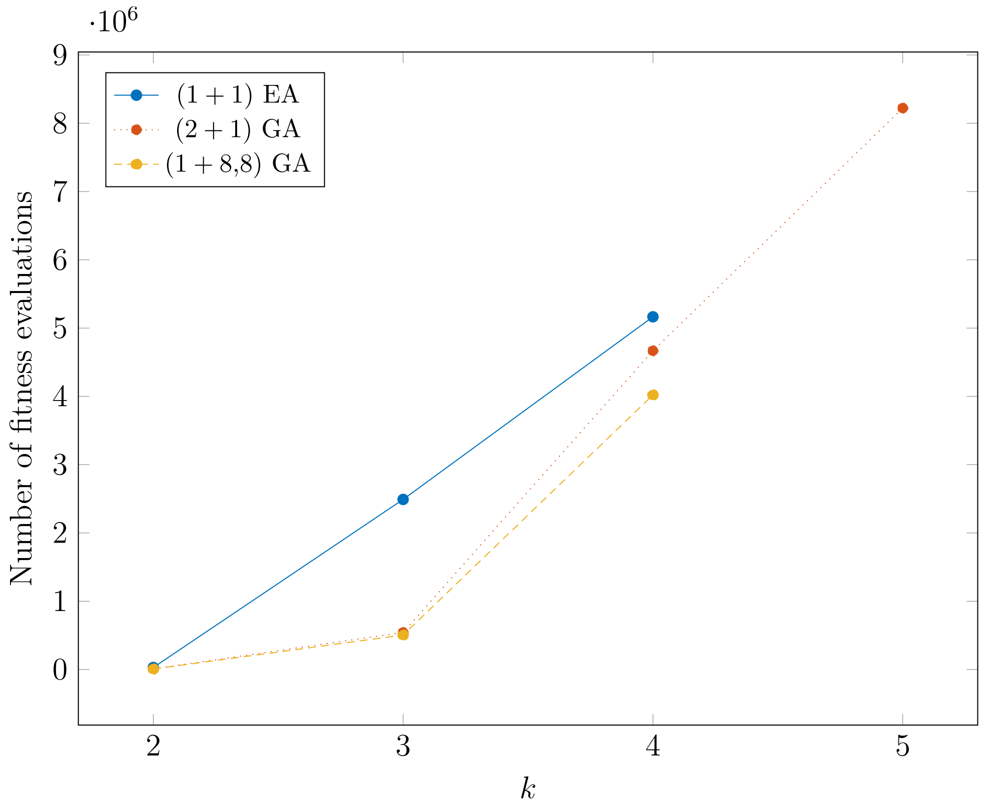


Figure 2.10: Number of fitness function evaluations to find the solution of the JUMP$_k$ for different gap lengths k - the $(1+1)$ EA (Alg 2), the $(2+1)$ GA (Alg 3), the $(1+\lambda, \lambda)$ GA (Alg 4). We show the mean values of 100 trials. For this experiment, we set $n = 100$ and limit the maximum number of fitness function evaluations to 10^7. Missing points are due to failure of the algorithm to finish within the allocated time.

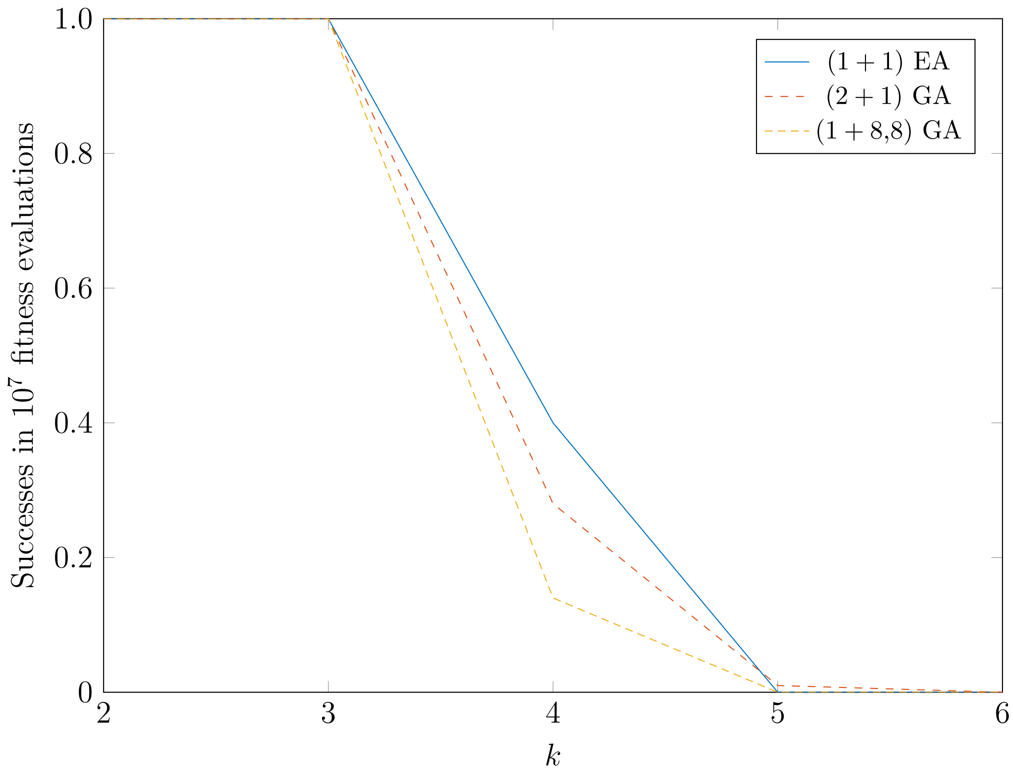


Figure 2.11: Proportion of the $\textsc{Jump}_k$ functions solved in 10^7 fitness function evaluations out of 100 trials for the mentioned EAs. We set $n = 100$.

alg/k	2	3	4	5
$(1+1)$ EA	29510.8 26792.26	2490706.98 231352.05	5165315.750 376196.266	– –
$(2+1)$ GA	9674.16 947.8	542521.15 51191.93	4668166.93 274470.23	8221911.0 0.00
$(1+8,8)$ GA	7496.68 636.48	504824.52 49943.03	4020991.86 325166.05	– –

Table 2.2: Results of the experiment depicted in Figure 2.10. For $n = 100$ we show the mean values (top) and the standard deviations (bottom) of 100 trials for different gap lengths k. Note that for the $(2+1)$ GA and for $k = 5$ only one run finished in 10^7 fitness function evaluations and thus the standard deviation is equal to zero.

3 Analysis of the Compact Genetic Algorithm

Results in this chapter are taken from a recent paper [1] where we provide a detailed analysis of the cGA on the $\textsc{Jump}_k$ function. We prove that the cGA solves the $\textsc{Jump}_k$ in time bounded by $\mathcal{O}(2\sqrt{2}Kn^{3/2}\log(\frac{n}{2}) + e^{\Theta(k)} + n)$ with probability $1 - o(1)$ for $k = o(n)$ and sufficiently large K. When $k = \Omega(\log(n))$, this translates to a super-polynomial speedup over mutation–only EAs (and other hillclimbing algorithms). We briefly mention these results in order to be able to compare behavior of the UMDA to the cGA.

Recall that EDAs maintain a product distribution p_t over $\{0, 1\}$. The cGA generates two offspring x, y in each generation that are then used to update the distribution as follows. Without loss of generality, assume $f(x) \geq f(y)$.

$$p_{i,t+1} = \begin{cases} p_{i,t} + \frac{1}{K} & \text{if } x_i - y_i = 1, \\ p_{i,t} - \frac{1}{K} & \text{if } x_i - y_i = -1, \\ p_{i,t} & \text{if } x_i = y_i. \end{cases}$$

Understanding the way of how the marginal probabilities in the distribution p_t change is the key to analyzing the estimation of distribution algorithms. As mentioned before, at the beginning of optimizing the $\textsc{Jump}_k$ the algorithm needs to climb up as if the function was the classical $\textsc{OneMax}$. This seems to be relatively easy for both the cGA and the UMDA. We call this phase the *subcritical phase.* In this phase, the algorithm generates only (with high probability) non-gap individuals.

Let us focus on the cGA only from now but the notation we introduce in the following section will be used for the UMDA as well. Once the cGA reaches the gap, and starts generating both gap and non-gap individuals, an interesting behavior occurs. Since it cannot simply keep climbing to the optimum, it starts to generate offspring with certain number of 0s in them. Given the property from Remark 2.2, the number of 0s is actually close to k. In other words, the number of 1s in individuals generated once the gap is reached is roughly distributed around $n - k$ (the edge of the gap). We call this stage the *critical phase.* This actually means that since all the frequencies are independent and follow Bernoulli distribution, the actual positions of 0 and 1 bits are also independent. Thus, if we have roughly k zero bits in each offspring, the marginal probabilities stabilize around the ratio

$$\frac{n-k}{n}.$$

The compact genetic algorithm stays in the critical phase for a while and that allows to eventually solve the $\textsc{Jump}_k$ function. The duration of how long it stays in this phase is determined by the parameter K. The bigger it is, the smaller the update $1/K$ is. However, large values of K also slow down the initial climbing phase since it takes smaller steps toward the gap. After the critical phase a *long-range diversity loss* follows. During this phase certain marginal probabilities converge toward 1 while others converge toward 0. We cannot be sure which bits converge to what values exactly but there is roughly $n - k$ bits converging to 1 and k converging to 0. See Figure 3.1 for better visualization and notice that the frequencies are really roughly distributed around the value $\frac{n-k}{n}$ $(200-20)/200 = 0.9)$ at the beginning of the critical phase. Lastly, we confirm through experiments that the mean of all frequencies throughout various runs is approximately $\frac{n-k}{n}$ for the whole critical phase and the long-range diversity loss.

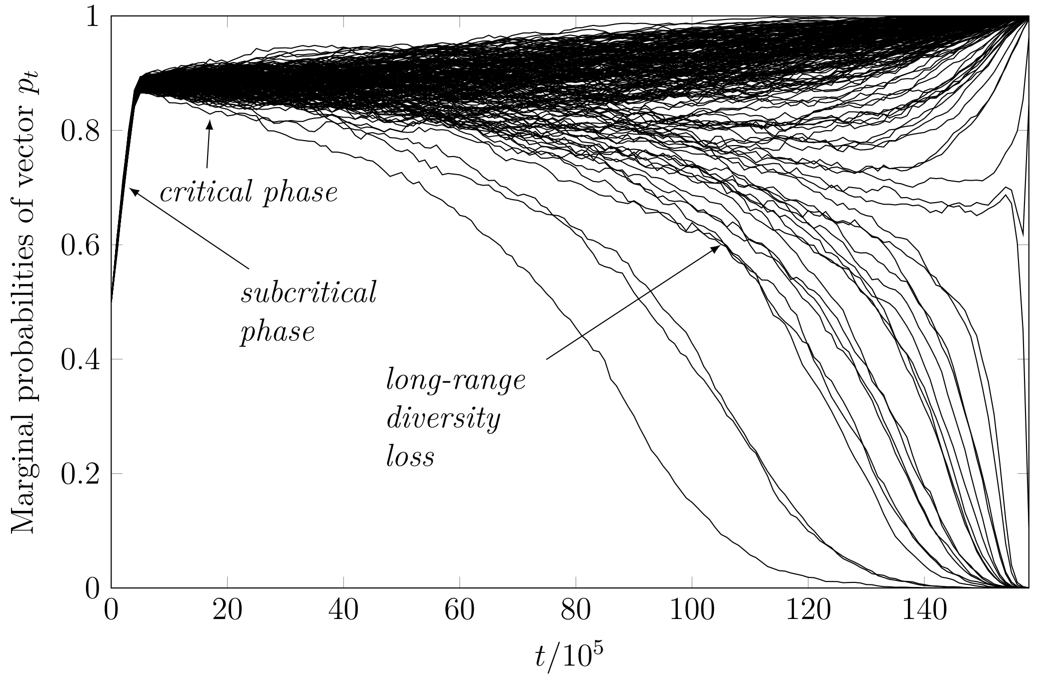


Figure 3.1: The cGA's marginal probabilities of vector p_t over long-range time for $\textsc{Jump}_{20}$ with $n = 200$. The probabilities are tightly concentrated in the subcritical phase while climbing to the gap, then eventually disperse. A run is successful if the optimal solution is generated before the marginals fix.

After understanding how the marginal probabilities change over time during a run of the cGA, we define a *drift* of that process. Bounding the drift of the process allows us to bound the hitting time of the process to a goal that coincides with some state we are interested in. This is possible via various established 'drift theorems' [25]. In general, a (stochastic) drift is the change of an expected value of a stochastic process $\{X_t : t \in \mathbb{N}\}$, i.e. $\mathbb{E}[X_{t+1} - X_t \mid X_t]$. We focus on the dynamics of two important entities during the execution of the cGA. The first is the stochastic process

$\{X_t : t \in \mathbb{N}\}$, defined as the sum of the marginal probabilities

$$X_t = \sum_{i=1}^{n} p_{i,t}. \tag{3.1}$$

Note that X_t is exactly the expected number of 1s of a string drawn from the time-t product distribution p_t. The second value we study is the stochastic process $\{\gamma_t : t \in \mathbb{N}\}$ that captures the absolute concentration of the marginal probabilities around their average value:

$$\gamma_t = \max_i |X_t/n - p_{i,t}|. \tag{3.2}$$

In particular, we show that early in the process, the marginal probabilities remain tightly concentrated.

Lemma 3.1. Let x and y be the offspring drawn from the product distribution in iteration t at lines 4 and 7 of Algorithm 5. Let $\Delta_t = |x| - |y|$. Then $\Delta_t = \Delta_{1,t} + \Delta_{2,t} + \cdots + \Delta_{n,t}$ is the sum of n independent random variables

$$\Delta_{i,t} = \begin{cases} 1 & \text{with probability } p_{i,t}(1-p_{i,t}), \\ -1 & \text{with probability } p_{i,t}(1-p_{i,t}), \\ 0 & \text{with probability } 1 - 2p_{i,t}(1-p_{i,t}); \end{cases}$$

with $1/n \le p_{i,t} \le 1 - 1/n$ for every $i \in \{1, \ldots, n\}$. Then

$$\mathbb{E}(|\Delta_t|) \ge \frac{\sqrt{2}}{n^{3/2}} X_t.$$

Lemma 3.2. The drift of the process X_t is

$$\begin{aligned} \mathbb{E}(X_{t+1} - X_t \mid X_t) &= \frac{|\Delta_t|}{K}\left(1 - 4P_{\mathcal{G}}(t) + 2P_{\mathcal{G}}(t)^2\right) \\ &\geq \frac{\sqrt{2}}{n^{3/2}K} X_t \left(1 - 4P_{\mathcal{G}}(t) + 2P_{\mathcal{G}}(t)^2\right) \end{aligned} \tag{3.3}$$

The drift of the process X_t is important because, as long as it stays positive, the cGA is making progress towards the solution. Clearly, the expression in the parenthesis, $1 - P_{\mathcal{G}}(t) + 2P_{\mathcal{G}}(t)^2$, from the equation (3.3) determines the sign of the drift. We have that if $P_{\mathcal{G}}(t) < 1 - \frac{1}{\sqrt{2}}$ the drift is positive. We call this value *critical threshold*. To bound the runtime of the cGA on the JUMP$_k$, we need to show that the drift stays positive long enough. In other words we must show that there is enough probability to generate the optimal solution before diversity is lost and the marginal probabilities fix (either at 0 or 1). By using the mentioned lemmas and focusing on the two processes 3.1 and 3.2 we are able to formulate the result that bounds the runtime of the cGA on the JUMP$_k$ as follows.

Theorem 3.1. *Let $\beta = \min\{1 + e^{4(c+k)}/n, n^{1.5+\epsilon/2}\}$ for a sufficiently large positive constant c and any small positive constant ϵ. Setting $K \geq n^2\beta$, the cGA generates an optimal solution for* JUMP$_k$ *with $k = o(n)$ in*

$$2\sqrt{2}Kn^{3/2}\ln(n/2) + e^{4(c+k)} + n$$

generations with probability $1 - o(1)$.

For more details, proofs and experiments refer to the paper [1]. Additionally to proving the bound on the runtime of the cGA, we also introduce a modification to the classical version of the cGA. We call this modification cGA$_\lambda$. The difference between the two is that in the cGA$_\lambda$, we generate λ offspring instead. Then, we choose the two fittest individuals breaking ties uniformly at random. The rest of the algorithm stays the same afterwards. See the pseudocode of the cGA$_\lambda$ in Algorithm 7.

Algorithm 7: The cGA$_\lambda$

```
1  $t \leftarrow 0$;
2  $p_{1,t} \leftarrow p_{2,t} \leftarrow \ldots \leftarrow p_{n,t} \leftarrow 1/2$;
3  while termination criterion not met do
4      for $j \in \{1, \ldots, \lambda\}$ do
5          for $i \in \{1, \ldots, n\}$ do
6              $x_i^{(j)} \leftarrow 1$ with probability $p_{i,t}$;
7              $x_i^{(j)} \leftarrow 0$ with probability $1 - p_{i,t}$;
8      Select $x$ and $y$ s.t. $x \neq y$ and $f(x) \geq f(y) \geq f(x^{(j)}$ for all $j \in \{1, \ldots, \lambda\}$
        with ties broken u.a.r.;
9      if $f(x) < f(y)$ then swap $x$ and $y$;
10     for $i \in \{1, \ldots, n\}$ do
11         if $x_i > y_i$ then $p_{i,t+1} \leftarrow \min\{p_{i,t} + 1/K, 1 - 1/n\}$;
12         if $x_i < y_i$ then $p_{i,t+1} \leftarrow \max\{p_{i,t} - 1/K, 1/n\}$;
13         if $x_i = y_i$ then $p_{i,t+1} \leftarrow p_{i,t}$;
14     $t \leftarrow t + 1$;
```

The great thing about this new algorithm is that it increases the duration that the drift stays positive (Fig 3.2). This means that the cGA$_\lambda$ stays longer in the critical phase and it has more time to solve the JUMP$_k$. This also implies that it can solve the function with increasing gap lengths k for the same length-n bitstring when allocated the same number of fitness evaluations. Despite the overhead incurred by evaluating additional offspring, numerical results support the improved performance as well. For the sake of completeness, we show a plot of runtimes of the cGA (Fig 3.4) and of the ratio of successes for different values of k (Fig 3.3).

In Figure 3.2, we can see the significance of generating $\lambda > 2$ offspring in every generation. Even a small change in λ has a large impact on the critical threshold. Despite the usage of the modified version of the cGA, even the classical version outperforms all the mentioned EAs in both the actual runtime and the gap length it

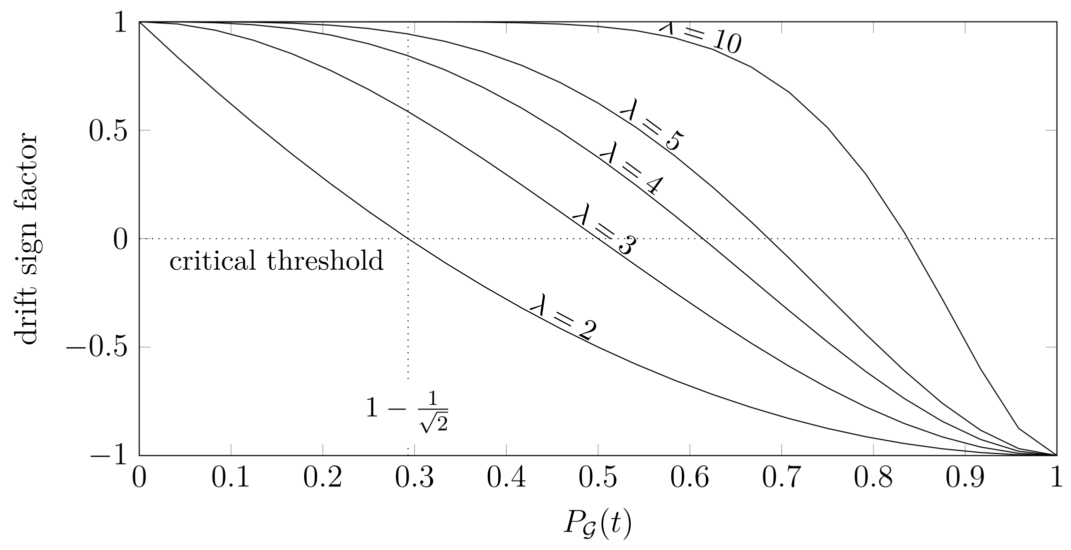


Figure 3.2: Drift sign factor as a function of gap probability $P_{\mathcal{G}}(t)$. As λ increases, the drift sign factor stays positive longer for gap probabilities approaching one.

can solve. In Figure 3.3, see that for $\lambda = 2$, the algorithm succeeds 100% times for $k \in \{2, \dots, 9\}$ in maximum allocated fitness function evaluations (10^7). For bigger values of k the performance starts to drop. Note that increasing the maximum number of allowed fitness evaluations does not necessarily lead to a higher success rate. The diversity loss still occurs at roughly the same time as in the previous case and, even though we give the algorithm more time, it does not solve the function anyway. We point out that for $\lambda = 20$, the cGA$_\lambda$ succeeds to solve the fitness function for $k \in \{2, \dots, 16\}$ in all cases (within 10^7 fitness evaluations). This is a significant jump given that the bitstring length is only $n = 100$. Clearly, for larger values of n, the ratio $\frac{n-k}{n}$ changes as well and hence, both the classical cGA and the new cGA$_\lambda$ are able to cross even bigger gaps. From this experiment, we can draw a conclusion that for $\lambda = 20$, the cGA$_\lambda$ can solve the Jump$_k$ function for any values of

$$k \in \{2, \dots, 0.16n\}$$

if given a reasonable number of fitness evaluations as time constraint. Regarding the actual runtime, the cGA scales more gracefully compared to the evolutionary algorithms. See that (Fig 3.4) for $k = 3$ and $n = 100$ the order of the runtime is 10^5 while the EAs already need around 10^6, and as we increase k just by one, the runtime changes dramatically.

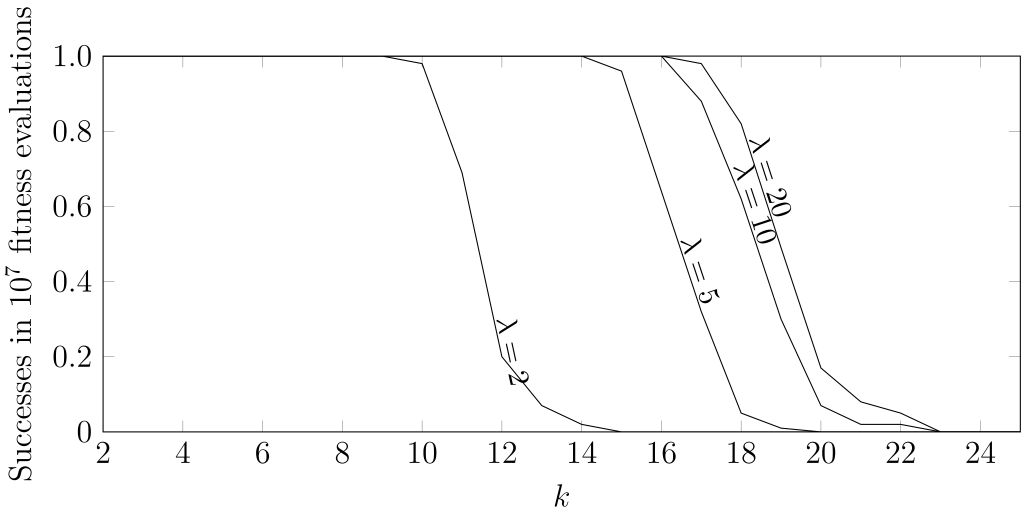


Figure 3.3: Proportion of the JUMP_k functions solved using the cGA_λ (Alg 7) in 10^7 fitness evaluations out of 100 trials for four different values of λ and $n = 100$.

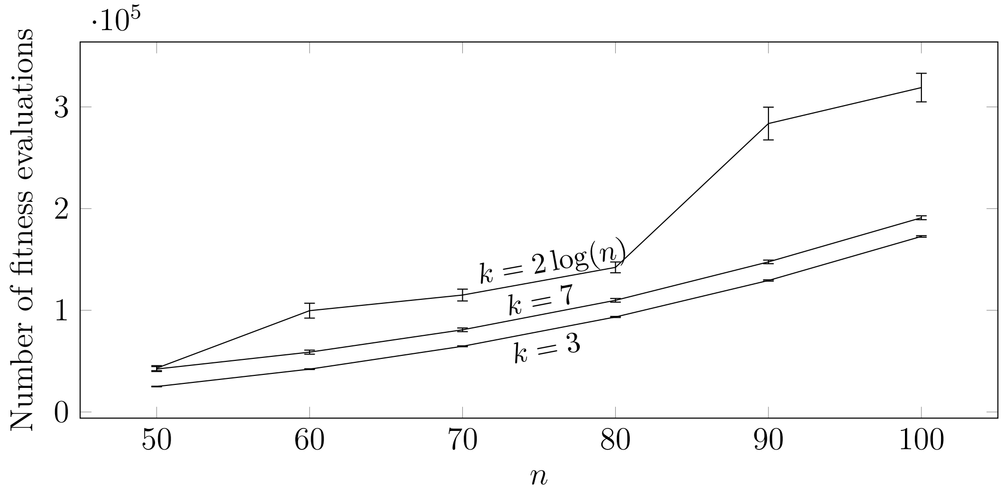


Figure 3.4: Number of fitness evaluations of the regular cGA (Alg 5) for various bitstring lengths n. We show the mean values and standard errors of 100 trials.

4 Analysis of the Univariate Marginal Distribution Algorithm

This chapter is the main contribution of this thesis. We analyze the UMDA algorithm on the $\textsc{Jump}_k$ fitness function. Our goal is to draw a comparison between the UMDA and the cGA based on the analysis we did for the cGA. Recall that for the cGA we were able to define its drift on the $\textsc{Jump}_k$ and that, as long as it stayed positive, the algorithm was making progress towards the solution. Once it reached the critical phase, we were able to show that the marginal probabilities stay concentrated around their mean value for a long time. Therefore, the cGA can solve the the $\textsc{Jump}_k$ fitness function for much bigger values of k compared to the traditional mutation and crossover only evolutionary algorithms.

First, we need to understand how the marginal probabilities are updated for the UMDA. We mentioned that the UMDA makes much bigger updates in every iteration than the cGA. Specifically, the update can easily exceed values like 0.1 (Fig 2.9) for every frequency while the cGA's marginals only change by $\frac{1}{n^2}$ (when $K = n^2$). This is a major difference between the two algorithms. To find out how the frequencies change during a run of the UMDA, we perform a series of experiments for different values of λ and μ. We already know that the ratio between these two parameters influences the updates of the marginals, and also the speed of the algorithm itself. See the following graphs in which we examine the marginal probabilities.

We can see (Fig 4.1) that the frequencies are much more dispersed compared to

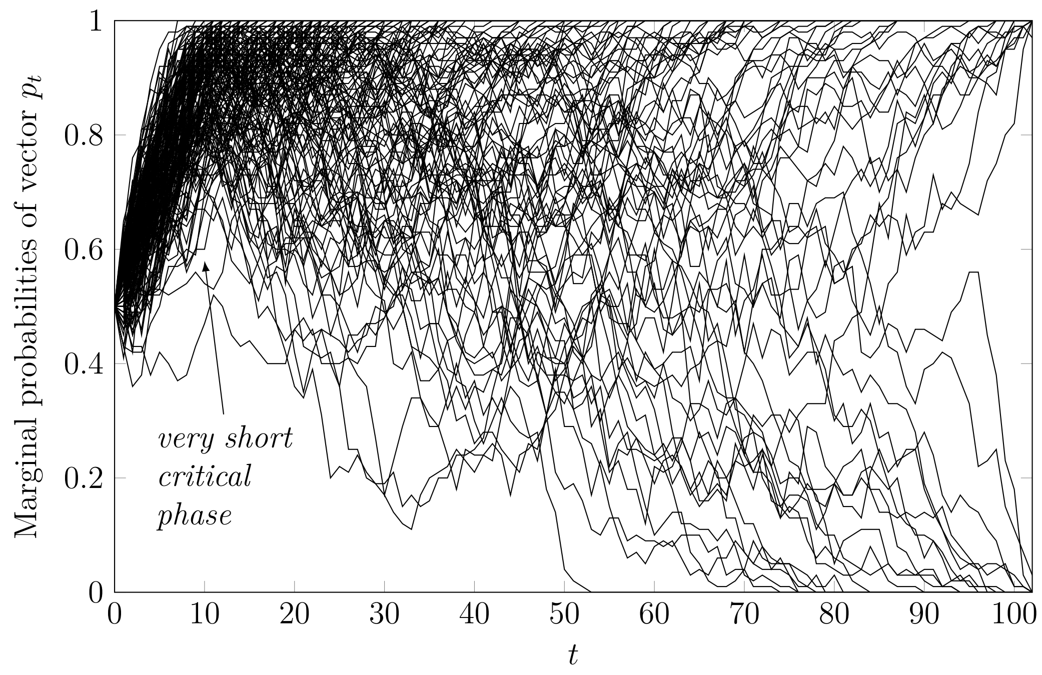


Figure 4.1: The UMDA's (Alg 6) marginal probabilities of vector p_t for JUMP_{20} with $n = 200$. Compared to the cGA (Alg 5) the marginals are much more dispersed. We set $\lambda = 4000$ and $\mu = 100$.

the cGA 3.1, which confirms the claims from above. In this experiment we accept 25% of all generated individuals in every generation. This means that there are potentially big differences between the fitness of the accepted offspring. Notice that some of the marginals drop even below the initial value of 0.5 right at the beginning. We can still observe an initial climb of the frequencies towards the ratio $\frac{n-k}{n}$ but there is definitely more randomness involved than for the cGA. However, the biggest issue the UMDA faces is the fact that its critical phase is very short. In fact, this example shows that the critical phase is roughly only from $t = 10$ to $t = 15$. That is only five iterations compared to tens of thousands for the cGA. This makes it harder for the UMDA to solve the JUMP_k as smoothly as the cGA does. One thing which remains the same is that the number of bits converging to 0 is k and the number of bits converging to 1 is $n - k$. This implies that the mean value of the marginal probabilities is again roughly $\frac{n-k}{n}$ after the short initial climbing stage. Let us repeat this experiment for different values of λ and μ. We set $\lambda = 4000$ and $\mu = 100$. We can see (Fig 4.2) an

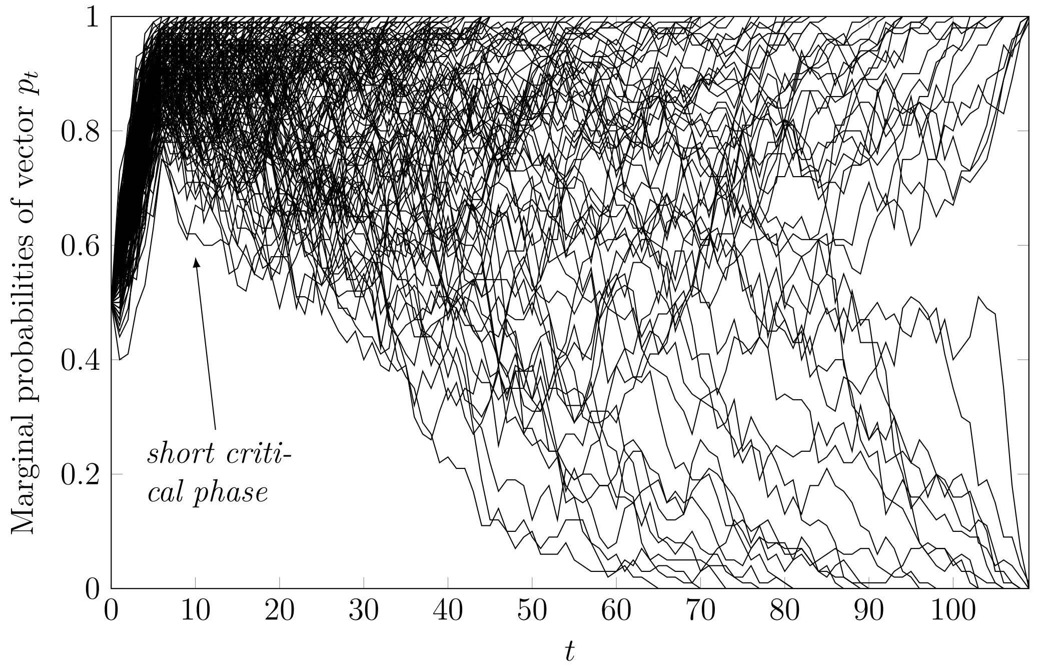


Figure 4.2: The UMDA's (6) marginal probabilities of vector p_t for $\text{J}\small{\text{UMP}}_{20}$ with $n = 200$. The marginals are still very dispersed. We set $\lambda = 4000$ and $\mu = 100$.

improvement of the marginals in terms of how disperse they are. In this case, the mean value of the frequencies stays closer to the value $\frac{n-k}{n}$ from even earlier iterations. However, the critical phase still remains very short. How is it possible that, as we show later in this chapter, the UMDA still solves the $\text{J}\small{\text{UMP}}_k$ function for relatively large values of k? We need to realize that we generate a large number ($\lambda = 4000$) of offspring under the current version of the probability vector p_t in every iteration. This means that if the critical phase is about 5 iterations long, then we still draw $20,000$ individuals from p_t before the marginals completely disperse. It is less than the cGA, but still enough to generate the optimum for various k.

At the initial stages of our UMDA analysis we were trying to mimic the approach we took to analyze the cGA. We attempted to define UMDA's drift and then bound the marginal probabilities. If we were able to do that then we would just apply the same lemmas and theorems. However, the nature of the univariate marginal distribution algorithm does not allow us to do that. The way in which the frequencies are updated makes it more difficult to formulate the drift formally. We cannot guarantee what individuals are accepted as the μ fittest ones to update the marginals. Recently, there have been papers published that show a way of analyzing OneMax [20] but their approach does not necessarily work the same for the Jump_k function. This is due to the problematic gap individuals. Recall that these bitstrings have more 1s in them compared to the non-gap individuals (besides the optimum) but their fitness is lower. Because of this we did not succeed in formulating the drift of the UMDA. Instead, we performed experimental work (Fig 4.3, 4.4) to see how UMDA performs on the Jump_k function.

For the sake of clarity, let us modify the UMDA's notation slightly. We will write $\text{UMDA}_{\lambda,\mu}$ to make it obvious what parameters are used in the algorithm. From the figures we can see that the runtimes are actually better than for the cGA but the UMDA cannot jump as far even though we exploit the UMDA's update policy by choosing large values for λ and μ. Note that we get the best updates by choosing large λ and small μ. However, we cannot pick μ too small otherwise the updates are too coarse-grained. Unfortunately, choosing such λ and μ also makes the individual frequencies converge towards 1 or 0 faster. Overall, we suggest choosing large λ and relatively small μ because in such case all the marginals get closer to the mean value $\frac{n-k}{n}$. The closer and concentrated the marginal probabilities get to the mean value the more likely it is that the UMDA finds the optimum.

For four out of five of our experiments we choose the ratio between λ and μ to be 25%. We suggest choosing an even smaller ratio in order to exploit the UMDA's

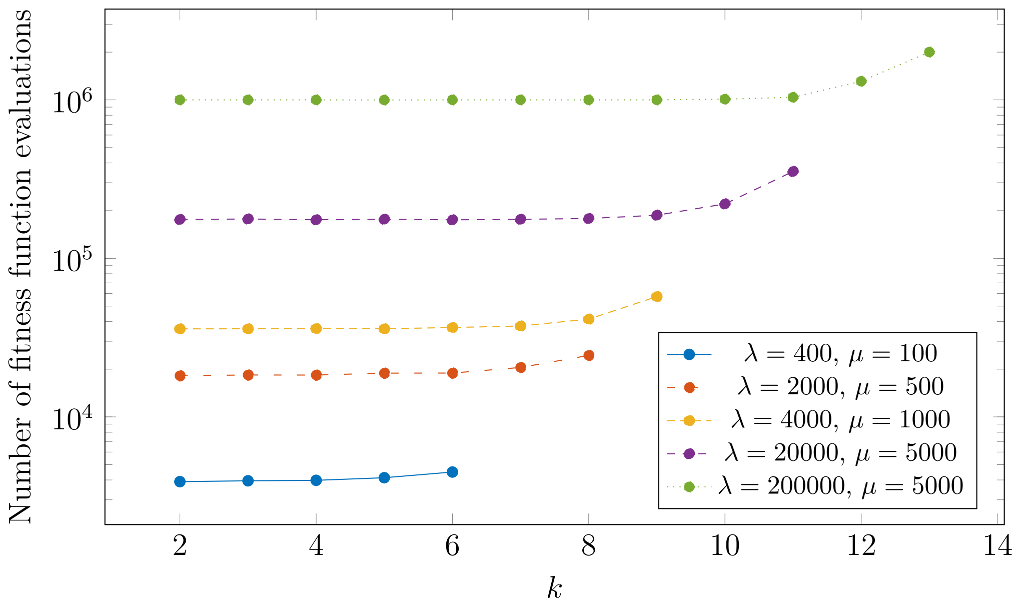


Figure 4.3: Number of fitness function evaluations to find the solution of the JUMP$_k$ for different gap lengths k for UMDA (Alg 6). We show the mean values of 100 trials. For this experiment, we set $n = 100$ and limit the maximum number of fitness function evaluations to 10^7.

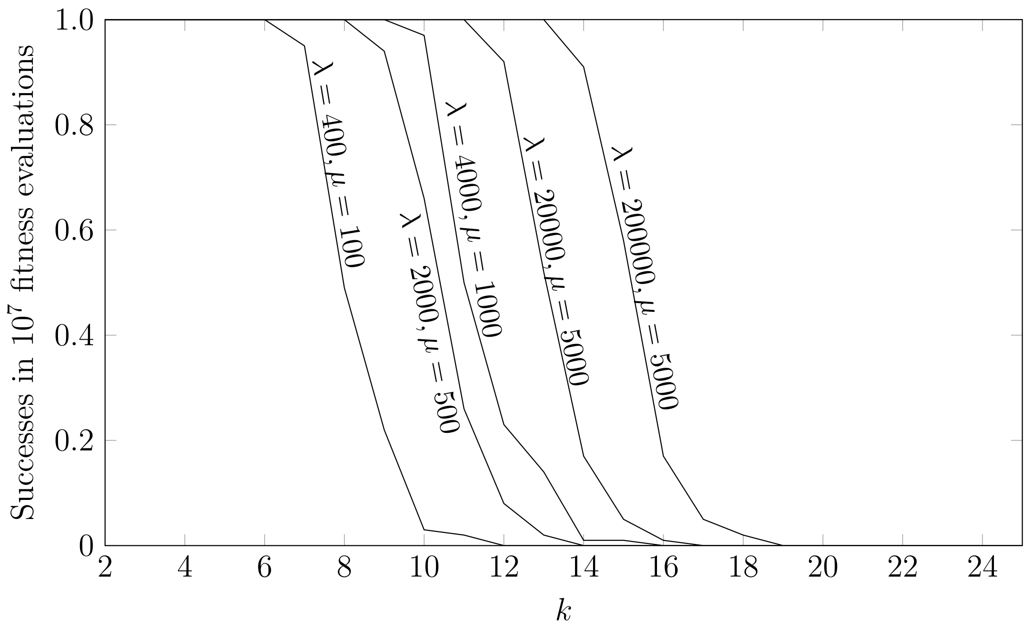


Figure 4.4: Proportion of the JUMP$_k$ functions solved using the UMDA$_{\lambda,\mu}$ (Alg 6) in 10^7 fitness function evaluations out of 100 trials for four different values of λ. We set $n = 100$.

update policy as much as possible. We conclude that the UMDA is slightly more efficient on JUMP_k, but does not solve the function for such large values of k. Since we are not able to analyze the UMDA in the same manner as the cGA we attempt to take a different approach. The idea for the following comes from trying to prolong the critical phase which we achieve with the cGA_λ.

4.1 Selection-free Univariate Marginal Distribution Algorithm

We introduce a new modified univariate marginal distribution algorithm. We call it Selection-free Univariate Marginal Distribution Algorithm and denote it by UMDA^{sel} or $\text{UMDA}^{sel}_{\lambda,\mu}$ (The Algorithm 8). For this algorithm we take an innovative approach to perform optimization. Traditionally, selection is what makes it possible for evolutionary algorithms to optimize problems. Removing selection in EAs would normally cause them to fail in finding an optimum efficiently for most fitness functions. For example, the $(1+1)$ EA without selection becomes just a random walk. However, turning off selection in an evolutionary distribution algorithm at the right time has a different effect. Basically, when we stop selecting the fittest individuals, we 'freeze' the marginals at their current values. What we mean by turning off selection is not choosing the μ fittest individuals to sum up their bits to update the frequencies, but update them by summing up all λ individuals instead. The update formula is as follows.

$$p_{i,t+1} = \sum_{j=1}^{\lambda} \frac{x^{(j)}[i]}{\lambda}$$

where $x^{(j)}$, $j \in \{1, \dots, \lambda\}$ denotes all the offspring in the generation t.

This new technique prevents the marginals from dispersing as fast as they do for

Algorithm 8: The UMDA^{sel}

```
1  $t \leftarrow 0$;
2  $p_{1,t} \leftarrow p_{2,t} \leftarrow \ldots \leftarrow p_{n,t} \leftarrow 1/2$;
3  **while** *termination criterion not met* **do**
4      $P_t \leftarrow \emptyset$;
5      **for** $j \in \{1, \ldots, \lambda\}$ **do**
6          **for** $i \in \{1, \ldots, n\}$ **do**
7              $x_i^{(j)} \leftarrow 1$ with probability $p_{i,t}$;
8              $x_i^{(j)} \leftarrow 0$ with probability $1 - p_{i,t}$;
9          $P_t \leftarrow P_t \cup \{x^{(j)}\}$;
10     **if** $t < t^*$ **then**
11         Sort all individuals in $P_t$ in descending order breaking ties u.a.r.;
12         $c \leftarrow \mu$;
13     **else**
14         $c \leftarrow \lambda$;
15     **for** $i \in \{1, \ldots, n\}$ **do**
16         $r = \frac{\sum_{j=1}^{c} x_i^{(j)}}{c}$;
17         **if** $r < 1/n$ **then** $p_{i,t+1} \leftarrow 1/n$;
18         **else** $p_{i,t+1} \leftarrow r$;
19         **if** $r > 1 - 1/n$ **then** $p_{i,t+1} \leftarrow 1 - 1/n$;
20         **else** $p_{i,t+1} \leftarrow r$;
21     $t \leftarrow t + 1$;
```

the regular UMDA after the algorithm performs the initial climb towards the gap. This gives the UMDA$_{\lambda,\mu}^{sel}$ more time to solve the JUMP$_k$ fitness function. One could argue that this goes against the very basic principles of evolutionary algorithms – being tools for *black–box optimization*. Here, it might seem that we are exploiting the properties of the JUMP$_k$ function while we should not require any knowledge about the fitness function at all. However, the idea which led to this modification of the UMDA was more based on studying the marginal probabilities. This can be done without any knowledge of the fitness function. This idea could potentially create a new branch of evolutionary algorithms in the future.

For comparison in what happens to the frequencies using the UMDA^{sel} refer to Figure 4.5. This experiment has the same parameter setup as the one described in Figure 4.2.

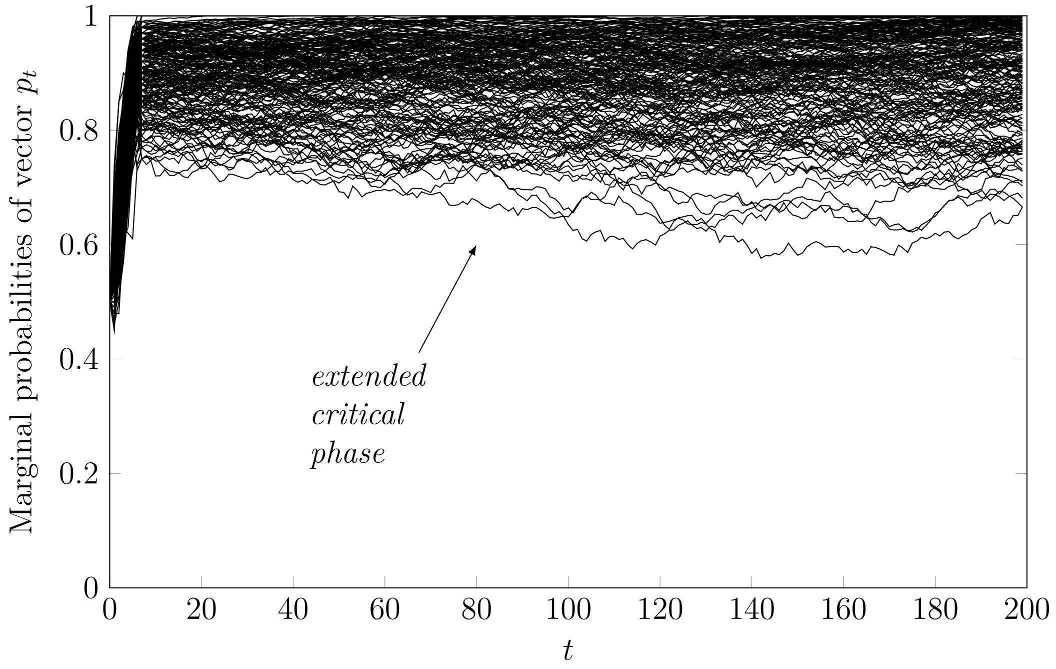


Figure 4.5: The UMDA$^{sel}_{\lambda,\mu}$'s (Alg 8) marginal probabilities of vector p_t for $\textsc{Jump}_{20}$ with $n = 200$. We zoom in to display only the first 200 iterations. The marginals are much less dispersed. We set $\lambda = 4000$ and $\mu = 100$.

The first important step in our analysis is to define a time $t^* \in \mathbb{N}$ when we turn off selection. See the following definition.

Definition 4.1. Let $t^* \in \mathbb{N}$ be the time when we turn off selection in the UMDA^{sel}. Also, define

$$m = \frac{n-k}{n}$$

where k is gap length and n is the bitstring length. We define t^* as follows.

$$t^* = \min_{t \in \mathbb{N}} \max_i |p_{i,t} - m|$$

for all $i \in \{1, \ldots, n\}$.

The Definition 4.1 says that we choose t^* so that the largest distance from any of the marginals to the ratio m is the smallest out of all iterations $t \in \mathbb{N}$. Note that we use m to approximate the mean value of all marginal probabilities after the initial climb towards the gap. As shown by the previous experiment, turning off selection helps each marginal probability $p_{i,t}$ stay closer to the value p_{i,t^*} for $t > t^*$. This is formalized in the following Remark.

Lemma 4.1. Denote the time when we turn off selection for the $\text{UMDA}_{\lambda,\mu}^{sel}$ by t^*. Then, the future expectation of the marginal probabilities becomes

$$\mathbb{E}[p_{i,t^*+j} \mid p_{i,t^*}] = p_{i,t^*}$$

for all $i \in \{1, \ldots, n\}$ and $j \geq 0$.

Proof. The UMDA generates its offspring from the probability vector p_t. Every bit position $i \in \{1, \ldots, n\}$ follows Bernoulli distribution. This means that every generated individual has 1 at position i with probability $p_{i,t}$ and 0 with probability $1 - p_{i,t}$. When we sum up λ generated individuals, the sum of each bit i then follows Poisson binomial distribution (sum of λ Bernoulli distributions denoted by $\mathcal{B}$). Therefore,

$$p_{i,t+1} = \frac{1}{\lambda} \sum_{l=1}^{\lambda} \mathcal{B}(p_{i,t}).$$

Then, given that the mean of the Poisson binomial distribution is $p_{i,t}$ in this case, we get $\mathbb{E}[p_{i,t+1} \mid p_{i,t}] = p_{i,t}$. By induction we get the same result for $p_{i,t+j}$ for any $j > 0$. This completes the proof. $\square$

In other words, Lemma 4.1 states that the ratio between the number of 1s and 0s at each bit i for $t > t^*$ is equal to the marginal probability p_{i,t^*}, or at least very close to it, for the rest of the run. Before we attempt to analyze this new version of the UMDA, let us perform experiments on runtimes and on the ratios of successes within the maximum allocated number of fitness evaluations. The experiment (Fig 4.7) confirms what we are hoping for. We can see an increase in the maximum possible jump for each configuration of λ and μ. The labels we use in Figure 4.7 are the following. The numbers $1-5$ denote the configurations of λ and μ where 1 is $\lambda = 400$ and $\mu = 100$, 2 is $\lambda = 2000$ and $\mu = 500$, 2 is $\lambda = 4000$ and $\mu = 1000$, 4 is $\lambda = 20000$ and $\mu = 5000$ and 5 is $\lambda = 200000$ and $\mu = 5000$. The letter a denotes the original UMDA$_{\lambda,\mu}$ while b stands for the new UMDA$^{sel}_{\lambda,\mu}$. The runtimes stay nearly the same as for the original UMDA$_{\lambda,\mu}$. This was not the case with the cGA$_\lambda$. For $\lambda > 2$ all the runtimes increased instead.

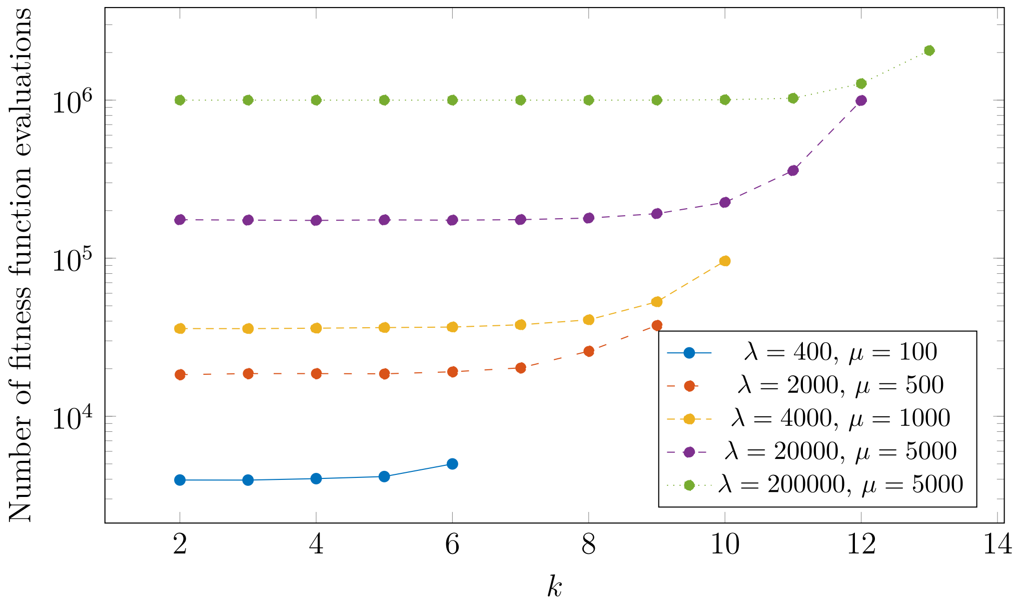


Figure 4.6: Number of fitness function evaluations to find the solution of the JUMP$_k$ for different gap lengths k for the UMDA^{sel} (Alg 8). We show the mean values of 100 trials. For this experiment, we set $n = 100$ and limit the maximum number of fitness function evaluations to 10^7. Selection was is turned off after first 9 iterations.

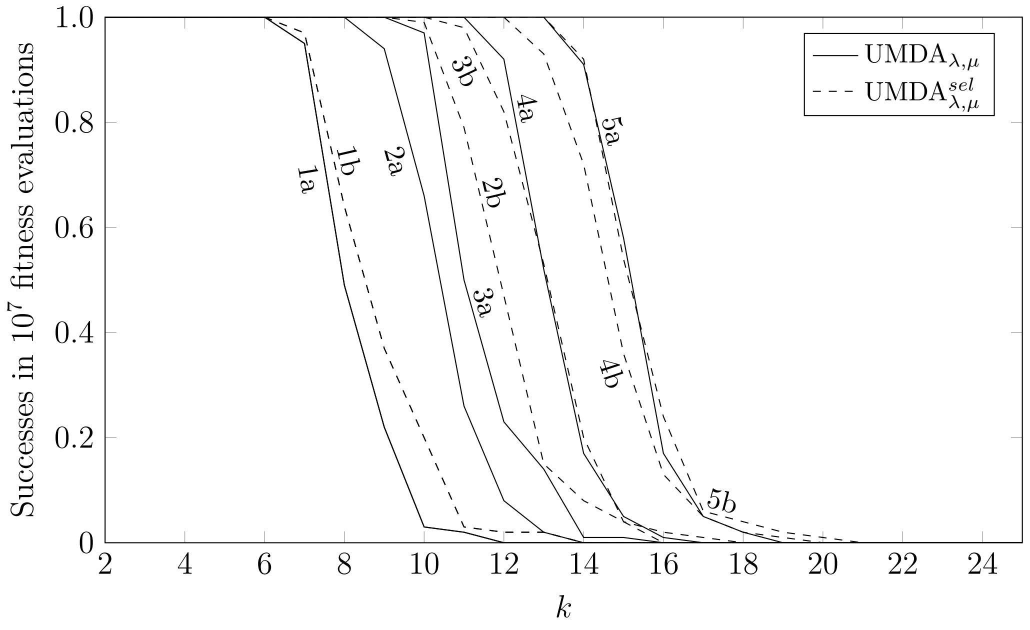


Figure 4.7: Proportion of the JUMP$_k$ functions solved using the UMDA$_{\lambda,\mu}$ (Alg 6) and the UMDA$^{sel}_{\lambda,\mu}$ (Alg 8) in 10^7 fitness function evaluations out of 100 trials for five different values of λ and μ. We set $n = 100$. For the UMDA$^{sel}_{\lambda,\mu}$ we turn off selection after 9 iterations.

We continue with the analysis. The important step is to bound all the marginal probabilities. We need to know what the frequencies look like around the moment when we turn off selection. We showed in the experiments that the mean of the marginals approaches the ratio $\frac{n-k}{n}$. However, this does not tell us anything about the individual marginal probabilities. Thus, we need to put a constraint on the maximum distance of the smallest of all the marginal probabilities from the mean value. We formulate a constraint on the marginal probabilities with the following lemma.

Lemma 4.2. Let $t^* \in \mathbb{N}$ and m be defined as in Definition 4.1. Then, with probability $1 - o(1)$, when selection is turned off the marginals are bounded s.t.

$$|p_{i,t} - m| < \frac{c}{n} = \mathcal{O}\left(\frac{1}{n}\right)$$

for $c > 0$ and $i \in \{1, \ldots, n\}$.

Lemma 4.2 bounds all the frequencies, most importantly, from below at the time when we turn off selection. We are not concerned as much about the upper bound because the higher the individual marginals get the easier it is for the algorithm to find the optimum. We are not able to prove this result formally due to the UMDA's complicated update policy but it is clear from the experiments (Fig 4.5) that the marginals actually approach the ratio $\frac{n-k}{n}$ fairly nice. Another fact is that for larger values of λ and μ the frequencies are concentrated around the mean as well. To support this claim even more we show one more plot of the marginal probabilities over time for very large λ for the UMDA^{sel} (Fig 4.9). For comparison we also display the same plot for the traditional UMDA (Fig 4.8) as well. Note that with $\lambda = 500000$ we generate very large number of offspring in every iteration. In Figure 4.9, the marginals almost do not change over time and stay the same for the rest of the run. We terminate the algorithm after 200 iterations but that still means that we generate 10^8 individuals. Given that the mean of the marginals climbs to the ratio $\frac{n-k}{n}$ really fast, we generate a majority of the individuals from probability vector p_t which has very high mean. This increases the chances of solving the Jump$_k$ fitness function significantly.

An important mechanism in many evolutionary algorithms analyses is knowing an expectation of a future state given the current state. In terms of EDAs, it translates to knowing what is the expectation of a marginal probability $p_{i,t+1}$ given $p_{i,t}$. This is

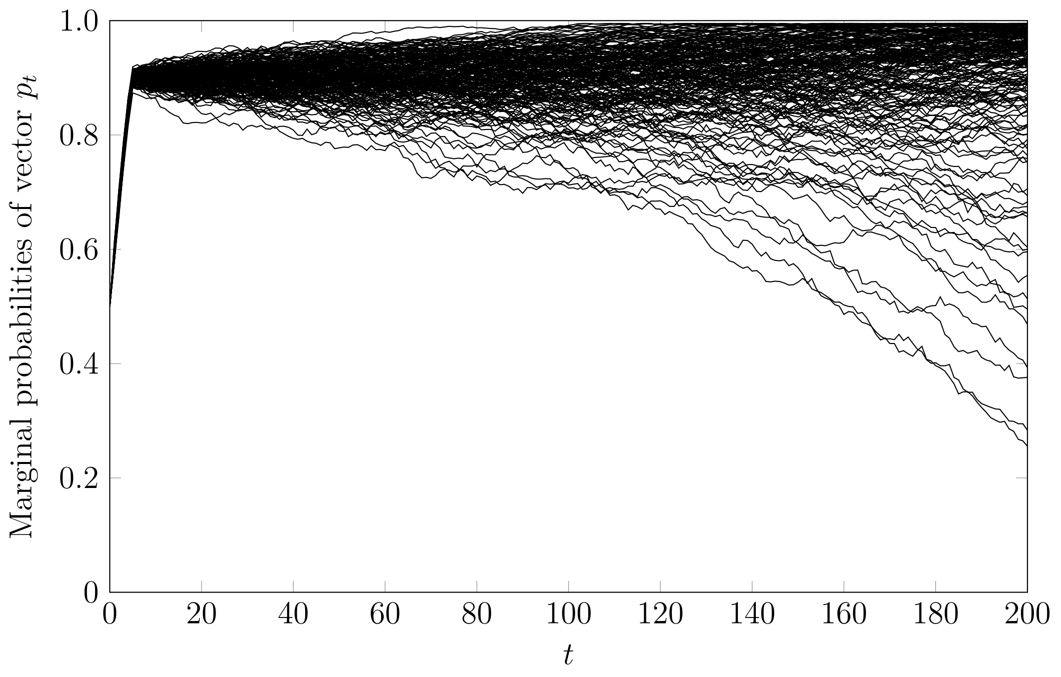


Figure 4.8: The marginal probabilities of vector p_t for JUMP_{20} with $n = 200$ for the standard UMDA (Alg 6). For this experiment we choose large λ ($5 \cdot 10^5$) and μ ($5 \cdot 10^3$). This marginals approach the ratio $\frac{n-k}{n}$ and stay concentrated around the mean longer.

formalized in the following definition.

Definition 4.2. A stochastic process $\{Z_n, n \geq 1\}$ is a *martingale* [26] if $\mathbb{E}[|Z_n|] < \infty$ and $\mathbb{E}[Z_{n+1} \mid Z_1, \ldots, Z_n] = Z_n$.

Definition 4.2 says that the expectation of the process Z_n does not change over time. The following lemma assess that the marginal probabilities of the UMDA^{sel} are martingale.

Lemma 4.3. For every bit position $i \in \{1, \ldots, n\}$ and its marginal probability $p_{i,t}$ we get

$$\mathbb{E}[p_{i,t+1} \mid p_{i,t}] = p_{i,t}. \tag{4.1}$$

This means that the marginal probabilities are a martingale sequence.

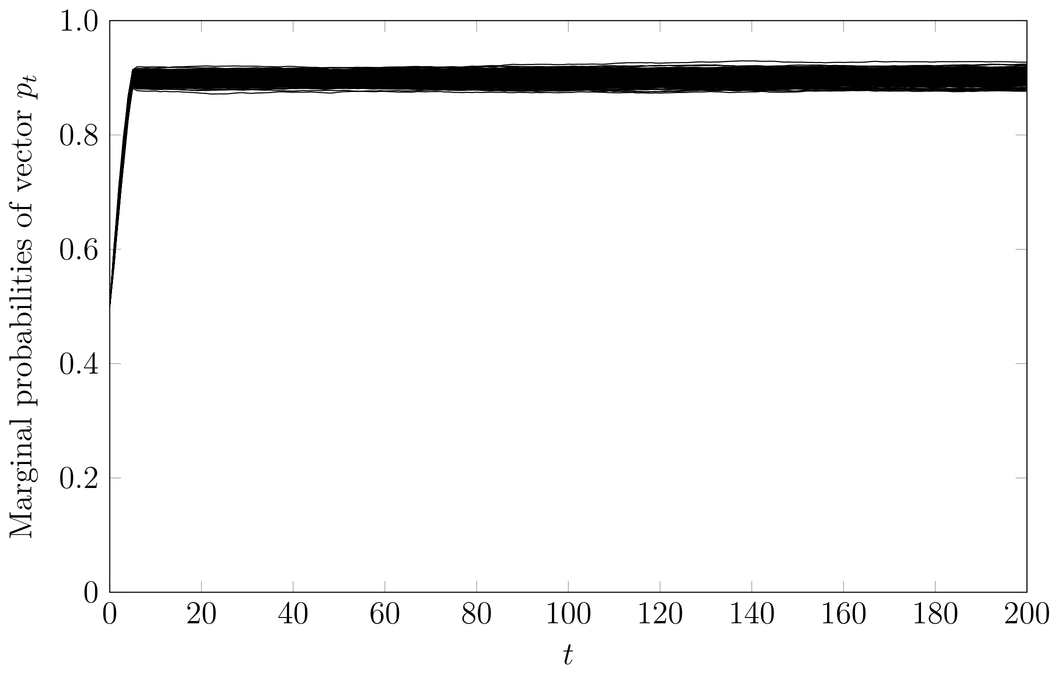


Figure 4.9: The UMDA^{sel}'s (Alg 8) marginal probabilities of vector p_t for JUMP$_{20}$ with $n = 200$. For this experiment we choose large λ ($5 \cdot 10^5$) and μ ($5 \cdot 10^3$). We can see that the marginals 'freeze' around their values when we turn off selection. Selection is turned off after 5 iterations.

Proof. The result (4.1) follows directly from Lemma 4.1. □

At this point we are ready to formally bound the frequencies of the UMDA^{sel} after turning off the selection for the first T iterations.

Lemma 4.4. Let t^* be the time when we turn off selection for the UMDA, then the marginals $p_{i,t}$ are sum of λ independent Bernoulli variables denoted by $\mathcal{B}(p)$, i.e.

$$p_{i,t+1} = \frac{1}{\lambda} \sum_{j=1}^{\lambda} \mathcal{B}(p_{i,t}),$$

For any $t \in \mathbb{N}$ s.t. $t^* \leq t \leq t^* + T$ and $T = poly(n)$ and for any $i \in \{1, \ldots, n\}$, with probability at least $1-\exp\left(-\frac{2\lambda}{a^2}+\mathcal{O}(\log(n))\right)$,

$$|p_{t+1}-p_{i,t}| \leq \frac{1}{a}$$

where $a = \Omega(n\sqrt{T})$.

Proof. Let $t \in [t^*, t^*+T]$ where $T = poly(n)$ and $i \in \{1,\ldots,n\}$ be given. Also let

$$X_j = \begin{cases} \frac{1}{\lambda} & \text{with probability } p_{i,t}, \\ 0 & \text{otherwise.} \end{cases}$$

Define $S = X_1 + X_2 + \ldots + X_\lambda = p_{i,t+1}$. Also,

$$\mathbb{E}[S] = \mathbb{E}[p_{i,t+1}] = \mathbb{E}\left[\frac{1}{\lambda}\sum_{j=1}^{\lambda}\mathcal{B}(p_{i,t}^{(j)})\right] = \frac{1}{\lambda}\,\mathbb{E}\left[\sum_{j=1}^{\lambda}\mathcal{B}(p_{i,t}^{(j)})\right] = \frac{1}{\lambda}\lambda p_{i,t} = p_{i,t}.$$

Then, $|p_{i,t+1}-p_{i,t}| = |S-\mathbb{E}[S]|$ and by Hoeffding's inequality [27] we get that

$$\Pr\left(|S-\mathbb{E}[S]| \geq \frac{1}{a}\right) \leq 2\exp\left(-\frac{\frac{2}{a^2}}{\lambda\frac{1}{\lambda^2}}\right) = 2\exp\left(-\frac{2\lambda}{a^2}\right).$$

By taking a union bound over all choices of $i \in \{1,\ldots,n\}$ and all $t^* \leq t \leq T$, we obtain a probability that $|S-\mathbb{E}[S]| \geq \frac{1}{a}$ happens

$$2nT\exp\left(-\frac{2\lambda}{a^2}\right) = 2\exp\left(-\frac{2\lambda}{a^2}+\log(nT)\right). \tag{4.2}$$

Since we are interested in when $|S-\mathbb{E}[S]| \leq \frac{1}{a}$ we take the complement of (4.2). This finishes the proof. □

The following theorem asserts the marginal probabilities stay concentrated around the mean long enough in the UMDA^{sel} process.

Theorem 4.1 (Azuma-Hoeffding Inequality [27]). *Suppose $\{Y_t : t \in \mathbb{N}\}$ is a martingale and $|Y_t - Y_{t-1}| < c_t$ almost surely. Then for any positive integer T and any $z > 0$*

$$\Pr(|Y_T - Y_0| \geq z) \leq 2 \exp\left(\frac{-z^2}{2\sum_{t=1}^{T} c_t^2}\right). \tag{4.3}$$

Theorem 4.2. *Suppose that the marginals are bounded as in Lemma 4.2 when we turn off selection at t^*. For a sufficiently large positive constants c and c', $a = \Omega(n^{1+\epsilon/2}\sqrt{T})$ and any small positive constant ϵ the UMDA$^{sel}_{\lambda,\mu}$ generates an optimal solution for the* Jump$_k$ *with $k = o(n)$ in*

$$\mathcal{O}(t^* + e^{k+c+c'} + n)$$

generations with probability $1 - o(1)$.

Proof. We split the run of the UMDA^{sel} into two phases. In the first phase, before we turn off selection, we know that the algorithm runs for

$$t^* \tag{4.4}$$

iterations. Because of the condition posed by Lemma 4.2 we have that the smallest marginal probability at t^* is at least $p_{i,t^*} \geq \frac{n-k}{n} - \frac{c}{n}$. From Lemma 4.1 we have that $|p_{i,t+1} - p_{i,t}| \leq \frac{1}{a}$ at least during the first T iterations and since the marginal probabilities are martingale (Lemma 4.3) we can apply Theorem 4.1. We get that

$$\Pr(|p_{i,t^*+T} - p_{i,t^*}| \geq \frac{c'}{n}) \leq 2\,\exp\left(-\frac{\frac{c'^2}{n^2}}{\frac{2T}{a^2}}\right) = 2\,\exp\left(-\frac{c'^2a^2}{2n^2T}\right).$$

Therefore, the probability that $|p_{i,t^*+T} - p_{i,t^*}| \leq \frac{c'}{n}$ is at least $1 - 2\,\exp\left(-\frac{c'^2a^2}{2n^2T}\right)$. All together, during the first T iterations after turning off selection the marginal probabilities are at least

$$\frac{n-k}{n} - \frac{c}{n} - \frac{c'}{n}$$

with probability $1 - 2\exp\left(-\frac{c'^2 a^2}{2n^2 T}\right)$. Since $a \geq n^{1+\epsilon/2}\sqrt{T}$ we have

$$1 - 2\exp\left(-\frac{c'^2 a^2}{2n^2 T}\right) = 1 - e^{-\Omega(n^\epsilon)}.$$

During each of the $e^{k+c+c'} + n$ iterations after we turn off selection, the probability that the UMDA^{sel} generates the optimal solution 1^n is

$$\prod_{i-1}^{n} p_{i,t} \geq \left(1 - \frac{k-c-c'}{n}\right)^n \geq e^{-\Omega(k)}. \tag{4.5}$$

The failure probability to produce the optimal solution during the $e^{k+c+c'}+n$ iterations is at most

$$\left(1 - e^{-\Omega(k)}\right)^{(e^{k+c+c'}+n)} = e^{-\Omega(n/k)}.$$

As $k = o(n)$, we have $e^{-\Omega(n/k)} = o(1)$. By combining (4.4) and (4.5) we get

$$\lambda t^* + e^{k+c+c'} + n$$

with probability $1 - o(1)$. □

This is the final step of our analysis. By conditioning on Lemmas 4.2 and 4.1 we were able to prove an upper bound on the UMDA^{sel} runtime. Even though we did not prove Lemma 4.2 formally, we believe that the experiments presented sufficiently support our results.

5 Conclusion

In this work we compare performance of two estimation of distribution algorithms, namely the cGA and the UMDA, on the $\textsc{Jump}_k$ fitness function. The analysis of the cGA is described in the chapter 3 and comes from the recent paper [1]. In this thesis, we focused on the analysis of the UMDA. By combination of experiments and formal proofs we bound the runtime of the algorithm and show its performance on $\textsc{Jump}_k$ for various parameter settings.

We also present a new innovative way of modifying the UMDA (and perhaps other EDAs) by turning off selection. This approach allows the UMDA's marginal probabilities to stay tightly concentrated around their mean value and gives the algorithm more time to succeed and find the optimum. It also allows the UMDA to jump across larger gaps of the $\textsc{Jump}_k$ function. Additionally, we show that the cGA$_\lambda$ can jump farther than the UMDA^{sel}. However, for certain choices of parameters λ and μ, the UMDA is faster than the cGA. Refer to Figure 5.1 for comparison. We limit the values of k so that both algorithms find the optimum in 10^7 fitness function evaluations with 100% success rate. Therefore, if the runtime is of importance we suggest using the UMDA and if we need to solve the $\textsc{Jump}_k$ for larger values of k, then the cGA should be the choice.

We observe a very interesting phenomenon for the marginal probabilities of the cGA. Unlike the UMDA's frequencies, whose mean stay very close to the ratio $\frac{n-k}{n}$, the cGA's mean value can go beyond that ratio. This is the main reason why the cGA can jump across larger gaps. Even a very small change in the overall mean

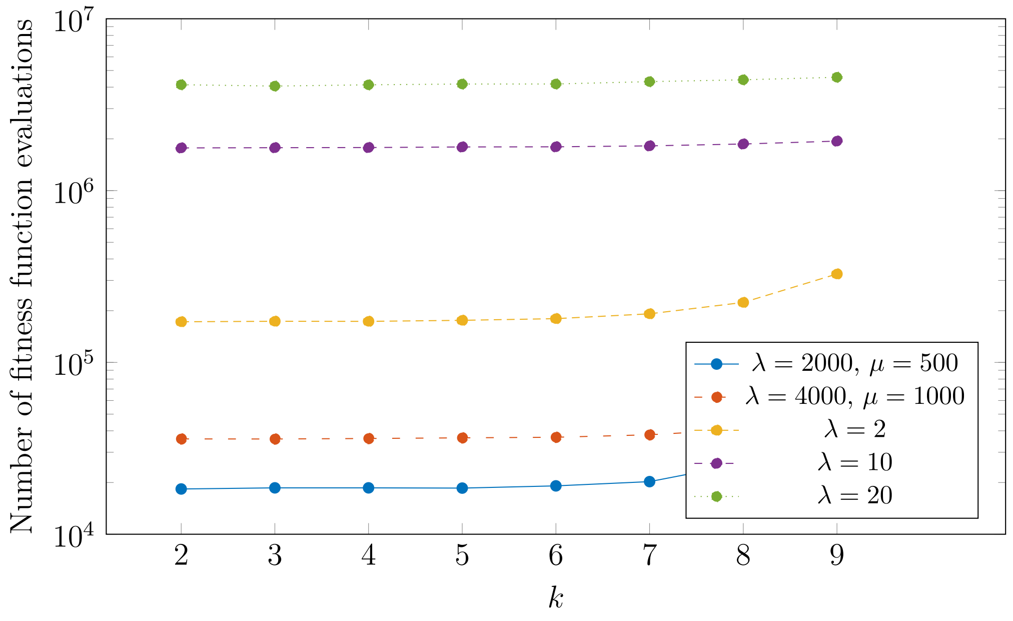


Figure 5.1: Number of fitness function evaluations to find the solution of the JUMP$_k$ for different gap lengths k for the cGA (Alg 5) and the standard UMDA (Alg 6). We show the mean values of 100 trials. For this experiment, we set $n = 100$ and limit the maximum number of fitness function evaluations to 10^7.

value can lead to a significant increase of the maximum potential gap length that the algorithm can solve. The Table 5.1 summarizes the highest mean values that we obtained throughout the cGA and the UMDA runs for different parameter settings. It also shows the absolute maximum gap length and the relative gap length in terms of n that the algorithms can solve with 100% success rate. The time when to turn off selection for the UMDA^{sel} is chosen experimentally and depicted in the parenthesis next to the mean value. It is important not to turn selection too early, so the marginals have enough time to climb to the ratio $\frac{n-k}{n}$. Also, we cannot switch it off too late, otherwise the frequencies are already too dispersed.

The reason why the mean of the marginals for the cGA$_\lambda$ goes beyond the ratio $\frac{n-k}{n}$ is due to the fact that in every iteration we generate very small number of individuals. This means that when the marginals get high enough we only generate offspring from

Algorithm	Highest mean	Absolute Largest gap	Relative Largest gap
cGA_2	0.8835	9	0.09n
cGA_5	0.9079	14	0.14n
cGA_{10}	0.9182	16	0.16n
cGA_{20}	0.9258	16	0.16n
$\text{UMDA}_{400,100}$	0.8987	6	0.06n
$\text{UMDA}_{2000,500}$	0.8995	8	0.08n
$\text{UMDA}_{4000,1000}$	0.8995	9	0.09n
$\text{UMDA}_{20000,5000}$	0.8993	11	0.11n
$\text{UMDA}_{200000,5000}$	0.9000	13	0.13n
$\text{UMDA}^{sel}_{400,100}$	0.8971 (12)	6	0.06n
$\text{UMDA}^{sel}_{2000,500}$	0.9012 (12)	9	0.09n
$\text{UMDA}^{sel}_{4000,1000}$	0.9008 (12)	10	0.10n
$\text{UMDA}^{sel}_{20000,5000}$	0.9012 (12)	12	0.12n
$\text{UMDA}^{sel}_{200000,5000}$	0.9002 (7)	13	0.13n

Table 5.1: Highest means of marginal probabilities during a run of both the cGA and the UMDA. The runs are terminated after 10^8 fitness evaluations. We also show the absolute largest gap and the relative largest gap related to n each algorithm can solve with 100% success rate within 10^7 fitness evaluations. Numbers in the parenthesis denote the iteration after which we turn off selection.

the gap. Clearly, the gap individuals have higher number of 1s among their bits and therefore, the overall mean can grow bigger. However, what is interesting is that for smaller λs, i.e. two, we see that the mean is even below the ratio $\frac{n-k}{n}$. The reason for that is because we only generate two offspring, the initial climb is not as efficient as it is for larger values of λ. For $\lambda > 2$ we climb more efficiently and when we start generating gap individuals, the mean is already higher than for smaller λs. Lastly, the mean of the marginals is not the only factor that determines how far the algorithm can jump. We can have a mean of exactly $\frac{n-k}{n}$ but there is k bits already very close to zero. Clearly, it is complicated to find the solution in such case.

This finishes the conclusion and summary of the results that we obtained in our analysis. All the experiments were performed in MATLAB and the code used to examine all the EAs and EDAs is in Apendix.

References

[1] V. Hasenöhrl and A. Sutton. "On the Runtime Dynamics of the Compact Genetic Algorithm on Jump Functions". In: *Genetic and Evolutionary Computation Conference*. GECCO '18. July 2018. DOI: 10.1145/3205455.3205608. URL: https://doi.org/10.1145/3205455.3205608 (cit. on pp. 2, 26, 27, 31, 35, 56).

[2] D. E. Knuth. *The art of computer programming*. Vol. 1. 1997, pp. 75–79. ISBN: 0-201-89683-4 (cit. on p. 10).

[3] R. Motwani and P. Raghavan. *Randomized algorithms*. Cambridge University Press, Cambridge, 1995, pp. xiv+476. ISBN: 0-521-47465-5. DOI: 10.1017/CBO9780511814075. URL: https://doi.org/10.1017/CBO9780511814075 (cit. on p. 10).

[4] L. Euler. *De Progressionibus harmonicus observationes*. 1735 (cit. on p. 11).

[5] T. Jansen. *Analyzing evolutionary algorithms from the computer science perspective*. Springer, 2013, pp. 7–29 (cit. on pp. 11, 12).

[6] T. Blickle and L. Thiele. "A Comparison of Selection Schemes Used in Evolutionary Algorithms". In: *Evol. Comput.* 4.4 (Dec. 1996), pp. 361–394. ISSN: 1063-6560. DOI: 10.1162/evco.1996.4.4.361. URL: http://dx.doi.org/10.1162/evco.1996.4.4.361 (cit. on p. 12).

[7] D. E. Goldberg and K. Deb. “A Comparative Analysis of Selection Schemes Used in Genetic Algorithms”. In: vol. 1. Foundations of Genetic Algorithms. Elsevier, 1991, pp. 69 –93. DOI: https://doi.org/10.1016/B978-0-08-050684-5.50008-2. URL: http://www.sciencedirect.com/science/article/pii/B9780080506845500082 (cit. on p. 12).

[8] B. Doerr and L. A. Goldberg. “Adaptive Drift Analysis”. In: *CoRR* abs/1108.0295 (2011). URL: http://arxiv.org/abs/1108.0295 (cit. on p. 12).

[9] C. Witt. “Tight Bounds on the Optimization Time of a Randomized Search Heuristic on Linear Functions”. In: *Combinatorics, Probability and Computing* 22.2 (2013), pp. 294–318. ISSN: 0963-5483. DOI: 10.1017/S0963548312000600 (cit. on p. 12).

[10] J. F. Crow and M. Kimura. “Efficiency of truncation selection”. In: *Proceedings of the National Academy of Sciences* 76.1 (1979), pp. 396–399. ISSN: 0027-8424. DOI: 10.1073/pnas.76.1.396. eprint: http://www.pnas.org/content/76/1/396.full.pdf. URL: http://www.pnas.org/content/76/1/396 (cit. on p. 15).

[11] S. Droste, T. Jansen, and I. Wegener. “On the Analysis of the (1+1) Evolutionary Algorithm”. In: *Theor. Comput. Sci.* 276.1-2 (Apr. 2002), pp. 51–81. ISSN: 0304-3975. DOI: 10.1016/S0304-3975(01)00182-7. URL: https://doi.org/10.1016/S0304-3975(01)00182-7 (cit. on p. 15).

[12] T. Jansen and I. Wegener. “On the Choice of the Mutation Probability for the (1+1) EA”. In: *Parallel Problem Solving from Nature PPSN VI*. Springer Berlin Heidelberg, 2000, pp. 89–98. ISBN: 978-3-540-45356-7 (cit. on p. 15).

[13] D. Sudholt. “How Crossover Speeds Up Building-Block Assembly in Genetic Algorithms”. In: *CoRR* abs/1403.6600 (2014). eprint: 1403.6600. URL: http://arxiv.org/abs/1403.6600 (cit. on p. 16).

[14] B. Doerr, C. Doerr, and F. Ebel. “Lessons from the black-box: Fast crossover-based genetic algorithms”. In: (July 2013), pp. 781–788 (cit. on pp. 16–18).

[15] M. Pelikan, M. W. Hauschild, and F. G. Lobo. “Estimation of Distribution Algorithms”. In: *Springer Handbook of Computational Intelligence*. 2015, pp. 899–928. ISBN: 978-3-662-43505-2. DOI: 10.1007/978-3-662-43505-2_45. URL: https://doi.org/10.1007/978-3-662-43505-2_45 (cit. on p. 19).

[16] G. R. Harik, F. G. Lobo, and D. E. Goldberg. “The Compact Genetic Algorithm”. In: *Trans. Evol. Comp* 3.4 (Nov. 1999), pp. 287–297. ISSN: 1089-778X. DOI: 10.1109/4235.797971. URL: http://dx.doi.org/10.1109/4235.797971 (cit. on p. 19).

[17] H. Mühlenbein. “The Equation for Response to Selection and Its Use for Prediction”. In: *Evol. Comput.* 5.3 (Sept. 1997), pp. 303–346. ISSN: 1063-6560. DOI: 10.1162/evco.1997.5.3.303. URL: http://dx.doi.org/10.1162/evco.1997.5.3.303 (cit. on p. 19).

[18] S. Droste. “A rigorous analysis of the compact genetic algorithm for linear functions”. In: *Natural Computing* 5.3 (2006), pp. 257–283. ISSN: 1572-9796. DOI: 10.1007/s11047-006-9001-0. URL: https://doi.org/10.1007/s11047-006-9001-0 (cit. on p. 20).

[19] T. Friedrich, T. Kötzing, M. S. Krejca, and A. M. Sutton. “The Compact Genetic Algorithm is Efficient Under Extreme Gaussian Noise”. In: *IEEE Transactions on Evolutionary Computation* 21.3 (2017), pp. 477–490. ISSN: 1089-778X. DOI: 10.1109/TEVC.2016.2613739 (cit. on p. 21).

[20] C. Witt. “Upper Bounds on the Runtime of the Univariate Marginal Distribution Algorithm on Onemax”. In: *Proceedings of the Genetic and Evolutionary Computation Conference*. GECCO ’17. ACM, 2017, pp. 1415–1422. ISBN: 978-1-4503-4920-8. DOI: 10.1145/3071178.3071216. URL: doi.acm.org/10.1145/3071178.3071216 (cit. on pp. 21, 42).

[21] D.-C. Dang and P. K. Lehre. “Simplified Runtime Analysis of Estimation of Distribution Algorithms”. In: *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*. GECCO ’15. ACM, 2015, pp. 513–518. ISBN: 978-1-4503-3472-3. DOI: 10.1145/2739480.2754814. URL: http://doi.acm.org/10.1145/2739480.2754814 (cit. on p. 21).

[22] M. S. Krejca and C. Witt. “Lower Bounds on the Run Time of the Univariate Marginal Distribution Algorithm on OneMax”. In: *Proceedings of the 14th ACM/SIGEVO Conference on Foundations of Genetic Algorithms*. FOGA ’17. ACM, 2017, pp. 65–79. ISBN: 978-1-4503-4651-1. DOI: 10.1145/3040718.3040724. URL: http://doi.acm.org/10.1145/3040718.3040724 (cit. on p. 21).

[23] P. K. Lehre and P. T. H. Nguyen. “Improved Runtime Bounds for the Univariate Marginal Distribution Algorithm via Anti-concentration”. In: *Proceedings of the Genetic and Evolutionary Computation Conference*. GECCO ’17. ACM, 2017, pp. 1383–1390. ISBN: 978-1-4503-4920-8. DOI: 10.1145/3071178.3071317. URL: http://doi.acm.org/10.1145/3071178.3071317 (cit. on p. 21).

[24] F. Neumann, D. Sudholt, and C. Witt. “A Few Ants Are Enough: ACO with Iteration-best Update”. In: *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*. GECCO ’10. ACM, 2010, pp. 63–70. ISBN:

978-1-4503-0072-8. DOI: 10.1145/1830483.1830493. URL: http://doi.acm.org/10.1145/1830483.1830493 (cit. on p. 26).

[25] B. Doerr and L. A. Goldberg. “Drift Analysis with Tail Bounds”. In: *Parallel Problem Solving from Nature, PPSN XI*. Springer Berlin Heidelberg, 2010. ISBN: 978-3-642-15844-5 (cit. on p. 33).

[26] D. Williams. *Probability with Martingales*. Cambridge mathematical textbooks. Cambridge University Press, 1991. ISBN: 9780521406055. URL: https://books.google.com/books?id=e9saZ0YSi-AC (cit. on p. 51).

[27] W. Hoeffding. “Probability Inequalities for Sums of Bounded Random Variables”. In: *Journal of the American Statistical Association* 58.301 (1963), pp. 13–30. ISSN: 01621459 (cit. on pp. 53, 54).

A Appendix

The following is the MATLAB code used to run all the experiments.

A.1 Main function

```
function varargout = main(alg, fcn_vec, param, varargin)
%MAIN Function to run all implemented evolutionary algorithms with an
%option to set parameters through the input.
%
%Call the 'main' function by: main(arg1, arg2, arg3, arg4)
%
% ————————— INPUT ARGUMENTS: ——————————
%arg 1 — integer
%   Choose an algorihtm you want to run
%      1 — RLS
%      2 — (mu + 1) EA
%      3 — (mu + 1) GA
%      4 — (1 + lambda,lambda) GA
%      5 — cGA
%      6 — UMDA
%      7 — UMDA selection free
%
%arg 2 — vector
%   Choose what fitness function you want to use (must be a vector)
%      [1] — OneMax
%      [2, k] — Jump, where k is the length of the gap
%
%arg 3 — vector
%   List of parameters specific for the chosen algorithm (vector)
%      RLS — [n]
%      (mu + 1) EA — [n, mu, mutation probability]
%       (mu + 1) GA — [n, mu, mutation probability]
%       (1 + lambda,lambda) GA — [n, lambda]
%       cGA — [n, lambda, 1/K]
%       UMDA — [n, lambda, mu]
%
%arg 4 — integer
%   varargin — not mandatory input
%   sets a limit to maximum fitness evaluations
%
% ————————— OUTPUT ARGUMENTS: ———————————
%Algorithms 1—4 have one output: out1 — integer
%   out1 — number of fitness evaluations
%Algortihms 5—7 have two outputs: out1 — integer, out2 — matrix
%   out1 — number of fitness evaluations
%   out2 — marginal probabilities
%
%Example — [t, freq] = main(5, [2, 10], [100, 5, 1/100], 10^7);
%   This runs the cGA with n = 100, lambda = 5, K = 100 on jump_20 and
%   if the solution is not found within 10^7 fitness evaluations then
%   the algorithm is terminated

switch alg
    case 1
        if nargin == 3
            t = hillclimber(fcn_vec, param);
        else
            t = hillclimber(fcn_vec, param, varargin{1});
        end
        varargout{1} = t;
    case 2
        if nargin == 3
            t = EA(fcn_vec, param);
        else
            t = EA(fcn_vec, param, varargin{1});
        end
        varargout{1} = t;
    case 3
        if nargin == 3
            t = GA(fcn_vec, param);
        else
            t = GA(fcn_vec, param, varargin{1});
        end
        varargout{1} = t;
    case 4
        if nargin == 3
            t = lambdalambdaGA(fcn_vec, param);
        else
            t = lambdalambdaGA(fcn_vec, param, varargin{1});
        end
        varargout{1} = t;
    case 5
        if nargin == 3
            [t, Freq] = cGA(fcn_vec, param);
        else
            [t, Freq] = cGA(fcn_vec, param, varargin{1});
        end
        varargout{1} = t;
        varargout{2} = Freq;
    case 6
        if nargin == 3
            [t, Freq] = UMDA(fcn_vec, param);
        else
            [t, Freq] = UMDA(fcn_vec, param, varargin{1});
        end
        varargout{1} = t;
        varargout{2} = Freq;
    case 7
        if nargin == 3
            [t, Freq] = UMDA_sel(fcn_vec, param);
        else
            [t, Freq] = UMDA_sel(fcn_vec, param, varargin{1});
        end
        varargout{1} = t;
        varargout{2} = Freq;
    otherwise
        fprintf('Invalid algorithm choice.')
end

```

```
end
```

Listing A.1: Main function.

A.2 Help functions

```
function Xfit = onemax(N, ~)
%ONEMAX onemax fitness function

Xfit = 0:N;

end
```

Listing A.2: Generate fitness values for the OneMaxfitness function.

```
function [Xfit] = jumpfunction(N, k)
%JUMPFUNCTION jump_k fitness function

Xfit = 0:N;

onemax = Xfit <= N - k | Xfit == N;
gap = Xfit > N - k & Xfit < N;
Xfit(onemax) = k + Xfit(onemax);
Xfit(gap) = N - Xfit(gap);

end
```

Listing A.3: Generate fitness values for the Jump$_k$ fitness function.

```
function [X] = initializepopulation(N, mu)
%SELECTION u.a.r chooses a bit string from {0,1}^n

X = randi(2, mu, N) - 1;

end
```

Listing A.4: Initialize a population of μ individuals of length n.

```
function [val] = bino(n, p)
%BINO Generates a random number from binomial distribution

val = random('bino', n, p);

end
```

Listing A.5: Generates a random number from Binomial distribution.

A.3 Algorithms

```
function t = hillclimber(fcn_vec, param, varargin)
% Random local search

% stopping criterion
T = Inf(1);
if nargin == 3
    T = varargin{1};
end

% param
N = param(1); % bitstring length

% init
k = 0;
f = onemax(N);
if length(fcn_vec) == 2
    k = fcn_vec(2);
    f = jumpfunction(N, k);
end
t = 0;
x = initializepopulation(N, 1);
xfit = f(sum(x) + 1);

while xfit ~= N + k
    e = randi(N);
    if x(e) == 0
        x(e) = 1;
        xfit = f(sum(x) + 1);
    end

    t = t + 1;
    if t > T
        t = -1;
        break;
    end
end

end
```

Listing A.6: Random Local Search

```
function t = EA(fcn_vec, param, varargin)
% (mu + 1) EA

% stopping criterion
T = Inf(1);
if nargin == 3
    T = varargin{1};
end

% param
N = param(1); % bitstring length
mu = param(2); % population size
p = param(3); % mutation probability

% init
k = 0;
f = onemax(N);
if length(fcn_vec) == 2
    k = fcn_vec(2);
    f = jumpfunction(N, k);
end
t = mu;
X = initializepopulation(N, mu);
```

```
Xfit = f(sum(X, 2) + 1);
Xmax = max(Xfit);

while Xmax ~= N + k
    y = mod((rand([1, N]) <= p) + X(randi(mu), :), 2);
    yfit = f(sum(y) + 1);
    [~, index] = min(Xfit);
    if Xfit(index) <= yfit
        X(index, :) = y;
        Xfit(index) = yfit;
    end

    if yfit > Xmax
        Xmax = yfit;
    end

    t = t + 1;
    if t > T
        t = -1;
        break;
    end
end

end
```

Listing A.7: $(\mu + 1)$ EA

```
function t = GA(fcn_vec, param, varargin)
% (mu + 1) GA

% stopping criterion
T = Inf(1);
if nargin == 3
    T = varargin{1};
end

% param
N = param(1); % bitstring length
mu = param(2); % population size
```

```
p = param(3); % mutation probability

% init
k = 0;
f = onemax(N);
if length(fcn_vec) == 2
    k = fcn_vec(2);
    f = jumpfunction(N, k);
end
t = mu;
pc = 1/2;
X = initializepopulation(N, mu);
Xfit = f(sum(X, 2) + 1);
Xmax = max(Xfit);

while Xmax ~= N + k
    % crossover
    g = rand([1, N]);
    p1 = randi(mu);
    p2 = randi(mu);
    while p1 == p2
        p2 = randi(mu);
    end

    y = X(p1, :);
    y(g >= pc) = X(p2, g >= pc);

    % mutation
    y = mod((rand([1, N]) <= p) + y, 2);
    yfit = f(sum(y) + 1);
    [~, index] = min(Xfit);
    if Xfit(index) <= yfit
        X(index, :) = y;
        Xfit(index) = yfit;
    end

    if yfit > Xmax
        Xmax = yfit;
    end
    t = t + 1;
    if t > T
        t = -1;
        break;
    end
end

end
```

Listing A.8: $(\mu + 1)$ GA

```
function t = lambdalambdaGA(fcn_vec, param, varargin)
% (1+(lambda,lambda)) GA

% stopping criterion
T = Inf(1);
if nargin == 3
    T = varargin{1};
end

% param
N = param(1); % bitstring length
lambda = param(2);

% init
kgap = 0;
f = onemax(N);
if length(fcn_vec) == 2
    kgap = fcn_vec(2);
    f = jumpfunction(N, kgap);
end
t = 1;
k = lambda;
p = k/N;
c = 1/k;
x = initializepopulation(N, 1);
xfit = f(sum(x) + 1);

```

```
while xfit ~= N + kgap
    X = zeros(lambda, N);
    l = bino(N, p);
    for i = 1:lambda
        mut = randperm(N, l);
        y = x;
        y(mut) = mod(y(mut)+1, 2);
        X(i, :) = y;
    end

    Xfit = f(sum(X, 2) + 1);
    [~, index] = max(Xfit);
    xprime = X(index, :);
    t = t + lambda;

    for i = 1:lambda
        g = rand([1, N]);
        y = x;
        y(g < c) = xprime(g < c);
        X(i, :) = y;
    end

    Xfit = f(sum(X, 2) + 1);
    [~, index] = max(Xfit);
    xprime = X(index, :);
    xprimefit = Xfit(index);

    if xprimefit >= xfit
        x = xprime;
        xfit = xprimefit;
    end

    t = t + lambda;
    if t > T
        t = -1;
        break;
    end
end

```

```
end
```

Listing A.9: $(1+\lambda, \lambda)$ GA

```
function [t, Freq] = cGA(fcn_vec, param, varargin)
% cGA

% stopping criterion
T = Inf(1);
if nargin == 3
    T = varargin{1};
end

% param
N = param(1); % bitstring length
lambda = param(2);
K = param(3);

% init
k = 0;
f = onemax(N);
if length(fcn_vec) == 2
    k = fcn_vec(2);
    f = jumpfunction(N, k);
end
t = 0;
xfit = 0;
p(1, 1:N) = 1/2;
Freq = zeros(10ˆ6, N);
Freq(1, :) = p;
counter = [1, 1];

while xfit ˜= N + k
    % stores every 10ˆ2 vector of marginal probabilities for
    % efficiency reasons (can be changed as desired)
    if mod(counter(1), 10ˆ2) == 1
        Freq(counter(2), :) = p;
        counter(2) = counter(2) + 1;
    end
    P = rand([lambda, N]) <= p;
    Pfit = f(sum(P, 2) + 1);
    if lambda == 2
        x = P(1, :);
        xfit = Pfit(1);
        y = P(2, :);
        yfit = Pfit(2);
    else
        [~, I] = sort(Pfit, 'descend');
        x = P(I(1), :);
        xfit = Pfit(I(1));
        y = P(I(2), :);
        yfit = Pfit(I(2));
    end

    if yfit > xfit
        temp = x;
        x = y;
        y = temp;
        xfit = yfit;
    end

    high = x > y;
    low = x < y;
    p(high) = p(high) + K;
    p(low) = p(low) - K;
    bounds = true;
    if bounds
        p(p < 1/N) = 1/N;
        p(p > 1 - 1/N) = 1 - 1/N;
    end

    t = t + lambda;
    counter(1) = counter(1) + 1;
    if t > T
        t = -1;
        break;
    end
    if all(p >= 1 - 1/N | p <= 1/N)
```

```
        t = -2;
        break;
    end
end

Freq = Freq(1:counter(2) - 1, :);
end
```

Listing A.10: cGA$_\lambda$

```
function [t, Freq] = UMDA(fcn_vec, param, varargin)
% UMDA

% stopping criteion
T = Inf(1);
if nargin == 3
    T = varargin{1};
end

% param
N = param(1); % bitstring length
lambda = param(2); % offspring size
mu = param(3); % number of best offspring to choose

% init
k = 0;
f = onemax(N);
if length(fcn_vec) == 2
    k = fcn_vec(2);
    f = jumpfunction(N, k);
end
t = 0;
xfit = 0;
p(1, 1:N) = 1/2;
Freq = zeros(100000, N);
Freq(1, :) = p;
counter = 1;

while xfit ~= N + k
```

```
    Freq(counter, :) = p;
    P = rand([lambda, N]) <= p;
    Pfit = f(sum(P, 2) + 1);
    xfit = max(Pfit);

    [~, I] = sort(Pfit, 'descend');
    update = sum(P(I(1:mu), :));
    update = update / mu;

    bounds = true;
    if bounds
        low = update < 1/N;
        high = update > 1 - 1/N;
        p(low) = 1/N;
        p(high) = 1 - 1/N;
        rest = ~(low | high);
        p(rest) = update(rest);
    else
        p(1, :) = update(1, :);
    end

    t = t + lambda;
    counter = counter + 1;
    Freq(counter, :) = p;
    if t > T
        t = -1;
        break;
    end
    if all(p >= 1 - 1/N | p <= 1/N)
        t = -2;
        break;
    end
end

Freq = Freq(1:counter - 1, :);
end
```

Listing A.11: UMDA

```
function [t, Freq] = UMDA_sel(fcn_vec, param, varargin)
% UMDA selection free

% stopping criterion
T = Inf(1);
if nargin == 3
    T = varargin{1};
end

% param
N = param(1); % bitstring length
lambda = param(2); % offspring size
mu = param(3); % number of best offspring to choose
sel_off = param(4); % time to turn off selection

% init
k = 0;
f = onemax(N);
if length(fcn_vec) == 2
    k = fcn_vec(2);
    f = jumpfunction(N, k);
end
t = 0;
xfit = 0;
p(1, 1:N) = 1/2;
Freq = zeros(100000, N);
Freq(1, :) = p;
counter = 1;

while xfit ~= N + k
    Freq(counter, :) = p;
    P = rand([lambda, N]) <= p;
    Pfit = f(sum(P, 2) + 1);
    xfit = max(Pfit);

    if counter > sel_off
        update = sum(P) / lambda;
    else
        [~, I] = sort(Pfit, 'descend');
        update = sum(P(I(1:mu), :));
        update = update / mu;
    end

    bounds = true;
    if bounds
        low = update < 1/N;
        high = update > 1 - 1/N;
        p(low) = 1/N;
        p(high) = 1 - 1/N;
        rest = ~(low | high);
        p(rest) = update(rest);
    else
        p(1, :) = update(1, :);
    end

    t = t + lambda;
    counter = counter + 1;
    if t > T
        t = -1;
        break;
    end
    if all(p >= 1 - 1/N | p <= 1/N)
        t = -2;
        break;
    end
end

Freq = Freq(1:counter - 1, :);
end
```

Listing A.12: UMDA^{sel}