Introduction of statistics in optimization

Fabien Teytaud

To cite this version:

Fabien Teytaud. Introduction of statistics in optimization. Machine Learning [cs.LG]. Université Paris Sud - Paris XI, 2011. English. NNT : . tel-00655731v1

HAL Id: tel-00655731
https://theses.hal.science/tel-00655731v1

Submitted on 2 Jan 2012 (v1), last revised 9 Jan 2012 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Thesis : Introduction of statistics in optimization

Fabien Teytaud

08/12/2011

Contents

I Evolutionary Optimization 5

1 Introduction 9
1.1 Evolutionary Algorithms . . . 9
1.2 Evolution Strategies . . . 10
1.2.1 Notions and Definitions . . . 10
1.2.2 Adaptation of the step-size . . . 12
One-fifth rule. . . . 12
Mutative Self-Adaptation. . . . 14
Cumulative Step-Size Adaptation. . . . 15
1.2.3 State of the Art Evolution Strategies . . . 16
The Covariance Matrix Adaptation Evolution Strategy 16
The Covariance Matrix Self-Adaptation Evolution Strategy 18
1.3 Estimation of Distribution Algorithms . . . 19
1.4 Other stochastic optimization algorithms . . . 20
Monte-Carlo Search. . . . 20
Differential Evolution. . . . 20

2 Issues with large population sizes 23
2.1 Introduction . . . 26
2.1.1 Notations and definitions . . . 26
2.1.2 Real world algorithms analysis . . . 27
One fifth rule. . . . 28
Self-Adaptation. . . . 28
Cumulative Step-Size Adaptation. . . . 30
2.1.3 Experimental analysis . . . 31
2.2 Log(λ) modifications for optimal parallelization . . . 45
2.2.1 New selection ratio . . . 45
2.2.2 Faster decrease of the step-size with reweighting . . . 58
2.2.3 Estimation of Multivariate Normal Algorithms analysis 60
2.3 Automatic parallelization . . . 77

3 Conclusion 81

II Multistage Optimization 85

4 Introduction 89
4.1 Evaluation function 89
4.2 The game of Go 90
4.3 The game of Havannah 92

5 State of the art 97
5.1 Alpha-Beta algorithm 97
5.2 Bandit Based Monte-Carlo Tree Search 98
5.2.1 Multi-Armed Bandits 99
5.2.2 Monte-Carlo Tree Search 100
Definitions. 100
Algorithm. 100
Modification of the bandit formula. 102
All Moves As First. 103
Rapid Action Value Estimates. 103
Last Good Reply. 104
5.3 Nested Monte-Carlo 104

6 Contributions 107
6.1 Application to the game of Havannah 107
6.1.1 Bandit Formula 108
6.1.2 Progressive Widening 109
6.1.3 Rapid Action Value Estimates 110
6.2 Improvement of the default policy 110
6.2.1 poolRave 111
6.2.2 Contextual Monte-Carlo 115
6.2.3 Decisive Moves 118
Complexity Analysis. 118
Experiments. 121
6.3 Tuning of the Nested Algorithm 123
6.4 Conclusion 132

Index 141

Bibliography 141

Acknowledgements

Now my thesis is almost over, I would like to acknowledge many people for their involvement in my work or in my life during these 3 years. First, I would like to thank Professor Damien Ernst and Professor Martin Müller for accepting to review my work and for their great comments. I would like to also thank Professor Abdel Lisser, Professor Frédéric Saubion and Professor Liva Ralaivola for accepting to be in my committee.
I will now switch to French, because I would like to thank a lot of French speakers.

Je voudrais remercier mes directeurs de thèse, Marc Schoenauer et Olivier Teytaud.
Merci Marc pour avoir accepté de m'encadrer durant cette thèse, mais aussi pour m'avoir accepté dans l'équipe (Merci à Michèle également). Je tiens tout particulièrement à remercier Olivier, tout d'abord pour m'avoir fait confiance en acceptant d'être co-directeur de cette thèse mais aussi pour tous les bons moments passés ensemble. Le contexte était particulier et difficile, mais tu as su très bien gérer tout cela. J'ai énormément appris à tes côtés, tant professionnellement que sur le plan humain.
Je tiens à remercier également plusieurs membres de l'équipe TAO: Jean-Baptiste, j'ai tout autant apprécié travailler à tes côtés que jouer aux échecs avec toi. Merci également d'avoir relu mon manuscrit. Hassen j‘ai été ravi de partager un bureau avec toi tout ce temps.
Je n'oublie pas non plus les anciens de TAO, avec qui j'ai passé de très bons moments. Arpad, cela a été un plaisir de travailler avec toi ces dernières années. Je mentirais si je disais que cela n'avait pas également été un plaisir de partir en conférence avec toi. Cédric merci pour tes conseils et ces moments de rire. Raymond, tes conseils ont toujours été très bons, et on se doit une belle au basket. Romaric et Philippe cela a été un plaisir de faire tous ces sig ensemble (Romaric, non je ne te remercie pas pour travian).
Un merci particulier à Anne, avec qui tout a commencé. Merci également à Adrien, Jean-Marc, Jacques, Julien, Ludovic et tous ceux que j'oublie pour

toutes ces grandes discussions au cesfo et tous ces bons moments.
Un merci également au staff du LRI. Vous avez toujours été très serviables et très réactifs. Un merci à Grid5000 aussi, qui permet de faire des expériences parallèlles rapidement et efficacement.
Bien sûr, je tiens à remercier chaleureusement ma famille, en particulier, ma mère Edith et ma soeur Valérie pour leur soutien permanent, à Maud qui est une des personnes les plus positives et enthousiastes que je connaisse. Je n'oublie pas mes amis, merci à Mathieu, avec qui j'ai eu la chance de découvrir la collocation, c'était génial. Merci à Julien, Imad pour tous ces bons moments.
Et enfin, un énorme merci à ma compagne Audrey, qui, pendant ces années, et surtout, durant les moments les plus difficiles, a su me donner un soutien et un réconfort indispensables pour un tel projet.

Introduction

In this document, we are interested in Artificial Intelligence (AI). AI is an important branch of computer-science. We can find a lot of different definitions of AI. For instance, we can find that AI is the branch of computer-science making computers behave like humans. This definition is a little simple, but it is interesting in the sense that if we ask non computer-scientist people to answer the question "What is Artificial Intelligence ?", this may be the most given definition. A more accurate possible definition, is that an intelligent agent is an agent able to perceive its environment and able to take actions that maximize its chance of success.

For a lot of people, the most famous example of success in AI is the 6-game matches between Gary Kasparov and the program Deep Blue. The first one was in 1996, and the human player defeated the program 4-2. In 1997, the program won 3.5-2.5, and this second match is considered as the most spectacular Chess event in history.

However, today, in a lot of problems (including games), computer are much weaker than humans. This is, in my opinion, one of the main reason why AI is still so interesting. It is important to be able to improve the level of computers for different reasons :

- Creating artificial intelligence stronger than humans at some tasks is interesting at a philosophical level, for understanding what is unique in humans

- Computers or machines could replace humans for difficult or dangerous tasks. The main popular examples are army (defuse a mine or a bomb for instance) or medicine (for instance doing difficult operations).

- Help the human when the decisions to make involve important calculations.

As we said, an important point in AI is the decision making process. The agent has to take decisions automatically. This criterion is restrictive, because with this notion we exclude a lot of algorithms or methods, such as,

for instance, all the methods specialized on a specific field (3D simulations, computer graphics, flow calculation ...). The decision process is domain dependent here, whereas our problematic is the decision making process in its independent aspect.

Another important point in computer science, is the raise of parallel structures. It seems more and more obvious that future of computer is based on parallel architectures. There are several reasons for this change. The main reason is that it becomes harder (and really expensive) to build more powerful processors.

In Section 2, we show how one can adapt some AI algorithms for parallel machines.

In this document, we are interested in two different types of decision making. First, if we take one unique decision, this case is called optimization. In Section I, we present different optimization algorithms. We are not interested in Gradient or Hessian based methods, because these methods are too specific. We study more precisely Evolution Algorithms, which are famous to be robust and therefore not problem-dependent. Second, if we take several decisions, this is called multistage optimization. In Section II, we present different methods dealing with this problem. We are particularly interested in Monte-Carlo Tree Search algorithms, because these algorithms are known for being generic.

Part I

Evolutionary Optimization

First, in Section 1 we introduce some state of the art optimization algorithms and more specifically evolution strategies. In a second part, in Section 2, we present the main problem studied in this section, which is the parallelization of evolution strategies. After that, in Section 3 we discuss and summarize the different problems ans solutions seen in Section 2.

Chapter 1

Introduction

First, in Section 1.1, we define what are Evolutionary Algorithms in general. After that, in Section 1.2 we present more specifically some state of the art Evolution Strategies. In Section 1.3, we present the Estimation of Distribution algorithm and finally in Section 1.4 some other stochastic optimization algorithms which are not used in this thesis.

1.1 Evolutionary Algorithms

Evolutionary Algorithms (EAs) belong to the family of stochastic optimization algorithms. They are biologically inspired algorithms that crudely mimic reproduction, mutations, recombination and selection. They are modelled according to Darwin's evolution theory. Individuals of the population represent candidate solutions of the optimization problem. A generation is an iteration of the algorithm (one step of the evolution process), mutations are represented by random blind variations on individuals. A full process can be described as follows : under the environment pressure, mutations and crossovers are applied to the individuals of the population. After a small number of generations, best individuals emerge from the population. Algorithmically, the fitness is represented by the objective function, the population of individuals denotes the set of possible solutions, mutations are blind random variations and a generation is an iteration of the algorithm. An evolutionnary algorithm is presented in Figure 1.1. During the initialization part, a random population of candidate solutions is generated. The evaluation of a population consists in computing the fitness function for each individual of the population. The selection part is the process consisting in choosing the *parents*. According to their fitness, individuals among the population are chosen and biased towards the ones with the best fitness..

Crossover recombines the parental genes and mutation is a little variation of an individual in a random way. One iteration of the whole process is called a generation.

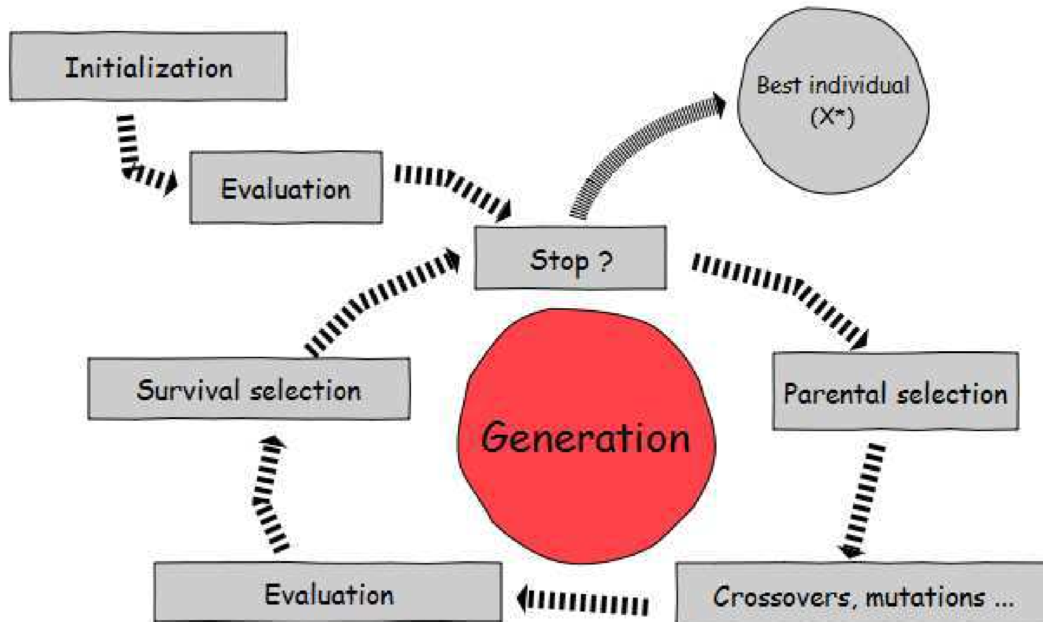

Figure 1.1: Evolutionary algorithm: the process.

1.2 Evolution Strategies

Evolution strategies (ESs) belong to the family of evolutionary algorithms, they are applied in the continuous case. We note λ the number of individuals belonging to the population. μ denotes the parent population size, i.e. the number of individuals in the selected population. First, in Section 1.2.1, we present the evolution strategies, and then in Section 1.2.2 we present some state of the art algorithms.

1.2.1 Notions and Definitions

Evolution Strategies have been initially introduced in [Rechenberg, 1973] and [Schwefel, 1981]. As shown in [Teytaud and Fournier, 2008], according to the

selection process, we can consider four families of algorithms :

- Selection-based non-elitist Evolution Strategies ($SB-(\mu,\lambda)-ESs$). In these algorithms, λ individuals are generated and constitute the population. Then, the μ best candidate individuals from the population are kept to generate the new population. Here the information needed is simply the μ points, in particular no ranking is necessary.

- Selection-based elitist Evolution Strategies ($SB-(\mu+\lambda)-ESs$). In these algorithms, λ individuals are generated, and the μ best individuals of the union between the population and the μ selected individuals at the previous generation are selected. As for $SB-(\mu,\lambda)-ESs$ we do not require the full ranking of the individuals.

- Full ranking non-elitist Evolution Strategies ($FR-(\mu,\lambda)-ESs$). Such the $SB-(\mu,\lambda)-ES$, after the generation of the λ individuals, the μ best individuals from the population are kept to generate the new population except that in this case, we also need to keep the complete ranking of these μ individuals. The complete ranking is necessary because in some cases the weight of each individual will be rank-dependent.

- Full ranking elitist Evolution Strategies ($FR-(\mu+\lambda)-ESs$) can be defined in the same way. The ranking of the μ best individuals of the union of the current population and the previous selected set is kept.

When the size of the selected population is larger than 1, recombination (i.e., the construction of a new parent by mixing selected individuals) can be applied in order to generate new parents. For fixed values of μ and λ, the two families using full ranking use more information than the two others. The expected consequence of this additional cost is that full ranking strategies should be faster.

In ES, mutations are mainly represented by Gaussian mutations. A parent x generates an offspring y as follows :

$$y = x + \sigma\mathcal{N}(0, C),$$

where σ denotes the step-size, $\mathcal{N}(0, C)$ denotes the standard multivariate normal variables with mean 0 and covariance matrix C.

The key point in ESs is the adaptation of the parameters of the process and in particular the adaptation of the step-size and the adaptation of the covariance matrix. In the next section, we focus our attention on the critical step-size σ. We now present in Section 1.2.2 the different possibilities for adapting this parameter.

1.2.2 Adaptation of the step-size

One important point in ES is the adaptation of the step-size σ. This adaptation is different from one algorithm to another, and this is this specification which is used to differentiate the different ESs. It is important to have an adaptative step-size, because if the step-size is constant and too small w.r.t. the distance to the optimum, the new individual will be close to the parent and the progression will be slow. In the other case, if the step-size is constant and too large w.r.t. the distance to the optimum, the probability that the new individual be better than its parent will be too small. In this section, we present different rules for adapting the step-size.

One-fifth rule.

The one-fifth rule has been successfully introduced in [Rechenberg, 1973]. The principle is to increase the step-size if the probability of success $\hat{p}$ is greater than $\frac{1}{5}$ and to decrease it otherwise. The value $\frac{1}{5}$ is an approximation of the optimal probability of success on the Sphere function and on the corridor function [Rechenberg, 1971, Rechenberg, 1973]. The probability of success is the probability that one offspring is better than its parent. Algorithm 1 presents this first method. The update of the step-size σ is done in lines 14 and 16 in Algorithm 1. Another update of σ found in literature is

$$\sigma = K^{(\hat{p}-\frac{1}{5})}$$

with K a constant greater than 1.

Algorithm 1 One-fifth rule.

1: **argument** dimension $d \in \mathbb{N}$
2: **Initialize** $\sigma \in \mathbb{R}$, $y \in \mathbb{R}^d$
3: **while** Halting criterion not reached **do**
4:     **for** $i \leftarrow 1$ **to** λ **do**
5:         $s_i \leftarrow \mathcal{N}_i(0, Id)$
6:         $y_i \leftarrow y + \sigma s_i$
7:         $f_i \leftarrow f(y_i)$
8:     Sort the individuals by increasing fitness ; $f_{(1)} < f_{(2)} \ldots < f_{(\lambda)}$
9:     Or define the set of the μ best individuals
10:     Compute p as the number of successful offsprings divided by λ $(\max\{i; f_{(i)} < f(y)\})$
11:     $s^{avg} \leftarrow \frac{1}{\mu} \sum_{i \leftarrow 1}^{\mu} s_{(i)}$
12:     $y' \leftarrow y + \sigma s^{avg}$
13:     **if** y' better than y **then**
14:         $y = y'$; $\sigma = 2\sigma$
15:     **else**
16:         $y = y'$; $\sigma = 2^{-1/4}\sigma$

Mutative Self-Adaptation.

The mutative Self-Adaptation Evolution-Strategy (SA-ES) has been first proposed in [Rechenberg, 1973] and [Schwefel, 1974]. The idea is to mutate the mutation parameters by working on the assumption that a bad mutation parameter will not generate a good offspring. If the mutation of the step-size is bad, then the resulting step-size will be either too large or too small. In the first case, the probability of having a good offspring is low, and in the second case, the improvement will be small. The main advantages are its simplicity and its genericity. Another important point is, as we will see in Section 2.1.3, that it is more amenable to parallelization than the Cumulative Step-size Adaption or the One-fifth rule. This algorithm is presented in Algorithm 2.

Algorithm 2 Mutative self-adaptation algorithm. For large population size τ is usually equal to $1/\sqrt{N}$; other tuning of τ can been used (e.g. $1/\sqrt{2N}$) which is sometimes found in papers.

argument dimension $d \in \mathbb{N}$
Initialize $\sigma^{avg} \in \mathbb{R}$, $y \in \mathbb{R}^d$.
while Halting criterion not reached **do**
  for $i = 1..\lambda$ **do**
    $\sigma_i = \sigma^{avg} e^{\tau \mathcal{N}_i(0,1)}$
    $z_i = \sigma_i \mathcal{N}_i(0, Id)$
    $y_i = y + z_i$
    $f_i = f(y_i)$
  Sort the individuals by increasing fitness; $f_{(1)} < f_{(2)} < \cdots < f_{(\lambda)}$.
  $z^{avg} = \frac{1}{\mu} \sum_{i=1}^{\mu} z_{(i)}$
  $\sigma^{avg} = \frac{1}{\mu} \sum_{i=1}^{\mu} \sigma_{(i)}$
  $y = y + z^{avg}$

Cumulative Step-Size Adaptation.

The last well-known algorithm for choosing the step-size is the Cumulative Step-size Adaptation (CSA). It was proposed in [Hansen and Ostermeier, 1996, Hansen and Ostermeier, 2001]. The principle of this method is to compare the length of the path followed by the algorithm to the length of the path followed under random selection. If the path followed by the algorithm is larger than the path under random selection then the step-size is increased. In the other case, the step-size is decreased. The detailed algorithm is presented in Algorithm 3.

Algorithm 3 Cumulative step-size adaptation.

argument dimension $d \in \mathbb{N}$
Initialize $\sigma \in \mathbb{R}$, $y \in \mathbb{R}^d$.
while halting criterion not fulfilled **do**
  for $i = 1..\lambda$ **do**
    $s_i = \mathcal{N}_i(0, Id)$
    $y_i = y + \sigma s_i$
    $f_i = f(y_i)$
  Sort the individuals by increasing fitness; $f_{(1)} < f_{(2)} < \cdots < f_{(\lambda)}$.
  Or define the set of the μ best individuals
  $s^{avg} = \frac{1}{\mu} \sum_{i=1}^{\mu} s_{(i)}$
  $y = y + \sigma s^{avg}$
  $p_\sigma = (1-c)p_\sigma + \sqrt{\mu c(2-c)} s^{avg}$
  $\sigma = \sigma \exp[\frac{||p_\sigma|| - \overline{\chi_d}}{D \overline{\chi_d}}]$

In Algorithm 3, $\overline{\chi_d}$ is approximated by $\sqrt{d} \times (1 - \frac{1}{4.0 \times d} + \frac{1}{21.0 \times d^2})$, and d is the dimension. Following [Hansen, 1998], $c = \frac{1}{\sqrt{d}}$ and $D = \sqrt{d}$. A main weakness of this formula is its moderate efficiency, for λ large, as pointed out in [Beyer and Sendhoff, 2008] and detailed in Section 2.

1.2.3 State of the Art Evolution Strategies

In this section, we present two state of the art ESs. The first one is the Covariance Matrix Adaptation Evolution Strategy, presented in Section 1.2.3. The second ES is the Covariance Matrix Self-Adaptation Evolution Strategy, presented in Section 1.2.3.

The Covariance Matrix Adaptation Evolution Strategy

The Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [Hansen and Ostermeier, 2001] is one of the most famous ES. The rule used for adapting the step-size is CSA (presented in Section 1.2.2, Algorithm 3). The key point is that the CMA-ES updates a full covariance matrix for the sample distribution. Consequently, CMA-ES learns all pairwise dependencies between all parameters. The CMA-ES is presented is Algorithm 4.

Algorithm 4 The Covariance Matrix Adaptation Evolution Strategy. $< . >$ denotes the recombination. $p_c = 0$, $p = 0$,

Initialize $\sigma \in \mathbb{R}$, $y \in \mathbb{R}^d$, $C = Id$.
while Halting criterion not fulfilled **do**
  for $i = 1..\lambda$ **do**
    $s_i = \sqrt{C} N_i(0, Id)$
    $y_i = y + \sigma s_i$
    $f_i = f(y_i)$
  $y = y + \sigma < s >_w$
  $p = \left(1 - \frac{1}{\tau_p}\right) p + \sqrt{\frac{1}{\tau_p}\left(2 - \frac{1}{\tau_p}\right)} \sqrt{\mu_{eff}} < s >_w$
  $C = (1 - \frac{1}{\tau_c})C + \frac{1}{\tau_c}\left[\frac{1}{\mu_{eff}} pp^T + (1 - \frac{1}{\mu_{eff}}) < ss^T >\right]$
  $p_\sigma = (1 - c_\sigma)\, p_\sigma + \sqrt{c_\sigma(2 - c_\sigma)} \sqrt{\mu_{eff}} < N(0, Id) >_w$
  $\sigma = \sigma \exp\left(\left(\frac{||p_c||}{\chi_d} - 1\right) \cdot \frac{c_\sigma}{d_\sigma}\right)$

c_σ,d_σ and μ_{eff} are parameters of the algorithm. The default values of these parameters are:

- $c_\sigma = \frac{\mu_{eff}+2}{d+\mu_{eff}+3}$
- $d_\sigma = 1 + 2\max(0, \sqrt{\frac{\mu_{eff}-1}{d+1}} - 1)$
- $\mu_{eff} = \frac{1}{\sum_{i=1}^{\mu}(w_i^2)}$

The default value of λ is $\lambda = 4 + \lfloor 3 + \ln(d) \rfloor$ and the default value of μ is $\mu = \frac{\lambda}{2}$.

The Covariance Matrix Self-Adaptation Evolution Strategy

The Covariance Matrix Self-Adaptation Evolution Strategy (CMSA-ES) [Beyer and Sendhoff, 2008] is based on the SA algorithm presented previously. In the CMSA-ES, the routine used to adapt the global step size σ is the Self-Adaptation one (Section 1.2.2, Algorithm 2). But as for CMA-ES, here a full covariance matrix is used, in order to learn the shape of the fitness function. This algorithm is presented in Algorithm 5.

Algorithm 5 Covariance Matrix self-adaptation. τ is equal to $1/\sqrt{d}$; $< . >$ represents the recombination. The initial covariance matrix C is the identity matrix. The time constant τ_C is equal to $1 + \frac{d(d+1)}{\frac{\lambda}{2}}$.

Initialize $\sigma^{avg} \in \mathbb{R}$, $y \in \mathbb{R}^d$, C.
while Halting criterion not fulfilled **do**
  for $i = 1..\lambda$ **do**
    $\sigma_i = \sigma^{avg} e^{\tau N_i(0,1)}$
    $s_i = \sqrt{C}\sigma_i N_i(0, Id)$
    $z_i = \sigma_i s_i$
    $y_i = y + z_i$
    $f_i = f(y_i)$
  Sort the individuals by increasing fitness; $f_{(1)} < f_{(2)} < \cdots < f_{(\lambda)}$.
  $z^{avg} = \frac{1}{\mu}\sum_{i=1}^{\mu} z_{(i)}$
  $s^{avg} = \frac{1}{\mu}\sum_{i=1}^{\mu} s_{(i)}$
  $\sigma^{avg} = \frac{1}{\mu}\sum_{i=1}^{\mu} \sigma_{(i)}$
  $y = y + z^{avg}$
  $C = (1 - \frac{1}{\tau_C})C + \frac{1}{\tau_C} < ss^T >$

1.3 Estimation of Distribution Algorithms

Estimation of Distribution Algorithms (EDAs) [Mühlenbein and Paass, 1996, Bosman and Thierens, 2000, Pelikan et al., 2002] are another well known stochastic optimization algorithms. The principle is to use a probability distribution to represent the potential solutions. The EDA iteratively estimates the parameters of the distribution by :

- sampling the domain with the current parametrized distribution
- evaluating the sampled points,
- selecting the best points,
- rebuilding the probability distribution described by these points.

Estimation of Multivariate Normal Algorithm (EMNA) [Larranaga and Lozano, 2002] belongs to the family of EDA. It is the case in which we use a Gaussian distribution estimated by maximum likelihood. This algorithm is presented in Algorithm 6.

Algorithm 6 The EMNA algorithm.

---

argument dimension $d \in \mathbb{N}$
Initialize $\sigma \in \mathbb{R}$, $y \in \mathbb{R}^d$.
while Halting criterion not reached **do**
  for $i = 1..\lambda$ **do**
    $z_i = \sigma \mathcal{N}_i(0, Id)$
    $y_i = y + z_i$
    $f_i = f(y_i)$
  Sort the individuals by increasing fitness; $f_{(1)} < f_{(2)} < \cdots < f_{(\lambda)}$.
  Or define the set of the μ best individuals
  $z^{avg} = \frac{1}{\mu}\sum_{i=1}^{\mu} z_{(i)}$
  $\sigma = \sqrt{\frac{\sum_{i=1}^{\mu} ||z_{(i)} - z^{avg}||^2}{\mu d}}$
  $y = y + z^{avg}$

1.4 Other stochastic optimization algorithms

We have seen Evolution algorithms which belong to stochastic algorithms. We now introduce other stochastic methods. First in Section 1.4 we see Monte-Carlo Search, and then in Section 1.4 the Differential Evolution algorithm.

Monte-Carlo Search.

Monte-Carlo Search, also known as Pure Random Search [Brooks, 1958], is one of the simplest optimization algorithms. The process consists in randomly sampling each point in the search space, according to a fixed probability distribution and keep the best point so far. This method converges asymptotically to the optimum [Zhigljavsky and Zilinskas, 2007]. Indeed, in practice, the time needed to find the optimum is really long, and increases exponentially with the search space dimension [Zhigljavsky and Zilinskas, 2007].

Differential Evolution.

Differential Evolution (DE) [Price et al., 2005, Price, 1996, Storn, 1996, Storn and Price, 1995, Storn and Price, 1997], belongs to the family of evolutionary algorithms. As in evolutionary algorithms, a population of candidate solutions evolves generation after generation. The four main steps are the same, they consist in initialization, mutation, recombination and selection. The difference with evolutionary algorithms is in the mutation process. Each individual, i.e. candidate solution, is represented by d-dimensional vectors. The initial vectors are randomly generated. At each generation we have :

$$P_x = (x_i), i = 1, \ldots, \lambda$$

$$x_i = (x_{i,j}); j = 1, \ldots, d$$

x_i denotes the ith individual).

The mutation step is defined as follows :

$$v_i = x_{i_1} + F(x_{i_2} - x_{i_3}), i = 1, \ldots, \lambda$$

i_1,i_2 and i_3 are indices randomly chosen in the population, i.e. in $1, \ldots, \lambda$ and F is a positive constant.

The recombination step defines a new vector u_i. A parameter, called crossover probability (Cr), determines the amount of information taken by v_i. Formally :

$$u_{i,j} = \begin{cases} v_{i,j} \text{ with probability Cr and} \\ x_{i,j} \text{ with probability 1-Cr} \end{cases}$$

The selection step is deterministic, the candidate u_i replaces x_i if it is better. Formally,

$$x_i = \begin{cases} u_i \text{ if } f(u_i) \leq f(x_i) \\ x_i \text{ otherwise} \end{cases}$$

Chapter 2

Issues with large population sizes

In this chapter we are interested by the case of large population sizes. The motivation behind the study of this particular case is that it is often said that evolutionary algorithms are well suitable for parallelization. In evolutionary algorithm, a population of candidate solutions evolves. Nowadays, the architecture of computers is changing, to become more and more parallel. Generally, in optimization problems, the critical step of the whole process (in term of time) is the evaluation of a candidate solution. With a large number of processors available, a natural and simple parallelization of evolutionary algorithms is to evaluate one candidate solution per processor. With this simple parallelization, the population size corresponds to the number of available processors. Generally in Optimization, the number of evaluations needed to reach a certain target fitness is measured [Hansen et al., 2009]. In this study, we are interested in measuring the number of iterations needed to reach a target function instead of the number of evaluations. The reason is simply that we consider that at each generation (i.e. iteration), the λ evaluations are done in parallel, then, if we increase the population size, the number of evaluations needed to reach a target fitness will also increase, but the number of iterations needed to reach the same target will decrease.

Let us take an example with the last version of CMA (Algorithm 4). The fitness function is the Sphere function (defined in Table 2.1), the target fitness is fixed to 10^{-10} and the dimension is 10. Results are plotted in Figure 2.1.

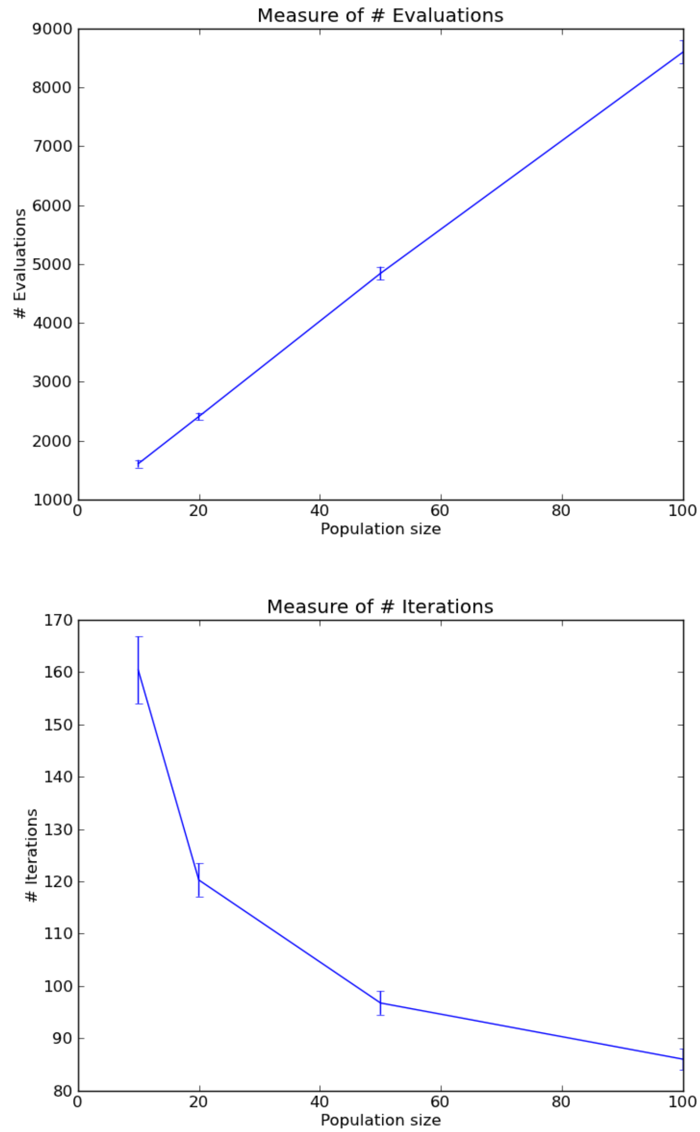

Figure 2.1: Top: measure of the number of evaluations as a function of λ. Bottom: measure of the number of iterations as a function of λ.

In this study, our goal is to measure the parallel performances (called the wall-clock time) of evolutionary algorithms, and not to minimize the number of evaluations (called the computational cost).

First, in Section 2.1 we study the behavior of evolutionary algorithms in the case of large population sizes, and we see that the results are far worse than what we could expect. In Section 2.2 we propose different solutions to solve this problem of large population size. Finally, in Section 2.3 we present an automatic parallelization of evolution strategies.

2.1 Introduction

2.1.1 Notations and definitions

Complexity bounds are here expressed with convergence ratio. Following [Fournier and Teytaud, 2011] the convergence ratio is defined as :

$$CR_\epsilon = \frac{\log N(\epsilon)}{dn_\epsilon}$$

where

- n_ϵ is the number of iterations necessary for ensuring, that with probability at least $\frac{1}{2}$, the algorithm estimates the location of the optimum with a precision ϵ. The constant $\frac{1}{2}$ is arbitrary and similar results have been found with a confidence $1-\delta$ ($\delta > 0$).
- d is the dimension.
- $N(\epsilon)$ corresponds to the 2ϵ-packing number, i.e. the maximum number k such that there exist k points in the domain with pairwise distance 2ϵ.

A faster algorithm means a larger convergence ratio. Usually, the convergence rate is define as :

$$-\log(\textit{convergence rate}) = \lim_{\epsilon \to 0} CR_\epsilon$$

According to O. Teytaud, S. Gelly in [Teytaud and Gelly, 2006] and O.Teytaud and H. Fournier in [Fournier and Teytaud, 2011], the advantage of using the convergence ratio over the convergence rate is that the speed-up between two algorithms corresponds to the ratio between their convergence ratios, and the number of iterations for reaching a given precision is proportional to the inverse of the convergence ratio.

H. Fournier and O. Teytaud have recently shown a several theoretical bounds for ESs [Fournier and Teytaud, 2011], from which we will retain :

- For $(\mu/\mu, \lambda) - ES$ the optimal speed-up is linear for $\lambda < d$. For larger values of λ the speed-up becomes logarithmic as a function of λ.
- For $(1, \lambda) - ES$ the speed-up is logarithmic as a function of λ.

It is important to bear in mind that $(\mu/\mu, \lambda) - ES$ corresponds to an evolution strategy with a population size λ and a selected population size of

μ, and the parent used to create the next population is the average of the individuals belonging to the selected population. $(1, \lambda) - ES$ is an evolution strategy with a population λ and a selected population size of 1, i.e. μ equals to 1 (the best individual in the population is the new parent).

Let us define the different objective functions used in this chapter. The first objective function is the Sphere function, which is a well-known test function in optimization. The second and third ones are the Schwefel Ellipsoid and the Cigar functions, for which the adaptation of the covariance matrix is helpful. The last one is the Rosenbrock function which requires, due to its shape, continuous changes of the covariance matrix. These functions are presented in Table 2.1.

Name	Objective function
Sphere	$f(y) = \sum_{i=1}^{d} y_i^2$
Schwefel	$f(y) = \sum_{i=1}^{d}(\sum_{j=1}^{i} y_i)^2$
Cigar	$f(y) = y_1^2 + 10000 \times \sum_{i=2}^{d} y_i^2$
Rosenbrock	$f(y) = \sum_{i=1}^{d}(100(y_i^2 - y_{i+1})^2 + (y_i - 1)^2)$

Table 2.1: : Objective functions considered in this chapter.

We now present that many real-world algorithms are far from these theoretical bounds. In Section 2.1.2, we show that the one-fifth rule, the self-adaptation algorithm and the cumulative step-size adaptation algorithm do not reach the optimal speed-up. In Section 2.1.3, we see experimentally that the speed-up of real world algorithms is far from the theoretical bounds.

2.1.2 Real world algorithms do not all reach the optimal speed-up

In this section, we show that the one-fifth rule (Algorithm 1), the Self-Adaptation algorithm (Algorithm 2) and the Cumulative Step-Size Adaptation algorithm (Algorithm 3) do not reach the optimal speed-up when the population becomes large but a bounded speed-up, i.e. in O(1) as $\lambda \to \infty$. As said previously, the optimal speed-up is $\log(\lambda)$ for $\lambda \to \infty$. The results on

this section have been published in the paper [Teytaud and Teytaud, 2010b].

We define $\eta^* = \frac{\sigma_{n+1}}{\sigma_n}$ (η^* depends on n, but we consider a fixed value of n here and therefore we drop this dependency in the notation η^*). η^* small means that σ decreases quickly and η^* close to 1 means a slow decrease. The important point is that the convergence rate is lower bounded by η^*. Hence, $\eta^* = \Omega(1)$ implies that the speed-up can not be $\Theta(\log(\lambda))$. This is the key point for showing that many Real-World algorithms do not reach the optimal speed-up.

One fifth rule.

The one-fifth rule has been introduced in Section 1.2.2. The one-fifth rule can be applied in different manners to $(\mu/\mu, \lambda)$ algorithms. Consider $\hat{p}$ equal to the ratio between (i) the number of generated individuals with fitness better than the center of the Gaussian generating the offspring (ii) the number of generated individuals; $0 \leq \hat{p} \leq 1$. A first possible implementation of the one-fifth rule is

$$\hat{p} \leq 1/5 \Rightarrow \eta^* = K_1 \in]0,1[\text{ and } \hat{p} > 1/5 \Rightarrow \eta^* = K_2 > 1 \tag{2.1}$$

$$\text{and a second version is } \eta^* = K_3^{(\hat{p}-1/5)} \text{ for some } K_3 > 1. \tag{2.2}$$

Proposition 1: *For the one-fifth rule, implemented as in Eq. 2.1 or in Eq. 2.2, there exists a constant $C > -\infty$ such that $\mathbb{E}\log(\frac{\sigma_{n+1}}{\sigma_n}) > C$.*

Proof: In the first case (Eq. 2.1, we see that $\eta^* \geq K_1 > 0$; and in the second case, $\eta^* \geq K_3^{-1/5} > 0$; in both cases, η^* is lower bounded by a positive constant; $\log(\eta^*)$ is therefore lower bounded by a constant $> -\infty$. The proposition is proved.□ Therefore, we have shown that with the one-fifth rule, the convergence ratio (and therefore the convergence rate) is $O(1)$ (as $\lambda \to \infty$).

Self-Adaptation.

The proof of the limited speed-up for SA requires the following lemma.

Lemma: *The expected logarithm of the average (arithmetic or geometric average) of the μ smallest of λ independent standard log-normal random variables, with $\frac{\mu}{\lambda} \to k > 0$ and $\mu > 0$, is lower bounded by some constant $> -\infty$. More formally, if $N_{(1)}, \ldots, N_{(\lambda)}$ are sorted realizations of standard independent Gaussian variables, and $L_{(i)}$ is $\exp(N_{(i)})$, then*

$$\inf_{\lambda > 0} \mathbb{E} \log \frac{1}{\mu} \sum_{i=1}^{\mu} \exp(N_{(i)}) > -\infty \textit{ and } \inf_{\lambda > 0} \mathbb{E} \frac{1}{\mu} \sum_{i=1}^{\mu} N_{(i)} > -\infty.$$

Proof: Define $N_1, \ldots, N_\lambda$ realizations of standard independent Gaussian variables (with mean 0 and variance 1). Define $L_i = \exp(N_i)$. Following usual notations in evolutionary computation, consider indices $(1), \ldots, (\lambda)$ such that $L_{(1)}, ..., L_{(\lambda)}$ are these independent log-normal standard random variables, sorted in non-decreasing order.

With probability 1, all the N_i are distinct; we therefore suppose without loss of generality that they are all distinct in calculus below. Then (we show that result for the arithmetic average, as the case of the geometric average is in fact simpler),

$$\begin{aligned}
& \mathbb{E}\left[\log \frac{1}{\mu} \sum_{i \leq \mu} \exp(N_i)\right] \\
\geq \quad & \mathbb{E}\left[\frac{1}{\mu} \sum_{i \leq \mu} N_{(i)}\right] \text{ (as exp is convex and log is non-decreasing)} \\
= \quad & \left(\mathbb{E}[N_{(1)}] + \mathbb{E}[N_{(2)}] + \mathbb{E}[N_{(3)}] + ... + \mathbb{E}[N_{(\mu)}]\right)/\mu \\
& \text{(as expectation and summation commute)} \\
= \quad & \mathbb{E}\left[N | \{(1) \leq \mu\}\right] \\
= \quad & \mathbb{E}\left[N | \{N_1 \leq N_{(\mu)}\}\right] \\
\underset{\lambda \to \infty}{\longrightarrow} \quad & CVAR_{\mu/\lambda}(N) \text{ thanks to [Chen, 2008, Theorem 1]} \qquad (2.3) \\
> \quad & -\infty.
\end{aligned}$$

$CVAR_\alpha$ is the conditional value at risk, i.e. the average of the α first quantiles, obviously finite for the Gaussian distribution; the convergence in line 2.3 is the convergence of the empirical CVAR to its asymptotic value, established in [Chen, 2008] in a general setting including this one. This concludes the proof.□

Proposition 2. *Consider a SA algorithm in which σ_{n+1} is the average (geometric or arithmetic average) of $\sigma_n \times L_1, \sigma_n \times L_2, \ldots, \sigma_n \times L_\lambda$, for $L_1, \ldots, L_\lambda$ as in the lemma above. Then, there exists some $C > -\infty$ such that $\mathbb{E}\log(\frac{\sigma_{n+1}}{\sigma_n}) > C$.*

Remark: Rescaling the N_i by any constant (equivalently, $L_i = \exp(kN_i)$ for some $k > 0$) does not change the result.

Proof: Consider a Self-Adaptation algorithm. In such algorithms, the new step-size is the average (arithmetic or geometric average) of selected step-sizes. Each mutation of a step-size is log-normal, we consider the mean of the μ selected mutation, which is lower bounded by the mean of the μ individuals with smallest step-sizes. Therefore, $\mathbb{E}\log(\eta^*) \geq \mathbb{E}\log mean_{i \leq \mu} \exp(N_i)$ (in

the arithmetic case, the other case is similar); the latter quantity is lower bounded by the lemma above and this concludes the proof.

Therefore, the lemma shows that the convergence ratio of (μ, λ)-ES with self-adaptation is bounded ($O(1)$) as $\lambda \to \infty$. □

Cumulative Step-Size Adaptation.

Now we show that Cumulative Step-size Adaptation (CSA) (presented in Section 1.2.2) does not reach optimal speed-up $\log(\lambda)$. We (classically) formalize an iteration of CSA in dimension d as follows:

$$w_i \geq 0, \sum_{i=1}^{\mu} w_i = 1 \tag{2.4}$$

$$\mu_{eff} = \frac{1}{\sum_{i=1}^{\mu}(w_i^2)} \tag{2.5}$$

$$\chi_d > 0 \quad , \quad ||p_c|| \geq 0 \tag{2.6}$$

$$d_\sigma = 1 + 2\max(0, \sqrt{\frac{\mu_{eff} - 1}{d+1}} - 1) \tag{2.7}$$

$$c_\sigma = \frac{\mu_{eff} + 2}{d + \mu_{eff} + 3} \tag{2.8}$$

$$\sigma_{n+1} = \sigma_n \exp\left(\left(\frac{||p_c||}{\chi_d} - 1\right) \cdot \frac{c_\sigma}{d_\sigma}\right).$$

($||.||$ does not have to be a norm, we just need Eq. 2.6). These assumptions, or equivalent variants of them, to the best of our knowledge, hold in all current implementations of CSA. We then show the following

Proposition 3. *For any dimension d, there exists $C > 0$ such that, for any λ, $\eta_n^* = \frac{\sigma_{n+1}}{\sigma_n} \geq C$.*

Proof: We write η^* as follows:

$$\eta^* = \frac{\sigma_{n+1}}{\sigma_n} = \exp\left((\frac{||p_c||}{\chi_d} - 1) \cdot \frac{c_\sigma}{d_\sigma}\right). \tag{2.9}$$

$$\text{Eq. 2.4 and Eq. 2.5 imply} \quad \forall \mu, \mu_{eff} \geq 1. \tag{2.10}$$

$$\text{Eq. 2.10 and Eq. 2.8 lead to} \quad \frac{3}{d+4} \leq c_\sigma \leq 1. \tag{2.11}$$

$$\text{Eq. 2.7 leads to} \quad d_\sigma \geq 1. \tag{2.12}$$

$$\text{By assumptions 2.6,} \quad \frac{||p_c||}{\chi_d} - 1 \geq -1. \tag{2.13}$$

$$\text{Eq. 2.9 and Eq. 2.13 lead to} \quad \eta^* \geq \exp(-\frac{c_\sigma}{d_\sigma}). \tag{2.14}$$

$$\text{Finally Eq. 2.11, Eq. 2.12 and Eq. 2.14 give} \quad \eta^* \geq \exp(-1) \tag{2.15}$$

which yields the expected result. □

This proposition shows that $\eta^* \geq \exp(-1)$; this implies that $CR \leq 1$, *i.e.* for cumulative step-size adaptation the speed-up is $O(1)$ for $\lambda \to \infty$.

2.1.3 Experimental analysis

In this section, we compare three step-size adaptation rules in the particular case on large population sizes. The first rule is the SA algorithm, define in Algorithm 2, the second rule is the CSA algorithm defined in Algorithm 3 and the last one is an EDA algorithm as defined in Algorithm 6.

These results have been published in [Teytaud and Teytaud, 2009b].

We have done our experiments on three different objective functions. The objective functions are the Sphere function, the Schwefel function and the cigar function. These functions are defined in Table 2.1. The first experiment confirms the superiority of SA over CSA when the population is large. This superiority has been first shown in [Beyer and Sendhoff, 2008]. In this experiment, Beyer and Sendhoff show the superiority of the Covariance Matrix Self-Adaptation Algorithm (presented in Section 1.2.3, Algorithm 5) against the Covariance Matrix Adaptation algorithm (presented in Section 1.2.3, Algorithm 4).

In our experiments, we measure the numbers of iterations needed to reach a halting criterion as a function of the dimensionality. The halting criterion is fixed and equals to $f_{stop} = 10^{-10}$. These experiments are presented in Figure 2.2 for $\lambda = 8$, Figure 2.3 for $\lambda = 4 \times d$ and Figure 2.4 for $\lambda = 4 \times d^2$.

The second set of experiments is the measure of $d \times \log(||x - x^*||)/n$ as a function of λ, where: d is the dimensionality; x is the best point in the last λ offspring; x^* is the optimum ($x^* = 0$ for our test functions); n is the number of iterations before the halting criterion is met. Each run is performed until $f(x) < 10e^{-50}$. For the sake of statistical significance each point is the average of 300 independent runs.

In these experiments, we compare SA-ES, CSA-ES and an EMNA algorithm as define in Algorithm 6. The EMNA algorithm is called SSA (standing for Statistical Step-size Algorithm) in our experiments. Results are presented

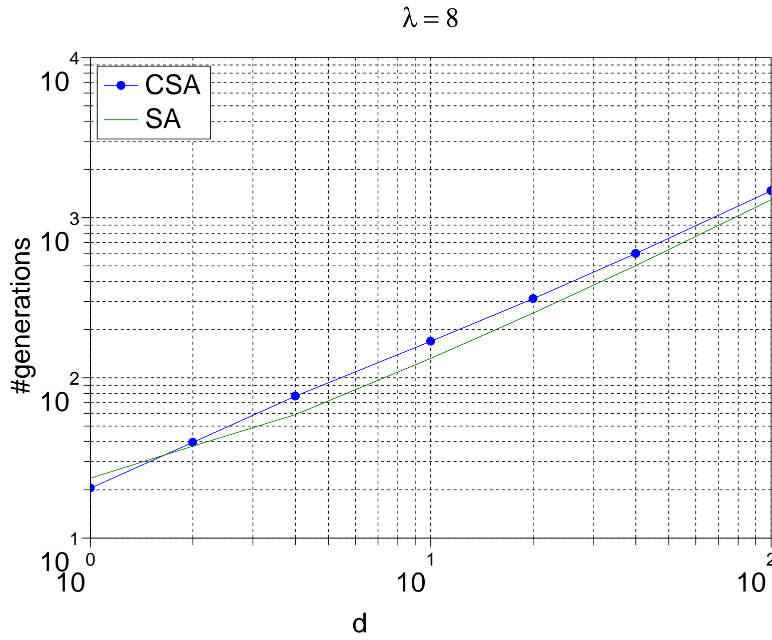

Figure 2.2: Results for $\lambda = 8$. Performance of CSA and SA on the Sphere function (number of iterations before $f(x) < 10^{-10}$) depending on the dimensionality. In these experiments, $\mu = \frac{1}{4}\lambda$.

in Figure 2.5 and Figure 2.6 for $\mu = \lambda/2$, Figure 2.7 and Figure 2.8 for $\mu = \lambda/4$, and Figures 2.9 and Figure 2.10 for $\mu = 1$.

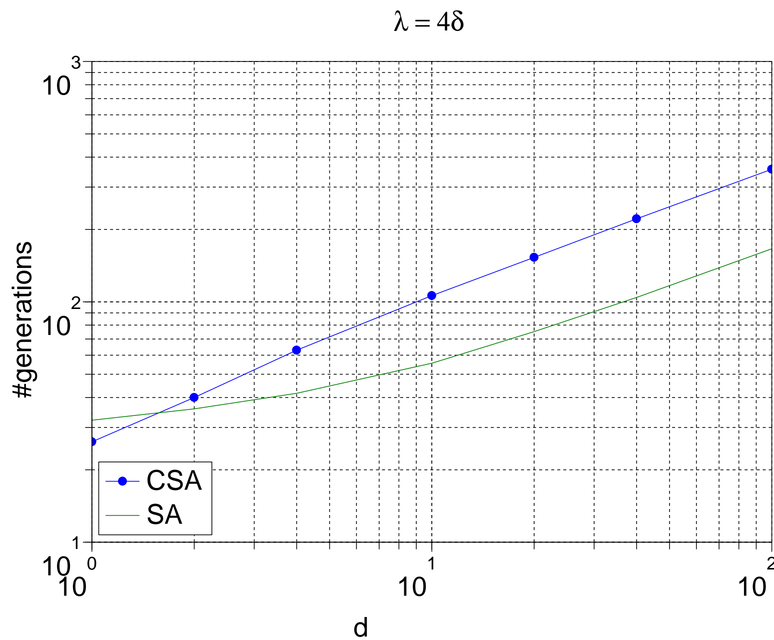

Figure 2.3: Results for $\lambda = 4 \times d$. Performance of CSA and SA on the Sphere function (number of iterations before $f(x) < 10^{-10}$) depending on the dimensionality. In these experiments, $\mu = \frac{1}{4}\lambda$.

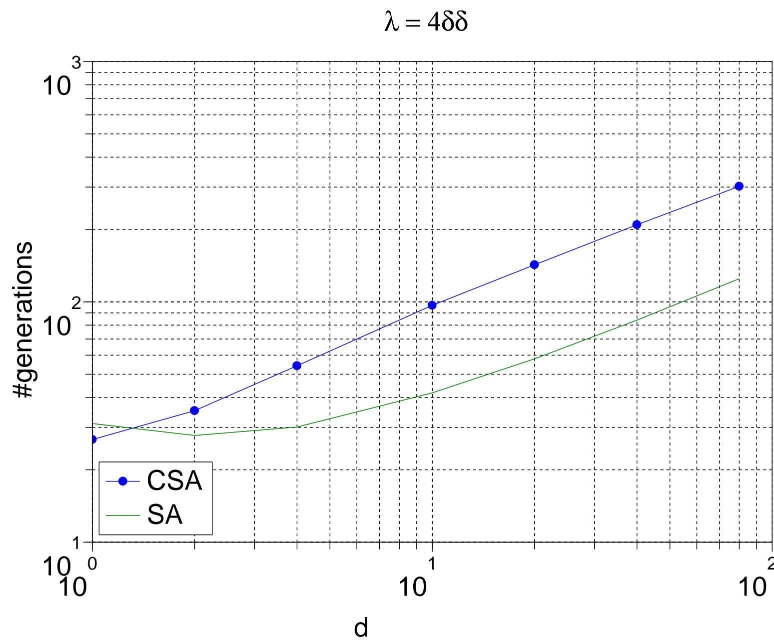

Figure 2.4: Results for $\lambda = 4 \times d^2$. Performance of CSA and SA on the Sphere function (number of iterations before $f(x) < 10^{-10}$) depending on the dimensionality. In these experiments, $\mu = \frac{1}{4}\lambda$.

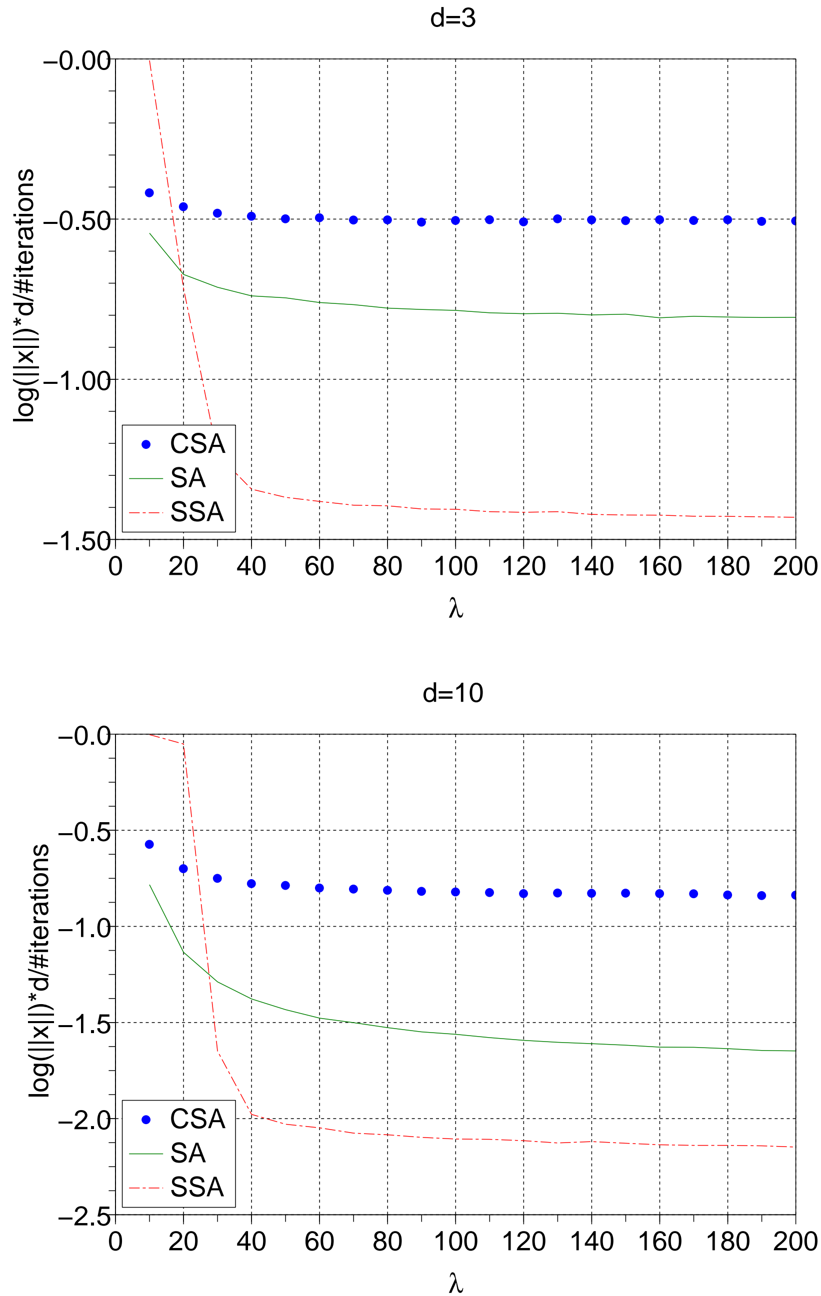

Figure 2.5: Log of distance to the optimum when the halting criterion ($f(x) < 10e^{-50}$) is met, normalized by the dimension and the number of iterations. Results for $\mu = \lambda/2$, on the Sphere function in dimension 3 (Top) and dimension 10 (Bottom). In all cases, SSA is the best rule for λ large.

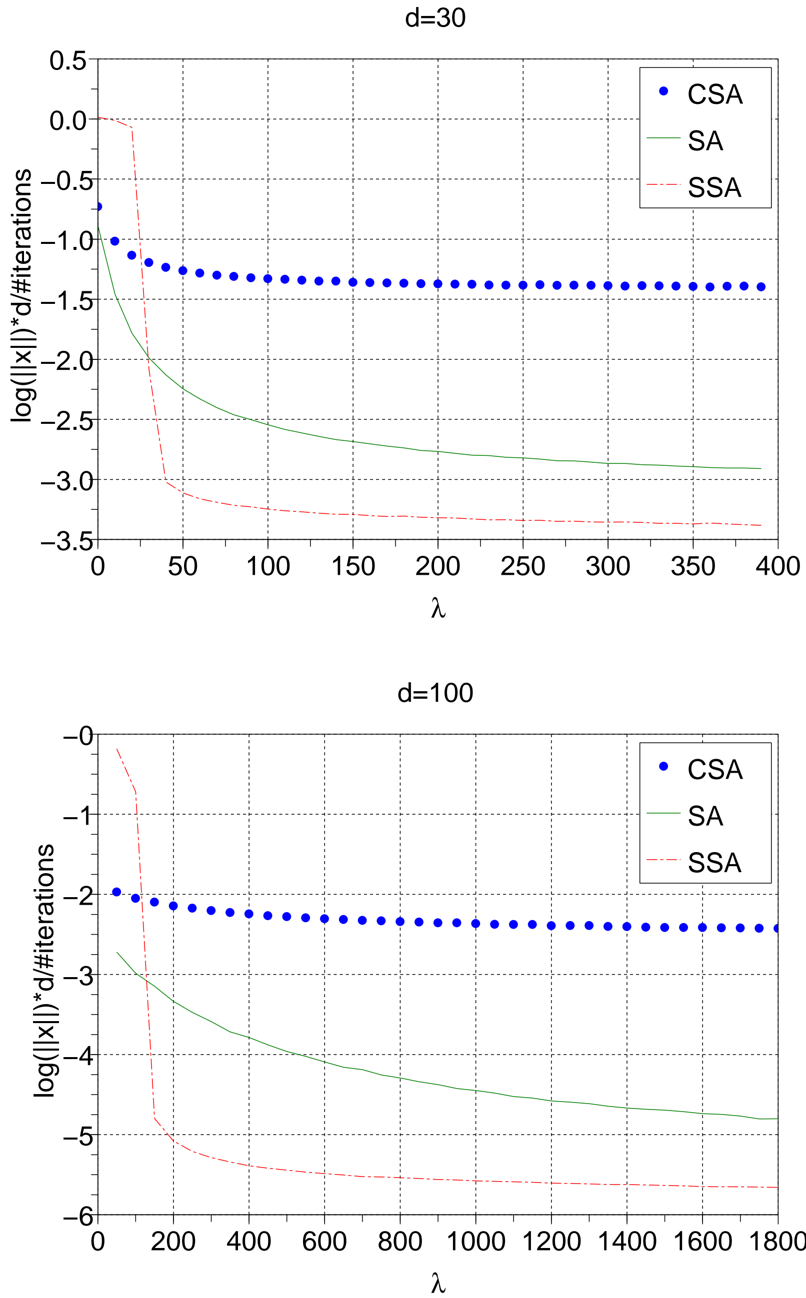

Figure 2.6: Log of distance to the optimum when the halting criterion ($f(x) < 10e^{-50}$) is met, normalized by the dimension and the number of iterations. Results for $\mu = \lambda/2$, on the Sphere function in dimension 30 (Top) and dimension 100 (Bottom). In all cases, SSA is the best rule for λ large.

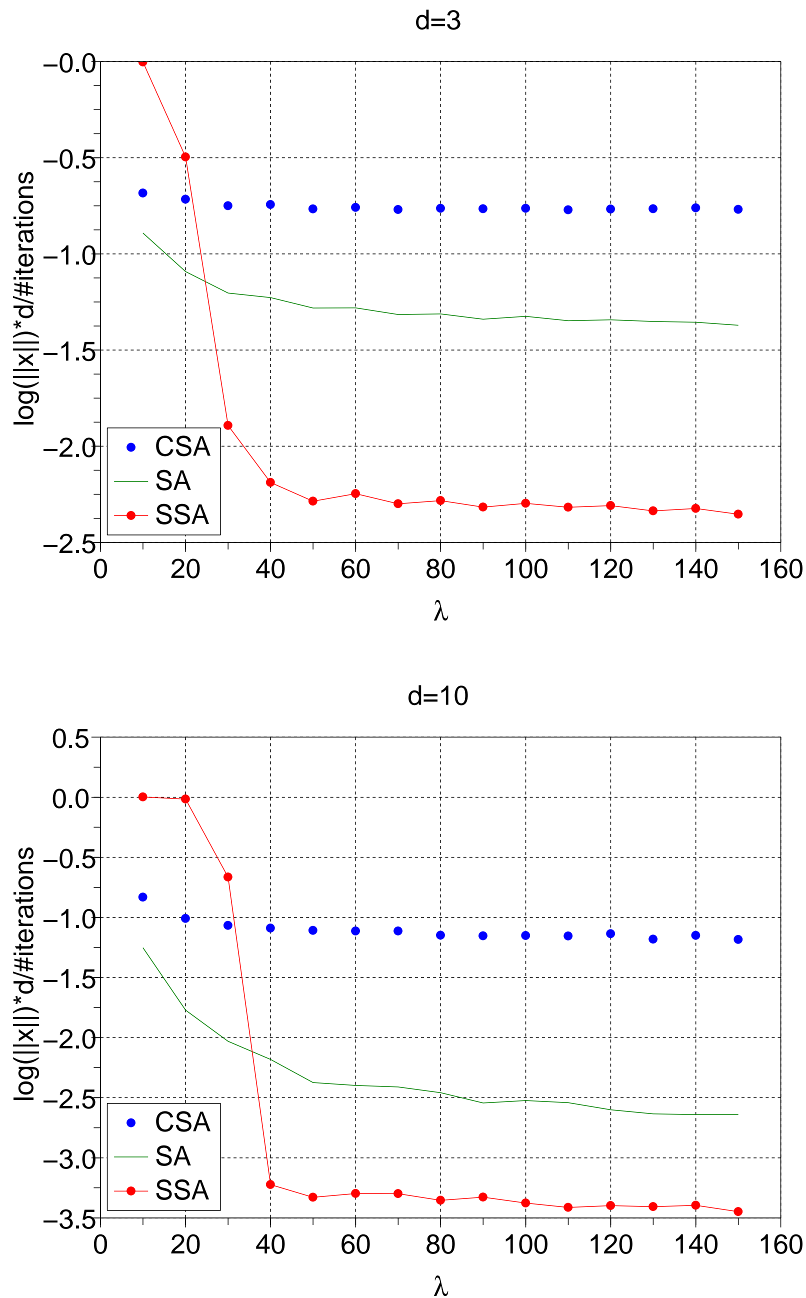

Figure 2.7: Results for $\mu = \lambda/4$ on the Sphere function in dimensions 3 (Top) and 10 (Bottom). All methods are better than for $\mu = \lambda/2$. In all cases, SSA is the best rule for λ large.

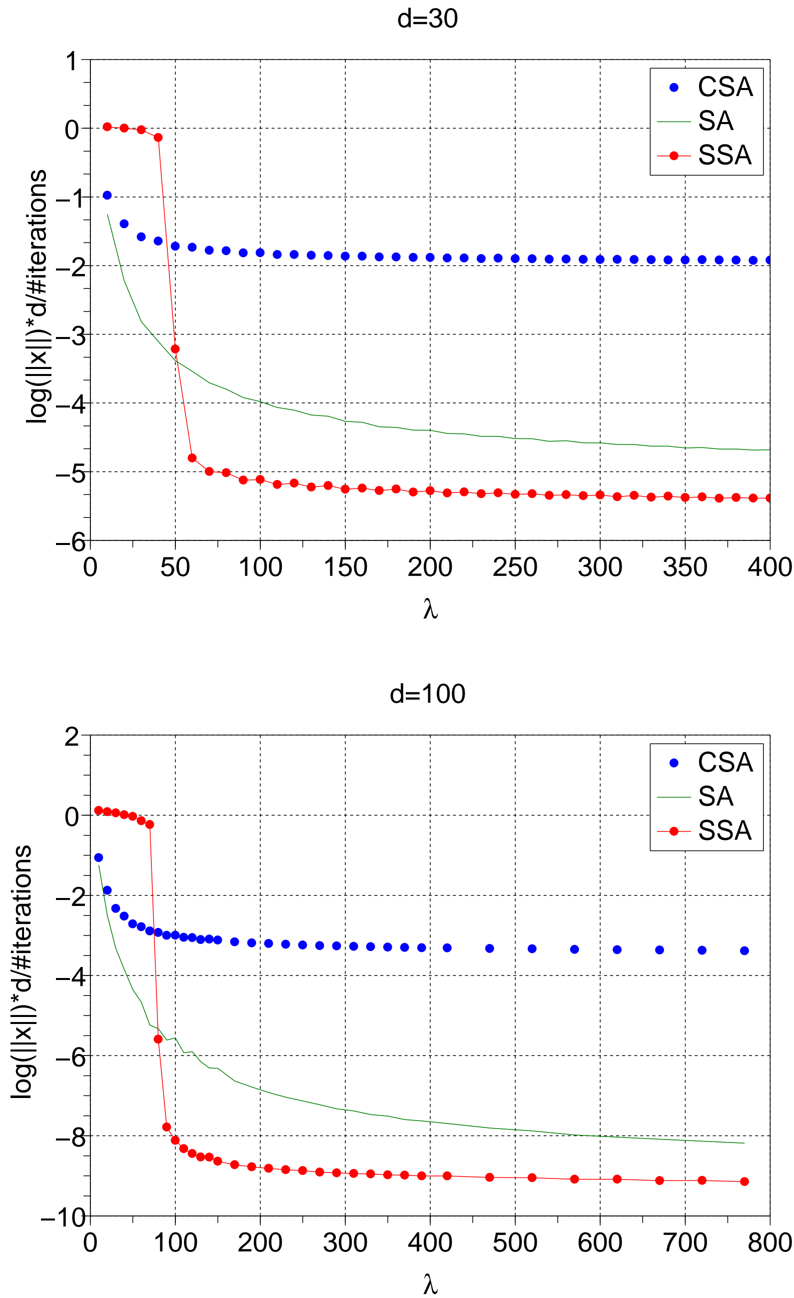

Figure 2.8: Results for $\mu = \lambda/4$ on the Sphere function in dimensions 30 (Top) and 100 (Bottom). All methods are better than for $\mu = \lambda/2$. In all cases, SSA is the best rule for λ large.

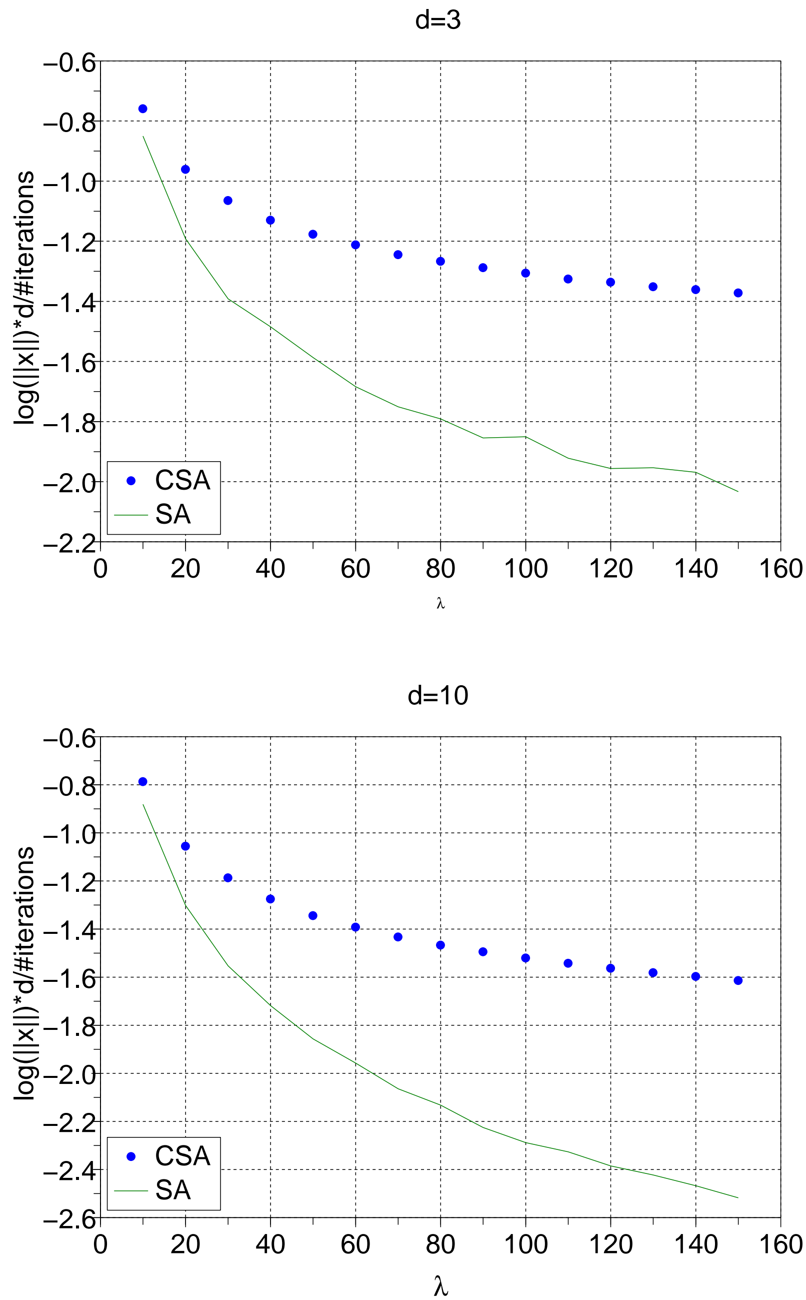

Figure 2.9: Results for $\mu = 1$ on the Sphere function in dimensions 3 (Top) and 10 (Bottom). SSA is not presented as it does not make sense for $\mu = 1$. As shown by this figure (compared to $\mu = \frac{1}{4}$ and $\mu = \frac{1}{2}$), $\mu = 1$ is quite weak for large dimension and the absence of SSA version for that case is therefore not a trouble.

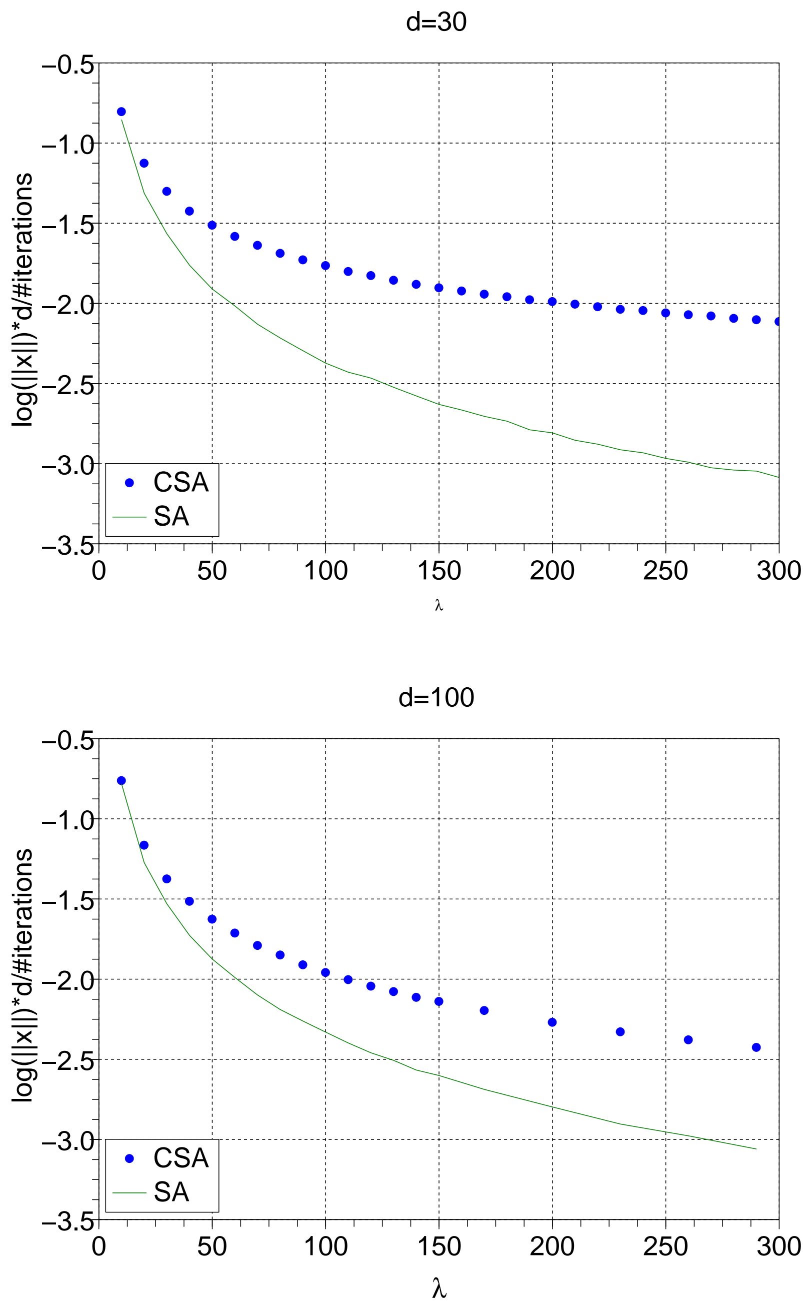

Figure 2.10: Results for $\mu = 1$ on the Sphere function in dimension 30 (Top) and 100 (Bottom). SSA is not presented as it does not make sense for $\mu = 1$. As shown by this figure (compared to $\mu = \frac{1}{4}$ and $\mu = \frac{1}{2}$), $\mu = 1$ is quite weak for large dimension and the absence of SSA version for that case is therefore not a trouble.

All these experiments have been done with simple algorithm without covariance matrix (the covariance matrix is represented by the identity matrix). We now repeat these experiments with the same algorithms but with the addition of a diagonal covariance matrix. This form of covariance matrix is intermediate between the full matrix and the matrix proportional to identity. It has been pointed out in [Ros and Hansen, 2008] that this separable version is in many cases faster than the complete covariance matrix. We have in this case to determine one step-size per axis. Figures 2.11 and 2.12 present the results of these experiments. We can see that

- On the Schwefel function, the results are the same as those on the Sphere function.
- On the Cigar function, the results are not as similar as the results on the Sphere function, but are not too far. Even on this ill-conditioned function, the EMNA algorithm gives good results.

In all these experiments two main results can be given.
First, we can note the superiority of EMNA algorithm when the population size becomes large. The advantages of EMNA are multiple. It is :

- very simple,
- fully-portable to the full-covariance matrix case,
- computationally cheap,
- highly intuitive,
- parameter-free (only classical parameters μ and λ),
- efficient for large population sizes.

It provides a speed-up of 100% (over mutative self-adaptation, which is already much better than CSA) on the Sphere function for $d = 3$, decreasing to 33% for $d = 10$ and 25% for $d = 100$ (in the case $\lambda = 150$). SA and CSA adaptation-rules use, as covariance matrix, a combination of an old covariance matrix and a new estimate based on the current population - in EMNA, we only keep the second term as λ large simplifies the problem.
Second, we show that the anisotropic version works in the sense that the convergence rate on the Sphere was recovered with the Schwefel and the Cigar function. The anisotropic version of the algorithm is the algorithm with a diagonal covariance matrix, i.e. there is one step-size per axis (the covariance

matrix in the isotropic version of the algorithm, presented in Algorithm 6 is the identity matrix).

Some by-products of our results are now pointed out, and compared with theoretical results from [Teytaud and Fournier, 2008]:

- The higher the dimensionality, the better the speed-up for $(\mu/\mu, \lambda)$-algorithms; [Beyer, 2001] has shown the linear speed-up as a function of λ as long as $\lambda << d$, and [Fournier and Teytaud, 2011] has proved that the speed-up is linear until λ of the same order of the dimension. Roughly, a number of processors linear as a function of the dimension is efficient. This is visible on our experimental results (Figures 2.5, 2.6, 2.7 and 2.8).

- Also, $(1, \lambda)$ algorithms (case $\mu = 1$) have a less-than-linear (logarithmic) speed-up as a function of λ. This is visible in our experimental results (Figures 2.9 and 2.10). This is also consistent with [Beyer, 2001] and [Teytaud and Fournier, 2008].

- On the Schwefel function, the results of the anisotropic version of the algorithm are the same as in the case of the Sphere function. The isotropic algorithms have, in this framework, a result close to 0 (i.e. much worse). Therefore, the anisotropic step-size adaptation works for this moderately ill-conditioned function.

- On the Cigar function, the results are not exactly the same as those of the Sphere function, but almost; even on this much more ill-conditioned function, the anisotropic SSA works.

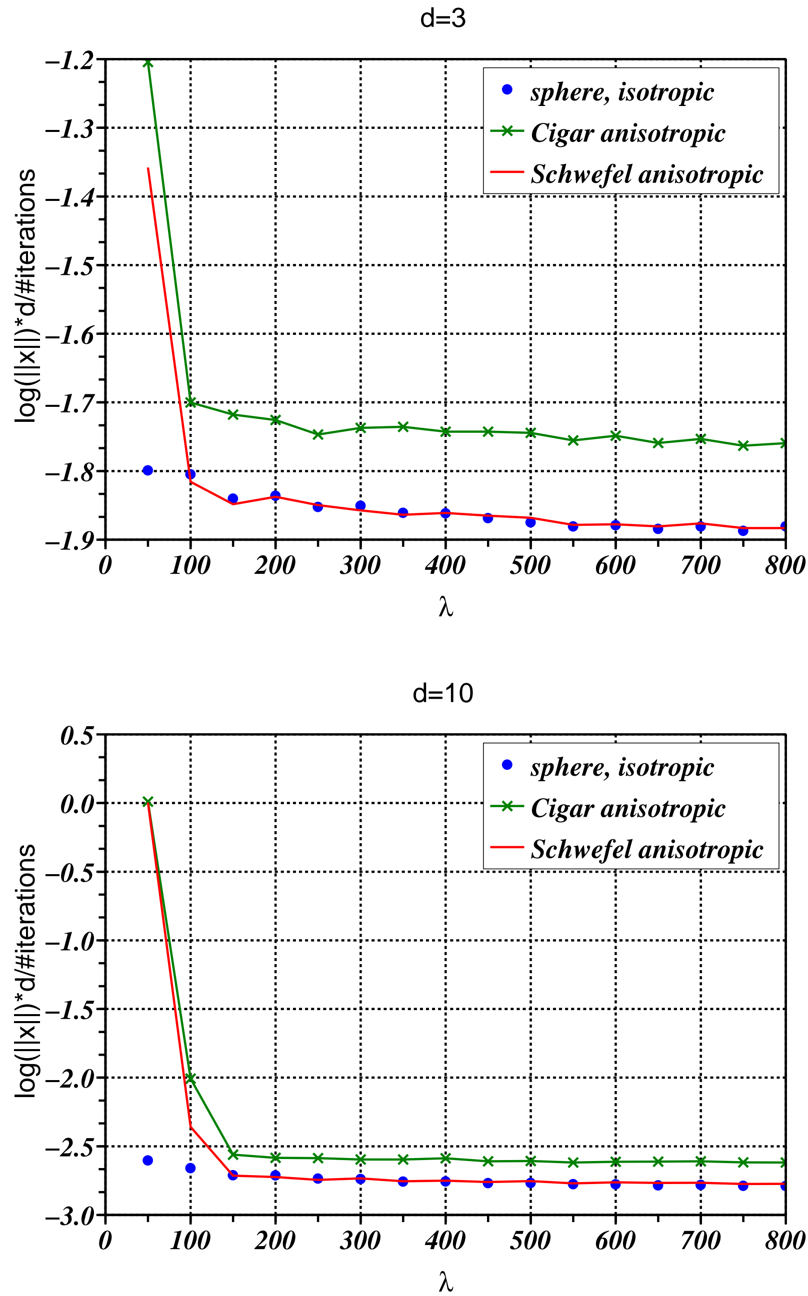

Figure 2.11: Results of the anisotropic version of the SSA algorithm in dimensions 3 (Top) and 10 (Bottom).

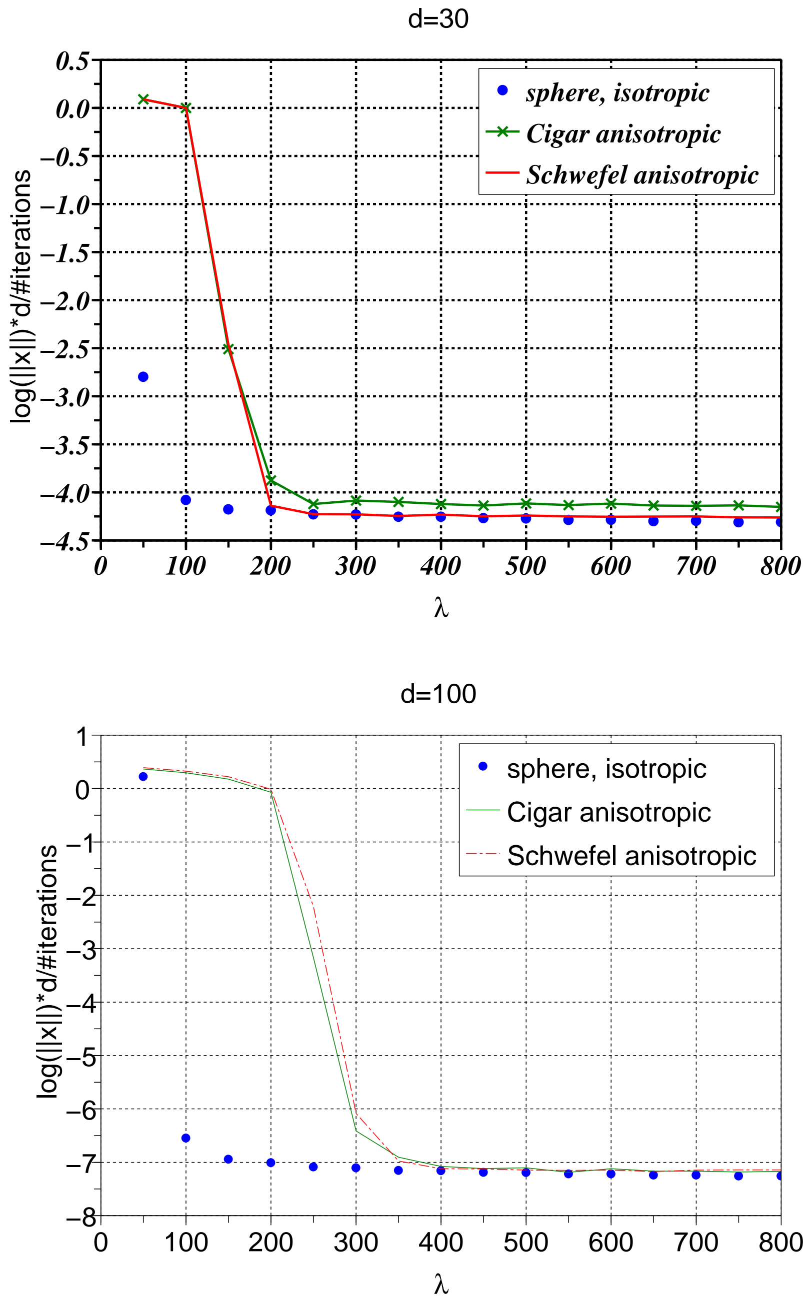

Figure 2.12: Results of the anisotropic version of the SSA algorithm in dimensions 30 (Top) and 100 (Bottom). It has, on ill-conditioned functions, nearly the same performance as the isotropic version on the Sphere function, in particular in high dimension.

2.2 Log(λ) modifications for optimal parallelization

As seen in the Section 2.1, there is a discrepancy between theory and practice. In this section we propose some modifications in order to improve the existing algorithm in case of large population size to solve this discrepancy. The first modification, in Section 2.2.1, is based on the adjustment of the selection ratio. The second modification, presented in Section 2.2.2 is a faster decrease of the step-size. We see, that in order to have a maximal benefit of this last modification, it is necessary to have a specific reweighting.

2.2.1 New selection ratio

The results of this section have been published in [Teytaud, 2010].

In this work, we compare a modified version of the SA algorithm, with the standard version (as defined in Algorithm 2. The modification of the SA algorithm consists in an adjustment of the selection ratio. In this new version the number of selected points μ is equal to the minimum between the dimensionality and the population size divided by 4. Formally, in that case, $\mu = \min(d, \frac{\lambda}{4})$. Intuitively, we want $\mu = d$ for λ large, and $\lambda/4$ for small value of the population size.

Before seeing the comparison between the standard version of the SA algorithm and the new one, we study the discrepancy between theory and practice, and in particular the importance of the choice of the selection ratio. The objective functions used for our experiments are the Sphere function, the Schwefel function and the Rosenbrock function. These functions are described in Table 2.1.

In section 2.1 and originally in [Fournier and Teytaud, 2011], several theoretical bounds have been proved. In particular, the speed-up should be $\Theta(\log(\lambda))$ with a $(\mu/\mu, \lambda)$-ES. In practice (*i.e.* with usual parametrizations of the algorithm), the SA-ES does not reach that speed-up, as demonstrated in Figure 2.13.

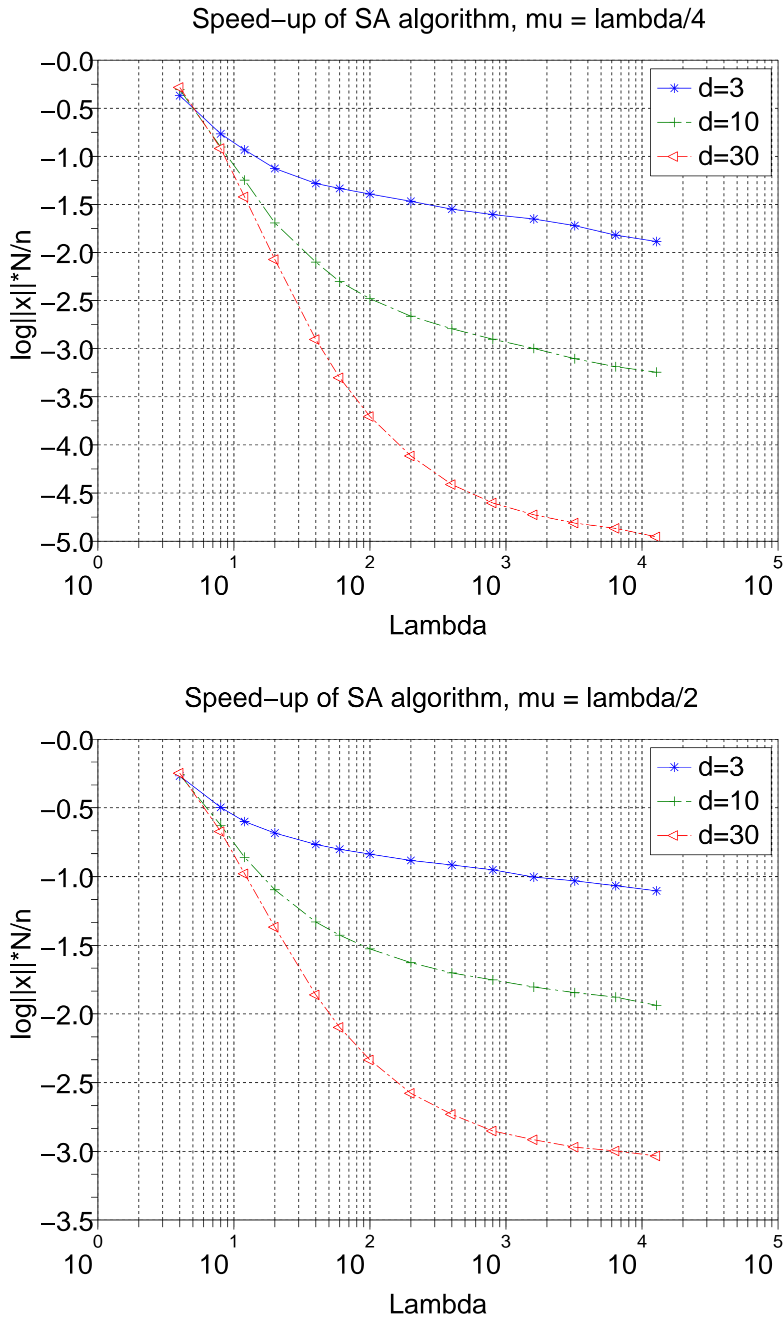

Figure 2.13: Speed-up of the Self-Adaptation algorithm as a function of λ, on the Sphere function. For this experiment, the initial step size is 1, the initial point is $(1, \ldots, 1)$ and the fitness function to hit is 10^{-10}. Top ($\mu = \frac{\lambda}{4}$) and bottom ($\mu = \frac{\lambda}{2}$) do not show a logarithmic speedup whatever the dimensionality.

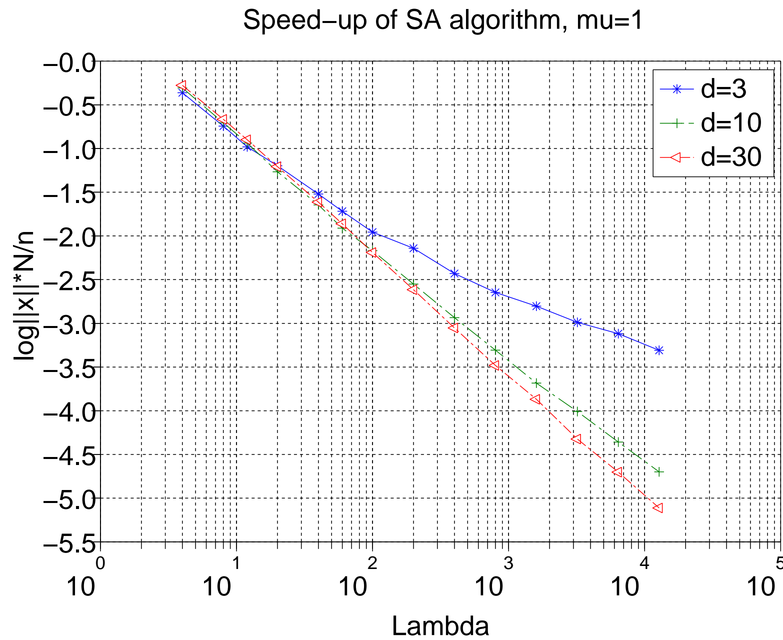

Figure 2.14: Speed-up of the Self-Adaptation algorithm as a function of λ, on the Sphere function for $\mu = 1$. The initial step size is 1, the initial point is $(1, \ldots, 1)$ and the fitness function to hit is 10^{-10}.

In Figures 2.13 and 2.14, we can note that the selection ratio μ is a crucial parameter for characterizing the behaviour of the SA algorithm. Unless $\mu = 1$, the speed-up reaches a plateau when λ is very large (Figure 2.13).
$\mu = \frac{\lambda}{2}$ is never a good choice whatever the population size, and if the population size becomes really large, the speedup tends to a constant. Figures 2.13 and 2.14 shows that in dimension 10, $\mu = \frac{\lambda}{4}$ is better than $\mu = 1$ up to a population size of 200. In dimension 30, $\mu = \frac{\lambda}{4}$ is also a good choice, in particular if the population size is smaller than 6400. The main problem is that the convergence rate tends to a constant (depending on the dimensionality) for large population sizes; theoretical results suggest it should be possible to reach better speed-up. As previously said, in [Teytaud and Fournier, 2008], it is shown that for $(\mu/\mu, \lambda)$-ES algorithms the convergence rate should be linear as a function of λ if the population size is smaller than the dimension, for a correctly tuned algorithm. If the population is larger than the dimension the speed-up should be $\Theta(\log(\lambda))$, here again in the case of an algorithm correctly tuned.
$(1, \lambda)$-ES only shows a logarithmic convergence rate as a function of λ, which suggests that this is the best choice when the population size is large.

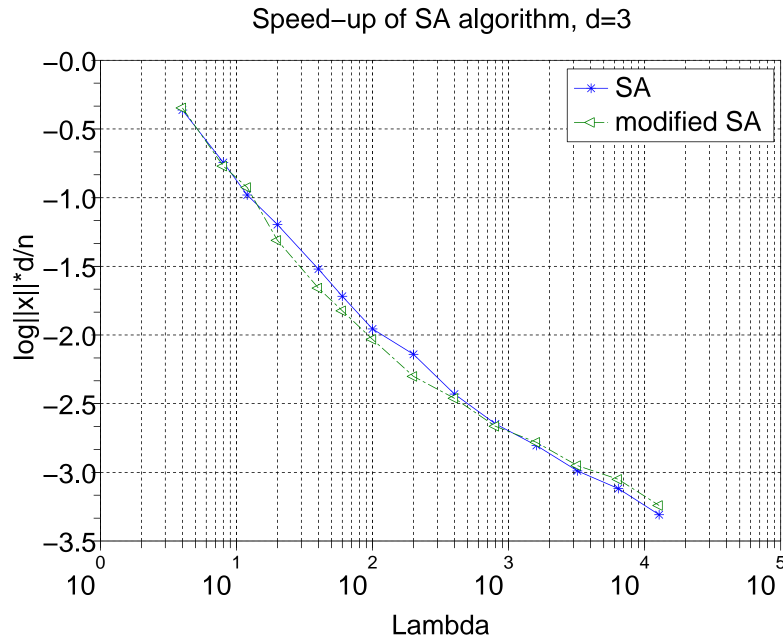

Figure 2.15: This figure shows the convergence rate on the Sphere function for the Self-Adaptation algorithm and for the modified version of this algorithm. In the modified self adaptation algorithm, the selection ratio is chosen equals to $\mu = \min(d, \lambda/4)$. For the standard self adaptation algorithm, we have chosen the best selection ratio for large population size ($\mu = 1$). Results are obtained for dimension 3, and are really close, simply because in that case, the selection ratio of both versions of the algorithm is almost the same (due to the small dimensionality).

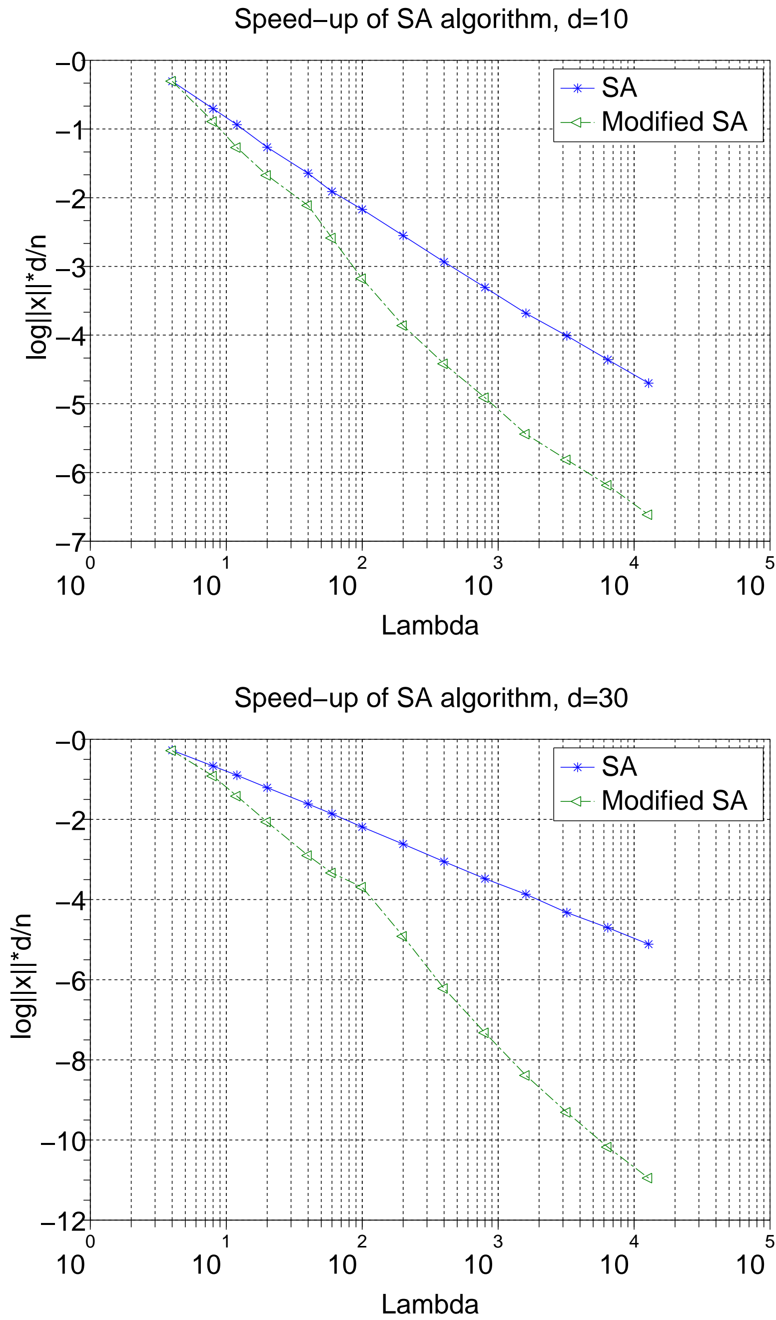

Figure 2.16: Same results as for Figure 2.15 but for dimensions 10 and 30. The modified self-adaptation algorithm outperforms the standard version.

We now compare two SA algorithms, the standard one (defined in Algorithm 2), and a new one, with a new selection ratio : $\mu = \min(d, \frac{\lambda}{4})$.

In Figures 2.15 and 2.16 we show that we can improve the convergence rate by using a simple rule for the selection ratio. Experiments have been done on the Sphere function. The initial point is $(1, \ldots, 1)$ and the initial step size is 1. The target fitness function is $f_{stop} = 10^{-10}$. In this figure, we compare the Self-Adaptation algorithm with $\mu = 1$ and a modified version of this algorithm. We have chosen $\mu = 1$ for the comparison because it is the best choice for the SA algorithm for λ large, as shown in previous section.

In dimension 3, both convergence rates are very close, because the value of the selection ratio of the Self-Adaptation algorithm and of the modified Self-Adaptation algorithm are close to each other.

In dimension 10, the modified version of the Self-Adaptation algorithm outperforms the standard version. We also have a logarithmic convergence rate as predicted by the theory. For a population size of 12800 we have a speedup of 41% over the Self-Adaptation algorithm with $\mu = 1$.

In dimension 30 finally, we have a very large improvement. We have a logarithmic convergence rate, as a function of the population size, and the speed-up reaches of 114% for a population size of 12800. In this case, for small population size, the convergence rate is also slightly better than $\mu = 1$.

We now experiment with the Covariance Self-Adaptation Evolution Strategy, first introduced in [Beyer and Sendhoff, 2008], and considered as state of the art algorithm for large population sizes. This algorithm has been presented in Algorithm 5.

Following the recommendations in [Beyer and Sendhoff, 2008] for the tuning of the CMSA-ES algorithm, a selection ratio μ/λ equals to $1/4$ should be used. In this section, we experiment the CMSA evolution strategy with two different selection ratios: the recommended $\mu = \lambda/4$, and our modified version $\mu = \min(d, \lambda/4)$ as above.

In our experiments, we have considered different dimensionalities, $d = 3, 10, 30$. We have compared different selection ratios for different population sizes, specially large population sizes. Each point the y-axis is the average of 60 independent runs. The plots represent the convergence rate. We plot $\frac{d \times log||y||}{number_of_Iterations}$ as a function of the population size. Choosing to measure the convergence rate (instead of the number of iterations as in [Beyer and Sendhoff, 2008]) is here justified by the fact that we know the theoretical limits of this criterion (up to a constant factor), and we know that

many usual algorithms have only a bounded speed-up; we want to emphasize this.

Figures 2.17 and 2.18 shows that using a bad selection ratio can be harmful, especially for large population size. Even for not so large population size (e.g. $\lambda = 20$) using the selection ratio $\mu = \min(d, \lambda/4)$ is a good choice. The best speed-ups are reached for large population sizes, more precisely we obtains a speed-up of 136% for the Sphere function and 139% for the Schwefel function for a population size equals to 10000 (more than twice faster in both cases), and 146% for the Rosenbrock function for a population size of 5000.

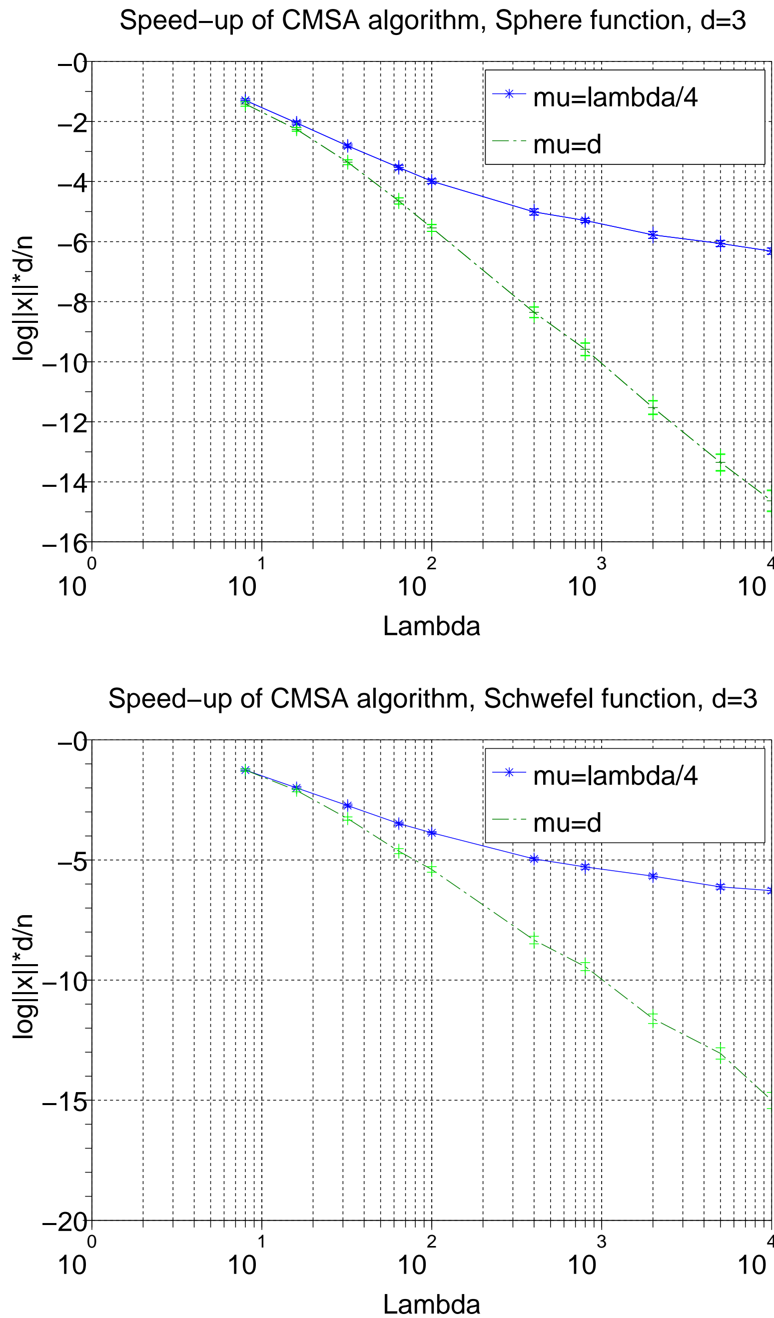

Figure 2.17: Results of the CMSA evolution strategy in dimension 3 for the Sphere function (Top) and the Schwefel function (Bottom). Convergence rate for the standard selection ratio ($\mu = \lambda/4$) and the new selection ratio ($\mu = min(d, \lambda/4)$) are plotted. Choosing a selection ratio equals to $\mu = min(d, \lambda/4)$ is good choice on both functions. We reach a logarithmic speedup.

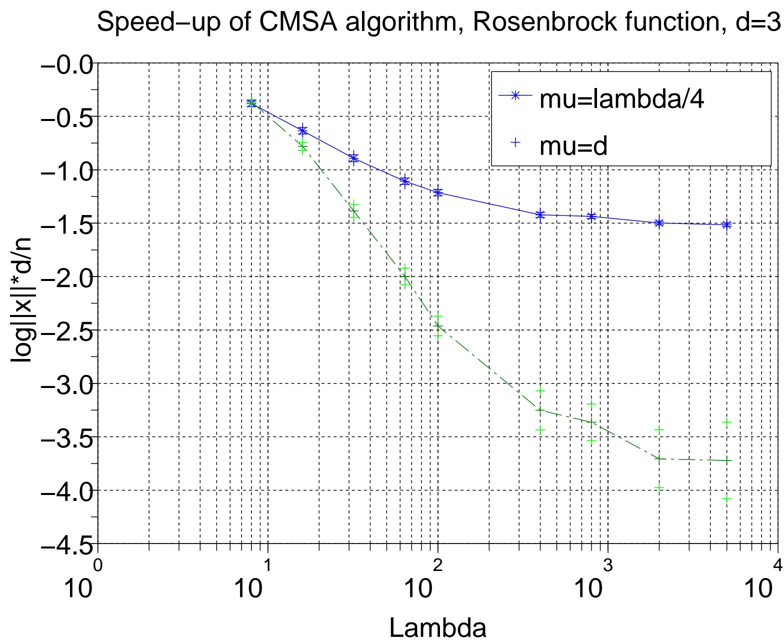

Figure 2.18: Results of the CMSA evolution strategy in dimension 3 for the Rosenbrock function. Convergence rate for the standard selection ratio ($\mu = \lambda/4$) and the new selection ratio ($\mu = min(d, \lambda/4)$) are plotted. Choosing a selection ratio equals to $\mu = min(d, \lambda/4)$ improves the algorithm.

In Figure 2.19 we experiments the Sphere function and the Schwefel function in dimension 10. The largest population size for this experiment is 800. Speedup of 37% is reached for the Sphere function and 41% for the Schwefel function for a population size of 400.

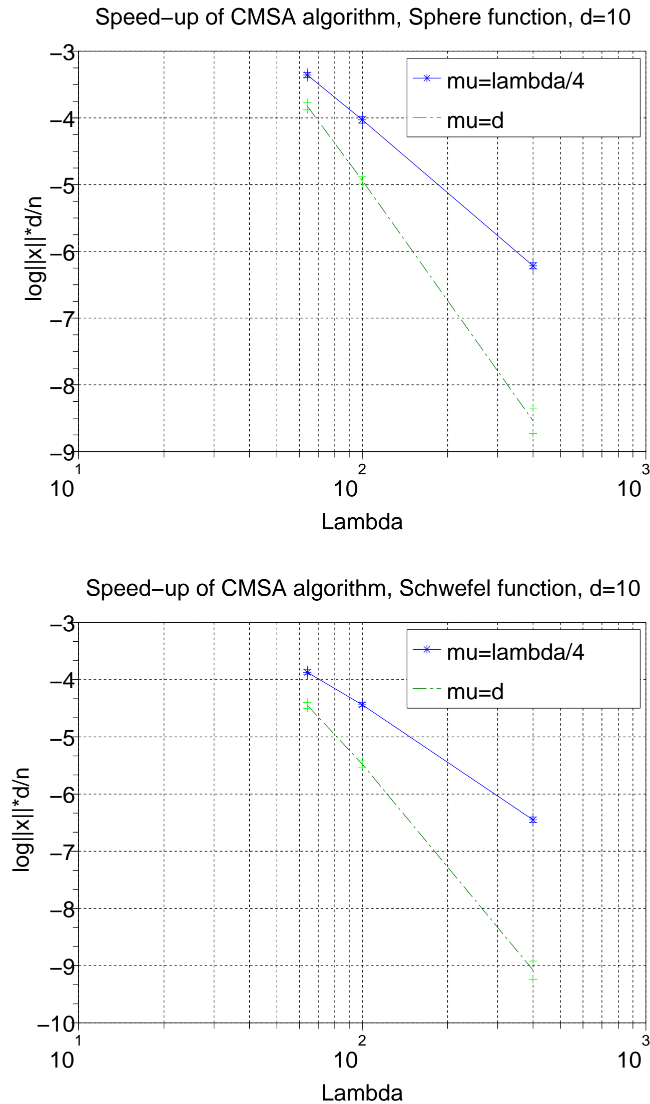

Figure 2.19: Results of the CMSA evolution strategy in dimension 10 for the Sphere function (Top) and the Schwefel function (Bottom). Convergence rate for the standard selection ratio ($\mu = \lambda/4$) and the new selection ratio ($\mu = min(d, \lambda/4)$) are plotted.

In Figure 2.20, we reproduce the experiments as previously, but in bigger dimensionality, 30. Results are similar, and we reach a speedup of 44% for the Sphere function and 34% for the Schwefel function for $\lambda = 800$.

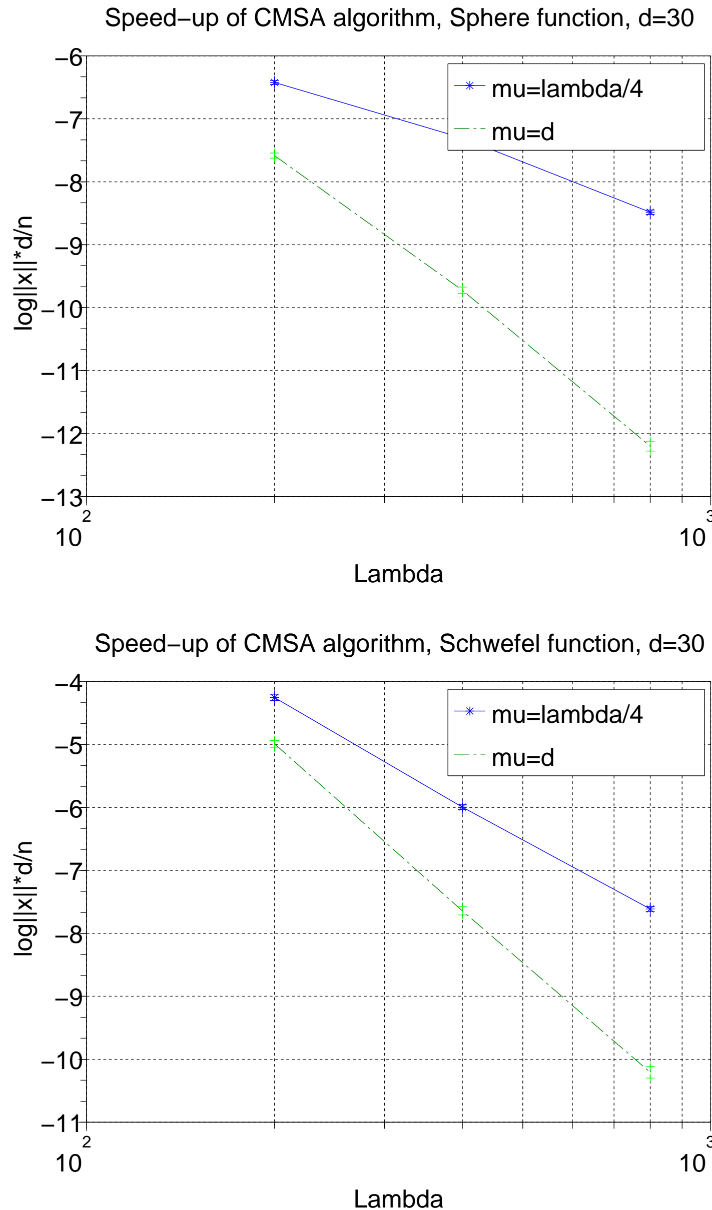

Figure 2.20: Results of the CMSA evolution strategy in dimension 3 for the Sphere function (Top) and the Schwefel function (Bottom). Convergence rate for the standard selection ratio ($\mu = \lambda/4$) and the new selection ratio ($\mu = \min(d, \lambda/4)$) are plotted.

To summarize, in this section we proposed a new rule for the selected population size $\mu = \min(d, \lambda/4)$. This rule is a simple modification, and has very good results for the CMSA evolution strategy. We can seemingly reach a logarithmic speed-up with this rule, which is consistent with the theoretical bounds.

We have first shown that the current version of the Self Adaptation rule only reaches the theoretical speedup (logarithmic as a function of λ) when the selection ratio is equal to $\frac{\mu}{\lambda} = \frac{1}{\lambda}$, *i.e.* $\mu = 1$.

A selection ratio of $\frac{1}{2}$ or $\frac{1}{4}$ is better at first view, but it's indeed harmful when the population size λ is large enough (larger than the dimension).

Then, we have shown that having $\mu = \min(d, \lambda/4)$ could lead to a very good speedup, better than both $\mu = 1$ and $\mu = \lambda/4$ or $\mu = \lambda/2$. Finally, we experiment the same rule on the CMSA algorithm (section 2.4), which is known to be a good evolution strategy when the population size is large.

Based of this idea of working on the selected population size, in [Jebalia and Auger, 2011], M. Jebalia and A. Auger have computed, with Monte-Carlo simulations the optimal values for μ on the Sphere function. They suggest that μ should monotonously increase in λ. They also confirm that, with their optimal choice for μ, the logarithmic speed-up behaviour of the ES. Indeed, experimentally their results are really close to our results, i.e. with the rule $\mu = \min(d, \lambda/4)$.

2.2.2 Faster decrease of the step-size with reweighting

The results of this section have been published in [Teytaud and Teytaud, 2010b].

In this section, we propose a modification, based of the idea of having a faster decrease of the step-size. We call such a modification, a $\log(\lambda)$-modification. We study the CMA-ES algorithm. This algorithm is, for many people, the state of the art algorithm for evolutionary strategies for sequential setting. We propose to add this line in CMA-ES :

$$\sigma = \frac{\sigma}{\max(1, (\zeta\lambda)^{1/d})}.$$

We have seen that the CSA algorithm does not reach the optimal speed-up. This bad behavior has been pointed out in Section 2.1.2. This formula avoids this bad behavior, and experimentally, improves the results. The fitness function considered here is the Sphere function (defined in table 2.1). We consider f as the best fitness found by the algorithm after a fixed number of evaluations. We report the mean of $\frac{d \cdot \log(f)}{\#evaluations}$ and the mean of $\log(f)$ in Table 2.2. The number of function evaluations is $100d^2$. Following [Beyer and Sendhoff, 2008], we experiment two sizes of population, $\lambda = 8d$ and $\lambda = 8d^2$. If the dimension is small (2) we almost have a speed-up of 2 independently of the size of the population. However, if the dimension becomes larger (10 or 30) we have a good speed-up only if the size of the population is large ($\lambda = 8d^2$). The results are positive, but not very good, and CMA with this correction is still far from the efficiency of CMSA or EMNA for large population size; we guess however that improvements of our formula above are possible, and also we guess that modifying the rule for computing the new parent should be adapted for λ large. The CMA-ES has a lot of parameters, as seen in Algorithm 4. All these parameters have been studied and tuned for sequential setting and therefore with a small population size (the default population size in CMA is $\lambda = 4 + \lfloor 3 + \log(d) \rfloor$). When one parameter is modified, the resulting behavior of the algorithm is hard to determine, because, the modified parameter can have influence on others. In section 2.1.2, the superiority of EMNA for large population sizes has been pointed out, then, for these reasons we investigate more deeply the $\log(\lambda)$-modification on the EMNA algorithm (Section 2.2.3).

λ	CMA	CMA with the log(λ)-correction
Dimension 2		
$8 \times d$	-0.100±0.001	**-0.177±0.001**
$8 \times d^2$	-0.0741±0.0009	**-0.134±0.001**
Dimension 10		
$8 \times d$	-0.0338±6e-05	**-0.0389±0.0001**
$8 \times d^2$	-0.00971±6e-05	**-0.0174±0.0001**
Dimension 30		
$8 \times d$	-0.0107±1e-05	**-0.0118±2e-05**
$8 \times d^2$	-0.00188±1e-05	**-0.00370±1.e-05**

Table 2.2: Comparison between CMA and CMA with log(λ)-correction in various dimensions. The maximum number of function evaluations is 400 (in dimension 2), 10 000 (in dimension 10) and 90 000 (in dimension 30), and the constant ζ involved in the log(λ)-correction (Eq. 2.2.2) is $0.4^{1/2}$ in dimension 2, 1 in dimension 10, $1.3^{1/30}$ in dimension 30.

2.2.3 Analysis of Estimation of Multivariate Normal Algorithms in case of large population size

The results of this work have been published in [Teytaud and Teytaud, 2009c] and [Teytaud and Teytaud, 2009a].

EMNA (Estimation of Multivariate Normal Algorithm), a widely known EDA (Estimation of Distribution Algorithms) is very efficient for parallel optimization. This result has been presented in Section 2.1, and in particular, it was shown that, for large population size λ, it outperforms SA-ES, which itself outperforms CSA-ES.

However, a detailed look at plots in Section 2.1.3, and more precisely at Figures 2.5, 2.6, 2.7 and 2.8, shows that the speed-up, as a function of λ, seemingly stagnates in EMNA for λ very large. This is not consistent with theoretical bounds in [Teytaud and Fournier, 2008]: the speed-up, for a well designed algorithm, should increase to infinity, linearly for small values of λ and then logarithmically, but never stop at a constant. By the way, the speed-up of SA-ES is seemingly still increasing when the speed-up of EMNA stagnates; this suggests that for λ larger than in previous works, the robust SA-ES might indeed be faster than EMNA.

We here first analyze the reasons why the speed-up of EMNA stagnates for λ large. We see that this is due to a bias/variance dilemma. The bias/variance dilemma is as follows. When x is an estimate of x^*, the bias/variance decomposition states that:

$$\mathbb{E}(x-x^*)^2 = \mathbb{E}\left((x-\mathbb{E}x)+(\mathbb{E}x-x^*)\right)^2 = \underbrace{\mathbb{E}(x-\mathbb{E}x)^2}_{variance} + \mathbb{E}\underbrace{(\mathbb{E}x-x^*)^2}_{squared\ bias}$$

We now see that, when using a specific reweighting for the EMNA algorithm, the bias disappears (the right hand side term is zero in Equation 2.2.3). Interestingly, when the bias is removed, then the following traditional statistical tricks for reducing variance become much more efficient (as bias is removed, reducing the variance decreases the error to zero!). The statistical tricks for reducing variance studied in this work are :

- use of quasi-random numbers, as in [Teytaud and Fournier, 2008];
- use of λ large, *i.e.* parallel case.

In this section, we define a version of EMNA including:

- reweighting as in [Teytaud and Teytaud, 2009c]
- quasi-random mutations as in [Teytaud, 2008b]

- a modification of the step-size adaptation rule based on [Teytaud and Teytaud, 2010b].

Now, we discuss precisely the three weaknesses of EMNA (or, more generally, of evolutionary algorithms based on random sampling and averages) which are tackled in this work: the weighting issue, the limited behavior for λ large, and the possible redundancies of the random mutations.

The weighting issue.

The American election of 1936 is famous for the error, in a very important moment of American history [Andersen, 1979], in the poll organized by the Literary Digest [lit, 1936], in spite of a huge sampling. More precisely, 10 million questionnaires have been mailed to readers and potential readers and over two million were returned. The Literary Digest, which had correctly predicted the winner of the last 5 elections, announced in its October 31 issue that Landon would be the winner with 370 electoral votes. The cause of this mistake is believed to be due to improper sampling: more Republicans subscribed to the Literary Digest than Democrats, and were thus more likely to vote for Landon than Roosevelt. Whereas Roosevelt finally got 61 % of votes, the Literary Digest predicted a comfortable win for Landon. This mistake by the Literary Digest proved to be devastating to the magazine's credibility, and in fact the magazine went out of existence within a few months of the election. That same year, George Gallup, an advertising executive who had begun a scientific poll, predicted that Roosevelt would win the election, based on a quota sample of 50,000 people. He also predicted that the Literary Digest would mis-predict the results. His correct predictions made public opinion polling a critical element of elections for journalists and indeed for politicians. The Gallup Poll would become a staple of future presidential elections, and remains one of the most prominent election polling organizations to this day.

Estimation of Distribution Algorithms are similar to polls: they are based on samplings. However, the reweighting has not been experimented or analyzed in this context, this section is based on this idea.

We here focus on EMNA, presented in Algorithm 6. This is not the only EDA, but it is an efficient one. Nevertheless, the reweighting technique emphasized here can be used for all algorithms which involve averaging or parameter estimation on a sample.

There are several advantages in EDA; simplicity, nearly parameter free, and better than SA and CSA evolution strategies for large *lambda*. However, an issue is premature convergence [Shapiro, 2005, Grahl et al., 2006, Liu and Teng, 2008, Posik, 2008] : for example, with a Gaussian EDA, if the initial point is too far from the target (formally, if the squared distance to the

optimum divided by the initial variance is too large), then the EDA might have premature convergence.

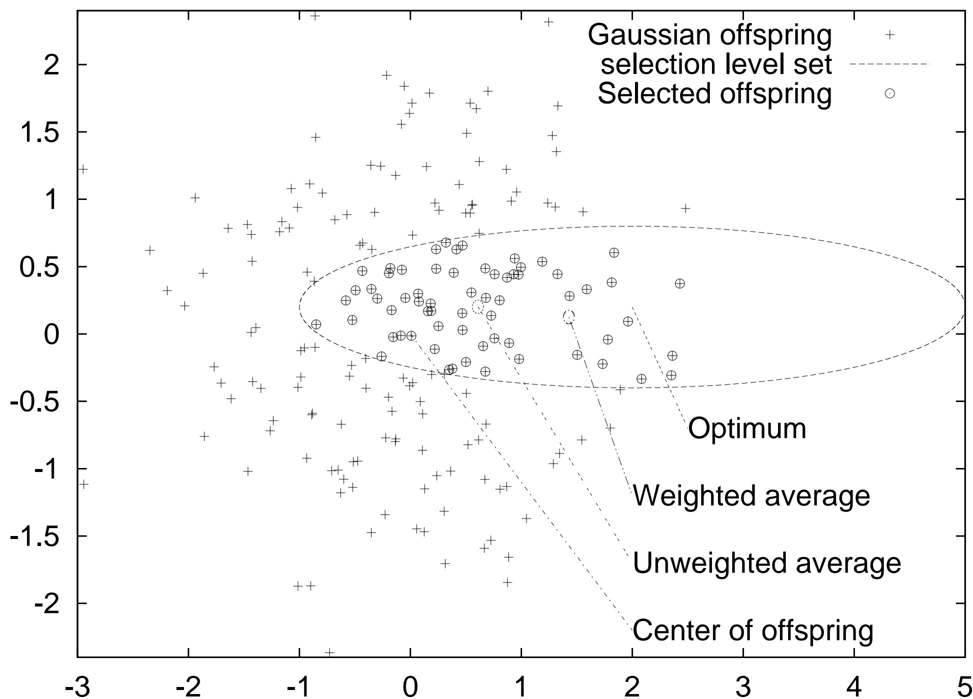

Figure 2.21: Illustration of the requirement of weighting. A Gaussian offspring (200 samples) is generated. The best individuals are selected. Their unweighted average is biased towards the center of the distribution. The weighted average, following [Teytaud and Teytaud, 2009c], is much better.

The Gaussian distribution induces a bias towards the center of the offspring, regardless of the selection. The reweighting idea proposed in [Teytaud and Teytaud, 2009c] is to reweight offsprings proportionally to the inverse of the Gaussian density, in order to correct this bias. The idea of this reweighting is illustrated in Figure 2.21. We study the weighting trouble because it is important for our bias/variance decomposition. We use this reweighting modification in the sequel, as it is necessary for removing the bias: thereafter, we can work efficiently on the variance.

Some important papers around reweighting are [Arnold, 2006, Arnold and Van Wart, 2008]; however, their goal is different: they choose weights for improving the convergence rates of ES for fixed values of λ, and in order to take into account the position of unselected points. Maybe their

reweighting can be combined to the reweighting by inverse density proposed here.

Limited behavior for large population sizes.

Another issue in EMNA, whenever we use reweighting, is that the convergence rate converges to a fixed constant when λ goes to infinity. This is because, as a function of λ, the step-size does not decrease to 0. More formally, with e.g. $\mu = \lambda/4$, and for a given parent x_n and step-size σ_n,

$$\lim_{\lambda\to\infty} \sigma_{n+1} = std(\text{first quartile of the offspring generated at iteration } n) \tag{2.16}$$

where std is the standard deviation of a set of points. This is definitely not reasonable: when λ runs to infinity, the optimum is estimated more and more precisely (at least now, as we have removed the bias by reweighting), and therefore σ should get smaller and smaller so that the search is more focused. Therefore, we might want to ensure that

$$\lim_{\lambda\to\infty} \sigma_{n+1} = 0. \tag{2.17}$$

With Eq. 2.17 instead of Eq. 2.16, we focus the search on the most important part. A question is then: to which extent should we force σ_{n+1}/σ_n to decrease to 0 ? We empirically choose the following formula:

$$\sigma \leftarrow \sigma / \max(1, (\log(\lambda)/2)^{1/d}) \tag{2.18}$$

where d is the dimension. This formula is probably too conservative and should be tuned carefully, but this is sufficient as a proof of concept as illustrated in later results. This will be done just after the classical EMNA estimate of the step-size; see Algorithm 6.

We here investigate the case of λ large. This is the case in which the problem consists in focusing the search more strongly than with standard values of λ, whilst avoiding premature convergence. In [Dong and Yao, 2008], the interested reader can find the reverse point of view: how to evaluate the Gaussian distribution (in particular when it includes a full covariance matrix and not only the diagonal case) properly in spite of the finiteness of λ.

Possibly redundant mutations. Random points can lead to a lot of redundancies among mutations. Therefore, many improved ways of generating more uniform points have been developed in the literature [Niederreiter, 1992, L'Ecuyer and Lemieux, 2005, Vandewoestyne and Cools, 2006]. It was shown in [Teytaud, 2008b] that in many cases using quasi-random sequences instead of random sequences provides big improvements, and that there is almost no case in which it is harmful. We do not present quasi-random points instead of random points here.

A detailed presentation of using quasi-random sequences in an evolutionary context can be found in [Teytaud and Gelly, 2007].

Please note that quasi-random sequences are not intended to change the distribution. We consider Gaussian distributions only. But as well as one can use random Gaussian numbers, one can use quasi-random Gaussian numbers - the distribution is the same, we just take care of avoiding unlucky (redundant) cases when we use quasi-random sequences.

Improving EMNA.

We want to combine three improvements. The first one is the use of reweighting [Teytaud and Teytaud, 2009c]. This is a very simple and efficient modification against premature convergence. It is implemented as follows: when computing the new parent and/or the new covariance matrix and/or the new stepsize, give a weight to each selected individual: this weight is inversely proportional to the density of the Gaussian distribution used for sampling individuals (see Algorithm 7 below) and Section 2.2.3 for more details. The second one is the use of quasi-random numbers. We refer to [Niederreiter, 1992, L'Ecuyer and Lemieux, 2005, Teytaud, 2008b] for more information around quasi-random numbers. It is just important to note that there exists quasi-random numbers as well as pseudo-random numbers, and they have some nice uniformity properties, and they provide improvements in continuous evolutionary algorithms. The third one is as simple as the two previous ones: after having estimated the step-size as in the classical EMNA, just divide it by $\max(1, (\log(\lambda)/2)^{1/d})$. The justification of this modification is discussed in 2.2.2. Our version including these 3 improvements is presented in Algorithm 7.

Algorithm 7 Our improved version of EMNA (IEMNA), with quasi-random mutations, reweighting, faster decrease of step-size.

Initialize $\sigma \in \mathbb{R}$, $y \in \mathbb{R}^d$.
while Halting criterion not fulfilled **do**
  for $l = 1..\lambda$ **do**
    $p_l = \sigma \times QRN_l(0, Id)$ // quasi-random Gaussian vector [Teytaud, 2008b]
    $w_l = 1/d(p_l)$ where d is the Gaussian density [Teytaud and Teytaud, 2009c]
    $z_l = \sigma p_l$
    $y_l = y + z_l$
    $f_l = f(y_l)$
  Sort the individuals by increasing fitness; $f_{(1)} < f_{(2)} < \cdots < f_{(\lambda)}$.
  $s = \sum_{i=1}^{\mu} w_{(i)}$
  Renormalize for all $i \leq \mu$, $w_{(i)} \leftarrow w_{(i)}/s$
  $$z^{avg} = \sum_{i=1}^{\mu} w_{(i)} z_{(i)}$$
  $$\sigma = \sqrt{\frac{\sum_{i=1}^{\mu} w_{(i)} ||z_{(i)} - z^{avg}||^2}{d}}$$
  $\sigma = \sigma / \max((\log(\lambda)/2)^{1/d}, 1)$
  $y = y + z^{avg}$

Experimental results.

All experiments below are based on EMNA (Algorithm 6) and IEMNA (Algorithm 7), and intermediate versions with only part of the improvements proposed in IEMNA. EMNA is the baseline from [Larranaga and Lozano, 2002]. EMNA+QR is EMNA, plus the quasi-random mutations as in [Teytaud and Gelly, 2007, Teytaud, 2008b]. EMNA+QR+reweighting is EMNA+QR, plus the reweighting as in [Teytaud and Teytaud, 2009c] (see the reweighting formula in Algorithm 7). IEMNA is EMNA+QR+reweighting+LB; this is the complete improved version in Algorithm 7. We perform our experiments with anisotropic Gaussians, with diagonal covariance matrix (i.e. one step-size per axis).

Case with good initialization.

We experiment IEMNA (Algorithm 7) with initial step-size $\sigma = 1$ on each axis, and $x_0 = (1, 1, \dots, 1)$. The number of generations is 50. The negative convergence rate, estimated by $d \log(||x_{50}||/||x_0||)/50$ (with d the dimension), is presented in Table 2.3 for the Sphere function $x \mapsto ||x||$, Table 2.4 for $x \mapsto \sum_i \log(|x_i|) + cos(1/x_i)$, Table 2.5 for the cigar function $x \mapsto \sum_i (10^4)^i x_i^2$. "QR" denotes the use of quasi-random mutations, "weight" the use of reweighting, "LB" the use of step-size decreasing as in section 2.2.2 ($\sigma \leftarrow \sigma / \max((\log(\lambda)/2)^{1/d}, 1)$). The results clearly show (i) the success of QR (always better than the baseline), (ii) the success of LB for λ large in the reweighted case, (iii) the poor performance of reweighting, which moderately but constantly decreases the performance. Result (iii) can be explained by the fact that we are in a case in which premature convergence is unlikely - we reproduce the experiments with a poor initialization of $\sigma = 0.01$ in the following section in order to clarify this idea.

Table 2.3: Experiments on the Sphere function. Numbers are the negative convergence rates (the lower the better). Each modification is validated or invalidated separately (p-values are for EMNA+QR versus the initial EMNA (Algorithm 6); then EMNA+QR+weighting against EMNA+QR; and finally the complete new version EMNA+QR+LB+reweighting as in Algorithm 7) against EMNA+QR+weighting. We see that IEMNA provides by far the best results.

Dimension, lambda	Baseline	+QR	+ weight	+LB (IEMNA)	P-value for QR	P-value for weight	P-value for LB
2,20	-0.345	-1.252	-1.743	**-2.103**	0	3.894e-07	0
2,60	-1.967	-2.086	-2.022	**-2.713**	0.001	1	0
2,200	-2.050	-2.089	-2.112	**-3.004**	0	3.308e-14	0
2,600	-2.080	-2.101	-2.061	**-3.190**	4.447e-08	1	0
2,2000	-2.103	-2.188	-2.111	**-3.434**	0	1	0
3,30	-0.697	-2.299	-2.277	**-2.398**	0	0.636	0.03
3,90	-2.330	-2.392	-2.282	**-3.047**	1.588e-13	1	0
3,300	-2.340	-2.404	-2.293	**-3.302**	0	1	0
3,900	-2.369	-2.443	-2.320	**-3.519**	0	1	0
3,3000	-2.404	-2.476	-2.367	**-3.726**	0	1	0
4,40	-0.749	-2.541	-2.397	**-2.578**	0	0.998	0.03
4,120	-2.543	-2.627	-2.443	**-3.271**	0	1	0
4,400	-2.601	-2.658	-2.480	**-3.547**	0	1	0
4,1200	-2.642	-2.717	-2.519	**-3.764**	0	1	0
5,50	-1.330	-2.885	-2.677	**-2.730**	0	1	0.31
5,150	-2.790	-2.858	-2.622	**-3.488**	0	1	0
5,500	-2.828	-2.908	-2.673	**-3.750**	0	1	0
5,1500	-2.886	-2.964	-2.718	**-3.975**	0	1	0

Table 2.4: Experiments on the multimodal function (see text). The lower, the better (as previous figures and following the definition of negative convergence rate in the text). QR is the only stable and efficient modification here.

Dimension, lambda	Baseline	+QR	+ weight	+LB (IEMNA)	P-value for QR	P-value for weight	P-value for LB
2,20	-0.709	**-1.301**	-1.105	-0.529	0	1	1
2,60	-1.139	**-1.181**	-0.655	-1.157	2.315e-07	1	0
2,200	**-1.104**	-1.074	-0.402	-0.822	1	1	0
2,600	-1.100	**-1.133**	-0.119	-0.534	0	1	0
2,2000	-1.124	**-1.146**	-0.124	-0.210	3.187e-09	1	0
2,6000	-1.144	**-1.162**	-0.173	-0.181	7.199e-11	1	0
3,30	-0.971	**-1.332**	-0.799	-0.537	1.721e-09	1	1.00
3,90	**-1.231**	-1.229	-0.481	-0.619	0.559	1	0.00
3,300	-1.210	**-1.243**	-0.178	-0.233	1.415e-05	1	0.00
3,900	-1.240	**-1.252**	-0.181	-0.179	0.031	1	0.61
3,3000	-1.269	**-1.307**	0.081	-0.031	6.342e-12	1	0.01
4,40	-1.204	**-1.388**	-0.713	-0.858	6.262e-06	1	0.01
4,120	**-1.357**	-1.353	-0.344	-0.480	0.624	1	0.00
4,400	-1.352	**-1.368**	-0.205	-0.183	0.013	1	0.98
4,1200	-1.389	**-1.427**	0.224	0.093	7.638e-10	1	0.01
5,50	-1.359	**-1.520**	-0.702	-0.445	0.000	1	1.00
5,150	-1.477	**-1.503**	-0.351	-0.391	0.010	1	0.05
5,500	-1.495	**-1.518**	-0.145	-0.161	0.001	1	0.34
5,1500	-1.539	**-1.579**	0.726	0.715	2.300e-08	1	0.37

We also show graphically the negative convergence rate (defined as in previous tables) in Fig. 2.22 in the case of the Sphere function.

Table 2.5: Experiments on the cigar function (see text). The number are negative convergence rates as defined in the text; the lower, the better. IEMNA is clearly the best algorithm here.

Dimension, lambda	Baseline	+QR	+ weight	+LB (IEMNA)	P-value for QR	P-value for weight	P-value for LB
2,20	-0.222	-1.833	**-1.885**	-0.758	0	0.024	1
2,60	-1.774	-1.954	-1.950	**-2.734**	0.002	0.796	0
2,200	-1.919	-2.003	-1.967	**-2.834**	0.016	1	0
2,600	-2.014	-2.060	-1.994	**-3.075**	1.064e-12	1	0
2,2000	-2.015	-2.059	-2.023	**-3.371**	0	1	0
2,6000	-2.047	-2.122	-2.019	**-3.476**	0	1	0
3,30	-0.209	-1.397	**-1.643**	-1.096	0	0.055	1.00
3,90	-1.586	-2.127	-2.018	**-2.686**	1.056e-07	1	0
3,300	-2.067	-2.143	-1.998	**-2.905**	7.749e-14	1	0
3,900	-2.106	-2.176	-2.025	**-3.084**	3.609e-08	1	0
3,3000	-2.157	-2.232	-2.070	**-3.288**	9.525e-12	1	0
4,40	-0.192	-1.071	-1.353	**-1.469**	2.635e-13	0.027	0.23
4,120	-1.391	-2.043	-1.842	**-2.498**	9.712e-11	1	0
4,400	-1.982	-2.095	-1.853	**-2.719**	0.001	1	0
4,1200	-2.072	-2.154	-1.855	**-2.894**	5.221e-10	1	0
5,50	-0.103	**-0.787**	-0.426	-0.566	1.607e-12	0.996	0.16
5,150	-1.034	-1.922	-1.579	**-2.212**	0	1	0
5,500	-1.798	-1.941	-1.527	**-2.356**	1.642e-05	1	0
5,1500	-1.936	-2.040	-1.603	**-2.520**	2.829e-06	1	0

Case with poor initialization

We now reproduce the experiments, comparing EMNA (Algorithm 6) and our improved version IEMNA (Algorithm 7), but in a different setting. Now, σ is initialized at a small value 0.01. Results are presented in Tables 2.6, 2.7, 2.8. QR is still always efficient (or has no effect). We clearly see that, now, with risk of premature convergence, reweighting is very efficient; on the other hand, LB is less efficient. We reproduce graphically in Fig. 2.24 the same results for the Sphere function; we see that only EMNA+QR+weight is nearly as efficient as IEMNA, and that these two algorithms (the only difference between them is that LB is used in IEMNA) are the only algorithms with non negligible convergence rates (all other algorithms have convergence rates nearly zero).

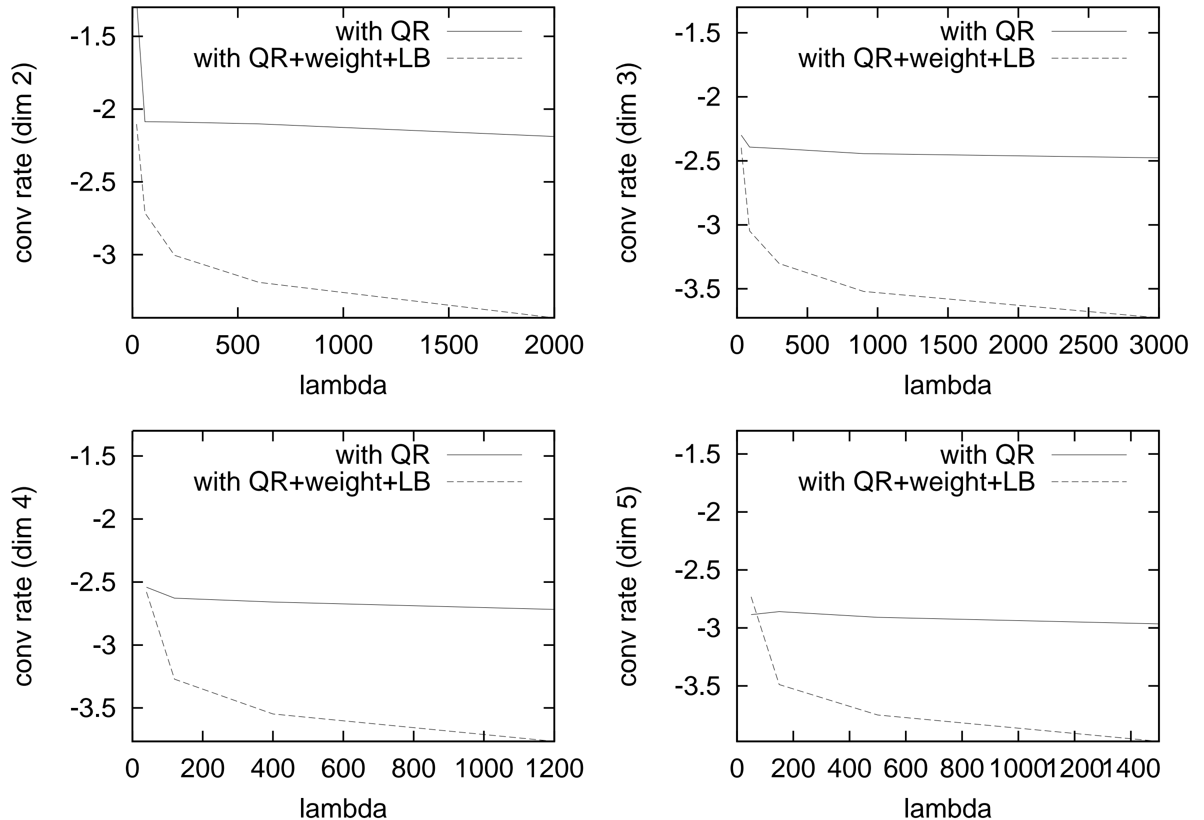

Figure 2.22: **Synthesis of the performance of LB in the case of the Sphere.** Convergence rate of EMNA with QR mutations (as in [Teytaud and Gelly, 2007, Teytaud, 2008b]), which provide the state of the art convergence rates for EMNA with λ large before this work, and convergence rates of IEMNA=EMNA+QR + reweighting + LB (Algorithm 7). The improvement is very clear for λ large. This is not in the case with poor initialization; with poor initialization (see text), results are much more impressive for reweighting (which avoids premature convergence, as shown in [Teytaud and Teytaud, 2009c]), but in that case LB would be harmful (LB makes it more difficult to recover from the bad initialization).

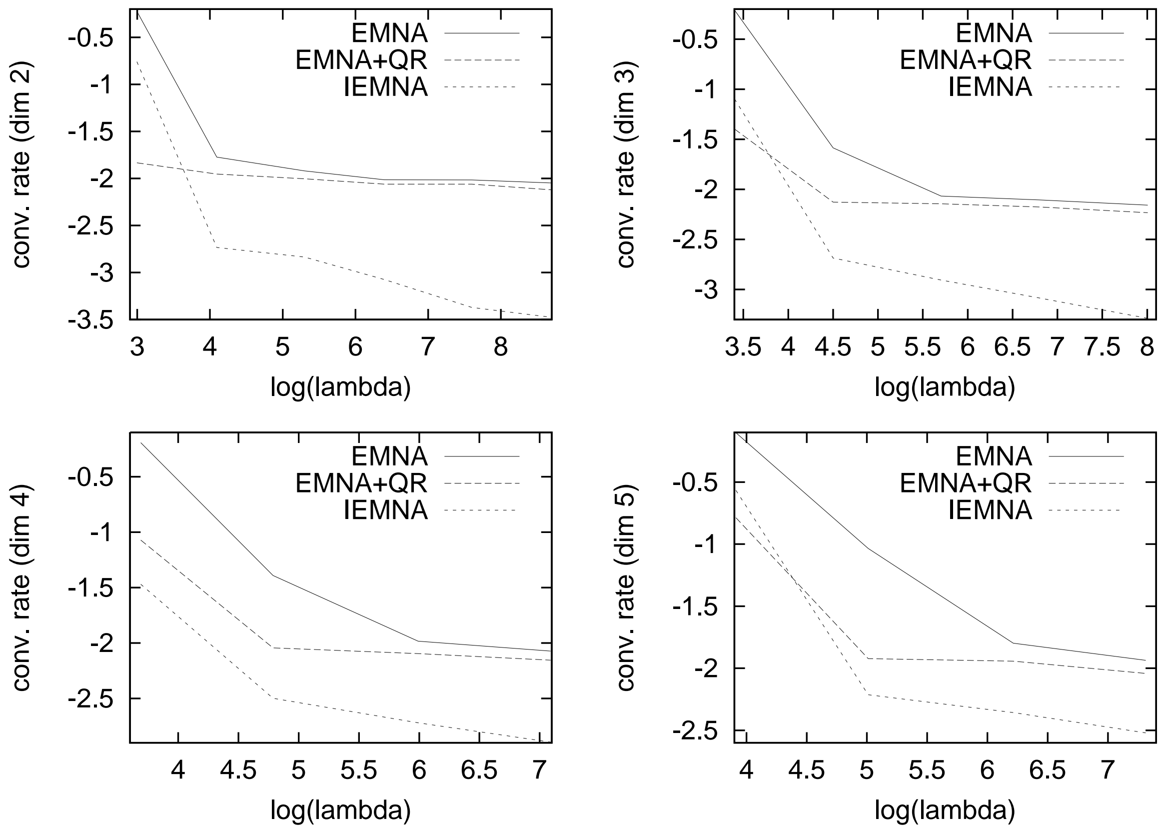

Figure 2.23: **Results on the cigar function.** Convergence rates for EMNA, EMNA+QR, IEMNA (the smaller, the better). For small values of λ, EMNA+QR already provides a great improvement over QR; but for λ large, only IEMNA makes a huge difference. This is due to the fact that when bias is removed (by reweighting) and step-sizes are reduced accordingly (by LB), then variance is the main term, and variance is reduced by QR mutations.

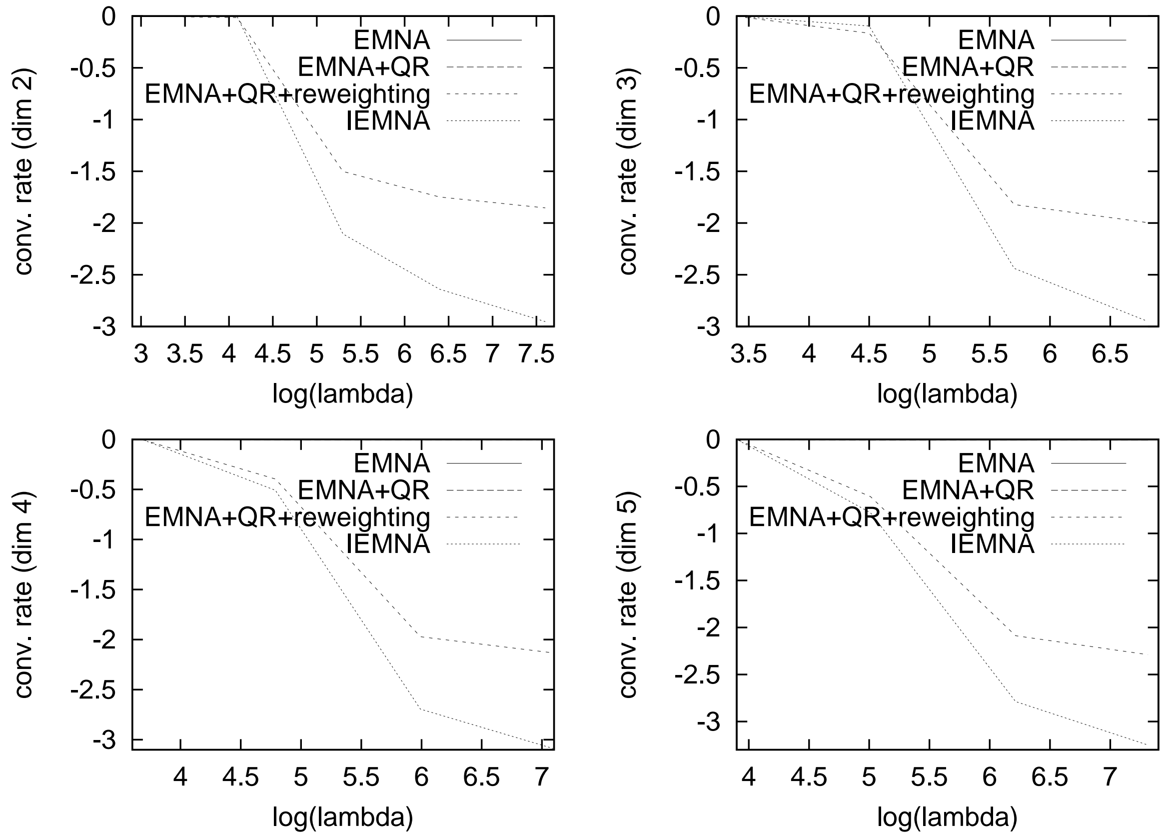

Figure 2.24: **Results on the Sphere function.** Convergence rates for EMNA, EMNA+QR, EMNA+QR+reweighting, and IEMNA=EMNA+QR+LB+reweighting, with very small initial step size. The smaller, the better the result. EMNA, EMNA+QR have convergence rate nearly 0. The main important tool here is reweighting, but IEMNA still improves the results over EMNA+QR+reweighting.

Table 2.6: Experiments on the Sphere function with small initial step-size. The lower the number, the better the result: EMNA+QR+LB+reweighting is usually the best algorithm for λ large enough; for λ small, EMNA+QR+reweighting is stronger - LB is only efficient for λ large, in particular with too small initial step-size. Each modification is validated (p-values are for EMNA+QR versus the initial EMNA (Algorithm 6); then EMNA+QR+LB against EMNA+QR; and finally the complete new version EMNA+QR+LB+reweighting as in Algorithm 7) in this case with small initial step-size (*i.e.* we test the robustness against premature convergence.).

Dimension, lambda	Baseline	+QR	+ weight	+LB (IEMNA)	P-value for QR	P-value for weight	P-value for LB
2,20	-0.000	-0.001	**-0.001**	-0.001	2.204e-05	0	1
2,60	-0.001	-0.001	**-0.014**	-0.005	0.000	0	1
2,200	-0.001	-0.001	-1.501	**-2.106**	5.588e-12	0	0
2,600	-0.001	-0.001	-1.748	**-2.640**	0.009	0	0
2,2000	-0.001	-0.001	-1.853	**-2.952**	0.002	0	0
3,30	-0.001	-0.001	**-0.003**	-0.002	1.056e-06	0	1
3,90	-0.002	-0.003	**-0.165**	-0.096	6.729e-12	7.252e-11	1.00
3,300	-0.003	-0.003	-1.821	**-2.437**	0.001	0	0
3,900	-0.003	-0.003	-1.995	**-2.945**	0.006	0	0
4,40	-0.002	-0.003	**-0.005**	-0.004	4.436e-11	0	1.00
4,120	-0.004	-0.004	-0.395	**-0.508**	4.912e-11	5.747e-11	0.13
4,400	-0.004	-0.005	-1.970	**-2.693**	1.701e-11	0	0
4,1200	-0.005	-0.005	-2.131	**-3.086**	5.772e-07	0	0
5,50	-0.003	-0.004	-0.007	**-0.008**	1.042e-08	0	0.04
5,150	-0.005	-0.006	-0.609	**-0.779**	0	0	0.08
5,500	-0.006	-0.007	-2.087	**-2.786**	0	0	0
5,1500	-0.007	-0.007	-2.288	**-3.250**	0.046	0	0

Table 2.7: Experiments on the multimodal function (see text) with small initial step-size (*i.e.* we test the robustness against premature convergence). Numbers are the negative convergence rates; the lower the better. LB is not efficient in this case in which we have a risk of poor local minima.

Dimension, lambda	Baseline	+QR	+ weight	+LB (IEMNA)	P-value for QR	P-value for weight	P-value for LB
2,20	-0.001	-0.001	**-0.001**	-0.001	0.032	0	1
2,60	-0.001	-0.001	**-0.018**	-0.006	7.251e-11	0	1
2,200	-0.001	-0.001	-0.250	**-0.693**	0.014	0	0
2,600	-0.001	-0.001	-0.240	**-0.312**	3.719e-05	0	0.01
2,2000	-0.001	-0.001	**-0.154**	-0.053	0.038	0	1
3,30	-0.001	-0.002	**-0.003**	-0.002	0.002	0	1
3,90	-0.002	-0.003	-0.109	**-0.144**	0	0	0.036
3,300	-0.003	-0.003	-0.191	**-0.210**	2.797e-09	0	0.02
3,900	-0.003	-0.003	**-0.177**	-0.168	3.195e-08	0	0.89
4,40	-0.002	-0.003	**-0.006**	-0.005	0.016	0	1.00
4,120	-0.004	-0.004	**-0.240**	-0.227	1.265e-14	0	0.70
4,400	-0.004	-0.005	**-0.219**	-0.202	0	0	0.962
4,1200	-0.005	-0.005	-0.200	**-0.202**	0.034	0	0.44
5,50	-0.004	-0.004	**-0.008**	-0.007	0.000	0	0.81
5,150	-0.005	-0.006	**-0.281**	-0.231	0	0	0.984
5,500	-0.007	-0.007	**-0.211**	-0.192	8.039e-12	0	0.89
5,1500	-0.007	-0.007	**-0.024**	-0.002	0.000	0.273	0.71

Table 2.8: Experiments on the cigar function (see text) with small initial step-size (*i.e.* we test the robustness against premature convergence). Numbers are the negative convergence rates; the lower the better. QR clearly works as usual; other modifications are unstable of harmful (this should be different for λ larger).

Dimension, lambda	Baseline	+QR	+ weight	+LB (IEMNA)	P-value for QR	P-value for weight	P-value for LB
2,20	-0.000	-0.000	**-0.000**	-0.000	0.062	8.180e-12	1
2,60	-0.000	-0.001	**-0.015**	-0.012	3.657e-11	0	1
2,200	-0.001	-0.001	**-0.016**	-0.016	0.112	0	0.97
2,600	-0.001	-0.001	**-0.017**	-0.017	0.009	0	1.00
2,2000	-0.001	-0.001	-0.006	**-0.020**	0.205	0	0
3,30	-0.000	**-0.000**	-0.000	-0.000	0.030	0.961	1.00
3,90	-0.001	-0.001	**-0.033**	-0.033	0.003	0	0.47
3,300	-0.001	-0.001	**-0.035**	-0.036	0.000	0	0.00
3,900	-0.001	-0.001	-0.009	**-0.026**	0.385	0.028	0.00
4,40	-0.000	**-0.000**	-0.000	6.412e-05	0.108	0.999	0.89
4,120	-0.001	**-0.001**	0.011	0.019	0.000	0.927	0.73
4,400	-0.001	**-0.001**	0.070	0.111	0.082	1	1.00
4,1200	-0.001	**-0.001**	0.230	0.165	9.548e-05	1	0.00
5,50	-0.000	**-0.001**	0.000	0.000	0.075	1	0.06
5,150	-0.001	**-0.001**	0.167	0.173	0.125	1	0.61
5,500	-0.001	**-0.001**	0.518	0.512	0.000	1	0.41
5,1500	-0.001	**-0.001**	0.799	0.782	0.752	1	0.22

2.3 Automatic parallelization

In previous sections, we have studied and analyzed known algorithms in the case of parallelization. Another idea to study is the case of automatic parallelization. Automatic parallelization refers to the conversion of sequential code into parallel code. The goal of automatic parallelization is to relieve the programmer from the tedious implementation task. A solution (in some cases) for automatic parallelization of an algorithm consists in developing the tree of possible futures, to compute separately all branches, and then to discard bad (non chosen) branches. This is a form of speculative parallelization [Calder and Reinman, 2000]. We here show that this simple approach can be applied to EAs. We have to introduce a somehow tedious formalization; this is necessary for the mathematical formalization of our proofs. As already pointed out in [Teytaud and Gelly, 2006], most EAs can be rewritten as follows:

$$(x^{O_1,O_2}_{n\lambda+1}, \dots, x^{O_1,O_2}_{(n+1)\lambda}) = O_1(\theta, I_n) \qquad \text{(generation)} \qquad (2.19)$$

$$\forall i \in [\![n\lambda+1, (n+1)\lambda]\!], y_i = f(x_i^{O_1,O_2}) \qquad \text{(fitness)} \qquad (2.20)$$

$$g_n^{O_1,O_2} = g(y_{n\lambda+1}, \dots, y_{(n+1)\lambda}) \qquad \text{(selection)} \qquad (2.21)$$

$$I_{n+1} = O_2(I_n, \theta, g_n^{O_1,O_2}), \qquad \text{(update)} \qquad (2.22)$$

for some fixed O_1, O_2, I_0, some random variable θ, and g with values in a set of cardinality K, where:

- I_0 is the initial state and I_n is the internal state at iteration n;
- θ is the random seed;
- g^{O_1,O_2} is the information used by the algorithm, typically in our case the indices of the selected points (and possibly their ranking in the FR case);
- $x_k^{O_1,O_2}$ is the kth visited point and y_k is its fitness value (y_k should, theoretically, be indexed with O_1, O_2 as well);
- (O_1, O_2) is the optimization algorithm, with:
  - O_1 is the function generating the new population (as a function of the random seed and of the internal state);
  - O_2 is the function updating the internal state as a function of the random seed and of the extracted information g.

(note that $g_n^{O_1,O_2}$ and $x_n^{O_1,O_2}$ both depend on θ and f; we drop the indices for the sake of clarity.) We term such an optimization algorithm a λ-optimization algorithm; this means that λ fitness values are computed at each iteration. The optimization algorithm is defined by O_1, O_2, I_0, θ; in cases of interest (below) we use the same θ and the same I_0 for all algorithms and therefore only keep the dependency in O_1 and O_2 in notations. In EAs, g_n has values in a discrete domain; typically, either g_n has values in the set of the finitely many possible ranking of the individuals; or g_n has values in the finite set of possible vectors of ranked indices of selected individuals. g_n is in both cases the only information that the algorithm extracts from the fitness function. In the FR case and $\mu = \lambda$, for example g_n is $(sign(y_{n\lambda+i} - y_{n\lambda+j})_{(i,j)\in[\![1,\lambda]\!]^2})$ where $sign(t) = 1$ for $t \geq 0$ and $sign(t) = -1$ otherwise. In the SB case for (μ, λ)-ES, the formulation is a bit more tedious:

$$g_n = \{I = \{i_1, \ldots, i_\mu\} \subset [\![1, \lambda]\!]^\mu; Card\ I = \mu \text{ and}$$
$$k \in I \wedge k' \in [\![1, \lambda]\!] \setminus I \Rightarrow y_{n\lambda+k} \leq y_{n\lambda+k'}\}.$$

An important property is that the set of possible values for g_n has cardinality $K < \infty$; K can be bounded as follows:

- (μ, λ)-ES (evolution strategies) with equal weights; then $K \leq \lambda!/(\mu!(\lambda - \mu)!)$;
- (μ, λ)-ES with weights depending on the rank; then $K \leq \lambda!/(\lambda - \mu)!$;
- $(1 + \lambda)$-ES; then $K \leq \lambda + 1$;
- $(1, \lambda)$-ES; then $K \leq \lambda$.

K is termed the *branching factor* of the algorithm. The branching factor, and bounds on the branching factor, have been used in [Teytaud and Fournier, 2008] for proving theoretical lower bounds of evolution strategies; we use it here for proving lower bounds on the parallelization of EAs; the lower the branching factor, the better the speed-up. We say that a λ'-optimization algorithm O'_1, O'_2 simulates a λ-optimization algorithm O_1, O_2 with speed-up D if and only if

$$\forall\theta, \forall n \geq 0, \forall i \in [\![1, \lambda]\!], x_{n\lambda'+i}^{O'_1,O'_2} = x^{O_1,O_2}{}_{nD\lambda+i}. \tag{2.23}$$

θ is the random seed; it is removed of indices for short as discussed above, rigorously all the x's depend on it. We now show how we can automatically build O', which is equivalent to O, but with $\lambda' > \lambda$ evaluations at the same time and a known speed-up.

Theorem 1. (Automatic parallelization of EAs and tightness of the $\log(\lambda)$ speed-up.) *Consider a λ-optimization algorithm (O_1, O_2) as in Eqs 2.19-2.22 with branching factor K, and consider λ' such that for some $D \geq 1$:*

$$\lambda \frac{K^D - 1}{K - 1} = \lambda'. \tag{2.24}$$

Then, there is a λ'-optimization algorithm which simulates (O_1, O_2) with speed-up D.

Remark: The speed-up is therefore

$$D = \frac{\log(1 + \frac{\lambda'}{\lambda}(K - 1))}{\log(K)}. \tag{2.25}$$

Proof: We are going to describe a λ'-optimization algorithm O'_1, O'_2, built from O_1, O_2, and we will show Eq. 2.23 for all $n \geq 0, \theta, f$. We do it by induction on n; we assume that it is true for $n - 1$ (unless $n = 0$), and show it for $n \geq 0$.

Consider the set of possible $g_{nD+i}^{O_1,O_2}$ for $i \in [\![0, D-1]\!]$, i.e.

$$\{g_{nD}^{O_1,O_2}, g_{nD+1}^{O_1,O_2}, g_{nD+2}^{O_1,O_2}, \dots g_{nD+D-1}^{O_1,O_2}\}$$

over all fitness functions f and for a fixed value of θ. It has cardinality bounded above by K^D.

Therefore, the set of points possibly visited during steps $nD, \dots, nD + D - 1$ is bounded above by λ times the number of possible $g_i^{O_1,O_2}$ for $i \in [\![nD, nD + D - 1]\!]$; it is therefore bounded above by $\lambda(1 + K^1 + \dots + K^{D-1}) = \lambda(K^D - 1)/(K - 1)$.

So, if

$$\lambda(1 + K^1 + \dots + K^{D-1}) = \lambda(K^D - 1)/(K - 1) = \lambda', \tag{2.26}$$

the algorithm O'_1, O'_2 which:

- computes all the possible $g^{O_1,O_2}{}_{(n-1)\lambda+i}$ for $i \in [\![1, D]\!]$;
- evaluates the fitness functions at all corresponding points of the domain (this is O'_1);
- and simulates the behavior of O_1, O_2 with these fitness values (this is O'_2)

has the following properties:

- it is a λ'-optimization algorithm;

- it simulates O_1, O_2 with speed-up D.

The proof is complete. □

The main weakness of this automatic parallelization is its application in continuous domains. If we use standard algorithms, to have a good speed-up the number of processors necessary will be far from reasonable. For instance, let us take two states of the art algorithms: CMA (Algorithm 4) and EMNA (Algorithm 6) in two dimensions: $d = 2$ and $d = 10$. The latest formulas to compute λ and μ in CMA are :

$$\lambda = 4 + \lfloor 3 * \log(d) \rfloor$$

$$\mu = \frac{\lambda}{2}$$

CMA uses weights depending on the rank, so the branching factor K is $\leq \lambda!/(\lambda - \mu)!$ and EMNA uses equal weights, so K is $\leq \lambda!/(\mu!(\lambda - \mu)!)$. We compute the speed-up by using the formula 2.25. In order to have a speed-up 2, in dimension 2 we need for CMA 726 processors and for EMNA 126 processors. For having a speed-up of 2, in dimension 10, we need for CMA 302410 processors and for EMNA 2530 processors. We can easily see that these numbers are not reasonable, especially if the dimension is not small. If we want a speed-up greater than two, we will need a number of processors even larger.

Chapter 3

Conclusion

The main point of this section was the study of evolutionary algorithm in case of parallelization. We consider that the parallelization consists in simply sending one individual per processor. It is then, somehow intuitive to consider the case of large population sizes. In a first part, we have shown that many real world algorithms do not reach the optimal bounds found in [Fournier and Teytaud, 2011]. It has been first shown theoretically in Section 2.1.2, and then experimentally in Section 2.1.3. In this Section (Section 2.1.3) we show two results:
First, we confirm the superiority of SA-ES over CSA-ES.
Second, we also show the better behaviour of EMNA against SA-ES (and consequently CSA-ES) in case of large population.

In a second part, in Section 2.2, several methods for improving ESs in case of large population sizes are presented, in order to reach the theoretical bounds. First, a new selection ratio rule ($\mu = \min(d, \lambda/4)$) has been proposed in Section 2.2.1. This rule is a simple modification, and has very good results for the SA and the CMSA evolution strategies. We can seemingly reach a logarithmic speed-up with this rule, which is consistent with the theoretical bounds. A selection ratio of 1/2 or 1/4 is better at first view, but it is indeed harmful when the population size λ is large enough (larger than the dimension). Using a selection ratio of $\mu = \min(d, \lambda/4)$ could lead to a very good speedup, better than both with $\mu = 1$ and with $\mu = \lambda/4$ or $\mu = \lambda/2$.

Another proposed improvement is the faster decrease of the step-size σ, the $\log(\lambda)$-modification. This has been presented in Section 2.2.2. With this modification, we have shown that the famous CMA-ES reaches the theoretical bounds on the Sphere function. This modification is also simple

to implement, such as the modification based on the modification of the selection ratio.

In Section 2.2.3, we proposed a new version of the EMNA algorithm. This new version consists in adding a reweighting rule, quasi-random mutations and faster decrease of the step-size. Thanks to a proper bias/variance decomposition, we have emphasized the reasons for some troubles in EMNA: premature convergence as emphasized in [Teytaud and Teytaud, 2009c], and the poor speed-up for λ very large. Thanks to this understanding, we could apply classical statistical techniques: reweighting and quasi-random. Also, thanks to these good estimates, the $\log(\lambda)$-modification, reducing the step-size, leads to improve convergence rates. Results show that using sound bias/variance principles, we can improve Estimation of Distribution Algorithms. All suggested modifications are simple and quite general. The improvements for $\lambda = 20, d = 2$ can reach 700 % for EMNA+QR over EMNA, 500 % for IEMNA over EMNA, and replace premature convergence by real convergence for small step-sizes (thanks to reweighting). For λ very large, the speed-up seemingly goes to infinity; we guess that a more carefully tuned formula for the step-size adaptation should provide much better results.

Quasi-random (QR) is for sure easier in continuous domains (not necessarily for monomodal EDA - quasi-random numbers can be used for many distributions as explained in e.g. [Teytaud and Gelly, 2007]), but they can also be experimented in discrete cases (this is, however, not straightforward). We confirm here results from previous publications, in the case of EMNA and different fitness functions. Interestingly, quasi-random seemingly comes as a free lunch and sometimes very strongly improves the result.

Reweighting can be used in all algorithms based on empirical distributions. The sample of selected points can be replaced by the corresponding sample of weighted selected points, for most (if not all) EDA, both in discrete and continuous domains. This makes sense also for evolution strategies. Importantly, reweighting is not a free lunch - as shown in Table 2.4, there are cases in which reweighting is harmful. On the other hand, it's the only efficient tool against premature convergence.

The fact that the step-size should decrease faster than the standard deviation of selected points when λ increases is also quite general. A simple and different way of developing this idea is to reduce μ, based on the same idea presented in Section 2.2.1 - perhaps this would have the same effect. Decreasing the step-size according to lower bounds (LB) is not a free lunch: it is possibly harmful in particular for moderate values of λ and when there is a risk of premature convergence.

As a summary of this work:

- Quasi-random mutations improve the results. The improvement can be moderate or huge, depending on the framework - almost always at least a few percents (with also decreased variance of performance), and up to 800 % improvement for $\lambda = 10d$ for the cigar function in dimension d or 300 % on the Sphere function for $\lambda = 10d$.

- Reweighting performs very well against premature convergence; on the other hand, it decreases the performance for cases in which the initialization avoids premature convergence.

- The $\log(\lambda)$-modification (forced decrease of the step-size for λ large) performs incredibly well when there is no risk of premature convergence. On the other hand, it can of course (as it decreases σ!) be harmful when there can be premature convergence. The main limitation of this work is that we only tested our ideas on EMNA, for a small set of fitness functions and small dimensions, and especially for λ not too small. It has been shown in previous papers that QR mutations are efficient in many other cases [Teytaud and Gelly, 2007, Teytaud, 2008b].

In Section 2.3, we proposed an automatically parallelization of ES. This parallelization is based on speculative parallelization. As we have seen, the main weakness of this automatic parallelization is its application in continuous domains. This kind of parallelization is only efficient in discrete cases or in small dimensions with $(1+\lambda)$-ES or $(1,\lambda)$-ES. If we do not have such conditions, it is better to use our $log(\lambda)$ modifications in case of large populations. Another possibility when we have a large number of processors (P) is to use a hybrid algorithm between speculative parallelization and classical algorithms. This hybrid algorithm is simply a classical algorithm (with λ processors) but the $P - \lambda$ inactive processors are used in order to do speculative parallelization. If we choose a branch already computed by the speculative parallelization, then the algorithm is faster; if we choose another branch we have the same speed as the classical algorithm. This idea has been presented in [Gardner et al., 2011] for the Particle Swarm Algorithm [Kennedy and Eberhart, 1995].

Part II

Multistage Optimization

In this part, we first introduce some important notions around games and algorithms used in games (Chapter 4). Then, in Chapter 5 we present the state of the art algorithms for tree exploration, including Monte-Carlo Tree Search algorithms. In Chapter 6, we see our contributions to the Monte-Carlo Tree Search algorithm.

Chapter 4

Introduction

First, in Section 4.1 we define the notion of evaluation function, that will be useful throughout this chapter. After this, in Section 4.2, we present the game of Go, which is now established as a standard benchmark for tree search algorithms. In Section 4.3, we present the game of Havannah, a recent game, known to be really difficult for computers.

4.1 Evaluation function

In games, an evaluation function is a function used to estimate the value of a situation. For instance, for the game of Chess, it is easy to build a crude evaluation function :

$$\begin{aligned} eval(s) = {} & (P - P') + 3(N - N' + B - B') \\ & + 5(R - R') + 9(Q - Q') + 200(K - K') \\ & - 0.5(D - D' + S - S' + I - I') \\ & + 0.1(M - M') \end{aligned}$$

where P, N, B, R, Q, K are respectively the number of White pawns, knights, bishops, rooks, queens and kings, D the number of doubled White pawns, S the number of backward White pawns, I the number of isolated White pawns and M the mobility of the White player (measured, for instance, as the number of possible moves). Primed letters correspond to the same representation but for the Black player. s corresponds to a situation, i.e. a position of a game. Such a symmetric evaluation function has been first presented in [Shannon, 1950]. For some game, it is not possible to build such a function. For instance, for the game of Go and the game of Havannah, the first programs playing these games were based on an evaluation function; however,

these programs programs quickly became outperformed by programs which were not based on evaluation functions, because of the imprecision of the evaluation function. The reasons why it is difficult for these games to build an evaluation function are several. First, obviously, there is no information based on the material on the board. In the game of Go, the number of stones is not a valuable information, and in the game of Havannah the number of stones is the same for both players. Second, the meaning of a move can be really deep and complex, even for a human. For example, it may be possible to capture some enemy stones by strengthening the opponent's stones elsewhere. In the game of Go there is an important notion which is the influence, and this is really difficult to measure, because this is not directly visible on the board, and it can depend on the position in its entirely.
The important thing is that if there is no evaluation function available, then we can not directly have a score for a position. Alpha-Beta algorithms, presented in Section 5.1 can not be used in that case. Monte-Carlo based algorithms, presented in Section 5.2 can handle such constraints, because the evaluation is done differently.

4.2 The game of Go

The game of Go is an ancient Asian two-player (Black and White) board game. It is mainly played on two different sizes, 9x9 and 19x19, but sometimes games are played on an intermediate size 13x13. The size represents the number of location of each side. For instance, in 19x19, there are 361 possible locations. Rules are quite simple and consist in the following. Each player puts a stone in an empty intersection on the board. Black player starts. Two definitions are important to well understand a game: A *group* is a connected (horizontally or vertically) set of stones of the same color. A *liberty* is an empty intersection next to a group. When a group has no more liberties, it is removed from the board. A *dead stone* is a stone on the board which will be captured by the opponent whatever its owner does. At the end of the game, after both players have passed consecutively, the dead stones are removed from the board. The score is counted for each player. The player with the higher score has won. The score for the black player (resp. white) is the sum of the number of black (resp. white) stones on the board, plus the number of empty intersections which belong to the black player (resp. white), i.e. the intersections which are surrounded by black stones (resp. white stones). As example of game is shown in Figure 4.1.

The game of Go, as in many board games, the first player has an advantage. To compensate this advantage, the second player gets a certain number

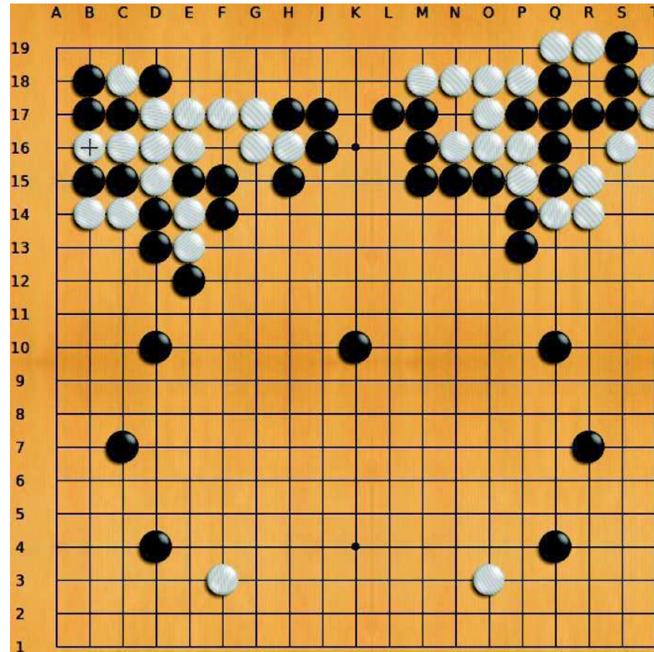

Figure 4.1: Example of a 19x19 game of Go between MoGo a top Go-program and Chou-Hsun Chou a top-player. For instance, the B16 white group is a group of 11 stones and has 6 liberties.

of compensation points, called *komi*.

Handicap rule is used to offset the strength difference between two players of different ranks. A game with X handicap is therefore a game for which the second player allows the first player to put X stones on the board before playing himself.

This game is much harder than the game of Chess, and is now a main challenge for computers. A complexity study has been done in Go. On the 3^{19x19} possible board positions, only 1.196% are legal positions, so the maximum legal game positions is $2.08168199382 \text{x} 10^{170}$ [Tromp and Farnebäck, 2007]. The average branching factor is about 200. This means that, on average, a player has approximately 200 legal moves at his disposal at each turn (whereas for the game of Chess, the average branching factor is about 35). In 1998, a strong amateur (6 dans) was able to win against Many Faces of Go which was at this time the best Go program with 29 handicap stones for the computer [Müller, 2002]. With new algorithms, this gap has been considerably reduce, and MoGo, a top Go program won in 2009 against a 9D professional, Chou-Hsun Chou, with only 7 handicap stones for the computer [Rimmel et al., 2010]. There are still a lot of difficulties remaining in

the game of Go. One of these difficulties, is the problem of semeais. A semeai is a situation where each side has to capture the opponent's group in order to save its own group. A position with a semeai is presented in Figure 4.2.

In Section 5.2 we present these new algorithms. The main reason for the difficulty of the game of Go for computers is that there is no evaluation function.

4.3 The game of Havannah

The game of Havannah is another two-player board game with complete information. It is a recent game, invented by Christian Freeling [Freeling, 2003]. This game is played on an hexagonal board of hexagonal locations with different sizes. Small sizes are 4 or 5 cells per side, and largest sizes are 8 or 10 cells per side. Rules are much simpler than for the game of Go and are as follows. Each player puts a stone alternatively in an empty location. The white player starts. If there is no more empty cell on the board and if no player has won yet, then the result is a draw. Such situations are very rare. To win a game, a player has to realize one of these three shapes:

- A ring, which is a connected string of stones around one or more cells. The surrounded cells can be empty or not, occupied by black or white stones.
- A bridge, which is a connected string of stones between two of the six corners.
- A fork, which is a connected string of stones between three of the six sides. Corner locations do not belong to any side.

An example of game is presented in Figure 4.3 and examples of the three winning shapes are shown in Figure 4.4. The game of Havannah is a nice challenge for computers because only few patterns and expert knowledge are known. The total number of possible positions is large. A large bound is 271! in size 10, but no complexity study has been done yet, in order to reduce this number. As for the game of Go, it is difficult to build an evaluation function for this game.

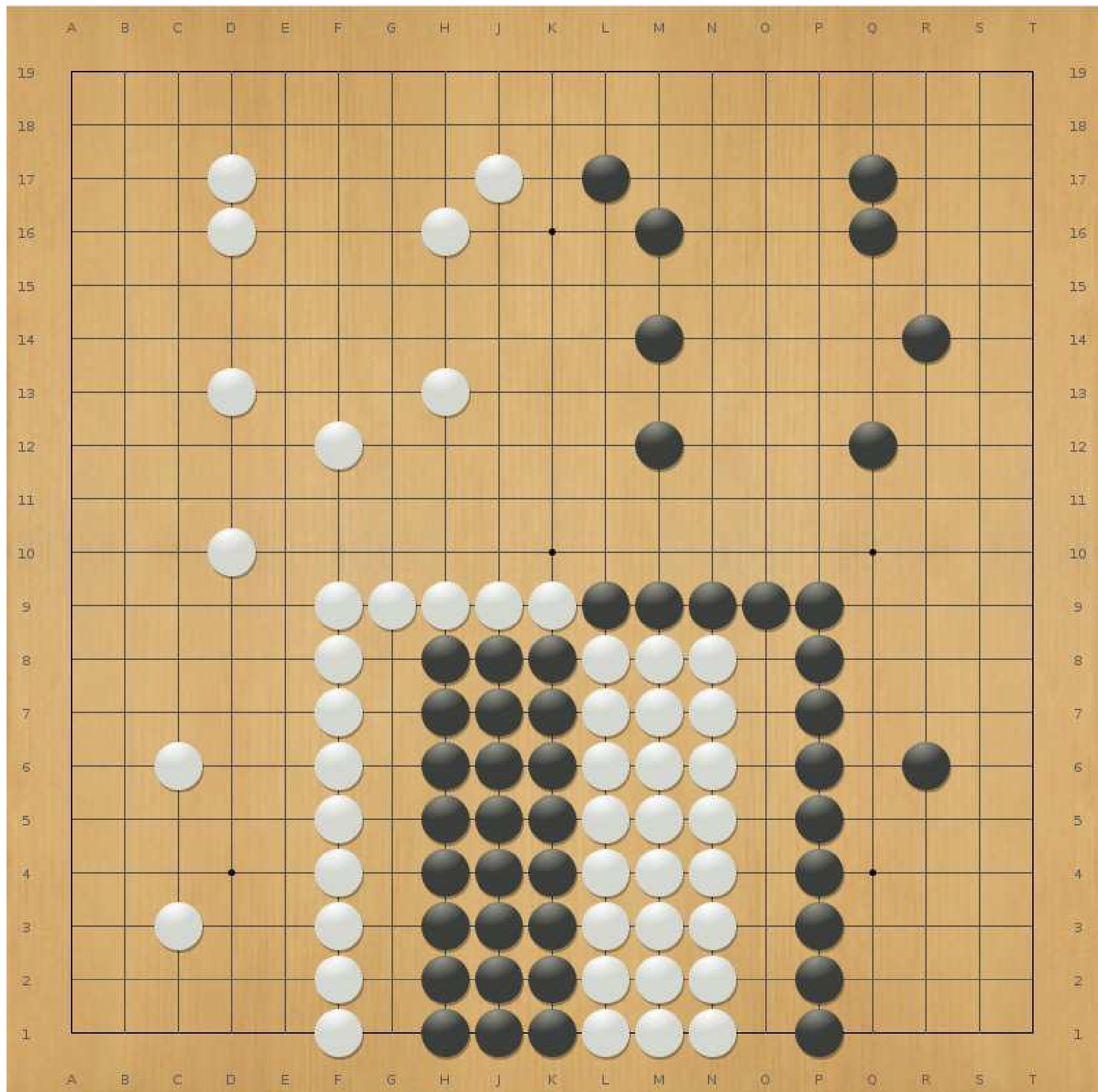

Figure 4.2: Example of a 19x19 position with a semeai. The black group J5 is alive if the black player is able to kill the white group M5. The white group M5 is alive if the white player is able to kill the black group J5. This is an easy example for human players, because the number of locations to fill is the same for both players. The player to play will therefore win this semeai.

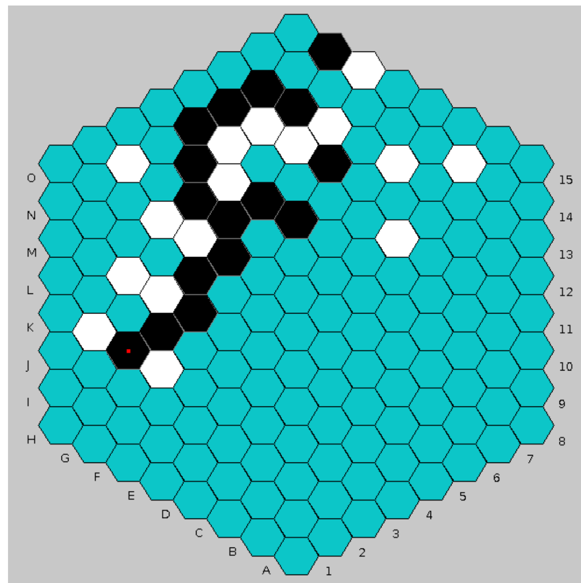

Figure 4.3: Example of a game of Havannah in size 8. In this position, Black wins because White can not do anything to avoid Black to be connected to three sides (and then to win with a fork).

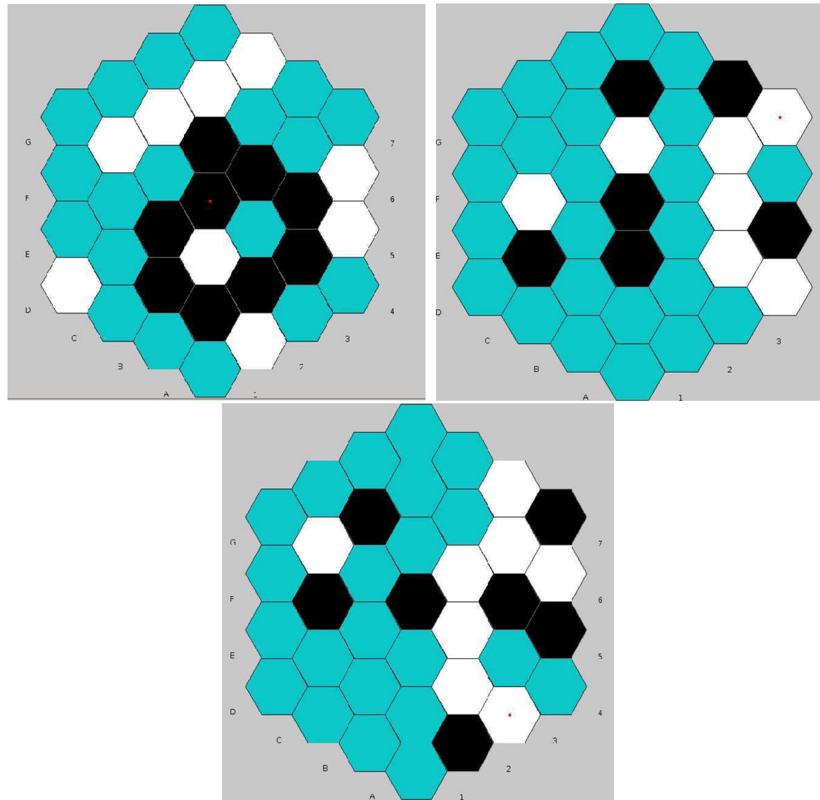

Figure 4.4: Top-Left: a won by ring for Black. Top-Right: a won by bridge for White. Bottom: a won by fork for White.

Chapter 5

State of the art

In this Chapter we present state of the art algorithms for tree exploration. First in Section 5.1 the Alpha-Beta algorithm. In Section 5.2 we present bandit based Monte-Carlo tree search methods. In Section 5.3 we present the Nested Algorithm.

5.1 Alpha-Beta algorithm

Alpha-Beta algorithm is a refinement of the classical Minimax algorithm. The Minimax algorithm is used for two-player zero sum games. Let d be the depth of the algorithm. The principle is to build the complete tree of the possible future states until depth d. In this algorithm an evaluation function is needed and called when the algorithm reaches a leaf. One player (noted player *Max*) tries to maximize the return score (given by the evaluation function) whereas the other player (noted *Min*) tries to minimize it. Algorithm 8 presents the pseudo-code of the Minimax algorithm.

Algorithm 8 Minimax algorithm.

```
arguments node n, depth d
if n is terminal or d = 0 then
  return evaluation of n
else
  α ← −∞
  for each child c of n do
    α ← max(α, −minimax(c, d − 1))
  return α
```

The most famous improvement of the Minimax algorithm is called Alpha-Beta [Shannon, 1950] and its theoretical analysis can be found in

[Knuth and Moore, 1975]. The principle is to prune branches of the tree that can not influence the final result. In order to do so, we need to maintain the minimum score that *Max* can obtain (α) and the maximum score that *Min* can obtain (β). If β becomes less than or equal to α then we do not need to explore further. The Alpha-Beta algorithm always returns the same answer than the Minimax algorithm. Pseudo-code for this algorithm is presented in Algorithm 9.

Figure 5.1 [Rimmel, 2009], shows a tree representation for the Minimax algorithm and the Alpha-Beta improvement.

Algorithm 9 Alpha-Beta algorithm.

```
arguments node n, depth d, α, β
if n is terminal or d = 0 then
  return evaluation of n
else
  α ← −∞
  for each child c of n do
    α ← max(α, −alphabeta(c, d − 1, −β, −α))
    if β ≤ α then
      break
  return α
```

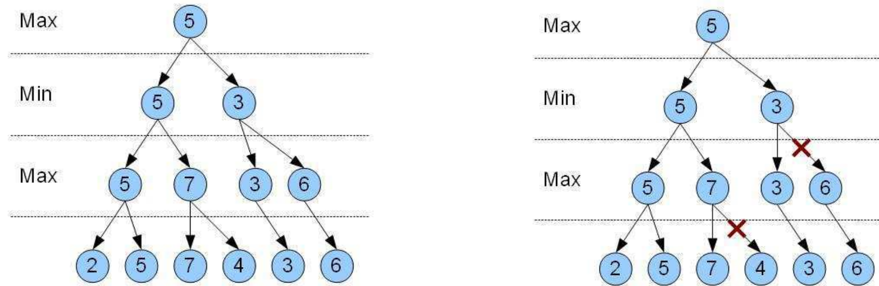

Figure 5.1: Left: tree representation of the Minimax algorithm. Right: tree representation of the Alpha-Beta algorithm. Crosses show the pruning done by the Alpha-Beta improvement.

5.2 Bandit Based Monte-Carlo Tree Search

Bandit Based Monte-Carlo Tree Search algorithms are based on an incremental construction of a tree representing the possible future states. To do

this construction, a bandit formula and Monte-Carlo simulations are used. First, we present multi-armed bandits in Section 5.2.1 and the Monte-Carlo Tree Search algorithm in section 5.2.2.

5.2.1 Multi-Armed Bandits

A k-armed bandit problem, in the usual stochastic framework (other frameworks, in particular the adversarial case, exist), is defined by the following elements:

- A finite set of arms $J = \{1, \ldots, k\}$ is given.
- When pulled, each arm $j \in J$ generates a reward r, whose value is an unknown random variable X_j; the expectation of X_j is denoted μ_j.
- At each time step $t \in \{1, 2, \ldots\}$, the algorithm chooses $j_t \in J$ depending on $(j_1, \ldots, j_{t-1})$ and $(r_1, \ldots, r_{t-1})$, and gets a reward r_t, which is an independent realization of X_{j_t}.

The goal is to minimize the so-called *regret*. Let $T_j(n)$ the number of times an arm has been selected during the first n steps. The *regret* after n steps is defined by

$$\mu^* n - \sum_{j=1}^{n} \mu_j \mathbb{E}[T_j(n)] \text{ where } \mu^* = \max_{1 \leq i \leq n} \mu_i.$$

[Auer et al., 2002] achieve a logarithmic regret (it has been proved that this is the best obtainable regret in [Lai and Robbins, 1985]) independently of the X_j with the following algorithm: first, try one time each arm; then, at each step, select the arm j that maximizes

$$\bar{x_j} + \sqrt{\frac{2ln(n)}{n_j}}. \tag{5.1}$$

$\bar{x_j}$ is the average reward for the arm j (until now). n_j is the number of times the arm j has been selected so far. $n = \sum_j n_j$ is the overall number of trials so far. This formula consists in choosing at each step the arm that has the highest Upper Confidence Bound (UCB). It is called the UCB formula. The first part of the formula is called the exploitation part, and the second part of the formula is called the exploration part.

5.2.2 Monte-Carlo Tree Search

First, in Section 5.2.2 we define two functions which are frequently used in this part. In Section 5.2.2 we present the Monte-Carlo Tree Search and eventually some improvements of this algorithm in Sections 5.2.2, 5.2.2 and 5.2.2.

Definitions.

Before presenting the Monte-Carlo Tree Search algorithm we first introduce two functions. The first one is the function $mc(s)$ which plays a random move from the situation s, i.e., the action is chosen randomly according to a uniform distribution and returns the new position.

The second one is the function $result(s)$ which returns the score of the final situation s taken as argument.

In this section, we note $f \rightarrow s$ the move which leads from a node f to a node s (f is the father and s the child node corresponding to move $m = f \rightarrow s$).

Algorithm.

Monte-Carlo tree Search (MCTS) has first been introduced in [Coulom, 2006] and the UCT version was proposed in [Kocsis and Szepesvari, 2006]. The main idea in MCTS algorithms is to construct a highly imbalanced partial game tree $\hat{T}$, in order to focus on and explore deeper the most interesting parts of the complete tree T and avoid constructing less interesting parts. The MCTS algorithm is presented in Algorithm 10 and illustrated on Figure 5.2.

Some statistics are attached to each node present in the tree, generally consisting in the number of simulations which have passed through this node, and the number of these simulations which are a win.

The construction of the tree $\hat{T}$ is done incrementally and consists in three parts: *descent, evaluation and growth*.

Descent. The descent in $\hat{T}$ is done by considering that selecting a new node is equivalent to a k-armed bandit problem. In each node s of the tree, the following information are stored:

- n_s: the total number of times the node s has been selected.
- $\bar{x_s}$: the average reward for the node s.

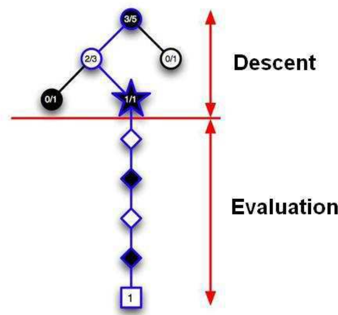

Figure 5.2: Illustration of the Monte-Carlo Tree Search algorithm. The Descent part consists in choosing a new node according to a bandit formula. The evaluation policy appears when we reach a situation outside the Tree. In the simplest case, the evaluation is done by doing a Monte-Carlo simulation.

The formula to select a new node s' is based on the UCB formula 5.1. Let C^s be the set of child of the node s:

$$s' \leftarrow \arg\max_{j \in C^s} \left[\bar{x_j} + \sqrt{\frac{2ln(n_s)}{n_j}} \right]$$

Once a new node has been selected, the same principle is repeated until we reach a situation S outside $\hat{T}$.
We call tree policy the policy for taking decision inside the tree (i.e. using the bandit formula 5.1 here).

Evaluation. At the end of the *descent* part, we have reached a situation S outside $\hat{T}$. Now there is no more information available to take a decision; we can not, as in the tree, use the bandit formula. As we are not at a leaf of T, we can not directly have a result for S. Instead, we use a Monte-Carlo simulation to have a value for S. The Monte-Carlo simulation is done by selecting a new node (a child of S) using the heuristic function $mc(S)$ and this process is repeated until a final situation is reached. $mc(S)$ returns one element of C^S based on a uniform distribution. In section 6.2, we see that better distributions than the uniform distribution are possible. The policy used for taking a decision where the situation is outside of the tree is called the default policy or the Monte-Carlo part. The basic default policy is a random policy.

Growth. In the *growth* step, we add the node S to $\hat{T}$. In some implementations, the node S is added to the node only after a finite fixed number of simulations instead of just 1. The goal of this trick is to keep in memory only nodes that are relevant.

After adding S to $\hat{T}$, we update the information in S and in all the situations encountered during the descent with the value obtained with the Monte-Carlo evaluation (the numbers of wins and the numbers of losses are updated), so that $\bar{x}_s$ and n_s are known for all s in $\hat{T}$.

Algorithm 10 MCTS(s).

```
argument node s
Initialization of T̂, n, x̄
while there is some time left do
  s' ← s
  Initialization of game
  //DESCENT
  while s' in T̂ and s' not terminal do
    s' ← arg max_{j∈C^{s'}} [x̄_j + sqrt(2ln(n_{s'}) / n_j)]
    game ← game + s'
  S ← s'
  //EVALUATION
  while s' is not terminal do
    s' ← mc(s')
  r = result(s') {//Function defined in Section 5.2.2}
  //GROWTH
  T̂ ← T̂ + S
  for each s in game do
    n_s ← n_s + 1
    x̄_s ← (x̄_s * (n_s − 1) + r) / n_s
```

Modification of the bandit formula.

A very simple but important improvement is the tuning of a parameter in the bandit formula proposed in 5.1. It is important to have such a parameter in this formula because the trade-off between exploration and exploitation is application-dependant. With this control parameter the new formula is now :

$$\bar{x_j} + \alpha \times \sqrt{\frac{2ln(n)}{n_j}}. \tag{5.2}$$

$\bar{x_j}$ is the average empirical reward for arm j, n_j is the number of times the arm j has been selected so far, n is the overall number of trials so far,

α the control parameter previously defined. A related improvement (scaling with the standard deviation) has been proposed in [Audibert et al., 2008].

All Moves As First.

All Move As First (AMAF) was initially introduced in [Bruegmann, 1993].

We can define the number of AMAF wins as the number of won simulations such that f has been encountered and m has been played after situation f by the player to play in f (but not necessarily in f!). Similarly, we can define the AMAF losses.

Rapid Action Value Estimates.

Rapid Action Value Estimates (RAVE) is a very efficient and generic improvement. Is was introduced in [Gelly and Silver, 2007].

The AMAF wins and AMAF losses numbers are termed RAVE wins and RAVE losses when they are used in MCTS.

The principle is to store, for each node s with father f,

- the number of wins (won simulations encountering s - this is exactly the number of won simulations playing the move m in f);
- the number of losses (lost simulations playing m in f);
- the number of AMAF wins.
- the number of AMAF losses.

The percentage of wins established with RAVE values instead of standard wins and losses is noted $\bar{x}_{f,s}^{RAVE}$. The total number of games starting from f and in which $f \to s$ has been played is noted $n_{f,s}^{RAVE}$. This number is much bigger than n_s ; therefore it is usually said that RAVE has lower variance (but larger bias)

From the definition, we see that RAVE values are biased; a move might be considered as good or bad (according to $\bar{x}_{f,s}$) just because it is good or bad later in the game.

Nonetheless, RAVE values are very efficient in guiding the search: each Monte-Carlo simulation updates many RAVE values per node on its path, whereas it updates only one standard win/loss value in standard MCTS. Thanks to this larger amount of statistical data, RAVE values are known to be more biased but to have less variance.

Those RAVE values are used to modify the bandit formula 5.1 used in the *descent* part of the algorithm. The new formula to chose a new node s' from the node s is given bellow; let C^s be the set of child of the node s.

$$s' \leftarrow \arg\max_{j \in C^s} (1-\beta)\bar{x}_j + \beta \bar{\mathbf{x}}^{\mathbf{RAVE}}_{\mathbf{s,j}} + \sqrt{\frac{2\ln(n_s)}{n_j}} \tag{5.3}$$

β is a parameter that tends to 0 with the number of simulations n and which is close to 1 when n is small. When the number of simulations is small, the RAVE term has a larger weight in order to benefit from the low variance. When the number of simulations gets high, the RAVE term becomes small in order to avoid the bias.

Last Good Reply.

This improvement has been proposed in [Drake, 2009]. This improvement is generic (it can be used for many applications) and appears in the default policy.

During a simulation, if the Black player replies action b to action a of the White player and if the simulation is a win for Black then for the next simulation, it seems reasonable for Black to play b if White tries action a, even if it is not in the same state. If the action b is not possible for Black at that time, then Black simply chooses the action according to the default policy (i.e. using the function $mc()$ in that case).

The Last Good Reply (LR) requires only a small amount of memory as each player only needs to maintain the last successful reply for each action.

5.3 Nested Monte-Carlo

The Nested Monte-Carlo (NMC) algorithm is an algorithm introduced in [Cazenave, 2009]. This algorithm is recursive and each level calls the lower level in order to determine which action has to be selected. The lowest level of the NMC algorithm, i.e. the level 0 is a Monte-Carlo simulation, in other words, each action is randomly chosen according to a uniform distribution until a final situation is reached. This corresponds to the function mc. More precisely, in each position, a NMC search of level l performs a NMC of level $l-1$ for each possible action and then select the one with the best score. For instance, a NMC search of level 1 does a Monte-Carlo simulation (corresponding to a NMC of level 0) for each possible action and selects the action with the highest score. Tristan Cazenave shows in [Cazenave, 2009], that the NMC algorithm can be easily improved by keeping in memory the

best sequence of actions found so far. It is useful when a future nested search gives worse results than the best sequence. This algorithm is presented in Algorithm 11.

Algorithm 11 Nested.

arguments node s, level l
$bestscore \leftarrow -1$
while not end of the game **do**
  if $l = 0$ **then**
    $move \leftarrow argmax_m(\text{result}(\text{mc}(play(position, m))))$
  else
    $move \leftarrow argmax_m(nested(play(position, m), l - 1))$
  if $currentscore > bestscore$ **then**
    $bestscore \leftarrow currentscore$
    $bestsequence \leftarrow currentsequence$
  $bestmove \leftarrow move$ of $bestsequence$
  $position \leftarrow play(position, bestmove)$

Chapter 6

Contributions

In this Chapter, we argue the generality of the Monte-Carlo Tree Search algorithm. In order to show this generality, we test the Monte-Carlo Tree Search algorithm on the game of Havannah, presented in Section 4.3. Different improvements of the Monte-Carlo tree Search, known to be efficient in the game of Go, are tested. This is presented in Section 6.1. Regarding to this result, we think that it is important to preserve this generality. Then, in Section 6.2, we present 3 generic improvements of the default policy of the Monte-Carlo Tree Search algorithm.

In Section 6.3, we present the use of evolutionary algorithms, presented in Section 1.1 for tuning the Nested algorithm (Algorithm 11), and we point out that this approach can be efficient on this application thanks to heuristics and a good tuning.

6.1 Application to the game of Havannah

As seen in Section 4.3 the game of Havannah is recent. Consequently, only a few programs playing Havannah exist, but it emerges more and more as a new challenge for computers. Programs were using Alpha-Beta algorithms first. Since [Teytaud and Teytaud, 2010a] UCT algorithms have been successfully introduced and are now used by all Havannah programs. The results of this section have been published in this paper. In this work the goal is to test the generality of UCT by experimenting it in the game of Havannah. We try different improvements of UCT known to be good in the game of Go.

$K_{bernstein}$	Score against Hoeffding's formula 6.1
0.000	0.439 ± 0.015
0.001	0.582 ± 0.012
0.010	$\mathbf{0.652 \pm 0.006}$
0.030	0.646 ± 0.005
0.100	0.578 ± 0.010
0.250	0.503 ± 0.011

Table 6.1: The first value 0.25 corresponds to Hoeffding's bound except that the second term is added. Experiments are performed with 1000 simulations per move, with size 5. We tried to experiment values below 0.01 for $K_{bernstein}$ but with poor results.

6.1.1 Bandit Formula

First, we experimentally tune, by self-play, the bandit formula 5.2, and found for the constant α the value $\sqrt{0.125}$. The new bandit equation is then :

$$\bar{x_j} + \sqrt{\frac{0.25 \times ln(2+n)}{n_j}}. \tag{6.1}$$

The added constant 2+ is here for avoiding special case for 0.

The exploration part is based on the result of the Hoeffding's bound. An improvement of the exploration part consists in using the variance of the rewards of the arms. If the variance for an arm is low, then it takes less iterations to know that this arm is not optimal. To use the variance a possibility is to use the Bernstein's bound instead of Hoeffding's bound [Audibert et al., 2006, Mnih et al., 2008]. Then, we experiment this new exploration term. The resulting exploration term is then:

$$\bar{x_j} + \sqrt{\frac{4K_{bernstein}\bar{x_j}(1-\bar{x_j})\log(2+n)}{n_j}} \tag{6.2}$$

$$+ \frac{3\sqrt{2}K_{bernstein}\log(2+n)}{n_j}. \tag{6.3}$$

Here again, the added constants 2+ are here for avoiding special cases for 0. This term is smaller for moves with small variance (and this whatever $score(d)$).

We tested several values of $K_{bernstein}$, results are presented in table 6.1.

However, when the number of simulations becomes larger the Formula using Hoeffding's bound (Formula 6.1) performs better [Lorentz, 2011].

6.1.2 Progressive Widening

With progressive widening [Coulom, 2007, Chaslot et al., 2007, Wang et al., 2008], we first rank the legal moves at a situation s according to some heuristic: the moves are then renamed 1, 2, ..., n, with $i < j$ if move i is preferred to move j for the heuristic. Then, at the mth simulation of a node, all moves with index larger than $f(m)$ have a score equal to $-\infty$ (i.e. are discarded), with f some non-decreasing mapping from $\mathbb{N}$ to $\mathbb{N}$. It was shown in [Wang et al., 2008] that this can work even with random ranking, with $f(m) = \lfloor K_{pw} m^{\frac{1}{4}} \rfloor$ for some constant K_{pw}. In [Coulom, 2007] it was shown that $f(m) = \lfloor K_{pw} m^{\frac{1}{3.4}} \rfloor$ for some constant K_{pw} performs well in the case of Go, with a pattern-based heuristic for move ranking. The algorithm proposed in [Chaslot et al., 2007] and now used also in MoGo is a bit different: an exploration term depending on a pattern-based heuristic and decreasing logarithmically with the number of simulations of this move is added to the score. In that case, for a move d the bandit formula is:

$$\hat{x_d} + \frac{H(d)}{\log(2+m)}$$

with H the pattern-based heuristic function.

In the case of Havannah, we do not have such heuristics. We decided to use heuristic-free progressive widening, as it was shown in [Wang et al., 2008] that an improvement can be provided even if no heuristic is available (i.e. the order is arbitrary). This idea of using progressive widening without heuristic is a bit counter-intuitive. However, consider for instance, a node of a tree which is explored 50 times only (this certainly happens for many nodes deep in the tree). If there are 50 legal moves at this node, if you use the standard UCB formula, then the 50 simulations will be distributed on the 50 legal moves (one simulation for each legal move). Meanwhile, progressive widening will sample a few moves only, e.g. 4 moves, and sample much more the best of these 4 moves - this may be better than taking the average of all moves as an evaluation function. We experimented with 500 simulations per move, size 5, various constants P and Q for the progressive widening $f(m) = Q\lfloor m^P \rfloor$. Results are presented in Table 6.2. These experiments were performed with the exploration formula given in Eq. 6.1. We tested various other parameters for Q and P, without any significant improvement.

Q, P	Success rate against no prog. widening
1, 0.7	0,496986 $\pm$ 0.014942
1, 0.8	0.51 $\pm$ 0.0235833
1, 0.9	0.50 $\pm$ 0.0145172
4, 0.4	0,500454 $\pm$ 0.0134803
4, 0.7	0.49 $\pm$ 0.0181818
4, 0.9	0,485101 $\pm$ 0.0172619

Table 6.2: Experiment of using progressive widening in the game of Havannah.

6.1.3 Rapid Action Value Estimates

In formula 5.3 we have defined the parameter β which controls the trade-off between the UCB score and the RAVE score. We use the following formula for defining β:

$$\beta = \frac{R}{R+n}$$

For small number of simulations (1000) $\alpha = 0$ is the best constant. This was also pointed out in [Lee et al., 2009] for the game of Go. We then tested larger numbers of simulations, i.e. 30 000 simulations per move. Disappointingly but consistently with [Lee et al., 2009], we had to change the coefficients in order to get positive results, whereas the tuning of UCT is seemingly less dependent on the number of simulations per move. The fifth line in Table 6.3 corresponds to the configuration empirically chosen for 1000 simulations per move; its results are disappointing, almost equivalent to UCT, for these experiments with 30 000 simulations per move. The sixth line in Table 6.3 uses the same exploration constant as UCT, but it's seemingly too much. Then, the following lines of Table 6.3, using a weaker exploration and a small value of R, gets better results. [Lee et al., 2009] points out that, with big simulation times, $\alpha = 0$ was better, but an exploration bonus depending on patterns was used instead.

As a conclusion, for large numbers of simulations, RAVE is not as efficient as for small number of simulations. (when compared to UCT).

6.2 Improvement of the default policy

In this section, we propose several methods to improve the default MCTS policy. A main weakness of MCTS is that choosing the right Monte-Carlo

size	R	α	number of simulations	success rate against no rave
5	50	0	1000	95.33% ± 0.01 %
5	50	0.05	1000	60.46% ± 2.9 %
5	**50**	**0.25**	**1000**	**47.26% ± 4.0 %**
8	**50**	**0**	**1000**	**100% on 1347 runs**
5	5	0.02	30 000	0.61 ± 0.06
5	**20**	**0.02**	**30 000**	**0.66 ± 0.03**
5	50	0	30 000	0.53 ± 0.02
5	50	0.02	30 000	0.60 ± 0.03
5	50	0.05	30 000	0.60 ± 0.02
5	50	0.25	30 000	0.47 ± 0.04

Table 6.3: Experiment of the RAVE improvement of the game of Havannah for sizes 5 and 8. α is the constant parameter defined in formula 5.2. R is the RAVE constant defined in 6.1.3.

formula ($mc(.)$ in Alg. 10) is very difficult. All the improvements proposed here are aimed at being generic, and consequently are independent of the application. We have seen in Section 5.2.2 the Last Good Reply improvement which is also a generic improvement of the default policy. Improving the MCTS algorithm with expert knowledge or domain-dependant knowledge is really important because these enhancements are in general very efficient, then the resulting algorithms are far better. However, having generic improvements is really important simply because it improves the general behavior of the algorithm. In all this section, we present generic improvements of the default policy. We first present a modification of the default policy based on the RAVE improvement in Section 6.2.1. In Section 6.2.2 we present an improvement based on multiple overlapping tiles and in Section 6.2.3 the "check mate in one" improvement.

6.2.1 poolRave

As said in Section 5.2.2, RAVE is a generic improvement of the tree policy. Here, we propose the "poolRave" modification. This modification is a generic way to improve the default policy of the MCTS algorithm. The results of this modification have been presented in [Rimmel et al., 2011b]. The modification here is really simple and is as follows:

- build a pool of the k best moves according to the RAVE values.

- choose one move m in this pool.
- with a probability p, play m, otherwise use $mc(s)$.

The RAVE values are those of the last node with at least 50 simulations. This improvement is very simple to implement if the RAVE improvement is already implemented. Two parameters are needed. The first one is k which defines the number of moves contained in the pool. The second parameter is the probability p of playing a move from the pool instead of a regular move. Such a probability is crucial because it is very important to limit the bias introduced by the Monte-Carlo part and then to keep diversity during the simulations.

The MCTS algorithm with the "poolRave" modification is presented in Algorithm 12.

The generality of this approach is demonstrated by experimenting this modification on two different applications: the classical application to the game of Go, and the interesting case of Havannah in which far less expertise is known.

We measure the success rate of our program with the new "poolRave" modification against the baseline version of our bot. We have experimented different numbers of simulations in order to see the robustness of our modification.

Results on the game of Havannah are shown in table 6.4. The best results are obtained with $p = \frac{1}{2}$ and a pool size of 10, for which we have a success rate of 54.3% for 1000 simulations and 54.5% for 10000 simulations. With the same set of parameters, for 20000 simulations we have 54.4%, so for the game of Havannah this improvement seems to be independent of the number of simulations.

We experiment this modification in the top Go-program MoGo. The probability p of using the modification is useful in order to preserve the diversity of the simulations. In MoGo, the diversity of the simulations is ensured by the fillboard modification [Chaslot et al., 2010]. This modification is simple: a location is randomly chosen on the board. If all the surrounding locations are empty then the move is played. If the move is not played (i.e. all the surrounding locations were not empty), a new location is randomly chosen and test. This process is repeated N times. Because of this behaviour, for the poolRave improvement, the probability p is set to 1, i.e. we always play a poolRave move when such a move exists.

The experiments are performed by making the original version of MoGo play against the version with the modification on 9x9 games with 1000 simulations per move.

Algorithm 12 RMCTS(s), including the poolRave modification.

Initialization of $\hat{T}$, n, $\bar{x}$, n^{RAVE}, $\bar{x}^{RAVE}$
while there is some time left **do**
  $s' \leftarrow s$
  Initialization of *game*, *simulation*
  //DESCENT//
  while s' in $\hat{T}$ and s' not terminal **do**
    $s' \leftarrow \arg\max_{j \in C^{s'}} [\bar{x}_j + \alpha \mathbf{\bar{x}^{RAVE}_{s',j}} + \sqrt{\frac{2ln(n_{s'})}{n_j}}]$
    $game \leftarrow game + s'$
  $S = s'$
  //EVALUATION//
  //beginning of the poolRave modification //
  $s'' \leftarrow$ last visited node in the tree with at least 50 simulations
  while s' is not terminal **do**
    if Random $< p$ **then**
      $s' \leftarrow$one of the k moves with best RAVE value in s''
        /* this move is randomly and uniformly selected */
    else
      $s' \leftarrow mc(s')$
    $simulation \leftarrow simulation + s'$
  //end of the poolRave modification //
  //without poolRave, just $s' \leftarrow mc(s')$//
  $r = result(s')$
  //GROWTH//
  $\hat{T} \leftarrow \hat{T} + S$
  for each s in *game* **do**
    $n_s \leftarrow n_s + 1$
    $\bar{x}_s \leftarrow \frac{(\bar{x}_s * (n_s - 1) + r)}{n_s}$
    for each s' in *simulation* **do**
      $\mathbf{n^{RAVE}_{s,s'} \leftarrow n^{RAVE}_{s,s'} + 1}$
      $\mathbf{\bar{x}^{RAVE}_{s,s'} \leftarrow \frac{\bar{x}^{RAVE}_{s,s'} * (n^{RAVE}_{s,s'} - 1) + r}{n^{RAVE}_{s,s'}}}$

# of simulations	p	Size of the pool	Success rate against the baseline
1000	1/2	5	52.7±0.62%
1000	1/4	10	53.19±0.68%
1000	1/2	**10**	**54.32±0.46%**
1000	3/4	10	53.34±0.85%
1000	1	10	52.42±0.70%
1000	1/4	20	52.13±0.55%
1000	1/2	20	52.51±0.54%
1000	3/4	20	52.9±0.34%
1000	1	20	53.2±0.8%
10000	1/2	10	54.45±0.75%
20000	1/2	10	54.42±0.89%

Table 6.4: Success rate of the poolRave modification for the game of Havannah. The baseline is the same program without the poolRave modification.

Size of the pool	Success rate against the baseline
5	54.2±1.7%
10	58.7±0.6%
20	**62.7±0.9%**
30	**62.7±1.4%**
60	59.1±1.8%

Table 6.5: Success rate of the poolRave modification for the game of Go. The baseline is the code without the poolRave modification. For this experiment, no expert-knowledge is used in the Monte-Carlo part.

In the case of the game of Go, we obtain up to $51.7 \pm 0.5\%$ of victory. The improvement is statistically significant but not very important. The reason is that Monte-Carlo simulations in our program MoGo possess extensive domain knowledge thanks to patterns. In order to measure the effect of our modification on application where no knowledge is available, we run more experiments with a version of MoGo without pattern. The results are presented on table 6.5.

When the size of the pool is too large or not large enough, the modification is not as efficient. When using the good compromise for the size (20 in the case of MoGo for 9x9 go), we obtain $62.7 \pm 0.9\%$ of victory.

It is also interesting to note that when we increase the number of simulations par move, we obtain slightly better results. For example, with 10000 simulations per move, we obtain $64.4 \pm 0.4\%$ of victory.

We presented a first generic way of improving the Monte-Carlo simulations in the Monte-Carlo Tree Search algorithm. This method is based on already existing values (the RAVE values) and is easy to implement.

We show two different applications where this improvement was successful: the game of Havannah and the game of Go. On the game of Havannah, we achieve 54.3% of victory against the baseline. On the game of Go, we achieve 51.7% of victory against the version without modification. Having a significant improvement in this case is a very good result, because in the program MoGo the default policy (with expert knowledge) is hard to improve. Moreover, without the domain specific knowledge, we obtain up to 62.7% of victory.

6.2.2 Contextual Monte-Carlo

The results of this section have been published in the paper [Rimmel and Teytaud, 2010]. In Section 6.2.1 we used statistics per move. In this section, we want to learn more complex patterns by using statistical information about pairs of moves. The idea is somehow similar to the last good reply improvement, presented in Section 5.2.2, except that for the Contextual Monte-Carlo improvement we keep statistics for each pair of moves and not only the last good answer. Let us define a tile $L_c(a_1, a_2)$ as the set of simulations where actions a_1 and a_2 have been played by player c. We can compute the empirical reward $V_c(a_1, a_2)$ for $L_c(a_1, a_2)$ based on previous simulations. We are interested in tiles which have a high reward (greater than a threshold B). If the last move of the player c is b,then with a probability p, we choose the action a which maximizes $V_c(a, b)$ if this move is still available. The idea is simply that if we notice that we can often reach a good situation after two actions, then we want to force the choice of the second one if the first one has already been played. The role of the probability p here is the same as it was in Section 6.2.1, which is to keep diversity between simulations. We call this improvement Contextual Monte-Carlo (CMC), the resulting algorithm is presented in Algorithm 13. The CMC modification is presented in Algorithm 13, line 14.

We have tested the effect of contextual Monte Carlo simulations on the game of Havannah. We experiment our Havannah program with the CMC improvement against the same player without this improvement. The experiments are done with 1000 simulations per move for each player. Results are shown in Figures 6.1 and 6.2. We study the impact of the two parameters

Algorithm 13 MCTS(s) with Contextual Monte-Carlo.

1: Initialization of $\hat{T}$, $\hat{x}$, n, $\hat{x}^{CMC}$, n^{CMC}
2: **while** there is some time left **do**
3: $\quad s' \leftarrow s$
4: $\quad$ Initialization of *game*, *simulation*
5: $\quad$ *//DESCENT//*
6: $\quad$ **while** s' in $\hat{T}$ and s' not terminal **do**
7: $\quad\quad s' \leftarrow \arg\max_{j \in C^{s'}} [\bar{x}_j + \sqrt{\frac{2ln(n_{s'})}{n_j}}]$
8: $\quad\quad game \leftarrow game + s'$
9: $\quad S = s'$
10: $\quad$ *//EVALUATION//*
11: $\quad$ //beginning of the CMC modification //
12: $\quad$ **while** s' is not terminal **do**
13: $\quad\quad$ **if** Random $< p$ **then**
14: $\quad\quad\quad s \leftarrow \texttt{play}(s, \arg\max_{a \in E_s} S(\hat{x}^{CMC}(a,b)))$
15: $\quad\quad$ **else**
16: $\quad\quad\quad s' \leftarrow mc(s')$
17: $\quad\quad simulation \leftarrow simulation + s'$
18: $\quad$ //end of the CMC modification //
19: $\quad r = result(s')$
20: $\quad$ *//GROWTH//*
21: $\quad \hat{T} \leftarrow \hat{T} + S$
22: $\quad$ **for** each s in *game* **do**
23: $\quad\quad n_s \leftarrow n_s + 1$
24: $\quad\quad \bar{x_s} \leftarrow \frac{(\bar{x_s} * (n_s - 1) + r)}{n_s}$
25: $\quad$ **for** each $(P(a_1), P(a_2))$ in s', P being one player **do**
26: $\quad\quad n^{CMC}(a_1, a_2) \leftarrow n^{CMC}(a_1, a_2) + 1$
27: $\quad\quad \hat{x}^{CMC}(a_1, a_2) \leftarrow (1 - \frac{1}{n_{CMC}(a_1,a_2)})\hat{x}^{CMC}(a_1, a_2) + \frac{r}{n_{CMC}(a_1,a_2)}$

$prob$ in 6.2 and B in 6.1.

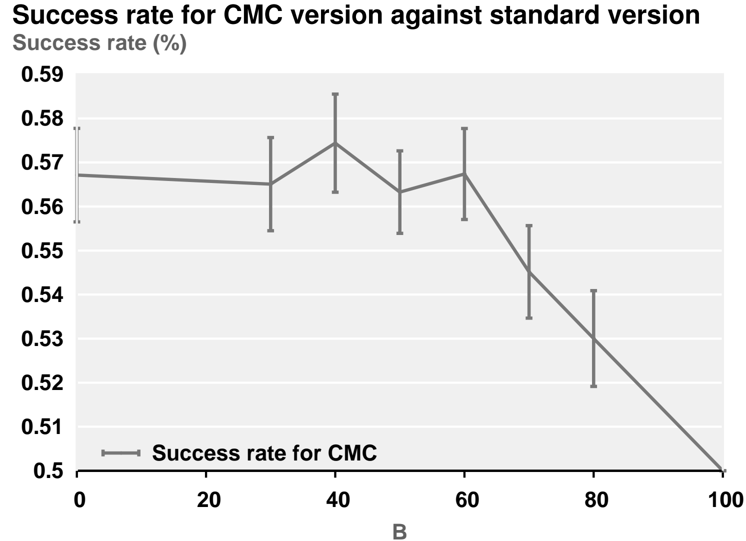

Figure 6.1: Winning percentage for the CMC version against the basic version as a function of B with $prob = 35\%$.

On Figure 6.1 we show the effect of changing B for a fixed value of $prob$ (75%). The winning percentage starts at 56% for $B = 0$ and is stable until $B = 60$ where it starts going down to 50% for $B = 100$. When B is too high, CMC is used less often and therefore the results are worse. When B is low, it means that we select the best tile even if all the possible tiles have a bad average reward. It seems that this is never worst than playing randomly. In the following, we use $B = 0$.

On Figure 6.2 we modify the value of $prob$ while keeping B fixed (0%). When $prob$ is too high, the diversity of the Monte Carlo simulations is not preserved and the results are worse. On the other hand, if $prob$ is too low, the modification has not enough effect. There is a compromise between this two properties.

First, we see that the utilization of CMC is efficient. It leads to 57% of victory against the base version for $prob = 35$ and $B = 0$.

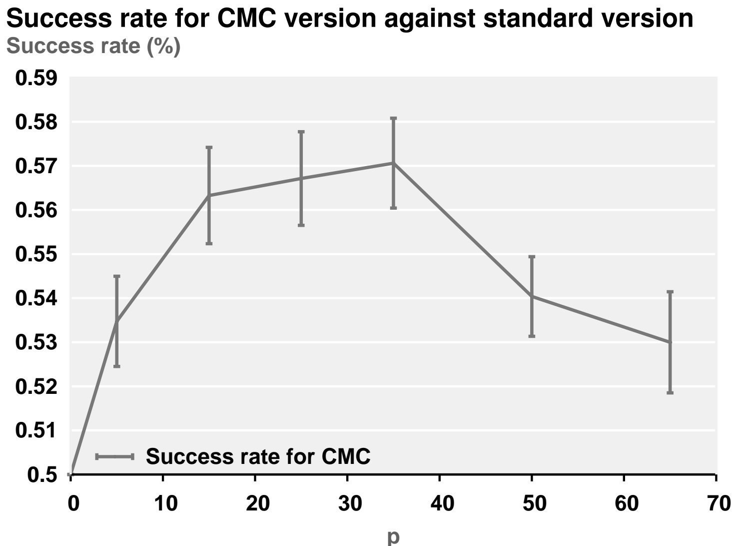

Figure 6.2: Winning percentage for the CMC version against the basic version with $B = 0$.

6.2.3 Decisive Moves

We here present the results published in the paper [Teytaud and Teytaud, 2010c]. For this modification, we use moves which conclude the game. We call such moves "decisive moves" but notion of "check mate in 1 move" can be found in literature. A decisive move is a winning move, a move which directly leads to a win. An anti-decisive move is a move which avoid a decisive move by the opponent one step later. On the one hand this improvement is generic because we do not need any knowledge on the application, we just have to know that this kind of moves exists, but on the other hand, it is true that this kind of moves can only be applied on application on which such moves exist. For instance, for the game of Go, there is no winning move so this modification can not be used.

We first discuss the computational cost of decisive moves, and in the second part experiment such moves on the game of Havannah.

Complexity Analysis.

Our complexity analysis probably holds for several connection games. However, we restrict here to the game of Havannah simply because we experiment this modification on the game of Havannah.

We present the data structures needed for our complexity analysis.

- For each location l, we keep as information in the state d for time step t the following $d(l)$:
  - the color of the stone on this location, if any;
  - if there is a stone, a group number; connected stones have the same group number;
  - the time steps $u \leq t$ at which this information $d(l)$ has changed; we see below why this list has size $O(\log(T))$; the group information and the connections for all stones in the neighborhood of l are kept in memory for each of these time steps.
- For each group, we maintain:
  - the list of edges/corners to which this group is connected (in Hex, corners are useless, and only some edges are necessary), and the timestep at which it was connected;
  - the number of stones in the group;
  - the location of one stone in this group.
- At each move, all the above information is updated. The $O(T \log(T))$ overall complexity in the update is due to the following: when k groups are connected, then the local information should be changed for the $k-1$ smallest groups and not for the biggest group (see Figure 6.3). This implies that each local information is updated at most $O(\log(T))$ times because the size of the group, in case of local update, is at least multiplied by 2.

Checking a win in $O(1)$ is easy by checking connections of groups modified by the current move (for fork and bridge) and by checking local information for cycles:

- a win by fork occurs if the new group is connected to 3 edges;
- a win by bridge occurs if the new group is connected to 2 corners;
- a win by cycle occurs when a stone connects two stones of the same group, at least under some local conditions which are fast to check locally.

Under conditions above, we can:

- initialize the state ($O(T)$);

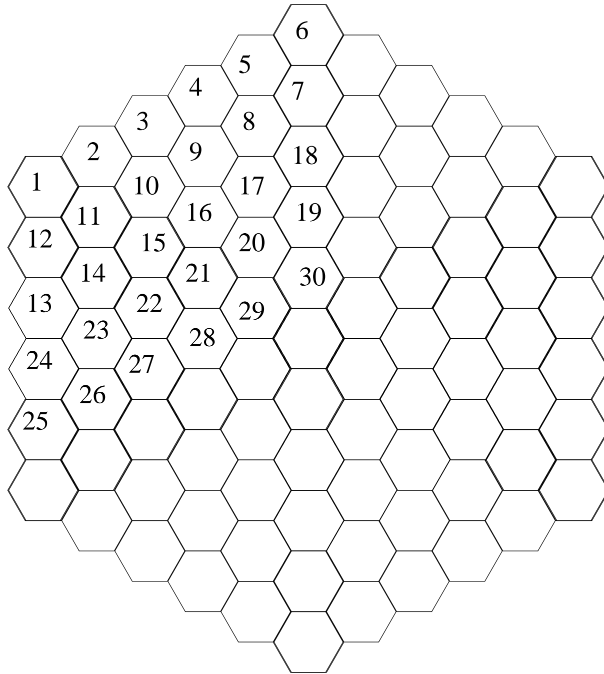

Figure 6.3: An example of sequence of $\Theta(T)$ moves for one of the players whose cost highly depends on the detailed implementation. The important feature of this sequence of moves are (i) it can be extended to any size of board (ii) the size of the group increases by 1 at each move of the player. In this case, if the biggest group has its information updated at each new connection, then the total cost is $\Theta(T^2)$; whereas if the smallest group is updated, the total cost is $\Theta(T)$ for this sequence of moves, and $\Theta(T \log(T))$ in all cases. Randomly choosing between modifying the 1-stone group and the big group has the same expected cost $\Theta(T^2)$ as the choice of always modifying the big group (up to a factor of 2).

- T times,
  - randomly choose a move (cumulated cost $O(T)$);
  - update the state (cumulated cost $O(T \log(T))$);
  - check if this is a win (cumulated cost $O(T)$ - exit the loop in this case).

therefore we can perform one random simulation in time $O(T \log(T))$. The strength of this data structure is that we can switch to decisive moves with no additional cost (up to a constant factor). This is performed as follows:

- initialize the state ($O(T)$);

- T times,
  - randomly choose a move (cumulated cost $O(T)$);
  - update the state (cumulated cost $O(T\log(T))$);
  - check if this is a win (cumulated cost $O(T)$) (exit the loop in this case).
- let $firstWin$ =time step at which the above game was won ($+\infty$ in case of draw).
- let $winner$ =the player who has won.
- for each time location l, $(O(T)$ times)
  - for each time step t (there are at most $O(\log(T))$ such time steps by assumption on the data structure) at which $d_t.s(l)$ has changed, $(O(\log(T))$ times)
    * check if $d_t.s(l)$ was legal and a win for the player p to play at time step t; $(O(1))$
    * if yes, if $t < firstWin$ then $winner = p$ and $firstWin = t$; $(O(1))$
    * check if $d_{t+1}.s(l)$ was legal and a win for the player p to play at time step $t+1$; $(O(1))$
    * if yes, if $t+1 < firstWin$ then $winner = p$ and $firstWin = t+1$. $(O(1))$

The overall cost is $O(T\log(T))$. We point out the following elements:

- The algorithm above consists in playing a complete game with the default policy, and then, check if it was possible to win earlier for one of the players. This is sufficient for the proof, but maybe it is much faster (at least from a constant) to check this during the simulation.
- We do not prove that it's not possible to reach $T\log*(T)$ with decisive moves in Hex or Havannah; just, we have not found better than $T\log(T)$.

Experiments.

Let us introduce the notion of anti-decisive moves. If the opponent p' has a winning move m', then the player p has to play it. Using decisive moves plus antidecisive moves is simple: if the player p is to play, if m is a winning move,

then p plays m and wins, else, if the opponent p' has a winning move m' then p has to play m' to avoid a loss. We perform experiments on Havannah. Please note that we do not implemented the complete data structure above, but some simpler tools which are slower but have the advantage of covering anti-decisive moves as well. We have no proof of complexity for our implementation and no proof that the $T \log(T)$ can be reached for anti-decisive moves.

We implement the decisive moves and anti-decisive moves in our Havannah program for measuring the corresponding improvement. We can see in Table 6.6 that adding decisive moves can lead to big improvements; the modification scales well in the sense that it becomes more and more effective as the number of simulations per move increases.

Number of simulations	100	250	500	1000
DM vs BL	98.6% $\pm$1.8%	99.1% $\pm$1.1%	97.8% 1.6%	95.9% $\pm$1.5%
DM + ADM vs BL	80.1% $\pm$1.2%	81.3% 2%	82.4 $\pm$1.7%	85% $\pm$1.4%
DM + ADM vs DM	49.3% $\pm$1.5%	56.1% $\pm$1.9%	66.6% $\pm$1.9%	78.1% $\pm$1.1%

Table 6.6: Success rates of decisive moves. BL is the baseline (no decisive moves). DM corresponds to BL plus the "decisive moves" (if there exists a winning move then it is played). DM + ADM, is the DM version of our bot, plus the "antidecisive moves" improvement: in that case, if player p is to play, if p has a winning move m then p plays m; else, if the opponent has a winning move m', then p plays m'.

Unfortunately, this improvement becomes less efficient when the number of simulations becomes really large. For instance, if we use 3 seconds per move, our program with decisive moves and antidecisive moves, is very good (80% of success rate) but when we have 30 seconds per move (approximately 1 000 000 of simulations) the success rate falls to a value close to 50%.

6.3 Tuning of the Nested Algorithm

In this section, we present the use of evolutionary algorithms for tuning the Nested Monte-Carlo algorithm for solving the traveling salesman problem with time windows.

The results of this section have been published in [Rimmel et al., 2011a].

The traveling salesman problem is a difficult optimization problem and is used as a benchmark for several optimization algorithms. In this section we tackle the problem of optimizing the Traveling Salesman Problem with Time Windows (TSPTW). For solving this problem, we combine a nested Monte-Carlo algorithm [Cazenave, 2009] and an evolutionary algorithm. With this system, the important point is that we have to optimize a function which is noisy and where the evaluation is not the score on average but the best score among a certain number of runs. When the noise is uniform on the whole function, optimizing for the mean or for the min is equivalent, so we focus on problems where the noise is non uniform. We show on an artificial problem that having a fitness function during the optimization that is different from the one we want to optimize can improve the convergence rate. We then use this principle to optimize the parameters of a nested Monte-Carlo algorithm for the TSPTW.

We have chosen to use Evolution-Strategies (ES [Rechenberg, 1973]) for the optimization part. These algorithms are known to be simple and robust. See [Rechenberg, 1973, Beyer, 2001] or Section I for more details on ES in general.

For this work, we use $(\mu/\mu, \lambda)$-ES. We have chosen the Self-Adaptation Evolution Strategy (SA-ES) for the optimization of our problem. See Section 1.2.2 for more details about the Self-Adaptation Evolution Strategy.

We first do experiment on an artificial problem. We have chosen to optimize the noisy sphere function.

The noisy sphere is a classical artificial problem for optimization experiments [Jebalia and Auger, 2008, Jebalia et al., 2010]. However, here, the noise function is original. The noise is Gaussian and non uniform. We use 5 dimensions.

Formally, the noisy sphere function used in our experiment is defined as the following:

$$f(y) = \sum_{i=1}^{N} (y_i^2 + N(0, (2y_i)^2)).$$

This noisy sphere has been designed such that the optimum according to the mean is different from the optimum according to the mean plus variance. It is represented on the top-left of figure 6.4.

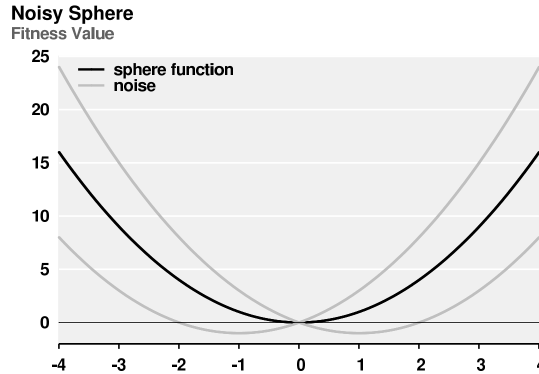

Figure 6.4: Representation of our noisy sphere function.

The evaluation function $eval(f(y))$ is the min over 1000 runs of f with parameters y. We optimize the expectation of this function:

$$eval(f(y)) = \min(f^i(y), i = 1..1000),$$

f^i being the i-th run of f.

During the optimization, we use a fitness function to evaluate an individual. Usually, people use the same function for the fitness function and for the evaluation function. However, we see that the function that we want to optimize is very unstable. For this reason, we propose to use a fitness function that can be different from the evaluation function.

The 3 fitness functions that we use are:

- $best_n(f(y)) = \min(f^i(y), i = 1..n)$
- $mean_n(f(y)) = \frac{\sum_{i=1}^{n} f^i(y)}{n}$
- $kbest_{k,n}(f(y)) = \frac{\sum_{i=1}^{k} f^i(y)}{k}$ with $f^1(y) < f^2(y) < ... < f^k(y) < ... < f^n(y)$

As the fitness function used during the optimization is not the same as the evaluation function, we compute for each generation the score of the best individual according to the evaluation function. This function is noisy, so the **true score** of an individual is the average over $NbEval$ runs of the evaluation function. This is very costly in number of evaluations but this true score is only used to show that the algorithm converges and is only used for the noisy sphere.

In the experiments, we used $k = 5$ for $kbest$ and $NbEval = 100$.

We compare $best_n$, $mean_n$ and $kbest_{5,n}$ for different values of n.

We compute the true score of the best individual in function of the number of the generation. Every curve is the average over 30 runs.

The results are given on Figure 6.5.

We see that in all cases, the convergence is slower with *best* than with *mean* and *kbest*. However, the final value is always better for *best*. This is because *best* is the fitness function the most similar to the evaluation function.

For high values of n, the convergence is equivalent for *kbest* and *mean*. Furthermore, the final value is better for *kbest* than for *mean*. This implies that for high value of n, it is always better to use *kbest* instead of *mean*.

For small values of n, *kbest* converges more slowly than *mean* but achieves a better final value.

As a conclusion, the choice of the fitness function should depend on the need of the user. If the speed of the convergence is important, one can use *mean* or *kbest* depending on n. If the final value is important, *best* is the function to use.

We now see if the conclusions we obtained on an artificial problem are still valid when we optimize difficult problems.

We now focus on the TSPTW problem. First, we describe the problem. Then, we present the nested Monte-Carlo algorithm. Finally, we show the results we obtain when we optimize the parameters of the algorithm on TSPTW.

The traveling salesman problem is an important logistic problem. It is used to represent the problem of finding an efficient route to visit a certain number of customers, starting and finishing at a depot. The version with time windows adds the difficulty that each customer has to be visited within a given period of time. The goal is to minimize the length of the travel. TSPTW is an NP-hard problem and even finding a feasible solution is NP-complete [Savelsbergh, 1985]. Early works [Christofides et al., 1981, Baker, 1983] were based on branch-and-bound. Later, Dumas et al. used a method based on Dynamic programming [Dumas et al., 1995]. More recently, methods based on constraints programming have been proposed [Pesant et al., 1998, Focacci et al., 2002].

Algorithms based on heuristics have also been considered [Solomon, 1987, Gendreau et al., 1998].

Finally, [López-Ibáñez and Blum, 2010] provides a comprehensive survey of the most efficient methods to solve the TSPTW and proposes a new algorithm based on ant colonies that achieves very good results. They provide a clear environment to compare algorithms on a set of problems that we used in this work.

The TSP can be described as follow. Let G be an undirected complete graph. $G = (N, A)$ where $N = 0, 1, ..., n$ is a set of nodes and $A = N * N$ is the set of edges between the nodes. The node 0 represents the depot. The n other nodes represent customers. A cost function $c : A \rightarrow \mathbb{R}$ is given.

For instance, it represents the distance between 2 customers. A solution to this problem is a sequence of nodes $P = (p_0, p_1, ..., p_n)$ where $p_0 = 0$ and $(p_1, ..., p_n)$ is a permutation of $[1, N]$. Set P_{n+1}=0, then, the goal is to minimize the function

$$cost(P) = \sum_{k=0}^{n} c(a_{p_k, p_{k+1}}). \tag{6.4}$$

In the version with time windows, each customer i is associated with a time interval $[e_i, l_i]$. The customer must not be served before e_i or after l_i. It is allowed to arrive at a node i before e_i but the departure time becomes e_i.

Let d_{p_k} be the departure time from node p_k. Then $d_{p_k} = \max(r_{p_k}, e_{p_k})$ where r_{p_k} is the arrival time at node p_k.

The function to minimize is 6.4 but a set of constraints must now be satisfied. Let $\Omega(P)$ be the number of windows constraints violated by tour P. The optimization of f must be done while satisfying the following constraints

$$\forall p_k, r_{p_k} < l_{p_k},$$

and

$$r_{p_{k+1}} = \max(r_{p_k}, e_{p_k}) + c(a_{p_k, p_{k+1}}).$$

With the addition of the constraints, the problem becomes much more complicated and classical algorithms used for TSP are not efficient any more. That is why we use Nested Monte-Carlo that has been presented in Section 5.3.

It is possible to improve the performance of NMC by modifying the Monte-Carlo simulations. An efficient way is to select actions based on heuristics instead of a uniform distribution. However, some randomness must be kept in order to preserve the diversity of the simulations.

To do that, we use a Boltzmann softmax policy. This policy is defined by the probability $\pi_\theta(p, a)$ of choosing the action a in a position p:

$$\pi_\theta(p, a) = \frac{e^{\phi(p,a)^T \theta}}{\sum_b e^{\phi(p,b)^T \theta}},$$

where $\phi(p, a)$ is a vector of features and θ is a vector of feature weights.

The features we use are the heuristics described in [Solomon, 1987]:

- the distance to the last node: $h1(p, a) = c(d, a)$
- the amount of time necessary to wait if a is selected because of the beginning of its time window: $h2(p, a) = \max(0, e_a - (T_p + c(d, a)))$

- the amount of time left until the end of the time window of a if a is selected: $h3(p,a) = \max(0, l_a - (T_p + c(d,a)))$

where d is the last node selected in position p, T_p is the time used to arrive in situation p, e_a is the beginning of the time window for action a, l_a is the end of the time window for the action a and $c(d,a)$ is the travel cost between d and a.

The values of the heuristic are normalized before being used.

The values that we will optimize are the values from the vector θ (the feature weights).

We use the set of problems given in [Potvin and Bengio, 1996].

As we have 3 different heuristics, the dimension of the optimization problem is 3.

We define $NMC(y)$, the function that associates a set of parameters y to the permutation obtained by a run of the NMC algorithm, with parameters y on a particular problem.

The NMC algorithm can generate permutations with some windows constraints violated; the score $Tcost(p)$ of a permutation p is the penalized travel cost.

$$Tcost(p) = cost(p) + 10^6 * \Omega(p),$$

$cost(p)$ is the cost of the travel p and $\Omega(p)$ the number of violated constraints.

10^6 is a constant high enough for the algorithm to first optimize $\Omega(p)$ and then $cost(p)$.

The exact equation of the function f that will be use is the following:

$$f(y) = Tcost(NMC(y)).$$

As the end of the evaluation, we want to obtain a NMC algorithm that we will launch for a longer period of time in order to obtain one good score on a problem. So the evaluation function should be the min on this period of time. As this period of time is not known and a large period of time would be too time-consuming, we arbitrarily choose a time of 1 second to estimate the true score of an individual.

The evaluation function $eval(f(y))$ is the min over r runs of f with parameters y. r being the amount of runs that can be done in 1 second. It means that we want to optimize the expectation of this function:

$$eval(f(y)) = \min_{1s}(f(y)).$$

As for the sphere problem, we use 3 different fitness functions instead of the evaluation function: $mean_n$, $kbest_n$ and $best_n$. In the experiments, we use $n = 100$.

We use a nested algorithm of level 2.

The first experiments are done on the problem rc206.3 which contains 25 nodes.

In this experiment we compare $best_{100}$, $kbest_{100}$ and $mean_{100}$. As in all this work, the population size λ is equal to 12 and the selected population size μ is 3, and $\sigma = 1$. The initial parameters are $[1, 1, 1]$ and the stopping criterion of the evolution-strategy is 15 iterations. Results are the average of three independent runs.

Iterations	BEST	KBEST	MEAN
1	2.7574e+06	2.4007e+06	2.3674e+06
2	5.7322e+04	3.8398e+05	1.9397e+05
3	7.2796e004	1.6397e+05	618.22
4	5.7274e+04	612.60	606.68
5	2.4393e+05	601.15	604.10
6	598.76	596.02	602.96
7	599.65	596.19	603.69
8	598.26	594.81	600.79
9	596.98	591.64	602.54
10	595.13	590.30	600.14
11	590.62	591.38	600.68
12	593.43	589.87	599.63
13	594.88	590.47	599.24
14	590.60	589.54	597.58
15	589.07	590.07	599.73

Table 6.7: Evolution of the true score on the problem rc206.3.

There are many differences between the initial parameters and optimized parameters in term of performances. This shows that optimizing the parameters is really important in order to obtain good performance.

Results are similar as in the case of our noisy sphere function. $best_{100}$ reaches the best score, but converges slowly. $mean_{100}$ has the fastest convergence, but finds the worst final score. As expected, $kbest_{100}$ is a compromise between the two previous fitness, with a nice convergence speed and is able to find a score really close to the best. For this reason, we have chosen to use $kbest$ for the other problems.

We launched the optimization algorithm on all problems from the set in [Potvin and Bengio, 1996]. We compare the best score we obtained on each problem with our algorithm and the current best known score from the literature. The results are presented in table 6.8. We provide the Relative Percentage Deviation (RPD): $100 * (value - bestknown)/bestknown$.

There are many differences between one set of parameters optimized on

one problem and one set of parameters optimized on an other problem. So, the optimization has to be done on each problem.

We perform as good as the state of the art for all problems with less than 29 nodes. We find at least one correct solution for each problem. When the number of nodes increases, this is not a trivial task. For problems more difficult with a higher number of nodes, we don't do as well as the best score. However, we still manage to find a solution close to the current best one and did this with little domain knowledge.

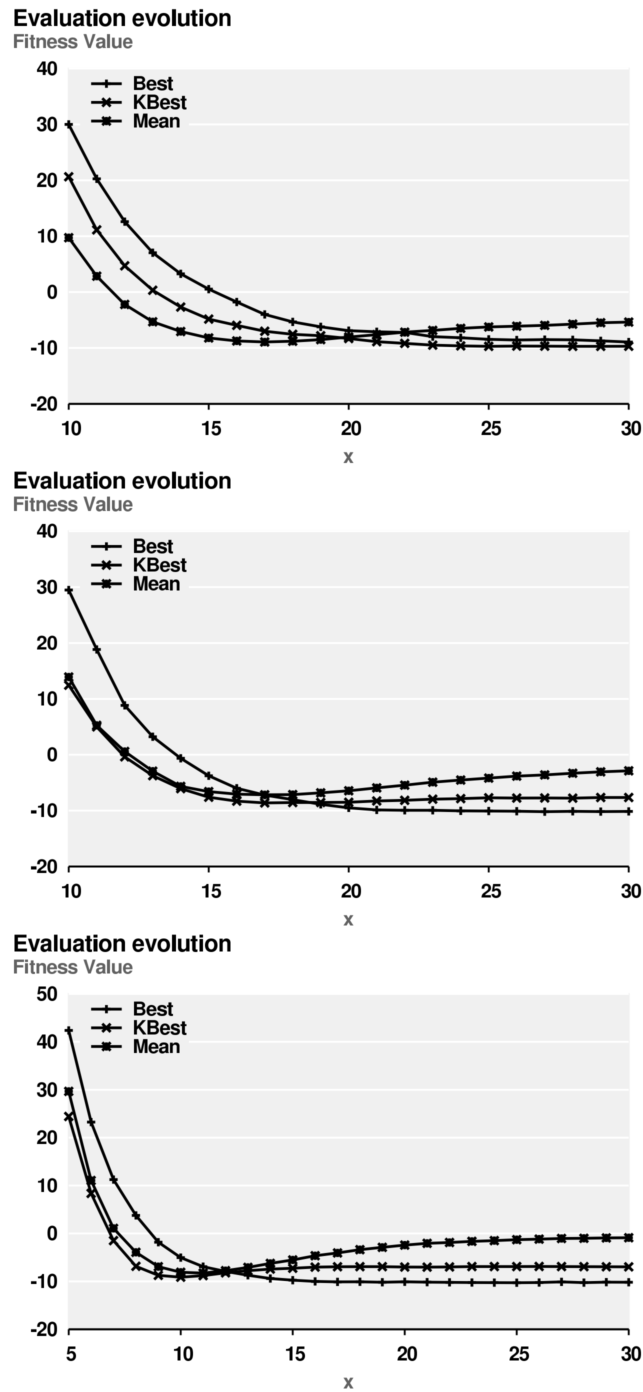

Figure 6.5: Evolution of the true score as a function of the iterations with $n = 10$ (Top), $n = 100$ (Middle) and $n = 300$ (Bottom).

Problem	n	State of the art	NMC score	RPD
rc206.1	4	117.85	**117.85**	0
rc207.4	6	119.64	**119.64**	0
rc202.2	14	304.14	**304.14**	0
rc205.1	14	343.21	**343.21**	0
rc203.4	15	314.29	**314.29**	0
rc203.1	19	453.48	**453.48**	0
rc201.1	20	444.54	**444.54**	0
rc204.3	24	455.03	**455.03**	0
rc206.3	25	574.42	**574.42**	0
rc201.2	26	711.54	**711.54**	0
rc201.4	26	793.64	**793.64**	0
rc205.2	27	755.93	**755.93**	0
rc202.4	28	793.03	**793.03**	0
rc205.4	28	760.47	**760.47**	0
rc202.3	29	837.72	**837.72**	0
rc208.2	29	533.78	536.04	0.42
rc207.2	31	701.25	707.74	0.93
rc201.3	32	790.61	**790.61**	0
rc204.2	33	662.16	675.33	1.99
rc202.1	33	771.78	776.47	0.61
rc203.2	33	784.16	**784.16**	0
rc207.3	33	682.40	687.58	0.76
rc207.1	34	732.68	743.29	1.45
rc205.3	35	825.06	828.27	0.39
rc208.3	36	634.44	641.17	1.06
rc203.3	37	817.53	837.72	2.47
rc206.2	37	828.06	839.18	1.34
rc206.4	38	831.67	859.07	3.29
rc208.1	38	789.25	797.89	1.09
rc204.1	46	868.76	899.79	3.57

Table 6.8: Results on all problems from the set from Potvin and Bengio [Potvin and Bengio, 1996]. First Column is the problem, second column the number of nodes, third column the best score found in [López-Ibáñez and Blum, 2010], fourth column the best score found by the NMC algorithm and fifth column it the RPD. The problems where we find the best solutions are in bold. We can see that for almost all problems with our simple algorithm we can find the best score.

6.4 Conclusion

MCTS are now well established as state of the art algorithms for observable problems with high dimensions. In Section 5.2, we have presented different works done on these algorithms. In Section 6.1, we have successfully introduced MCTS algorithms to the game of Havannah. We can clearly validate, in the case of the game of Havannah, the efficiency of some well known techniques, showing the generality of the MCTS approach. Essentially:

- The efficiency of Bernstein's formula, in front of Hoeffding's formula, is clear (up to 65%). Unfortunately, this efficiency reduces when the number of simulations becomes larger.

- The constant success rate of UCT with $2k$ simulations per move, against UCT with k simulations per move, nearly holds in the case of Havannah (nearly 75%, whereas it is usually around 63% for MCTS in the game of Go [Gelly et al., 2008]). However, this ratio becomes smaller when the number of simulations becomes larger. A large number of simulations reduces the variance but not the bias, that is why this ratio decrease to 50% with large number of simulations. This is a limitation of the parallelization of MCTS algorithms.

- The efficiency of the RAVE heuristic is clearly validated. The main strength is that the efficiency increases with the size, reaching 100 % on 1347 games in size 8. On the other hand, RAVE becomes less efficient, and requires tuning, when the number of simulations per move increases.

- Progressive widening, in spite of the fact that it was shown in [Coulom, 2007, Wang et al., 2008] that it works even without heuristic, is not significant for us. In the case of Go, progressive widening was shown very efficient in implementations based on patterns [Coulom, 2007, Chaslot et al., 2007].

In Section 6.2, we present three different techniques in order to improve the default policy of MCTS algorithms in a generic way. We strongly believe that the next step in improving the MCTS algorithm will be reached by finding an efficient way of modifying the Monte-Carlo simulations depending on the context.
The first method is called "poolRave" and is presented in Section 6.2.1. This method is based on already existing values (the RAVE values) and is easy to implement.
In order to show the generality of this improvement, we tried it on two

different applications: the game of Havannah and the game of Go. On the game of Havannah, we achieve 54.3% of victory against the version without the modification. On the game of Go, we achieve only 51.7% of victory against the version without modification. However, on the version of our Go program without domain specific knowledge, we obtain up to 62.7% of victory. So we can say that the integration of this method in MCTS is successful.

The second method is called "Contextual Monte-Carlo" and is presented in Section 6.2.2.
This improvement is based on the modification of the default policy by using a reward function learned on a tiling of the space of MC simulations. It achieves very good results for the game of Havannah with a winning percentage of 57% against the version without CMC. It is, for the moment, tested only in the case of one example of two-player game. An immediate perspective of this work is to experiment CMC on other problems. It is possible to try a different choice of tiling. We proposed a successful specific one but others can surely be found as we used only a small part of the information contained in this space. We really believe that this kind of improvement is the next step in order to solve specific problems in the game of Go. If we look at semeai (Section 4.2 and Figure 4.2), the problems for solving it are several. First, we have to detect all the locations involved in the semeai, and second, to play in these locations in the right order. Trying to solve this problem by simply increasing the number of simulations is not reasonable (because the number of simulations needed is too large), then we think that the CMC improvement could be really helpful if we increase the number of actions per tile.

The last generic improvement is presented in Section 6.2.3 and is called "decisive moves". This is a generic modification in the sense that it does not use any expert-knowledge but only a feature of some problems. Unfortunately, this improvement can not be used on all problems, but only on those where a "check mate in one" action is available. We have shown that (i) decisive moves have a small computational overhead (ii) they provide a big improvement in efficiency.

Anti-decisive moves might have a bigger overhead, but they are nonetheless very efficient as well even with fixed time per move. A main lesson, consistent with previous works in Go, is that having simulations with a better scaling as a function of the computational power, is usually a good idea whenever these simulations are more expensive.

The main limitation of this modification is that it becomes less efficient when the number of simulations becomes (really) large.

Improvement	# of simulations	Best score against the baseline
poolRave	1000	54.32±0.46%
	10000	54.45±0.75%
	20000	54.42±0.89%
CMC	1000	57%
DM	1000	95.9±1.5%
DM+ADM	1000	85±1.4%

Table 6.9: Summary of the improvements of the default policy on the game of Havannah. poolRave is the modification presented in Section 6.2.1, CMC is the Contextual Monte-Carlo, presented in Section 6.2.2. DM stands for Decisive Moves, and ADM is Anti-Decisive Moves. DM and ADM have been presented in section 6.2.3.

Extending decisive moves to moves which provide a sure win within M moves, or establishing that this is definitely too expensive, would be an interesting further work. We have just shown that for $M = 1$, this is not so expensive if we have a relevant data structure (the cost of a simulation of length O(T) on a board of size O(T) is O(Tlog(T)) for many connection games).

Decisive moves naturally lead to proved situations, in which the result of the game is known and fixed; it would be natural to modify the UCB formula in order to reduce the uncertainty term (to 0 possibly) when all children moves are proved, and to propagate the information up in the tree. To the best of our knowledge, there's no work around this in the published literature and this might extend UCT to cases in which perfect play can be reached by exact solving.

A summary of the efficiency of these methods can be found in Table 6.9

In Section 6.3 we present the tuning of the Nested Monte-Carlo algorithm. In this work we used a new method for solving the TSPTW problem based on the optimization of a nested Monte-Carlo algorithm with the Self-Adaptation algorithm. This algorithm is known for its robustness. The only adaptation of the nested algorithm to the TSPTW was to add 3 heuristics. Even in this case, for all the problems with less than 29 nodes, we were able to reach state of the art solutions with small computation times. However, a clear limitation of our algorithm is to deal with a large number of nodes. A solution could be to prune some moves at the higher level of the nested algorithm. Another solution could be to add new heuristics. In this case, because of the higher dimensionality, we will try other evolution algorithms

and increase the population size. A natural choice is the Covariance Matrix Self-Adaptation algorithm [Beyer and Sendhoff, 2008], known to be robust and good for large population sizes. Adding covariance and allowing large population sizes should increase the speed of the convergence.

Conclusion

This thesis covers two important fields of optimization. The first one is the study of Evolution Strategies in a parallel context. The second one concerns the improvement of multistage optimization algorithms.

Regarding the study of Evolution Strategies in a parallel context, we have first demonstrated that the implementation of the current Evolution Strategies is not adapted to parallelization with a large number of nodes. In order to solve this issue, we proposed several rules. We hope to have asked the good questions. However, a lot of works is still necessary in order to well understand and adapt current algorithms to the special case of large population sizes. A summary of our results can be found in Chapter 3. It should be noted that, during this work, we only looked at unimodal functions. A possible approach for multimodal optimization is the use of some restart strategy [Auger and Hansen, 2005]. This technique can be extended to the parallel case, by simply doing the restarts (and then the searches) in parallel, as suggested in [Schoenauer et al., 2011]. There is no doubt that parallelization is an important part of the future in optimization.

In the second part of this thesis, we have worked on Monte-Carlo Tree Search algorithms. The generality of these techniques is now well established, and they are not only used in games but also in planning [Xie et al., 2011], active learning [Rolet et al., 2009], feature selection [Gaudel and Sebag, 2009], or within industrial applications, like the automatic generation of libraries for linear transform [De Mesmay et al., 2009]. They become more and more visible in these fields, because they are efficient in front of the dimensionality and they do not require any expert knowledge.
Adding expert knowledge is nevertheless possible, and can improve the results of the algorithm. However, in the present work, we focused on adding generic improvements, in order to keep the generality of MCTS algorithms. A summary of the proposed improvements can be found in Section 6.4.
Many interesting works remain to be done, for instance adapting these methods to partially observable applications. Another issue is that, in order to be able to apply these methods, a decision model is needed, and an interesting further development will be to adapt these methods to model-free applications. Works on performing Monte-Carlo simulations without any model have been recently published [Fonteneau et al., 2011], and extending this to MCTS methods is a very important, but difficult, task.

Bibliography

[lit, 1936] (1936). Literary digest.

[Akiyama et al., 2010] Akiyama, H., Komiya, K., and Kotani, Y. (2010). Nested Monte-Carlo Search with AMAF Heuristic. In *2010 International Conference on Technologies and Applications of Artificial Intelligence*, pages 172–176. IEEE.

[Andersen, 1979] Andersen, K. (1979). *The creation of a Democratic majority, 1928-1936*. University of Chicago Press.

[Arnold, 2006] Arnold, D. (2006). Weighted multirecombination evolution strategies. *Theoretical computer science*, 361(1):18–37.

[Arnold and Van Wart, 2008] Arnold, D. and Van Wart, D. (2008). Cumulative step length adaptation for evolution strategies using negative recombination weights. *Applications of Evolutionary Computing*, pages 545–554.

[Audibert et al., 2008] Audibert, J., Munos, R., and Szepesvári, C. (2008). Variance estimates and exploration function in multi-armed bandit. *Research report*, pages 07–31.

[Audibert et al., 2006] Audibert, J.-Y., Munos, R., and Szepesvari, C. (2006). Use of variance estimation in the multi-armed bandit problem. In *NIPS 2006 Workshop on On-line Trading of Exploration and Exploitation*.

[Auer et al., 2002] Auer, P., Cesa-Bianchi, N., and Fischer, P. (2002). Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2/3):235–256.

[Auger and Hansen, 2005] Auger, A. and Hansen, N. (2005). A restart cma evolution strategy with increasing population size. In *Proceedings of the IEEE Congress on Evolutionary Computation*, volume 2, pages 1769–1776, Piscataway, NJ, USA. IEEE Press.

[Auger et al., 2005] Auger, A., Jebalia, M., and Teytaud, O. (2005). XSE: quasi-random mutations for evolution strategies. In *Proceedings of Evolutionary Algorithms*, page 12.

[Baker, 1983] Baker, E. (1983). An exact algorithm for the time-constrained traveling salesman problem. *Operations Research*, 31(5):938–945.

[Beyer, 2001] Beyer, H. (2001). *The theory of evolution strategies*. Springer Verlag.

[Beyer and Sendhoff, 2008] Beyer, H. and Sendhoff, B. (2008). Covariance Matrix Adaptation Revisited–The CMSA Evolution Strategy. *Parallel Problem Solving from Nature–PPSN X*, pages 123–132.

[Bishop, 1995] Bishop, C. (1995). *Neural networks for pattern recognition*. Oxford university press.

[Bosman and Thierens, 2000] Bosman, P. and Thierens, D. (2000). Expanding from discrete to continuous estimation of distribution algorithms: The id\ mathbbe e a. In *Parallel Problem Solving from Nature PPSN VI*, pages 767–776. Springer.

[Brooks, 1958] Brooks, S. (1958). A discussion of random methods for seeking maxima. *Operations Research*, 6(2):244–251.

[Broyden, 1970] Broyden, C. (1970). The convergence of a class of double-rank minimization algorithms 1. general considerations. *IMA Journal of Applied Mathematics*, 6(1):76.

[Bruegmann, 1993] Bruegmann, B. (1993). Monte-carlo Go (unpublished draft http://www.althofer.de/bruegmann-montecarlogo.pdf).

[Calder and Reinman, 2000] Calder, B. and Reinman, G. (2000). A comparative survey of load speculation architectures. *Journal of Instruction-Level Parallelism*, 2:1–39.

[Cazenave, 2009] Cazenave, T. (2009). Nested Monte-Carlo search. In *IJCAI*, pages 456–461.

[Chaslot et al., 2010] Chaslot, G., Fiter, C., Hoock, J., Rimmel, A., and Teytaud, O. (2010). Adding expert knowledge and exploration in Monte-Carlo Tree Search. *Advances in Computer Games*, pages 1–13.

[Chaslot et al., 2009] Chaslot, G., Hoock, J., Teytaud, F., and Teytaud, O. (2009). On the huge benefit of quasi-random mutations for multimodal optimization with application to grid-based tuning of neurocontrollers.

[Chaslot et al., 2007] Chaslot, G., Winands, M., Uiterwijk, J., van den Herik, H., and Bouzy, B. (2007). Progressive strategies for monte-carlo tree search. In Wang, P. et al., editors, *Proceedings of the 10th Joint Conference on Information Sciences (JCIS 2007)*, pages 655–661. World Scientific Publishing Co. Pte. Ltd.

[Chen, 2008] Chen, S. (2008). Nonparametric estimation of expected shortfall. *Journal of financial econometrics*, 6(1):87.

[Christofides et al., 1981] Christofides, N., Mingozzi, A., and Toth, P. (1981). State-space relaxation procedures for the computation of bounds to routing problems. *Networks*, 11(2):145–164.

[Coulom, 2006] Coulom, R. (2006). Efficient selectivity and backup operators in monte-carlo tree search. In *Proceedings of the 5th international conference on Computers and games*, pages 72–83. Springer-Verlag.

[Coulom, 2007] Coulom, R. (2007). Computing elo ratings of move patterns in the game of go. *ICGA Journal*, 30(4):198–208.

[De Mesmay et al., 2009] De Mesmay, F., Rimmel, A., Voronenko, Y., and P
"uschel, M. (2009). Bandit-based optimization on graphs with application to library performance tuning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 729–736. ACM.

[Devroye et al., 1996] Devroye, L., Gy
"orfi, L., and Lugosi, G. (1996). *A probabilistic theory of pattern recognition*. Springer Verlag.

[Dong and Yao, 2008] Dong, W. and Yao, X. (2008). Unified eigen analysis on multivariate gaussian based estimation of distribution algorithms. *Information Sciences*, 178(15):3000–3023.

[Drake, 2009] Drake, P. (2009). The Last-Good-Reply Policy for Monte-Carlo Go. *ICGA Journal*, 32(4):221–227.

[Dumas et al., 1995] Dumas, Y., Desrosiers, J., Gelinas, E., and Solomon, M. (1995). An optimal algorithm for the traveling salesman problem with time windows. *Operations Research*, 43(2):367–371.

[Efron and Tibshirani, 1993] Efron, B. and Tibshirani, R. (1993). *An introduction to the bootstrap*, volume 57. Chapman & Hall/CRC.

[Fletcher, 1970] Fletcher, R. (1970). A new approach to variable metric algorithms. *The Computer Journal*, 13(3):317.

[Focacci et al., 2002] Focacci, F., Lodi, A., and Milano, M. (2002). A hybrid exact algorithm for the tsptw. *INFORMS Journal on Computing*, 14(4):403–417.

[Fonteneau et al., 2011] Fonteneau, R., Murphy, S., Wehenkel, L., and Ernst, D. (2011). Estimation monte carlo sans modèle de politiques de décision. *Revue d'Intelligence Artificielle*, 25.

[Fournier and Teytaud, 2011] Fournier, H. and Teytaud, O. (2011). Lower Bounds for Comparison Based Evolution Strategies using VC-dimension and Sign Patterns. *Algorithmica*, pages 1–22.

[Freeling, 2003] Freeling, C. (2003). Introducing Havannah. In *Abstract Games*, volume 14, page 14.

[Fukumizu and Amari, 2000] Fukumizu, K. and Amari, S. (2000). Local minima and plateaus in hierarchical structures of multilayer perceptrons. *Neural Networks*, 13(3):317–327.

[Galil and Italiano, 1991] Galil, Z. and Italiano, G. (1991). Data structures and algorithms for disjoint set union problems. *ACM Computing Surveys (CSUR)*, 23(3):319–344.

[Gardner et al., 2011] Gardner, M., McNabb, A., and Seppi, K. (2011). Speculative evaluation in particle swarm optimization. *Parallel Problem Solving from Nature–PPSN XI*, pages 61–70.

[Gaudel and Sebag, 2009] Gaudel, R. and Sebag, M. (2009). Feature selection as a one-player game. In *Proceedings of the second NIPS Workshop on Optimization for Machine Learning, OPT*. Citeseer.

[Gelly et al., 2008] Gelly, S., Hoock, J. B., Rimmel, A., Teytaud, O., and Kalemkarian, Y. (2008). The parallelization of monte-carlo planning. In *Proceedings of the International Conference on Informatics in Control, Automation and Robotics (ICINCO 2008)*, pages 198–203.

[Gelly and Silver, 2007] Gelly, S. and Silver, D. (2007). Combining online and offline knowledge in UCT. In *ICML '07: Proceedings of the 24th*

international conference on Machine learning, pages 273–280, New York, NY, USA. ACM Press.

[Gendreau et al., 1998] Gendreau, M., Hertz, A., Laporte, G., and Stan, M. (1998). A generalized insertion heuristic for the traveling salesman problem with time windows. *Operations Research*, 46(3):330–335.

[Goldfarb, 1970] Goldfarb, D. (1970). A family of variable metric methods derived by variational means. *Mathematics of Computation*, 24(109):23–26.

[Grahl et al., 2006] Grahl, J., Bosman, P., and Rothlauf, F. (2006). The correlation-triggered adaptive variance scaling idea. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 397–404. ACM.

[Hansen, 1998] Hansen, N. (1998). Verallgemeinerte individuelle Schrittweitenregelung in der Evolutionsstrategie. *Mensch & Buch Verlag, Berlin.*

[Hansen et al., 2009] Hansen, N., Auger, A., Finck, S., and Ros, R. (2009). Real-parameter black-box optimization benchmarking 2009: Experimental setup. Technical report, Citeseer.

[Hansen and Ostermeier, 1996] Hansen, N. and Ostermeier, A. (1996). Adapting arbitrary normal mutation distributions in evolution strategies: The covariance matrix adaptation. In *Evolutionary Computation, 1996., Proceedings of IEEE International Conference on*, pages 312–317. IEEE.

[Hansen and Ostermeier, 2001] Hansen, N. and Ostermeier, A. (2001). Completely derandomized self-adaptation in evolution strategies. *Evolutionary computation*, 9(2):159–195.

[Jebalia and Auger, 2008] Jebalia, M. and Auger, A. (2008). On multiplicative noise models for stochastic search. In et a.l., G. R., editor, *Conference on Parallel Problem Solving from Nature (PPSN X)*, volume 5199, pages 52–61, Berlin, Heidelberg. Springer Verlag.

[Jebalia and Auger, 2011] Jebalia, M. and Auger, A. (2011). Log-linear convergence of the scale-invariant (μ/μ w, λ)-es and optimal μ for intermediate recombination for large population sizes. *Parallel Problem Solving from Nature–PPSN XI*, pages 52–62.

[Jebalia et al., 2010] Jebalia, M., Auger, A., and Hansen, N. (2010). Log-linear convergence and divergence of the scale-invariant (1+1)-es in noisy environments. *Algorithmica*, pages 1–36. online first.

[Kennedy and Eberhart, 1995] Kennedy, J. and Eberhart, R. (1995). Particle swarm optimization. In *Neural Networks, 1995. Proceedings., IEEE International Conference on*, volume 4, pages 1942–1948. IEEE.

[Kimura and Matsumura, 2005] Kimura, S. and Matsumura, K. (2005). Genetic algorithms using low-discrepancy sequences. In *Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1341–1346. ACM.

[Kloetzer et al., 2007] Kloetzer, J., Iida, H., and Bouzy, B. (2007). The Monte-Carlo Approach in Amazons. In *Proceedings of the Computer Games Workshop*, pages 185–192. Citeseer.

[Knuth and Moore, 1975] Knuth, D. E. and Moore, R. W. (1975). An analysis of alpha-beta pruning* 1. *Artificial intelligence*, 6(4):293–326.

[Kocsis and Szepesvari, 2006] Kocsis, L. and Szepesvari, C. (2006). Bandit-based monte-carlo planning. In *European Conference on Machine Learning 2006*, pages 282–293.

[Korpaas et al., 2003] Korpaas, M., Holen, A., and Hildrum, R. (2003). Operation and sizing of energy storage for wind power plants in a market system. *International Journal of Electrical Power & Energy Systems*, 25(8):599–606.

[Lai and Robbins, 1985] Lai, T. and Robbins, H. (1985). Asymptotically efficient adaptive allocation rules. *Advances in Applied Mathematics*, 6:4–22.

[Larranaga and Lozano, 2002] Larranaga, P. and Lozano, J. (2002). *Estimation of distribution algorithms: A new tool for evolutionary computation*, volume 2. Springer Netherlands.

[Lee et al., 2009] Lee, C., Wang, M., Chaslot, G., Hoock, J., Rimmel, A., Teytaud, O., Tsai, S., Hsu, S., and Hong, T. (2009). The computational intelligence of MoGo revealed in Taiwan's computer Go tournaments. *Computational Intelligence and AI in Games, IEEE Transactions on*, 1(1):73–89.

[Liu and Teng, 2008] Liu, J. and Teng, H. (2008). Model learning and variance control in continuous edas using pca. In *Innovative Computing Information and Control, 2008. ICICIC'08. 3rd International Conference on*, pages 555–555. IEEE.

[López-Ibáñez and Blum, 2010] López-Ibáñez, M. and Blum, C. (2010). Beam-ACO for the travelling salesman problem with time windows. *Computers & OR*, 37(9):1570–1583.

[Lorentz, 2011] Lorentz, R. (2011). Improving Monte–Carlo Tree Search in Havannah. *Computers and Games*, pages 105–115.

[Lunacek et al., 2008] Lunacek, M., Whitley, D., and Sutton, A. (2008). The impact of global structure on search. *Parallel Problem Solving from Nature–PPSN X*, pages 498–507.

[L'Ecuyer and Lemieux, 2005] L'Ecuyer, P. and Lemieux, C. (2005). Recent advances in randomized quasi-monte carlo methods. *Modeling uncertainty*, pages 419–474.

[Mnih et al., 2008] Mnih, V., Szepesvári, C., and Audibert, J. (2008). Empirical bernstein stopping. In *Proceedings of the 25th international conference on Machine learning*, pages 672–679. ACM.

[Mühlenbein and Paass, 1996] Mühlenbein, H. and Paass, G. (1996). From recombination of genes to the estimation of distributions I. Binary parameters. *Parallel Problem Solving from Nature—PPSN IV*, pages 178–187.

[Müller, 2002] Müller, M. (2002). Computer go. *Artificial Intelligence*, 134(1-2):145–179.

[Niederreiter, 1992] Niederreiter, H. (1992). Quasi-monte carlo methods.

[Owen, 2003] Owen, A. (2003). Quasi-monte carlo sampling. *Monte Carlo Ray Tracing: Siggraph*, pages 69–88.

[Pelikan et al., 2002] Pelikan, M., Goldberg, D., and Lobo, F. (2002). A survey of optimization by building and using probabilistic models. *Computational optimization and applications*, 21(1):5–20.

[Pesant et al., 1998] Pesant, G., Gendreau, M., Potvin, J., and Rousseau, J. (1998). An exact constraint logic programming algorithm for the traveling salesman problem with time windows. *Transportation Science*, 32(1):12–29.

[Posik, 2008] Posik, P. (2008). Preventing premature convergence in a simple eda via global step size setting. *Parallel Problem Solving from Nature–PPSN X*, 5199:549–558.

[Potvin and Bengio, 1996] Potvin, J. and Bengio, S. (1996). The vehicle routing problem with time windows part II: genetic search. *INFORMS journal on Computing*, 8(2):165.

[Price, 1996] Price, K. (1996). Differential evolution: a fast and simple numerical optimizer. In *Fuzzy Information Processing Society, 1996. NAFIPS. 1996 Biennial Conference of the North American*, pages 524–527. IEEE.

[Price et al., 2005] Price, K., Storn, R., and Lampinen, J. (2005). *Differential evolution: a practical approach to global optimization*. Springer Verlag.

[Quante et al., 2009] Quante, R., Fleischmann, M., and Meyr, H. (2009). A stochastic dynamic programming approach to revenue management in a make-to-stock production system. *ERIM Report Series Reference No. ERS-2009-015-LIS*.

[Rechenberg, 1971] Rechenberg, I. (1971). *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution. Dr.-Ing.* PhD thesis, Thesis, Technical University of Berlin, Department of Process Engineering.

[Rechenberg, 1973] Rechenberg, I. (1973). Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution, Fromman-Holzboog. *Stuttgart. German*.

[Rimmel, 2009] Rimmel, A. (2009). Thesis: Improvements and Evaluation of the Monte-Carlo Tree Search Algorithm.

[Rimmel and Teytaud, 2010] Rimmel, A. and Teytaud, F. (2010). Multiple Overlapping Tiles for Contextual Monte Carlo Tree Search. *Applications of Evolutionary Computation*, pages 201–210.

[Rimmel et al., 2011a] Rimmel, A., Teytaud, F., and Cazenave, T. (2011a). Optimization of the nested monte-carlo algorithm on the traveling salesman problem with time windows. *Applications of Evolutionary Computation*, pages 501–510.

[Rimmel et al., 2011b] Rimmel, A., Teytaud, F., and Teytaud, O. (2011b). Biasing Monte-Carlo simulations through RAVE values. *Computers and Games*, pages 59–68.

[Rimmel et al., 2010] Rimmel, A., Teytaud, O., Lee, C.-S., Yen, S.-J., Wang, M.-H., and Tsai, S.-R. (2010). Current Frontiers in Computer Go. *IEEE*

Transactions on Computational Intelligence and Artificial Intelligence in Games, page in press.

[Rolet et al., 2009] Rolet, P., Sebag, M., and Teytaud, O. (2009). Optimal active learning through billiards and upper confidence trees in continous domains. In *Proceedings of the ECML conference*, pages 302–317.

[Ros and Hansen, 2008] Ros, R. and Hansen, N. (2008). A simple modification in cma-es achieving linear time and space complexity. *Parallel Problem Solving from Nature–PPSN X*, pages 296–305.

[Savelsbergh, 1985] Savelsbergh, M. (1985). Local search in routing problems with time windows. *Annals of Operations Research*, 4(1):285–305.

[Schoenauer et al., 2011] Schoenauer, M., Teytaud, F., Teytaud, O., et al. (2011). A rigorous runtime analysis for quasi-random restarts and decreasing stepsize.

[Schwefel, 1974] Schwefel, H. (1974). Adaptive Mechanismen in der biologischen Evolution und ihr Einfluß auf die Evolutionsgeschwindigkeit. *Interner Bericht der Arbeitsgruppe Bionik und Evolutionstechnik am Institut f "ur Mess-und Regelungstechnik Re*, 215(3).

[Schwefel, 1981] Schwefel, H. (1981). Numerical optimization of computer models.

[Shanno et al., 1970] Shanno, D. et al. (1970). Conditioning of quasi-newton methods for function minimization. *Mathematics of computation*, 24(111):647–656.

[Shannon, 1950] Shannon, C. E. (1950). XXII. Programming a computer for playing chess. *Philosophical Magazine (Series 7)*, 41(314):256–275.

[Shapiro, 2005] Shapiro, J. (2005). Drift and scaling in estimation of distribution algorithms. *Evolutionary computation*, 13(1):99–123.

[Siu et al., 2001] Siu, T., Nash, G., and Shawwash, Z. (2001). A practical hydro, dynamic unit commitment and loading model. *Power Systems, IEEE Transactions on*, 16(2):301–306.

[Sloan and Wozniakowski, 1998] Sloan, I. and Wozniakowski, H. (1998). When are quasi-monte carlo algorithms efficient for high dimensional integrals? *Journal of Complexity*, 14(1):1–33.

[Sobol, 1979] Sobol, I. (1979). On the systematic search in a hypercube. *SIAM Journal on Numerical Analysis*, 16(5):790–793.

[Solomon, 1987] Solomon, M. (1987). Algorithms for the vehicle routing and scheduling problems with time window constraints. *Operations Research*, 35(2):254–265.

[Storn, 1996] Storn, R. (1996). On the usage of differential evolution for function optimization. In *Fuzzy Information Processing Society, 1996. NAFIPS. 1996 Biennial Conference of the North American*, pages 519–523. IEEE.

[Storn and Price, 1995] Storn, R. and Price, K. (1995). Differential evolution-a simple and efficient adaptive scheme for global optimization over continuous spaces. *INTERNATIONAL COMPUTER SCIENCE INSTITUTE-PUBLICATIONS-TR*.

[Storn and Price, 1997] Storn, R. and Price, K. (1997). Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. *Journal of global optimization*, 11(4):341–359.

[Teytaud, 2010] Teytaud, F. (2010). A new selection ratio for large population sizes. *Applications of Evolutionary Computation*, pages 452–460.

[Teytaud and Teytaud, 2009a] Teytaud, F. and Teytaud, O. (2009a). Bias and variance in continuous eda.

[Teytaud and Teytaud, 2009b] Teytaud, F. and Teytaud, O. (2009b). On the parallel speed-up of estimation of multivariate normal algorithm and evolution strategies. *Applications of Evolutionary Computing*, pages 655–664.

[Teytaud and Teytaud, 2009c] Teytaud, F. and Teytaud, O. (2009c). Why one must use reweighting in estimation of distribution algorithms. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 453–460. ACM.

[Teytaud and Teytaud, 2010a] Teytaud, F. and Teytaud, O. (2010a). Creating an Upper-Confidence-Tree program for Havannah. *Advances in Computer Games*, pages 65–74.

[Teytaud and Teytaud, 2010b] Teytaud, F. and Teytaud, O. (2010b). Log(lambda) modifications for optimal parallelism. In *PPSN (1)*, pages 254–263.

[Teytaud and Teytaud, 2010c] Teytaud, F. and Teytaud, O. (2010c). On the huge benefit of decisive moves in Monte-Carlo Tree Search algorithms. In *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*, pages 359–364. IEEE.

[Teytaud, 2008a] Teytaud, O. (2008a). Conditioning, halting criteria and choosing lambda. In *Artificial Evolution*, pages 196–206. Springer.

[Teytaud, 2008b] Teytaud, O. (2008b). When does quasi-random work? *Parallel Problem Solving from Nature–PPSN X*, pages 325–336.

[Teytaud and Fournier, 2008] Teytaud, O. and Fournier, H. (2008). Lower bounds for evolution strategies using VC-dimension. *Parallel Problem Solving from Nature–PPSN X*, pages 102–111.

[Teytaud and Gelly, 2006] Teytaud, O. and Gelly, S. (2006). General lower bounds for evolutionary algorithms. *Parallel Problem Solving from Nature-PPSN IX*, pages 21–31.

[Teytaud and Gelly, 2007] Teytaud, O. and Gelly, S. (2007). DCMA: yet another derandomization in covariance-matrix-adaptation. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 955–963. ACM.

[Teytaud et al., 2006] Teytaud, O., Gelly, S., and Mary, J. (2006). On the ultimate convergence rates for isotropic algorithms and the best choices among various forms of isotropy. *Parallel Problem Solving from Nature-PPSN IX*, pages 32–41.

[Tromp and Farnebäck, 2007] Tromp, J. and Farnebäck, G. (2007). Combinatorics of go. *Computers and Games*, pages 84–99.

[Van der Vaart and Wellner, 1996] Van der Vaart, A. and Wellner, J. (1996). *Weak convergence and empirical processes*. Springer Verlag.

[Vandewoestyne and Cools, 2006] Vandewoestyne, B. and Cools, R. (2006). Good permutations for deterministic scrambled halton sequences in terms of l2-discrepancy. *Journal of computational and applied mathematics*, 189(1-2):341–361.

[Wang et al., 2008] Wang, Y., Audibert, J., and Munos, R. (2008). Algorithms for infinitely many-armed bandits. *Advances in Neural Information Processing Systems*, 21.

[Xie et al., 2011] Xie, F., Nakhost, H., and Müller, M. (2011). A local monte carlo tree search approach in deterministic planning.

[Yakowitz et al., 1978] Yakowitz, S., Krimmel, J., and Szidarovszky, F. (1978). Weighted monte carlo integration. *SIAM Journal on Numerical Analysis*, 15(6):1289–1300.

[Zhigljavsky and Zilinskas, 2007] Zhigljavsky, A. and Zilinskas, A. (2007). *Stochastic global optimization*. Springer.