Artificial Intelligence and Optimization with parallelism

Olivier Teytaud

Artificial Intelligence with Parallelism

Olivier Teytaud, habilitation à diriger des recherches

Committee:

- Hans-Georg Beyer (reviewer)
- Louis Wehenkel (reviewer)
- Marc Schoenauer
- Frédérick Garcia
- Dirk Arnold
- Olivier Sigaud (reviewer)

Université Paris-Sud

Defense April 22, 2011

Artificial Intelligence and Optimization with parallelism

Olivier Teytaud

October 2, 2010

Contents

Part I

Introduction

Chapter 1

Introduction: artificial intelligence and optimization with parallelism

This document is devoted to artificial intelligence and optimization. This part will be devoted to having fun with high level ideas and to introduce the subject. Thereafter, Part II will be devoted to Monte-Carlo Tree Search, a recent great tool for sequential decision making; we will only briefly discuss other tools for sequential decision making; the complexity of sequential decision making will be reviewed. Then, part III will discuss optimization, with a particular focus on robust optimization and especially evolutionary optimization. Part IV will present some machine learning tools, useful in everyday life, such as supervised learning and active learning. A conclusion (part V) will come back to fun and to high level ideas.

This document is not intended to be read as a novel - therefore a big index is provided. Several summaries are provided (see "summary" in the index).

Section 1.1 will discuss links between computational intelligence and human intelligence - the main purpose of Artificial Intelligence (AI) is to reduce the gap between both for domains in which the humans are stronger. In section 1.2 we will discuss the evolution of hardware, and its impact on the mentioned gap. Section 1.3 will discuss what is covered by this document. All the document assumes that the reader is familiar with some linear algebra, and understands some notations; we will use $[[a,b]] = \{a, a+1, a+2, \dots, b\}$, $\mathscr{L}(f|E)$ is the conditional distribution of f conditionally to E, $\mathbb{E}_x f$ is the expectation of f for random variable x, $\arg\min_x f$ is any x such that $\forall y, f(x) \leq f(y)$ (when such and x exists), $\limsup f_n = \lim_{n\to\infty} \sup_{m\geq n} f_m$.

1.1 Computational intelligence and human intelligence

This document is devoted to parallel Artificial Intelligence (AI) and parallel optimization. As multi-core machines and clusters or grids are invading computer science (see

Fig. 1.1 and Fig. 1.2), I guess this is not so different from just ”artificial intelligence and optimization”. The idea is just that we will not neglect the fact that networks exist, and that multi-core machines are now the standard in computer science. We do not claim that all artificial intelligence tasks would benefit from parallelization, but we claim that all artificial intelligence tasks should be considered from the point of view of their possible parallelization, so that we know the benefit of parallelism.

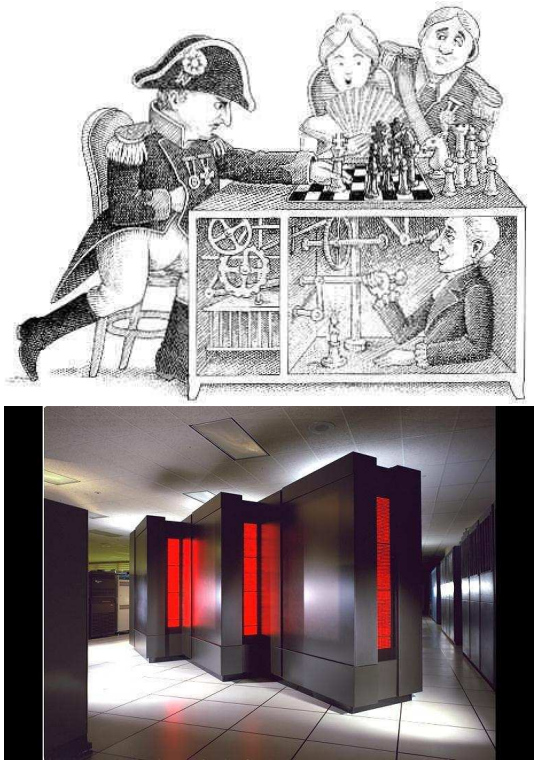


Figure 1.1: Top: using humans is a good solution for efficient artificial intelligence; so-called “GWAP” systems consist in rephrasing boring problems (e.g. annotating databases of images with keywords) into games (e.g. finding a keyword cited by at least one other player connected to the game, and not yet by other people playing the games). Bottom: A beautiful connection machine. From `http://blog.websourcing.fr/files/2009/03/total-number-websites.gif`.

What is artificial intelligence (AI) ? I like the definition I often heard by M. Sebag: AI is research in computer science for tasks which are currently absolutely not solvable by a computer. This means that AI does not consist in improving a convergence rate by 10 %; it is about solving something which was either impossible to solve with current technologies, or about reducing the time requested for a task by a factor of 20. Parallelization is a good tool (not the only one) for this. As pointed out by M. Girard [235], AI often hosts parasites: many AI research papers are somehow strange

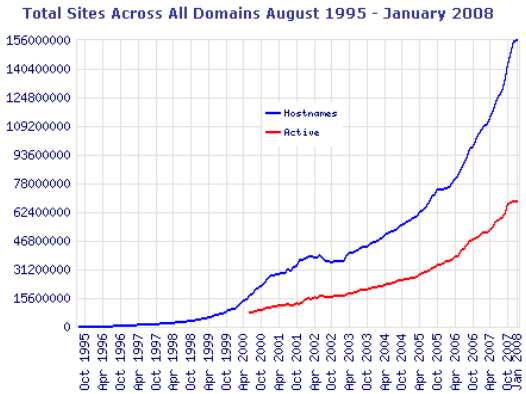


Figure 1.2: Number of hostnames in the Internet (Amit Agarwal/DI/Netcraft, from `netlog.org` and `http://blog.websourcing.fr/files/2009/03/total-number-websites.gif`.

and far from science. Nonetheless, there are notable successes in AI, and AI invades applications and popularization (Fig. 1.3).

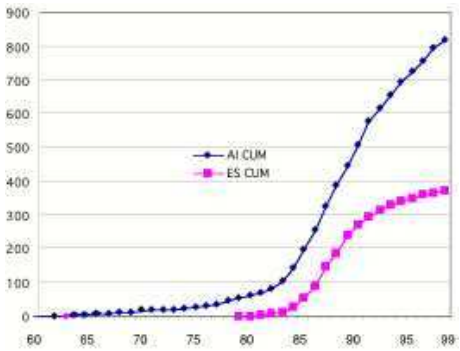


Figure 1.3: The number of books about artificial intelligence and expert systems in libraries (from B.R. Gaines, Knowledge Science and Technology: Operationalizing the Enlightenment, KWAB00).

Are machines more clever than humans, or not ? Obviously, machines are stronger for arithmetic computations. They are also stronger for memorizing quantities of information, and various tasks which are (therefore) excluded from what we consider as intelligence. In particular, people can laugh at Alan Turing, who predicted that in 2000 a machine would be able to pass the Turing test (i.e. fooling an average conversational partner into believing it was a human being). It has been said that it is clear that computers are not able of discussing about Joseph Conrad [235]. Nonetheless, if I want to get reliable information about Joseph Conrad, I'll request "wikipedia Joseph Conrad" on Google, and thanks to Google, its algorithms, Wikipedia, its algorithms, and their

contributors, I'll get much more information about Joseph Conrad than can be given by most humans.

Beyond things which are usually considered as some form of "computation", for which activities are computers better than humans ?

Combinatorial problems ? Computers are strong for combinatorial evaluations; this is illustrated by their ability in establishing a win in chess many moves in advance. They will also solve combinatorial problems better than you, and solve many (but not all) puzzles faster than humans. Go provides a simply defined combinatorial problem which is very difficult for computers (section 5.3).

Probabilities ? Computers are often strong for problems involving probabilities. It has been said [164, 185] that humans are not efficient in the trading between prior knowledge and online knowledge (equivalently, humans don't compute a posteriori probabilities efficiently as a function of prior probability and observations). This can be illustrated by the superiority of computers for cases in which exact computations of probabilities are involved, as in the so-called bookbag and pokerchips problem, or for checking exact risk of errors in DNA or fingerprint identification, but also in cases in which exact computation is impossible, as in e.g. the game of Backgammon. Computers are also strong at exploiting the weakness of human players in the shi-fu-mi game (also known as "RoShamBo", or "Rock-Paper-Scissors"). However, for complex phenomena humans might be stronger than expected; humans still resist to computers in Backgammon and poker [1], and are still stronger for many tasks.

Fast behaviors ? Computers are strong for fast and precise continuous answers: e.g., an autopilot is better than a human for saving up fuel in a plane. Nonetheless, they are less versatile and less robust: "You may be able in a manual mode to sense something sooner than the autopilot can sense it" according to National Transportation Safety Board investigator Steve Chealander, after the crash of a plane which was in autopilot mode in spite of ice built up on the wings. Also, a computer is, by far, not able of receiving a subtle spiny table tennis serve, and computers can drive a car only in very easy environments. Computers are also weaker than children for many tasks in vision (see Fig. 5.14). Finally, they are desperately weak for versatile activities, involving several tasks concurrently, as in a game of Go when several local fights are involved; this author needed a long time for understanding why its favorite algorithms can not play as strong humans and finds this example really deep and interesting (section 5.3.5).

Resistance, low price. Computers are useful also for tasks in which they are weaker than humans. Computers are less "expensive" than humans, accept boring tasks (Fig. 1.4), and therefore we prefer to use them in dangerous situations: demining, radioactivity, or tedious tasks. Also, they have low physiological constraints (need for sleep, reasonable temperatures, food, reasonable pressure, oxygen... all these constraints which make humans obsolete in various environments) and can therefore perform very long flights (for surveying borders[2], looking for survivors of a disaster, protection of nuclear power plants).

Repetitive tasks. Computers are very fast and robust for some simple repetitive

[1] Human professional players won by a small margin in the humans vs machine poker competition organized by the university of Alberta in 2007.

[2] According to Wikipedia, unmanned aerial vehicles made possible a lot of arrests and the seizure of tons of marijuana.

tasks; as a consequence, for finding the best program for a very small task like sorting 7 items, they can test very quickly many possible programs and pick up the best, leading to optimal programs. This is one of the successes of genetic programming. Computers can also handle quickly repetitive and not so stupid tasks: for example, which human is able to recognize a speaker, from a new audio record, within hundreds of audio records ? Computers can also recognize (with limited precision) individuals with a camera in a crowd in an airport. Incidentally, with very good cameras, computers are seemingly now stronger than humans for recognizing individuals (in "nice" environments, however).

Figure 1.4: The humanoid version (left) is more poetic, but the practical version (Trilobit vacuum cleaner on the right) is not so bad (but needs help sometimes). Dish washers are useful too, but an arm, a camera and an electronic brain, with the ability to pick up all the dishes in the kitchen, using the thrash with pertinence and putting relevant objects in the dish washer would be better.

1.2 The evolution of hardware: is parallelism crucial for successful artificial intelligence ?

How did computers (from the point of view of the hardware capabilities) evolve in the recent years ? Below a quick point of view:

- First, memory became crucial. Consider hard drives. They become bigger and bigger, but the disk transfer rate does not increase as fast as the size of the disk (see Fig. 1.6) - therefore, we must be careful when using disks.

- Second, Moore's law, stating the increase of processor speed at a regular rate (doubling each 24 months), is seemingly stopped. Now, increasing the computational power involves improving something else: memory bandwidth, other input/output devices, but also multiplicating the number of computation units (see Fig. 1.5).

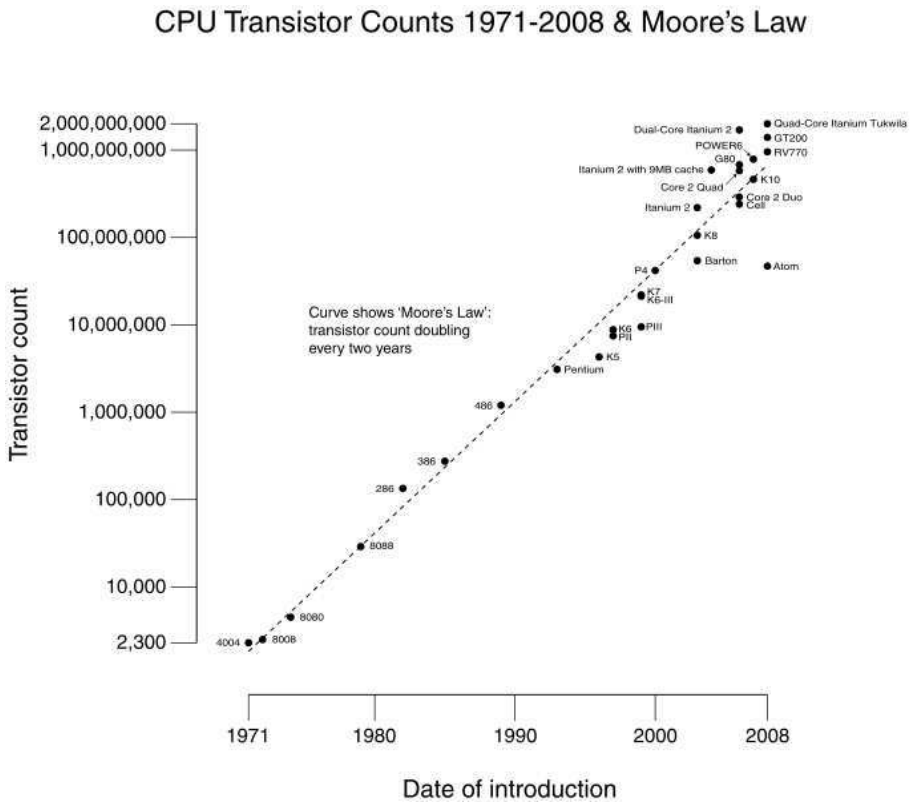


Figure 1.5: Moore's law illustrated by Wikipedia. All the recent improvements are based on multi-core machines.

Parallelism consists in distributing the computation among several computational units. The **load balancing** is the art and the science of distributing the tasks over the computation units, so that all of them are efficiently contributing. Parallelism has the two following immediate advantages:

- The number of instructions per second increases linearly as a function of the number of machines, as well as the transfer to the *local* memory (see Fig. A.1 from [170]).

- The price of a fixed computational power decreases regularly, as shown by funny Figure 1.7 (from [170]).

There are various forms of parallelization:

- Parallelization within the "logical" computational units: then, instructions are decomposed (e.g. thanks to a pipeline within the computation unit). This is not in the scope of this document.

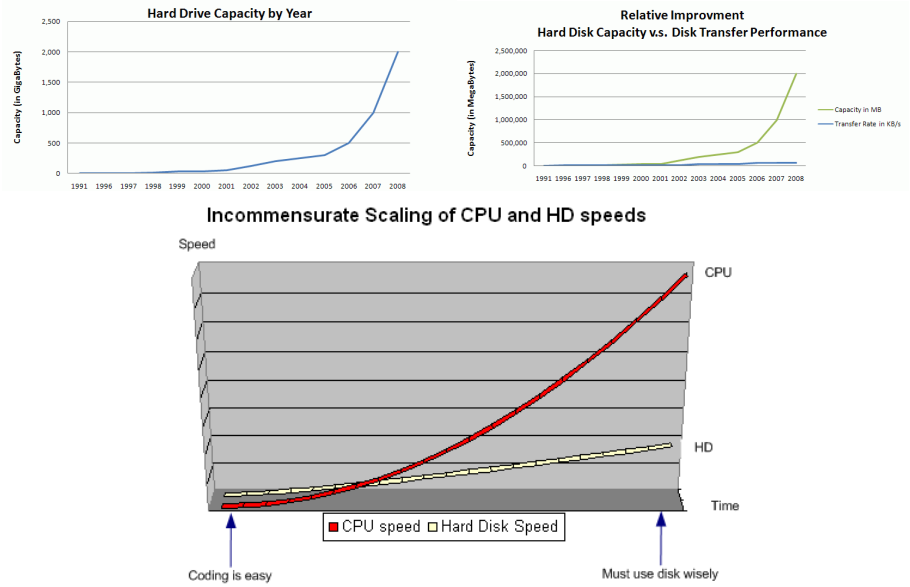


Figure 1.6: Top left: disk capacity as a function of years. Top right: transfer rate as a function of years. Bottom: transfer rate and disk capacity as a function of time (within a small margin for the same years as top graphs). Graphs from UCLA's Computer Science and Engineering (CSE 111), in the Introduction to Operating Systems lectures; and from David Wartell's web article " Why Are Server Backups So Painful? " (from `http://wiki.r1soft.com/pages/viewpage.action?pageId=3016608`).

- **Multi-core machines**: then, several computational units, termed cores, have access to the same memory: the memory is **shared**. Various levels of cache (a cache is an intermediate relatively fast memory, see Fig. 1.8, which keeps copies of the most frequently used parts of the memory) improve the transfer rate (the rate at which information can be sent/received from the memory), which is known as being a main limitation. Multi-core machines are limited by the transfer rate to the memory: having 1024 cores on a same machine is very expensive and not very interesting if you have no advantage over 1024 single cores on separate machines because the memory cannot be shared between the cores. NUMA (non-uniform memory access) is the case in which the access to the memory is slower of faster depending on which core requests an access to which part of the memory, whereas in SMP (symmetric multiprocessing) machines the rate is the same for all cores and all parts of memory. Asymmetric multiprocessing (ASMP) is the case in which the processors have different abilities - programming such a machine requests a careful look at the architecture.

- **Machines connected with a network.** In this case, machines have to explicitly send messages to each other, in order to work together. The computation units

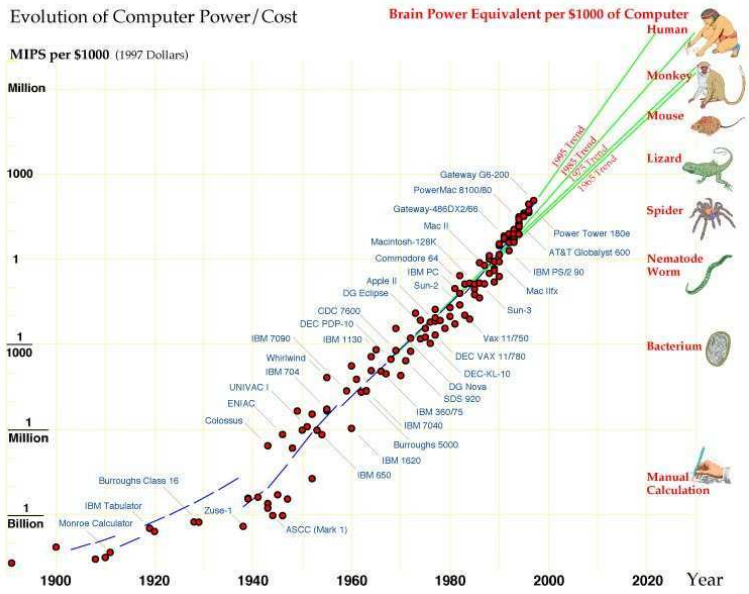


Figure 1.7: The computational power of $1000 of computers, according to H. Moravec (1998).

using the network are also termed nodes: if each of the node is itself parallel (typically a multi-core machine), then the number of computation units might be greater than the number of nodes (for examples, a network connecting 80 quad-core machines has 320 computation units, and 80 nodes when considered as is; if the program uses the cores of each quad-core as if they were not sharing memory, then we have 320 computation units and 320 nodes - the number of nodes is therefore dependent of the algorithmic choices). The performance of the machine strongly depends on the capacity of the network and of the needs of your specific application; an example of extremely slow connection is a set of machines connected with Internet, and the other extremal case is a high-performance network between machines on a cluster. Various types of parallel machines/parallel computations without shared memory are:

– **Clusters.** Then, machines are more or less homogeneous and not too far the ones from the others (the distance is in terms of communication delay). There are clusters with high-performance networks, and clusters with low-performance networks. The load balancing is easier when clusters are made of homogeneous machines. A **Beowulf** cluster is a cluster made of off-the-shelf computers with a standard Ethernet network.

– **Massively parallel machines.** The difference with clusters is not so clear: essentially, we should have then “much more” than 100 computation units (according to wikipedia) and should have a very fast network. Also, the

computation units should be nearly identical.

– **Grids** is the other extremal case of clusters: in a grid there's no fast network between the nodes, the nodes are not supposed to be homogeneous. A Grid can be the union of several clusters, and we might consider that Internet is a very big grid.

Other specific hardwares for parallel machines exist: look for GPGPU (general purpose computing on graphics processing units) or FPGA (field-programmable gate array), vector processors, for more on this.

A program is **sequential** when it is written for a non-parallel architecture. The **speed-up** of a parallel program is the improvement in speed when using a parallel machine instead of a sequential program: it depends on the number of computation units, on the size of the problem, on the characteristics of the parallel machine. The **efficiency** is the speed-up, divided by the number of computation units: it is usually at most 1, except when there are specific memory requirements or resource-related limitations of the sequential program.

Sequential consistency, for a program developed on a parallel machine, is the property (relative to a sequential program) that the result and the computation, within its detailed order, is exactly the same as the result of the sequential version. It is sometimes also requested that the operations are exactly the same as those of the sequential program (but not in the same order, as some operations are simultaneous on the parallel program). Strict sequential consistency is usually not required, and not a good idea: its performance is strongly affected by **Amdahl's law**, i.e. the fact that if p% of the program cannot be parallelized (and p is often quite big in case of sequential consistency) then the speed-up cannot be more than $1/p$ - this is in fact a big limitation when using many computational units, and with our point of view on AI (i.e. AI consists in solving previously unsolvable tasks, and not in moderately fastening existing tasks). **Gustafson's law** is an improved version of Amdahl's law, taking into account the number of computation units. We'll see in section 8 an example of application of Amdahl's law, in which we could get order of magnitudes of speed-up thanks to a non sequentially consistent parallelization. Usually (but not always), sequential consistency is a bad idea.

Computation units connected by a network have usually a lower computational power than multi-core machines *for a given number of computation units* because we loose the ability of sharing the memory: on the other hand, it is technologically possible to have huge clusters (or grids) of computation units, what is usually impossible for multi-core machines. Coding in multi-threading is also often much more tricky than in message-passing, when there are critical sections to handle carefully.

A machine with multiple CPU (central processing unit) is not the same thing as a machine with multiple cores: cores are supposed to be more closely integrated, e.g. on a single integrated circuit die. A **many-core** machine is a machine with too many cores for "easy" natural techniques for multi-core machines. A SMP (symmetric multiprocessing) machine is a machine with multiple CPU; possibly, the SMP technology can be used for multiple cores. A cluster of SMP is a cluster of machines, each of them having a SMP architecture. This is the most classical form of supercomputing. There's

a recent interest in the use of modern videocards as a model of highly-parallel ASMP[3]; using this possibility however implies a strong rewriting of your program and it does not work for all forms of problems.

A problem is said **fine-grained** if its parallelization (or at least its natural parallelization) involves a lot of communication between the computation units. It is said **coarse-grained** if it involves only a few communications per second. It is said **embarrassingly parallel** if it involves very few communications, and it is said **multi-sequential** when it communicates only briefly at the very beginning and the very end of the computation.

Some quantitative ideas:

- Some of the works involved in this document involve machines with 32 cores.
- Some of the works involved in this document with high-performance clusters involved 800 cores.
- Some of the works involved in this document with grids involved 5000 cores (thanks to Grid5000).

A particular form of parallelization, which is essential in some mathematical proofs and also for some real algorithms, is **speculative** parallelization. Consider a program of the form:

> "$x = a()$
> IF $x \in b$ THEN RETURN $c()$ ELSE RETURN $d()$"

where b is a set, $a()$, $c()$ and $d()$ are functions; $c()$ and $d()$ are supposed to be nearly as expensive as a. Then speculative parallelization replaces this by:

> Compute in parallel:
>
> - $x = a()$;
> - $y = c()$;
> - $z = d()$.
>
> IF $x \in b$ THEN RETURN y ELSE RETURN z.

This parallel version takes time $\max(T(a), T(c), T(d))$ where $T(t)$ denotes the computational cost of task t (plus the cost of the IF ... THEN ... ELSE ..., supposed to be small), whereas the sequential version takes time $T(a) + T(b)$ or $T(a) + T(c)$; if $T(b) = T(c) = T(a)$ there is a speed-up 2.

[3]The asymmetry in videocards comes from the fact, in this case, that in such cases, there is a CPU (or possibly several CPUs) *plus* a GPU made of many block of many "cores".

Speculative optimization is in particular interesting when k levels of ”IF ... THEN ... ELSE ...” are nested; then, we can have a speed-up $2k$. The drawback is that we use a number of processors exponential in k; the speed-up is at best logarithmic in the computational power. For large numbers of processors, it is sometimes impossible to do better and speculative parallelization can then provide optimal speed-up (an example in section 15.2). Speculative optimization is sequentially consistent (see definition below).

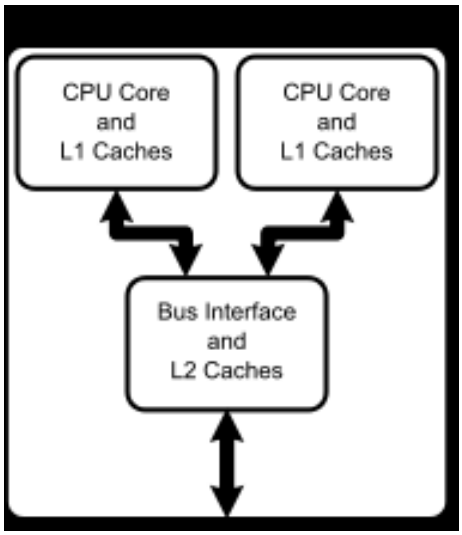


Figure 1.8: A schema of a multi-core machine (from Wikipedia). There are caches local to each core, and possibly caches between these caches and the “real” memory, for accelerating the use of frequently accessed parts of the memory.

Is it worth parallelizing and is it worth highly optimizing the performance of a program for a specific parallel machine ? One can have measurable improvements by taking into account the very details of the architecture of a parallel machine. Nonetheless, from the personal experience of this author, improving the representation of your problem provides very big improvements, then parallelizing it on a multicore machine and/or a cluster or network provides a big improvement, and taking into account the rough nature of your parallel machine (e.g. one layer of parallelization for highly integrated computational units with fast access to a common memory, and one layer for less integrated computational units) provides most of the improvement that can be brought by parallelization; in some cases it was worth separating in three levels (shared memory, clusters, grids). I’ve not seen cases in which taking into account the very details of the architecture (like non-uniform memory access) provides more than +100 % speed-up - and such a speed-up is already very rare. My personal rule consists in optimizing very carefully the representation of the problem, then the algorithm, then choosing its parallel counterpart from an algorithmic point of view, and only in very specific cases the detailed parallelization.

1.3 Problems under analysis

This document is about artificial intelligence and optimization. The problems under consideration in this document are typically the followings:

- First, **sequential decisions with uncertainty** (SDU). In SDU, I have to take a sequence of decisions $d_1, d_2, d_3, \ldots$, and I get rewards or losses. I want as much rewards as possible; which decisions should I take ? This very general problem will be here considered in the easier case in which we have a model of the uncertainties, i.e. a model of the world: this means that if we know all decisions, all random outcomes, and the initial state, then we know the intermediate states and the reward (this is not, by far, a negligible assumption). SDU is also termed **planning**. This is incredibly general, and incredibly difficult in the general case; an easier case if the fully observable one, such as the game of Go; we will also consider, in two separate parts, two special cases of SDU, described below: optimization and active learning. These two cases are so specific that they are usually not considered as SDU; nonetheless they can use the same techniques, as will be discussed later.

  Applications of SDU include:

  - The **game of Go**. Here, you have one decision to take at each move, the uncertainties are the moves of the opponent, and the reward is either a win or a loss (in some rare cases, a draw). This is a particularly convenient testbed. Incidentally, there are millions of players in Asia, and an increasing interest for this game even in Europe and America; if the intellectual property around source codes developed in the French academic world was manageable, then research institutes and universities would have earned a lot of money with the program we developed. Games are the most popular applications of Monte-Carlo Tree Search (MCTS), and many people consider MCTS as a tool for games; we will nonetheless define and study MCTS as a general tool for planning (part II).
  - **Power plant management**. If you have a few tenths of nuclear power plants, a few tenths of hydroelectric plants, and various other power plants, for producing electricity for millions of user, then it is of crucial importance to minimize the cost (economical and ecological cost) of production, whilst satisfying the demand. The uncertainties are meteorological outcomes, technical troubles, economical factors influencing the demand.
  - Other stock management problems have been considered [61].
  - Martin Müller and his colleagues published applications in **classical planning benchmarks**[172].
  - Other games (than the game of Go) in which MCTS performs quite well are Havannah[221], Hex[5], and (with a very interesting trick detailed later, see section 9) Amazons[160]. The case of General Game Playing (GGP) is also very interesting: in this game, the rules are not known in advance; the program is given the rules, in a specified format, just before the game

starts, and then the program must play. In these games, MCTS performed extremely well as soon as the chosen game is somehow difficult.

- Second, **optimization**. In this very important case of SDU, to which many books are dedicated: we are given an oracle, which, given x, computes $f(x)$; we want to find a minimum of f, or an approximation of this minimum. This is **optimization**. In accordance with the fundamental idea of this document that AI should work on problems as moderately structured as possible, we will here only consider non-linear optimization; we will restrict our attention essentially to unconstrained optimization; see part III. We will consider also the noisy case (i.e. when the oracle provides noisy information). Optimization is the most immediately applicable research field in artificial intelligence; almost any industrial process, or even economical policy, or even any program (through parameter tuning) can be improved by non-linear (possibly noisy) optimization.

- Third, **active learning.** In this case, we are given an oracle, which, given x, computes $f(x)$; we want to find f, or an approximation (usually denoted $\hat{f}$) thereof, from examples $(x_1, f(x_1))$, $(x_2, f(x_2))$, $\ldots$, $(x_n, f(x_n))$. This is **active learning**; see part 16. For example, x can be a text in English, and $f(x)$ its translation in french; in this example (**automatic translation**), what is expensive is the request of the exact value $f(x)$ (this implies an expert); in order to save up some of this cost (i.e., in order to save up the number of requests) it is important to take care of choosing examples. Indeed, $f(x)$ can be the result of a big finite-elements computation: typically, $f(x)$ can be a measure of the efficiency of a mechanical piece parametrized by x, and $\hat{f}$ is an approximation of the heavy computation evaluating this efficiency, so that $\hat{f}$ can be used instead of f (this is in particular useful if f is very expensive and $\hat{f}$ is much cheaper); $\hat{f}$ is then termed a **surrogate model**. Active learning can perform very well in some cases (in particular classification with not too big sets of models), and not so much in others (e.g. regression by neural networks or SVM). Don't try to implement active learning without a clear understanding of whether it will provide a big improvement or not: AL makes things more complicated, more difficult to parallelize, and should be used only when it is really relevant.

The main assumption limiting the generality of the approaches in this document is that we always assume that a model of the world is given (in particular in part II). This is not a small restriction, and it strongly restricts the applicability of the approaches here. A work around consists in replacing the knowledge of the model of the world by a worst case analysis on a family of models; this is exactly the classical approach in games. We don't know how the opponent will play, we have no model of it; so we consider the worst case on the possible opponents. This provides a solution for problems in which we don't know the model of the opponent.

Links between biology and artificial intelligence are somehow controversial, but there's always some fun and philosophical insights in these comparisons. After the conclusion, an appendix will therefore discuss links between AI and biological aspects.

Chapter 2

Acknowledgements

I am grateful to all the people with whom I made most of the works in these pages. Hassen Doghmen, Jean-Baptiste Hoock, Sylvain Gelly, Arpad Rimmel, Philippe Rolet, Jérémie Mary, Fabien Teytaud, are the main collaborators I had here; many thanks to you for all the fun, all the stress, all the hopes we had together. After a long and intense day of work, we often naturally keep working in the RER, and after the RER we used Ssh, Gtalk, Email, for keeping working a few hours more at home. There was just something special in that. I was even sometimes reached by the feeling of doing something important and useful; thanks to you I feel that research, science, is something great that must be pushed forward.

I've had plenty of collaborators these last 17 years, since I started studying mathematics and computer science. I am infinitely grateful to all the people with whom I learnt, in particular in my young years. I hope I could transmit some of the knowledge they gave to me to some other people. I also hope that the young people working with me have had as much fun as I had with them, and as much fun as I had with older people who spend some time for teaching me plenty of useful things. Special thanks here to A. Esbelin, the LLAIC, Yuri Matiyasevich and people in the Steklov institute in Saint-Petersburg; and thanks a lot to the reviewers of my earliest papers. I am also grateful to E. Loyer, A. Moreau, C. Finot, E. Piètre, G. Laffite, for the many things I learnt with them.

I have also special thanks to Marc De Crisenoy, who was a great friend when I somehow randomly arrived in Ecole Normale Supérieure, in which I was essentially because I wanted to have a salary young without having a very boring job; Jean-Michel Friedt, whose ideas around science influenced a lot my own ideas on it, and in particular my conviction that solving $P \neq NP$ is less important than curing Aids or Cancer; Gérald Gavin, whose energy for attacking some big and new problems has been great for me; Stéphane Lallich, for all reasons for acknowledging the people acknowledged in this paragraph. I am also grateful to the people who gave me confidence in the strength of science, Jacques, Nicole, and Yves Guiet. Last but not least, I'm very, very, very grateful to Maud, Clement, Simon, Elise, and Capucine for support, fun, opportunity of learning something else than computer science or mathematics.

I am grateful to Artelys (the company in which I was working before I join Inria).

It was much more possible, there, to do various applications, than anywhere else. I really had fun there, and I left Artelys essentially because I wanted to do mathematical investigations, what was not really the job in Artelys. I sometimes feel that a perfect professional happiness would be 50% of my time in academic research (without taking care of ANR and other such stuff) and 50 % to do some applications.

I would not be here without the help of Marc Schoenauer and Michèle Sebag; thanks for considering my candidacy for a research position. This position is just great for doing interesting things, in spite of the poor evolution we see currently in academic research, with low quality people at the head of research institutes or governments, and I like many of the research works performed in the Tao team `http://tao.lri.fr`; if we had less frequent evaluations (there are so many evaluations in Tao that sometimes it is difficult to find time for working between two evaluations), and if Tao was somewhere in which lodgings[1] was less expensive than Ile-De-France, it would just be the perfect place for freely doing important research with a lots of great collaborators.

I have often the feeling of a debt to Anne Auger, who spent time teaching me the basics of evolutionary algorithms, and the less basic things, and who pointed out the main ideas that were proved, that remained to be proved, that were not yet considered. Thank you for this Anne. I also want to thank Nicolas Bredèche for the interesting points of views he provided to me, and in particular a lot of "Yes, but...". Nicolas is a 9 Dan pro of "Yes, but...". Anne, Nico, your help was beyond publication records and H-index.

Many thanks also to Merve Amil; Pierre Audouard; Nicolas Baskiotis; Vincent Berthier; Yann Bonnemay; Guillaume Chaslot; Hervé Fournier; Christian Gagné; Romaric Gaudel, Julien Pérez; Francois-Michel De Rainville; Sylvie Ruette, Nataliya Sokolovska, Atsushi Takahashi, Yizao Wang, Nicolas Omont, Vincent Danjean, Thomas Hérault, Vincent Berthier, Adrien Couëtoux, Yann Kalemkarian. I am grateful also to the staff of the Lri and to Vincent Néri. I probably forgot a lot of fantastic people met during these years - sorry, sorry, sorry - please forgive me!

I am also very grateful to human Go players who accepted to play against our program; in particular Li Yue, Frédéric Donzet, Junfu Dai, Bernard Helmstetter, Motoki Noguchi, who spent time playing against MoGo and who provided interesting feedbacks.

I am extremely grateful to our Taiwanese friends for their impressive motivation for improving MoGoTW and for their energy in organizing events, whenever I was in many cases desperately tired of competitions and demonstrations against humans. Incidentally, opportunities of meeting the Taiwanese culture were just great. I am not grateful to the people who decided, for unknown reasons, that we should not welcome Taiwanese people in our university, but I am grateful to the unknown reasons which, after all, made their venue possible. Unfortunately, it was difficult for me to keep in mind all the names of the people I met in Taiwan, and I'm afraid of forgetting some people; I'll thank Chang-Shing Lee, Mei-Hui Wang, Shi-Jim Yen, M. Tsai, M. Dong, TonTon, M. Hsu, and all others (sorry for being unable to keep in memory so many Asian names simultaneously). I am also grateful to Marc Jégou for starting all the

[1] And in particular lodgings sufficiently big to two adults and four children.

stuff with pro players, Catherine Girard, Maud Oger, Christopher Couzelin, Florent Madelaine, Simon Billouët, Jean-Yves Papazoglou, Eric Saves, Chantal Gajdos, Arnaud Boucherie, Eudeline Arnaud, Mario Nolla, Juan Jesus Ligero, Michael Robson for many things around Go; Michel Boutin for informations around connection games.

Thanks also to Wikipedia for plenty of fruitful information, references, pictures. Thanks to Bob Hearn for very interesting comments around complexity in games through Facebook. Thanks to Linux for being my main and favorite operating system since 1995. Thanks to Grid5000 (`http://www.grid5000.fr`) for facilitating my parallel experiments.

Part II

Monte-Carlo Tree Search for games and planning

Chapter 3

On games and planning and why they are important[1]

Games are important far beyond the field of games. When a new algorithm emerges in games, usually it can be adapted for many other applications. A typical example is a great recent algorithm for artificial intelligence termed Monte-Carlo Tree Search (MCTS [72, 62, 143]). Three (almost) simultaneous papers defined this great, new tool for discrete time planning under uncertainty; it is based on developping an incremental tree in the neighborhood of the current situation, in a more depth-first manner than in alpha-beta[2]. Essentially, planning under uncertainty occurs when:

- at different time steps, you have to make decisions;
- after a given number of time steps, you get a reward (or a loss);
- there are some unknown elements, to be learnt later (e.g. the decision of an opponent, or some random outcomes).

My favorite *test bed* for this is the game of Go. My favorite *application* is power plant management. The MCTS algorithm is surprisingly simple, relevant in both continuous and discrete cases, and needs far less expertise than alpha-beta (for games), and far less simplifying assumptions than dynamic programming. It appeared in games (in particular in the fascinating game of Go, discussed below), but it has been adapted to many other applications as well. Two important families of problems from artificial intelligence, namely expensive optimization and active learning, could be tackled efficiently with this tool. The algorithm (which has essentially no value function) has some limitations. First, it should probably be mixed with other techniques when the number of time steps is huge (or infinite). Second, when there are automatic answers

[1] This part is based on collaborations with J.-Y. Audibert, P. Audouard, G. Chaslot, A. Couëtoux, R. Coulom, V. Danjean, H. Doghmen, S. Gelly, R. Gaudel, T. Hérault, J.-B. Hoock, J. Mary, A. Rimmel, P. Rolet, M. Schoenauer, M. Sebag, A. Takahashi, F. Teytaud, Y. Wang, Z. Yu.

[2] Incidentally, the idea of developping a tree is usual in games, but not in stochastic dynamic programming in which Bellman values are usually approximated once for all; see however [182].

to learn (e.g., for the case of the game of Go, reducing the number of liberties of group B when group A lost one liberty and A and B are in a liberty race), then MCTS needs some ”patches”. This has been handcrafted in the case of the game of Go, for some simple cases, but not for the famous “semeai” problem (i.e. liberty races - more on this later). I like the example of semeai, as it clearly shows why MCTS is not ready for conquering the world: some simple things, trivial for humans, are extremely hard for MCTS. Unfortunately, understanding this weakness is a good first step, but we could not do the second step: finding a general tool for this.

We'll first see in this chapter why all this is crucial, both philosophically, industrially, ecologically, economically (yes, I'm convinced of this!). Section 3.1 will discuss the importance of planning. Section 3.2 will discuss the importance of games. Section 3.3 will survey some terminology, and section 3.4 will present key elements.

Next chapters will then give an overview of games, planning, and algorithms for games and planning. We'll see a brief survey of tools for games (chapter 4) and then fundamental elements on the complexity of games (chapter 5); we'll see that this is highly relevant for understanding which tools are the best ones. We will then present MCTS, a great recent tool (chapter 6) which have big advantages in terms of scalability, practicability, readability; section 7 is devoted to showing the generality of this approach, and section 8 to its parallelization. Chapter 9 will then discuss the limitations of this MCTS technique and some current works on it.

3.1 Planning is important

Planning consists in making a sequence of decisions so that everything is fine. Choosing who will work at which time in the hospital is a planning task. Choosing in which order you will do your tasks today is another planning problem. Scheduling the works of you and your collaborators is another planning problem. You can also consider power plant scheduling: which power plants will be switched on and which ones will be switched off at each time step this week ?

If you don't decide everything from the very beginning, but decide which strategy will be used for making decisions (depending on the information you get), this is another planning task; you have now observability. If everything is deterministic and known in advance, then observability is useless; otherwise, it matters.

In the extremal case in which there's no observability at all, you can decide right now all your future actions (without loss of generality): you then have to decide your actions independently of what happens - in most cases, however, you have partial observability: there are things you see, and things you don't see. Unfortunately, partial observability is very complicated, as shown both by theory (chapter 17) and by experiments.

3.2 Games are important, too

It is often argued that games are important because people learn a lot by games, and because many sophisticated animals play. I'd like here to emphasize other reasons for

the importance of games.

First, the best techniques in games are also very good techniques for other problems. For example, works emphasizing the use of computational power at each time a decision is requested, and not only off-line, are very similar to alpha-beta techniques from the game community. Also, the Monte-Carlo Tree Search technique, a revolution in games, is now a revolution in difficult planning (not in the non-reactive forms of planning).

Moreover, games provide a very clear framework for discussions and experimentations: the understanding of partial observability, and the difference between partial observability in exogenous random processes (case in which partial observability can be replaced by stochasticity) and real partial observability appears clearly in games - this is the difference between Kriegskiel and Memory. In the Memory game, there's an unobserved part, but we can completely remove it and replace it by a stochastic component (the hidden unseen cards can just be randomly sampled among unseen cards when they become visible). This is not possible in Bridge, Poker or Coinche as the hidden information is hidden only to some players. Therefore, Bridge and Poker fall in the same category as Kriegspiel whereas Memory falls in the category of Backgammon[3].

Another important argument is the **similarity between games and robust optimization**. Robust optimization consists in choosing a solution which works well independently of some outcomes: for example, the best planning for my power plants, provided that the risk of black-out is lower than 10^{-3} independently of the weather, if there are at most 5 failures in my plants. This is exactly as having an opponent, who can decide 5 failures, and who wins in case of black-out: you look for a strategy so that the opponent is surely defeated, or defeated with a maximum probability. For example, many works around the "fictitious play" algorithm are applied to both games and robust linear optimization[149].

At a more fundamental level, games are useful for understanding the current limitations of artificial intelligence in front of human intelligence; the example of the semeai situations in the game of Go is a perpetual wonder to me (see section 5.3.5). Games are also useful for understanding some of the great strengths of computers: the fact that computers win against humans in Shi-Fu-Mi (also known as Rock/Paper/Scissors) is also a perpetual wonder to me.

3.3 Terminology

We use the classical vocabulary on graphs; a directed graph is made of a set of vertices (also termed nodes), and a set of edges; an edge is a pair (a,b), where a and b are vertices; b is termed a child of a. A **Markov Decision Process** (MDP) is a finite directed graph with both decision nodes and random nodes.[4] **Random nodes** are equipped with a distribution of probability on their children. **Decision nodes** are either max-nodes or min-nodes. A max-**player**, a.k.a max-**strategy** or max-**policy**, is a (possibly stochastic) function which takes as input a max-node and gives as output one of the sons of

[3]We here neglected the number of players in the category, and only categorized games by the nature of partial observability.

[4]We here consider only finite graphs for simplicity.

this node. A min-**player** (min-strategy, min-policy) is defined similarly for min-nodes (for the partially observable case, definitions differ as we'll see later). If there's no min-nodes, or no max-nodes, then we will simply refer to policies or strategies.

Given a MDP, a max-player and a min-player, a **game** (also known as a **simulation** is the random path through the graph from the root to a leaf in which transitions are chosen by the max-player, the min-player, or at random, for max-nodes, min-nodes, and random nodes respectively. **Leaf** nodes conclude the game (game over).

In some cases, **leaf nodes** are equipped with a **reward** $\in \mathbb{R}$; the **loss** is the opposite quantity. We consider a reward, to be maximized by a player and minimized by the other (if there are two players); more general cases, involving different and non-opposite rewards for the two players, are possible but not (not much) considered here.

In some cases, there's a reward for each state (which is added to the total reward each time the player arrives in this state), or a reward for each pair (state,action), which is taken when the agent chooses this action in this state: then, there's a reward for each time step; r_t is by definition the reward obtained after the t^{th} move in the graph. We will note $r(x,a)$ the reward obtained when choosing action a at state x.

When a **discount factor** $\gamma \in]0,1]$ is used, then the total reward is $R = \sum_t \gamma^t r_t$. The case with no discount factor is equivalent to $\gamma = 1$. $\gamma < 1$ and bounded rewards per time step imply that the overall reward R is well defined. This is not ensured if $\gamma = 1$; when $\gamma = 1$ and if the reward might be infinite due to cycling, one can consider the average reward $R = \lim_{t\to\infty} \frac{1}{t} \sum_{i=0}^{t-1} r_i$, when it is defined.

Viability refers to various concepts, based on the idea that you take care of survival. For example, you might have no reward until you loose, and then reward -1; then, the best you can do (and this is obviously not possible for all games) is to ensure that the game is never ended. Some versions of Tetris involve viability. In some cases, you have reward -1 when you loose, and a discount factor $\gamma < 1$; then, even if the problem is such that you loose with probability 1 independently of your choices, you can find strategies optimizing the expected reward, and these strategies will try to ensure a long life.

A **Partially Observable Markov Decision Process** (POMDP) is a MDP modified so that:

- each decision node is equipped with an observation;
- the max (resp. min) player compute his (possibly stochastic) output as a function of the vector of the observations along the visited max nodes (resp. min nodes) - as a consequence, strategies are now functions from finite sequences of observations to actions and not from nodes to actions.

The finite sequence of observations can obviously be very big, and then it is not realistic to take a decision directly as a function of all this. Therefore, a compact (but imperfect) representation consists in having an internal state in the controller, and the equations become:

$$\begin{aligned} state_{t+1} &= f(state_t, decision_t) \quad //\text{transition function} \\ observation_{t+1} &= observation(state_{t+1}) \quad //\text{observation step} \\ (decision_{t+1}, memory_{t+1}) &= strategy(observation_{t+1}, memory_t) \quad //\text{decision making} \end{aligned}$$

A MDP can theoretically be recovered, from a POMDP, by considering $memory_{t+1} = (memory_t, observation_{t+1})$. This is a brute-force algorithm[5] and usually it is not realistic. Recurrent neural networks are a possible solution for non brute-force POMDP solving (see also reservoir computing). Another solution is based on Monte-Carlo simulations conditionally to observations, as discussed in Chapter 4.

3.3.1 Bellman values, value functions and reinforcement learning

The notions discussed in this section are presented in the one-player case, or for one player against random. The extension to 2 players is easier when the reward of player 2 is the opposite of the reward for player 1; this case is termed the zero-sum case. We refer to [213] for more information on the general case of $N > 2$ players or 2 players with non-opposite rewards, as well as for more information on notions presented in this section.

The **Bellman value** or **Massé-Bellman**[165] value of a state in a MDP is the expected reward of the terminal node if both players choose optimal decisions (max for maximizing the expected reward, min for minimizing the expected reward). The **value function** associated to a strategy π (in a one-player game here), denoted V_π, is the function such that $V_\pi(s)$ is the expected reward if you start at state s and use strategy π (in a multi-player games, V must be indexed by k strategies if there are k players). When no strategy is mentioned, then $V = V_{\pi^*}$ where π^* is an optimal strategy (optimal strategies do not always exist, but they do in many important cases).

When the transition function is only approximately known (typically, only a sample of transitions and rewards is observed), it is often more comfortable to work with the **Q-function**: for a state x and an action a, the Q function is defined as $Q(x,a) = r(x,a) + \gamma V(f(x,a))$. In particular, Q functions can be approximated in cases in which there are samples, but no model of the transition.

Q-functions can be directly learnt by Q-learning or Sarsa, based on simulations (the main advantage of these algorithms is that they don't require a model - they can directly learn from simulations). The simulations are usually performed by the ε-greedy strategy: with probability ε, play randomly, and with probability $1-\varepsilon$, play the action with maximum Q-value (i.e. in state s, choose action a maximizing $Q(s,a)$). Usually, ε decreases linearly with time i.e. with t the time step $\varepsilon = \varepsilon_t = K/(K_0 + t)$.

These algorithms (Q-learning and Sarsa) are incremental algorithms, updating a Q function as follows:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha\left(r + \gamma Q'(s,a,t)\right) \tag{3.1}$$

where

- $Q'(s,a,t) = Q(s',a')$ (for the Q-learning algorithm);
- or $Q'(s,a,t) = \max_b Q(s',b)$ (for the SARSA learning algorithm);

[5] It's a brute-force algorithm in the sense that it keeps in memory *all* the past observations, without trying to build a compact synthetic representation.

when a transition is made from the state s to the state s' by action a and the action to be chosen in s' is b. In Eq. 3.1 α is typically chosen linearly decreasing as a function of the number of visits in a. The algorithm is easily implemented for Q-functions which are just look-up tables (i.e. functions written in extenso, with one value for each (s,a)), or discretizations; implementing Q-learning with function approximation is more difficult [27, 1].

Dynamic programming is a classical tool for solving acyclic MDP; then, the Bellman value is computed for each state:

- for final states (leaf nodes), the computation is easy;
- for nodes with only leafs as sons, the value can be computed by averaging (for random nodes) or maximizing (for max nodes) or minimizing (for min nodes) over the children;
- we then compute the Bellman value by induction on the length of the longest path to a leaf.

This procedure has cost linear in the number of edges for an acyclic graph. It does not terminate in case of graph with cycles (but might converge in particular if $\gamma < 1$). Dynamic programming works well with 2-players in the zero-sum case.

Stochastic dynamic programming refers to dynamic programming applied to a MDP which contains random nodes. For two-player games, the natural counter-part of dynamic programming is the Minimax approach (see Fig. 6.1), except that in the Minimax approach the Bellman value (for time steps beyond a given horizon) is approximated by some value function so that we don't have to consider a too big tree. The minimax algorithm for choosing an action is presented in Alg. 1; the alpha-beta version is presented in Alg. 2.

Dynamic programming cannot be used as is for cyclic MDP. Value iteration (defined below) is a natural transformation of dynamic programming in the cyclic case. We consider that we are in maximization of reward below; minimization of loss is obviously equivalent.

Value iteration is an algorithm for solving (non-necessarily acyclic) MDP. It consists in iteratively improving a value function: if making decision a in state x leads to state $f(x,a)$ and reward $r(x,a)$, then you can update the approximate value function $\hat{V}$ by

$$\hat{V}(x) \leftarrow \max_a r(x,a) + \gamma \hat{V}(f(x,a)).$$

If $\hat{V}_i$ is the value function after i iterations of value iteration, the equation becomes:

$$\hat{V}_{i+1}(x) \leftarrow \max_a r(x,a) + \gamma \hat{V}_i(f(x,a)).$$

This implies that we can simulate the transition $f(.,.)$ as much as we want; in many real world applications, transitions are only observed when effectively applied to a plant - therefore, simulating $f(x,a)$ implies that the plant is effectively sent to state x and that action a is applied. This is obviously extremely difficult in many cases. Therefore, value iteration can often not be applied as is. Usual variants are based on incremental

minimaxChooseAction(s)
Input: a state s
Parameter: a depth $d \in \mathbb{N}$
Output: an action a
for each possible action a **do**
  Compute the state x that we reach if we choose action a in state s
  Define: $value(a) = minimaxValue(x,d)$
end for
Return the action which leads to the best value

minimaxValue(x,d)
Input: a state x, a depth $d \geq 0$
Parameter: a (usually handcrafted) evaluation function
Output: a value $\in \mathbb{R}$
if $d = 0$ or x is a game over **then**
  Return $evaluationFunction(x)$
else
  if we are in a max node **then**
    Return $\max_a$ $minimaxValue(t(x,a),d-1)$
  end if
  if We are in a min node **then**
    Return $\min_a$ $minimaxValue(t(x,a),d-1)$
  end if
  if We are in a random node **then**
    Return $\text{mean}_a minimaxValue(t(x,a),d-1)$
  end if
end if

Algorithm 1: The minimax algorithm *minimaxChooseAction* for choosing a move, which uses the minimax algorithm *minimaxValue* for estimating the value. The parameters are (i) an usually handcrafted evaluation function (used in *minimaxValue*) (ii) a depth d.

Function $alphabetaChooseMove(s,d)$
Input: a situation x, a depth d.
Output: a move a.
Return $alphabeta(x,d,-\infty,+\infty)$.

Function $alphabeta(x,d,\alpha,\beta)$
if $d=0$ or s is a game over **then**
  Return $evaluationFunction(x)$
end if
for each possible action a in heuristic order (probably best before) **do**
  $\alpha \leftarrow \max(\alpha, -alphabeta(t(x,a),d-1,-\beta,-\alpha)$
  if $\beta < \alpha$ **then**
    Break
  end if
end for
Return α

Algorithm 2: The alpha-beta algorithm as presented in [245]; it is an improvement of the minimax algorithm (Alg. 1). We simplified the presentation by removing random nodes. α and β are memories of the best/worst action found in other branches; alphaBeta is the pruning of useless moves thanks to this history.

or local families of functions for $\hat{V}$, so that $\hat{V}$ can be modified dynamically, more or less locally, during simulations; we will not consider a lot this important case and the interested reader is reported to [27] (in particular for $TD(\lambda)$ algorithms).

In **policy iteration**, we maintain a policy instead of a value function. More precisely, policy iteration works as follows[6]:

$$\hat{\pi}(x) \leftarrow \arg\max_a r(x,a) + \gamma V_{\hat{\pi}}(f(x,a))$$

or with index i for the i^{th} iteration:

$$\hat{\pi}_{i+1}(x) \leftarrow \arg\max_a r(x,a) + \gamma V_{\hat{\pi}_i}(f(x,a))$$

Policy iteration converges faster in terms of convergence rate per iteration, but it involves the computation of $V_{\hat{\pi}}$, which is expensive (possibly done by modified policy iteration as below). **Modified policy iteration**[176, 187] is policy iteration in which the computation of $V_{\hat{\pi}_i}$ is made by iterating:

$$V_{\hat{\pi}} = \lim_{k\to\infty} V_{\hat{\pi},k} \text{ (approximated by a finite number of iterations)} \quad (3.2)$$

$$V_{\hat{\pi},k+1}(x) = \max_a r(x,a) + \gamma \hat{V}_{\hat{\pi},k}(f(x,a)). \quad (3.3)$$

[6] $\arg\max f$ denotes (by a slight abuse of notation, as existence and unicity are not guaranteed) any x such that $\forall y, f(x) \leq f(y)$, for a minimization problem.

(the iterative method suggested in Eq. 3.2 can also be replaced by an exact solving (by linear programming). Value iteration and policy iteration are then a particular case of modified policy iteration.

Approximate dynamic programming is the use of dynamic programming together with supervised learning for approximating the function value (in order to compute it only at a restricted set of states, before generalization by supervised learning). It allows the use of dynamic programming in continuous problems. **Approximate value iteration** is similar in the case of value iteration.

POMDP with one player only (no min node) can be solved by dynamic programming as well: the solution consists in replacing the POMDP by a much bigger MDP, in which the state is the complete memory of the past observations (which might be arbitrarily large, if the POMDP has unbounded length!). Therefore, computing an exact optimal solution for a POMDP is possible only for very small POMDP; even for MDP, the computation is very expensive when the dimension is large.

POMDP with at least two players are much more difficult (however, the problem can be simplified if the observations are the same for both players - in other cases, the POMDP is said decentralized). Decentralized problems lead to higher complexity classes; indeed, in the case of two players per team, and two teams, with partial observability and **private information** (what is observed by a player is not observed by other players), the existence of winning strategies is undecidable[127][7], whenever we only look for strategies which are a sure win (the existence of strategies winning with probability $\geq \frac{1}{2}$ is much harder, as it is undecidable even with a finite graph with cycle (even with just two players and a deterministic transition function).

A **plant** is often a POMDP with continuous set of states; this terminology is usual in control theory; a plant is sometimes defined as a combination between a process and the actuators with an effect on it.

An extension of dynamic programming for the partially observable case has been proposed in [155] but is not tractable; it has indeed been shown that the complexity of such problems is very high[22]. [2] proposes essentially to work by **direct policy search**: define a parametric policy, with parameters, and optimize the parameters of the policy by non-linear programming.

3.3.2 Examples: board games, card games, sports, and other games

Fully observable games with simultaneous decisions by the two players can be rephrased as POMDP without simultaneous decisions (see Fig. 3.1 for an example).

Chess, Draughts, Go, Shogi, Hex, Havannah, are non-stochastic MDPs with two players. Shi-Fu-Mi (Fig. 3.1) is a two-player deterministic POMDP, in which computers are surprisingly strong thanks to their ability in finding our human weaknesses as random generators.

[7] The complexity of the management of decentralized POMDP can be illustrated by the high percentage of death by so-called “friendly fire” in modern wars (a friendly fire is an inadvertent firing towards friendly forces.

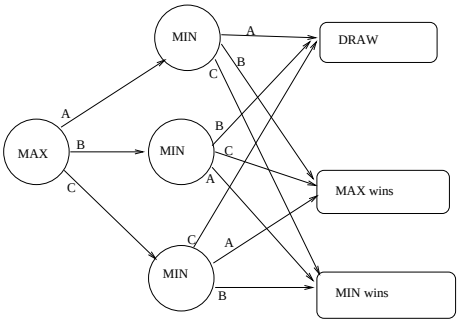


Figure 3.1: The POMDP corresponding to Shi-Fu-Mi. Informally, and with less poetic action names than in the original version of the game, max and min simultaneously choose *a*, *b* or *c*: if max = min, there is equality; *a* wins against *b* but looses against *c*; *b* wins against *c*; the game is symmetric. We see on this graph (in which the observation is always empty) that a game with simultaneous decisions for the two players can be rephrased as a turn-based game in the POMDP case.

Tennis[8] is a two-player nearly deterministic POMDP. Phantom-Go is a deterministic two-player POMDP. Bridge is a decentralized stochastic POMDP with more than 2 players. Football is a decentralized nearly deterministic POMDP with 22 players (11 per team). The management of power plants in front of random uncertainty is a one-player stochastic POMDP. First Person Shooters (games like Wolfenstein or Doom) with one player trying to kill fixed non-player characters is a one-player POMDP. Real Time Strategy Games (games like Populous or Dune 2) are usually one-player POMDP (except network versions, or if we consider the strategy of each soldier alone[9] - then we switch to a decentralized multi-player POMDP).

3.3.3 Succinct representation and limited horizon

Succinct representations are a usual tool in graph. The idea is that, instead of representing a graph by an explicit set of transitions on a finite set of nodes, we can represent it by a compact representation: typically, the set of states is the valuations of a finite number of binary variables, and the transitions are represented by Boolean formula. With such representations, the length of the input is much smaller than the number of transitions; as complexities are expressed with respect to the size of the input, this leads to much higher complexity classes for succinct representations. Typically, a simple MDP with one initial state and deterministic transitions switches from P-complete to PSPACE-complete when switching from non-succinct to succinct representations[192]. Complexity in succinct representations are related to general game playing: the complexity which is considered is the complexity for solving a whole family of games and not just one game, and rules are at a sufficiently abstract level to be much smaller (ex-

[8] Please note that we here include games with continuous states in these illustrations; obviously, continuous domains introduce new troubles in many cases; we will not develop this further here.

[9] In usual versions on a single personal computer, a player takes care of all his individuals in the game.

ponentially smaller, usually) than the complete state space. Nonetheless, we'll see that many games are as difficult (at least from the point of view of computational complexity classes) as these whole families of games: for two-player deterministic fully-observable problems, the complexity is EXPTIME-complete [192], and we have the same complexity for chess, Go with no superko, draughts, ...

Short games. Succinct representations make games (artificially) much harder, and limited horizon make games much easier: for example, a MDP with polynomial length is in P; a POMDP with exponential length is EXP-complete, NEXP-complete and EXPSPACE-complete in the observable, non-observable, and partially observable cases respectively, whereas without unbounded length it is EXP, EXPSPACE and 2EXP respectively[171, 192].

In the non-succinct case with a number of time steps equal to the number of states, some complexities from [116] and references therein are as follows:

- Computing the optimal value of a MDP is FP-complete[10].
- Computing the optimal value of an unobservable MDP is $FP^{[NP]}$-complete[11].
- Computing the optimal value of an unobservable MDP is $FPSPACE$-complete[12].

We can express these results (still for MDP with a number of time steps equal to the number of states) in terms of decision problems, i.e. existence of a policy with positive reward. Then, the problem is P-complete for fully observable MDP, NP-complete for unobservable MDP, NP^{PP}-complete for partially observable MDP[13], $PSPACE$-complete for partially observable MDP.

3.4 Some key notions in games

This section will briefly recall some crucial notions (many of which are more precisely defined elsewhere in this book) in computer games. This chapter is not supposed to be very precise; it is more here as an informal introduction to games and to several important techniques:

- **tree search**, and in particular **alpha-beta**, the most well known algorithm for games. Alpha-beta is an improvement of **Minimax**; alpha-beta is much faster in particular when a heuristic ordering on legal moves is available; see section 3.3. Alpha-beta has had many improvements since the early times, in particular iterative deepening which iteratively increases the horizon of the search. In iterative deepening, each iteration is an alpha-beta search with limited depth, and each iteration helps for improving the ordering used in later alpha-beta iterations thanks to the values memorized (see section 3.3.1).

[10] FP is the set of functions computable in polynomial time.

[11] $FP^{[NP]}$ is the set of functions computable in polynomial time with a NP oracle.

[12] $FPSPACE$ is the set of functions computable in polynomial space.

[13] NP^{PP} is the sets of problems solvable in polynomial time by a non-deterministic machine with PP oracles; a PP-oracle is an oracle which solves a problem for which a polynomial time solution exists on a probabilistic Turing machine with error rate less than $\frac{1}{2}$ on all instances.

- for **partially observable games**, the notion of **belief states**, i.e. the distribution of probability of the unknown internal states (in problems in which this distribution makes sense, e.g. partially observable stochastic one player games or partially observable two player games at Nash equilibrium), or the set of possible states, defined as a function of past observations. The belief state can be approximated by Monte-Carlo sampling (more or less[14] theoretically consistently, depending on the framework).

- **opponent modelling**, which is crucial when we want to benefit from the opponent's weakness, rather than reaching the Nash equilibrium. In some cases (see in particular rock-paper-scissors), the Nash equilibrium is easy but uninteresting, whereas some strategies (which are useless against a Nash strategy) have good rewards against realistic opponents.

- **evaluation function**: an evaluation function takes a state as input (or a sequence of observations in the partially observable case) and outputs a value; it might be handcrafted, or it can be based on Monte-Carlo evaluation, or it might be learnt by self-play (reinforcement learning), as in e.g. TDgammon[219].

- **ordering of moves**, which is crucial in alpha-beta. The heuristic ordering of moves (thanks to human expertise) can be greatly improved by some generic techniques:

  - **refutation tables**, which use statistics from already visited nodes for biasing the search in a new node of the tree search, depending on the last move. Informally speaking, with refutation tables, if move b is often a good answer to move a, then b is tested early when exploring nodes in which the last move is a.
  - the **killer heuristic**, which introduces a bias towards moves which are good at a given depth (according to earlier evaluations). Thanks to this bias, the cutoffs are more frequent, and the alpha-beta is faster.
  - **iterative deepening**, in which early iterations provide heuristic values for ordering later iterations (as discussed above). Thanks to the better ordering of moves, the cutoffs are more frequent, and the alpha-beta is faster.

- **Monte-Carlo tree search**, a recent very promising approach; as this beautiful technique takes a lot of room in the next chapters we will not insist too much here.

[14] Sometimes not at all, as shown by undecidability theorems later in this document.

Chapter 4

Games and tools: brief state of the art

This chapter is devoted to briefly summarize the main techniques used for various games. Section 4.1 discusses computer chess, the most classical testbed in the old times. We then briefly discuss computer draughts (section 4.2), computer backgammon (section 4.3), computer rock-paper-scissor (section 4.4), computer bridge and poker (section 4.5), computer connection games (section 4.6); a final section will be devoted to other games.

4.1 Computer chess: alpha-beta and its many improvements

The most well known challenge for computers is probably chess; nonetheless, chess is now easy for computers which clearly outperform humans. The tools for this are:

- An alpha-beta algorithm with various improvements:
  - quiescence search: a leaf is not evaluated without playing a few moves on it until the situation is somehow "stable"; before evaluating, play captures and obvious improvements of the position.
  - good ordering of moves for fastening the alpha-beta (but almost no hard pruning usually).
  - iterative deepening[144], which provides a better ordering of moves for the alpha-beta pruning (each iteration makes later iterations faster by favouring cutoffs thanks to the better ordering).
- A good evaluation function: the evaluation functions in chess typically sum a score for each piece on the board, including pawns, plus modifiers depending on some degrees of freedom. The evaluation function is used both:

  - for evaluating leafs;
  - for choosing the ordering of moves to be tested in the alpha-beta.

  Endgames databases are used also in the evaluation function.

- Endgames databases: there are proved families of situations, for which it is known that they lead to a win for a given side. Almost all six-pieces endgames are known. Endgames databases are built by retrograde analysis, i.e. by the analog of dynamic programming from a final state (a win for one side) backwards in time.

- Transposition tables are used for keeping in memory already analyzed positions in order to ensure that a same situation is not analyzed twice.

- Refutation tables keep in memory statistics on the best answers to each given move, dynamically during a search; this allows a bias in the order in which moves are considered.

Interestingly, computers sometimes play, against humans, moves which lead to endgames which are, for perfect play, a draw - they "know" that humans don't play endgames perfectly. This is a limitation to the classical "same strength assumption", according to which both players are considered equivalently in the tree search.

Computers are now extremely strong and win against the best humans; however, strong players can ensure many draws. Humans usually try a specific technique, specialized against computers, namely moves based on very long term effects. It is widely believed that computers became stronger than humans in 1997, with the win by Deep Blue against Gary Kasparov; nonetheless, at that time, humans were still stronger, and the really clear victory of computers occured in 2005 with 5 wins and 1 draw out of 6 games by Hydra against Mickael Adams.

4.2 Computer draughts: exact solving

The case of draughts uses roughly the same methodology as chess, with a big emphasis on the retrograde analysis; it has been pushed to the point that even the initial situation is solved (for the draughts 8x8 version termed "go as you please"). The conclusion is that draughts are a draw, i.e. both players can ensure a draw if they make no mistake.

4.3 Computer backgammon: smooth evaluations for an old randomized game

Backgammon is a fully observable randomized game in which until the pioneering work [219] humans were better than computers. Ancestors of backgammon, such as Senet, seemingly go back to 5000 years ago, probably earlier that Go - at that time there were still mammoths in Wrangel Island (nothing, however, shows that backgammon was ever played in Wrangel Island, so maybe no mammoth has ever seen a backgammon or a go game). Alpha-beta can theoretically be adapted to stochastic games, by

e.g. averaging the values of the possible random outcomes. In Backgammon however, the crucial point is the evaluation function, built by self-play: the really strong programs appeared in the late 80s, with neural networks evaluation functions optimized by reinforcement learning (temporal difference learning $TD(\lambda)$). The tree search on top of the evaluation function does not have to be very deep.

Computers won games against the top players in the world in July 1979; the program was based on some sets of rules, with a "fuzzy logic" approach for switching in a non-abrupt manner from a strategy to another[21]. Nonetheless, the win was a lucky win, due to more favorable dice rolls, and the real strength appeared with optimized evaluation functions as discussed above[246].

4.4 Rock-paper-scissors: beyond Nash strategies

Rock-paper-scissors is an interesting game (rules are given in Fig. 3.1) by its apparent simplicity: the Nash equilibrium is trivial, each player should play rock, paper or scissor with probability $\frac{1}{3}$. But humans are not able to be perfect random players: as a consequence, the best algorithm against humans is not the Nash equilibrium; algorithms which perform *opponent modelling* in order to guess, with a bit more than probability $\frac{1}{3}$, what the human will play, win against humans, whereas the Nash equilibrium would only have a tie (on expectation).

4.5 Computer bridge & computer poker: handling partial observability

Bridge is a partially observable randomized game; alpha-beta search can't be applied as is. Consistently with works like [10], some algorithms use the belief state (i.e. the distribution of possible states, including the unknown part) - they generate thousands of possible states (they reject those which are not consistent with past observations), and choose between a finite set of strategies depending on the best average result for this distribution. A mathematically interesting point is that the distribution of probability on unknown states, in cases with two players, should take into account the choices made by the opponent, and doing this perfectly is not computable (see chapter 5 for more on this) in the general case of partially observable games. This explains why real world algorithms are usually quite heuristic and use Monte-Carlo simulations as a feature rather than for developing a rigorous tree search.

The quantity of scientific work on computer bridge is much smaller than in chess or go; it is therefore difficult to know exactly how strong will be computers in the future and if the state of the art in computer games is enough for playing at the top human level in bridge.

The situation in poker is close to the case of bridge, but with a strong increase of attention these last years[33]. There's a lot of opponent modelling in poker (as in the case of Rock-Paper-Scissors, guessing the opponent's style (his/her bias) is crucial for winning against weak players), and a lot of work for hiding one's information / for guessing the opponent's information. The multiplayer aspect increases the importance

of the opponent modelling, as playing with a one-against-all style would be highly suboptimal.

4.6 Computer connection games: a big AI challenge

Connection games are games which involve connections; a player wins (or has a strong advantage) if he/she can connect something to something. The oldest known connection game is lightning[38]; the most well known are Hex, Havannah, Go. In these difficult games, Monte-Carlo Tree Search is now the main approach. We will give more information on connection games and on Go in the MCTS chapter (Chapter 6).

4.7 Other computer games: so many new challenges

We here follow the excellent lines of the introduction of [33] and refer to it for further information. In Othello, the program Logistello won 6 out of 6 games against the human world champion Takeshi Murakami. In a competition in 2000 of Lines of Action, the program Mona won all its games against humans; it has never been defeated by a human. In 2002, Awari was strongly solved (i.e. the value of each possible state was computed). In Scrabble, the opponent modelling is not necessary for performing better than humans; it is established (in Scrabble) that just Monte-Carlo simulations and conditioning to past observations is enough for having a good distribution of probability on possible future outcomes, without taking into account the possible deceptive actions by the opponent - in fact, just checking all possible actions at each time step and playing greedily the action providing the best reward is better than what humans can do. Theoretically, Scrabble has partial information, to be taken into account for perfect play; but this effect is moderate in Scrabble.

Partially observable board games (in particular Kriegspiel and Phantom-Go) are a beautiful challenge for future theoretical or experimental research - as undecidability results will show later, these problems are extremely different from the classical cases.

An interesting case is general game playing, in which the rules of the games are given just before the game starts; in a competition of general game playing, programs have to read the rules (in a given format), and then play games. In this difficult setting as well as for connection games, Monte-Carlo Tree Search is now the standard approach.

Video games (First Person Shooting (FPS), Real Time Strategy Games (RTS)) are a great recent challenge; AIs are usually based on scripts and sometimes cheat (they might be informed of your “private” information or have better rates than you).

4.8 Summary

As a summary,

Human / computer comparison. Computers are stronger than humans in chess, draughts, lines of action, and many other games; a few games are even completely solved in the sense that perfect play is reachable. Surprisingly, computers are stronger than humans in rock-paper-scissors.

Classical tools are alpha-beta, killer heuristics, quiescence search, Monte-Carlo, opponent modelling, belief states, evaluation functions, transposition tables, retrograde analysis, refutation tables, RAVE values, parallelization.

New challenges. Partially observable game, real time games, connection games, distributed games, are important new challenges.

Chapter 5

Games and their complexity

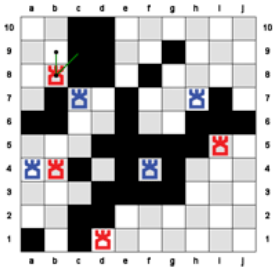
We will here consider games with **two players** (*i.e.* we do not consider puzzles, and we do not consider (at least, not a lot) games with 3 or more players (which require a lot of opponent modelling), neither in the collaborative case, nor in the case in which coalitions might arise. In many cases below, we will discuss games with real numbers and/or with continuous time (like sports); the complexity of such games is obviously not formalized in the same terms as discrete board games - they are mainly here for illustration.

We will consider mainly (but not only) **completely observable games** (Fig. 5.1); the partially unobservable case is nonetheless extremely interesting. Some examples of partially observable games are given in Fig. 5.2. In kendo, people are dressed to that their legs are not fully observed; however this is probably a minor effect. In table tennis, the rules now forbid that people hide the ball when the racket hits it during the serve, but there are still many efforts for hiding the spin given to the ball. Football is an important nice example: it's decentralized, and communication between cooperating agents (players of the same team) is hidden to the opponents: the computational complexity of soccer is probably extremely high, much more than the computational complexity of Go. Shi-Fu-Mi, when formalized with simultaneous decisions by the two players, is fully observable - when formalized with sequential decisions, it is partially observable (see Fig. 3.1).

There exist big theoretical developments around partially observable problems, with the interesting big strength of the simplest approach, namely direct policy search (possibly, however, based on sophisticated features taking into account the past observations), for these complicated cases. Direct policy search converts the problem of planning (or playing) into a noisy non-linear optimization; therefore, partially observable planning is indirectly treated in chapter 14.2 (stochastic optimization). Sometimes, policies for partially observable games use random sampling of the possible states as a key feature.

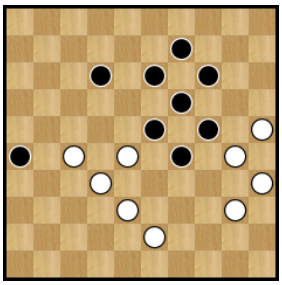
We will not (not much) consider:

- iterated games;
- games with continuous states or actions;

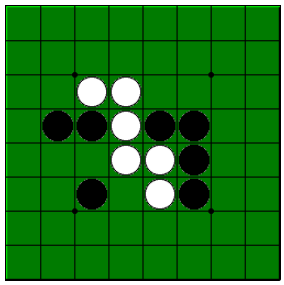

Amazons

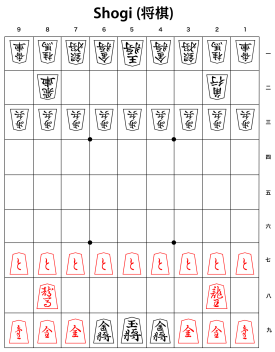

Chinese checkers


Hnefatafl

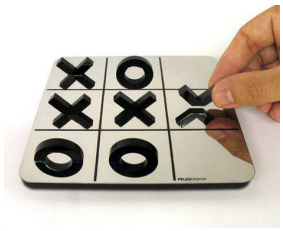

Draughts


Hex


Reversi



Shogi


Tic-Tac-Toe

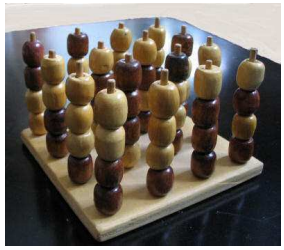

Qubic

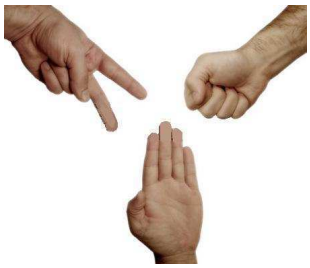
Figure 5.1: Fully observable games. Nonetheless, for some of these games, focusing one's attention on a given part of the board is not very different from observing a priori hidden parts of the state.

- games with random parts (yet, the considered algorithms can often be used in a straightforward manner and efficiently in the stochastic case).

Kendo

Table tennis

Football



Shi-Fu-Mi

Figure 5.2: Some partially observable games (see text for more).

5.1 The complexity measures of games

There are plenty of complexity measures for games; none of them clearly indicates whether finding an optimal strategy is difficult, and none of them clearly indicates whether computers are better than humans or not.

For many complexity measures, the complexity is defined as a function of the size of the board; this means that we have to consider a generalization of the game to various sizes. This is easy for Hex, Havannah, Amazons, Draughts (several variants are however possible for generalizations, e.g. for the initial position), but not for chess.

The main measures are as follows:

- **State-space complexity.** Number of possible states in the game. Please note that this should include only legal states, and keep in mind that when some rules take into account the memory (e.g. forbidding twice[1] the same situation with the same player to play, or claiming a draw in case of 50 moves without moving a pawn in chess, or variants of positional or situational superko in Go), then the state should include a sufficiently big information for making the rule explicit.

- **Game-tree size.** This is the number of leaf nodes in the tree of possible situations. Some games with huge game tree size are trivial, some games with moderate values are harder than chess for computers.

- **Decision complexity** is the minimum number of leaf nodes in a decision tree

[1] In some games like draughts, draw occurs if the same situation occurs three times.

that establishes the min-max value of the root node[2]. This measures looks quite appealing, but I am not aware of a lot of works around it. We have nonetheless in [25] a proof that some versions of MCTS visit infinitely often only a set of nodes which is minimal for inclusion (not for cardinal, unfortunately) among the subsets which are sufficient for establishing the minimax value of the root.

- **Game-tree complexity** is based on the same idea as the decision complexity, except that we consider all the nodes until the minimum depth of a "flat" tree (flat tree = a tree with all paths having the same length) sufficiently deep for establishing the min-max value.

- The **computational complexity** of a game considers the resources necessary for guessing if it is possible to have reward better than a given threshold for a position of size n. For example, given a position in draughts, on a board of size $n \times n$, how many time and space do we need for knowing if (in case of perfect play) the player to play will ensure a win ? It is probably the most studied measure after some easy bounds on state-space complexity and game tree size, probably because it's mathematically more fun, and because it uses the vocabulary of classical complexity (see Fig. 5.3; this vocabulary makes people who use it look much more clever than their neighbours who implement and test complicated algorithms) and, somehow, provides some hints in the understanding of games and in the understanding of families of situations (the case of Go is particularly rich from this point of view). The results around partially observable games (only very briefly cited here) are particularly impressive to this author. This measure will be discussed further in section 5.2.

- There are also works around the complexity (in terms of resources like time and space) necessary for playing optimally on a board of size n. This can be upper bounded by computational complexity, but it is not equivalent (see counter-examples later in this document).

- An empirical complexity measure is the number of games won by computers against top-level humans. For example in chess, it is now impossible for humans to compete with machines. In 9x9 Go, some computers have won games against professional players; and even in the disadvantageous situation of black, MoGoTW won as black against top level humans (a top pro, i.e. a professional player with rank 9p and recent winner of a major open tournament, lost as white in 2009 against MoGoTW as black) - this looks like the point at which computers reach the best human level.

- Finally, empirically, there are solved games:

  - very weakly solved games, namely games in which is it known that the first player necessarily wins in case of perfect play (like Hex, for which it has been shown by J. Nash by a nice strategy stealing argument - but we don't know the optimal strategy).

[2] For understanding this definition, you have to understand that (at least for many games) you can compute the min-max value of the root with a finite subtree of the complete tree of possible states.

– weakly solved games (the intuitive notion of solving) in which a computer can reasonably[3] play optimally; this is the case for English Draughts in 8x8, for which a long parallel computation has shown that both players can ensure a draw. [205] pointed out that this perfect play was probably reached by the top level humans as well.
– completely solved games, in which a computer can play optimally from any position. This is a strong requirement, as will be shown by sections below: there are very difficult situations such as those built by Turing-reductions for proving complexity results, and which are very unlikely to occur in a real game between players who are not desperately trying to make the game extremely strange and complicated (only mathematicians might play a game such as the one in [194]).

5.2 Computational complexity of games

In this section, we will consider the computational complexity of games. Part 5.2.1 will discuss the main ideas. Part 5.2.2 will discuss advanced topics.

5.2.1 Generalities and introduction to complexity

The most classical complexity classes considered in computer science (including the famous P and NP classes) are complexity classes for decision problems. This means that problems under consideration must be binary problems: an input of size n is given, and the program must reply yes or no. The problem is said to be in P if their is a program (on a Turing machine) which solves the problem in polynomial time. It is said to be in NP if their is a program on a non-deterministic Turing machine which solves the problem in polynomial time.

The reader unfamiliar with non-deterministic Turing machines is refered to classical complexity textbooks for more information around that; non-deterministic Turing machines are sometimes popularized as a (bad) model of parallel complexity. It is widely believed that non-deterministic Turing machines are much more powerful than classical Turing machines which are reasonably analogous to usual computers. A poor popular version of NP therefore considers that NP means "non-polynomial", because a problem which is NP-complete is usually supposed to be non-solvable in polynomial time; yet, "being in NP" does not mean "not being in P", and "$P \neq NP$" has never been proved.

This complexity measure makes sense only if we can consider problems of variable size n. For games, we will therefore consider e.g. boards of variable size n; this is somehow difficult for chess, but easy for e.g. the games of Go or Draughts. Then, we will consider the complexity class of the question:

Here is a situation, with a player to play. Can the player to play ensure a win in this situation ?

We first point out some simple facts around this kind of question. First, there

[3] i.e. with reasonable time.

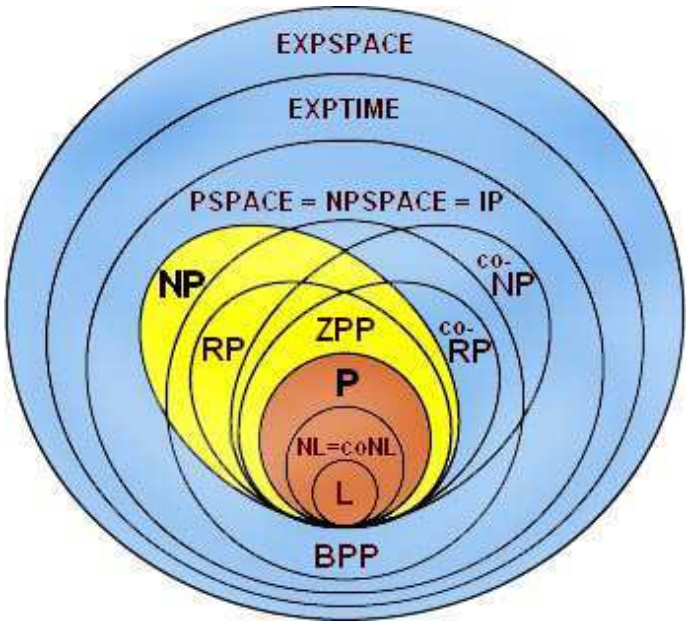


Figure 5.3: Some complexity classes for decision problems. These classes are not easily used for partially observable problems as the input is not straightforward (the complete history might be required); their use for fully observable or non-observable cases is easier. Computational complexity might be very misleading for games as problems for which the optimal strategy is obvious can be in a very high complexity class, just because the complexity usually involved is the decision problem for arbitrary initial state, independently of the initial state of the game; the game might be much easier with knowledge of its initial state.

are games for which the optimal strategy if trivial for both players, and which have nonetheless a very high complexity for this measure. Consider for example the game of Go, modified as follows:

- The first player, instead of playing, can just claim ”I have won”, and in that case he wins the game.
- Otherwise, he plays; then, the second player chooses between claiming that he has won, or playing a move.
- In the second case, the game is played as in the usual game of Go.

Let’s term this game the ”stupid-Go”. The complexity classes of stupid-go are exactly those of the game of Go. It is nonetheless very easy for a 2-years old child to win a game as soon as he is black (the first player), and to have the optimal strategy as white.

Also, the tree of a game can be reduced, in the sense of just removing some nodes (therefore removing legal moves), and nonetheless increasing the complexity. For example, another variant of stupid-go would be that each player can, at anytime, claim

that he has won; removing this rule just removes nodes from the tree, and nonetheless makes the game much more complicated (and more interesting, except maybe from a psychological or sociological point of view).

Turing machines will not be completely formalized here; the interested reader is referred to [236, 137, 235] or any standard computational complexity book for more on this. The reader might just keep in mind that Turing machines are abstract machines, roughly equivalent to the intuitive idea we have of what is an "abstract" computer. Then,

- PTIME (also noted P) means computable with polynomial time,
- PSPACE means computable with polynomial memory requirement,
- EXPTIME (also noted EXP) means computable with exponential time (i.e. $O(2^{n^{O(1)}})$),
- EXPSPACE means computable with exponential memory requirement,
- 2EXPTIME (also noted 2EXP) means computable with exponential of exponential time, and so on.

Less-than-linear complexities (e.g. logarithm time, LTIME) are a bit more difficult to define as they must take into account the fact that reading the input already takes a linear time and storing the input takes a linear memory; L means logarithmic time but only for additional time (i.e. we are allowed to have a linear time - linear with coefficient 1 - spent for reading the input one letter at a time) and such a class is useful when we consider very big datasets than can be stored in e.g. a set of DVDs but not in memory. The suffix "time" is often removed for short, i.e. $P = PTIME$, $L = LTIME$.

There are variants of Turing machines which are convenient for mathematical analysis. **Non-deterministic Turing machines** are Turing machines which contain "$\exists$" states with several transitions: in these states, the machine simulates simultaneously all the transitions and there is acceptance if and only if at least one of the transition leads to acceptance. **Alternating Turing machines** are less widely known: they contain both "$\forall$" states and "$\exists$" states, with several transitions: in $\forall$ states, the machine simulates simultaneously all the transitions but there is acceptance if and only if *all* these transitions lead to acceptance. Complexity classes corresponding to non-deterministic Turing machines are prefixed with a *N* and complexity classes corresponding to alternating Turing machines are prefixed with a *A*. Complexity classes corresponding to $\forall$ states only are prefixed with $Co-N$. One can immediately see that $NP \cup Co-NP \subset AP$: NP is the class of problems solvable (in poly time) with $\exists$ states, $Co-NP$ is the class of problems solvable (in poly time) with $\forall$ states, and AP is the class of problems solvable (in poly time) with both kinds of states.

A "*N*" obviously increases the computational power of a machine, but less than a *A* (see [129]):

$$\begin{aligned} P \subset NP \subset AP \\ PSPACE \subset NPSPACE \subset APSPACE \\ EXP \subset NEXP \subset AEXP \\ \dots \end{aligned}$$

The following results are well known since [60]:

$$\begin{aligned} AP &= PSPACE \\ APSPACE &= EXP \\ AEXP &= EXPSPACE \\ AEXPSPACE &= 2EXP \end{aligned}$$

Let's now consider some complexity classes and their inclusions:

$$NL \subset P \subset NP \subset PSPACE = NPSPACE \subset EXPTIME = APSPACE \subset NEXPTIME \ldots$$
$$\cdots \subset EXPSPACE = NEXPSPACE = AEXP \subset 2EXP = AEXPSPACE \ldots$$
$$\cdots \subset 2NEXP \subset 2EXPSPACE = 2NEXPSPACE$$

with some strict inequalities $NL \neq PSPACE$, $P \neq EXPTIME$; and all $\subset$ above widely believed to be strict.

If X is a class, a problem p is said to be X-complete, if the two following requirements are met:

- the problem p is in X;
- the problem p is "more difficult" than any problem in X.

The second requirement is more difficult to formalize; it is based on the concept of reduction. For showing the X-completeness, it is sufficient to show that if the problem p' is in X, then there is a transformation t which can be computed within the resources allowed in class X such that

$$\forall q, t(q) \in p \leftrightarrow q \in p'.$$

Usually, there are problems which are known complete for some classes, and in order to show that your game is complete for a class X, you have to show:

- that it is in X (you can show that by exhibiting an algorithm which solves the problem within resources allowed in X);
- that some problem q (carefully chosen), already known for being X-complete, can be *reduced to your game*; *i.e.* there is a transformation (using only resources as in X) encoding a problem of q in a situation of your game, so that e.g. black wins if and only if the good answer to the problem is "yes".

Many important fully observed games are either PSPACE-complete or EXPTIME-complete; we need more results for partially observable games. An important game used for showing PSPACE-completeness of games is the so-called geography game, in particular in its planar version which can be encoded in many games (this is far from being straightforward). All games with a finite board, in which an empty location is filled at each move and is never freed, is in PSPACE (but not necessarily PSPACE complete), as can be shown by an easy depth-first search (which works in polynomial space).

The following games are PSPACE-complete (and the list on Wikipedia is usually up-to-date):

- Tic-Tac-Toe;
- Qubic;
- Reversi;
- Hex;
- Go-moku;
- Connect-6;
- Amazons.

The following games are EXPTIME-complete (for the usually accepted generalization in dimension n, which is not always so clear):

- English draughts;
- International draughts;
- Chinese checkers;
- Chess;
- Shogi;
- Go for some set of rules.

This classification becomes much more complicated when including partial information or decentralized controllers: just formalizing the concept of complexity in that case is much more complicated. There are nonetheless results showing that much higher complexity levels are involved in that case - more on this later.

The case of Go is particularly rich:

- There are families of tsumegos[4] with forced moves only for one player (i.e. there is a fast algorithm for showing that all but one move are immediately fatal), and with 2 possible moves only for the other player, and with polynomial horizon, which are nonetheless NP-complete.
- Go without ko rule is PSPACE-hard, i.e. at least as difficult as PSPACE.[5]
- Go with ko rule and no superko (Japanese rules) is EXPTIME-complete.
- The variant termed "ponnuki-go" (in which the first capture makes the win) is PSPACE, but PSPACE-completeness has never been proved. Ponnuki-Go in which the first move of each player cannot be "pass" and then each player can pass when he wants, is a draw.
- The complexity of Go with ko and superko is not known.

[4] A Tsumego is a situation of Go; an exercise consists in finding if the player to play can win or not.

[5] The ko rule forbids, in case of capture, to come-back to the state before the capture.

- The complexity of Phantom-Go is not known. For the general family of POMDP with an opponent and succinct representation, the problem (of finding, if any, a strategy which wins with probability 100%) is 2EXP-complete, but this is not known for phantom-Go.
- The very end of a Go game, termed "small yose", is PSPACE-complete[248].
- To the best of our knowledge, the complexity of Go with superko rule is not known.
- To the best of our knowledge, the complexity of Go without ko is not known.

Interestingly, if you can show that Go with ko rule is more difficult than Go without ko rule, then you have solved an important open problem of computer science, i.e. you have shown $EXPTIME \neq PSPACE$. The same if you can show that Go without superko is more difficult that Go with superko. Other examples of families of situations and their complexity will be given below.

5.2.2 Succinct representations, general game playing, team games, partial observability

We will here consider cases which were not discussed above:

- games with partial observability;
- complexity for programs taking as input, instead of just the position, the position and the rules of the game; this is highly related to general game playing;
- decidability issues. It is known since [114] that many problems are undecidable, i.e. that some facts cannot be computed even with unlimited time and space: e.g. provability of mathematical results, existence of bugs in a program [236, 137]; decidability is not an old dead topic, as shown by recent results answering old conjectures[163]. This is, to some (large) extent, independent of the representation and therefore considering the succinct case has no impact on decidability. A particularly surprising result is that there are games on a finite board, with partial observability and private information between teammates, such that the existence of a winning strategy is undecidable [127].

Are computational complexity results with succinct representations a good model for general game playing ? More or less. The drawbacks of computational complexity (discussed above) also hold for succinct representations. Nonetheless, the main results are very informative. Another drawback is that there are various models of succinct representations. They have usually nearly the same complexity results[192], but this is not an absolute rule, and you should not apply results discussed below for your particular case without checking the details.

Some important results can be summarized as follows (see [192, 171, 162] and references therein):

Complexities in the succinct case for win/loss/draw games	
MDP (1 initial state) deterministic, one goal state)	PSPACE-complete
MDP with an opponent, no probability (i.e. non-deterministic case in complexity terminology, without probability)	EXPTIME-complete
POMDP with an opponent, no probability, problem of separating cases with proba of winning > 0.6 and cases < 0.4	undecidable *****
POMDP without opponent**	AEXPTIME=EXPSPACE-completeness
POMDP with an opponent, no observability, no probability (i.e. 1 initial state, non-determinism, one goal state)	EXPSPACE-complete
POMDP with no opponent, no observability, probabilities and goal = success with proba 1	EXPSPACE-complete
POMDP with opponent and no proba	2EXP-complete
0-player deterministic game with infinite state (e.g. Conway's game of life)***	undecidable
POMDP (even no opponent, but unbounded horizon)*	undecidable
POMDP with one goal state	undecidable****
Finite partially observable non-stochastic team games (finite state, three players, two of them in the same team)**	undecidable[183, 127]

(* here the undecidable question is: is there a solution for having average reward $> c$ with proba 1 ? the reward for each state is known)
(** here the undecidable question is: is the situation a sure win ? the horizon is not bounded)
(*** here the question is: will a given Boolean variable be true one day ?)
(**** here the question is: is there a solution for ensuring reachability with proba $> c$?)
(***** this result is not often cited, as most people focus on the question "is there a strategy for winning with probability 1"; yet, this case might be the most natural formalization of the idea of decidability of a partially observable game; the proof is by reduction to the case (****) thanks to the simulation of random transitions by hidden information)

Importantly in undecidability results above in partially observable (PO) cases, games are allow to have cycles. With PO, players can't know that they come for the second time in the same state as they don't observe the state.

The terminology above might be puzzling for people who are not familiar with non-determinism as used in the vocabulary of complexity: non-determinism here means that an opponent can choose some moves. This is in fact a two-player case. No observability means that there's absolutely no information that can be used by the strategy except the rules of the game and the initial position - no idea of the current state. The last undecidability result, in the table above, for team games, consider games with players having private information, not shared with their teammates. Surprisingly enough, such games can be undecidable even in the easy case in which we only want to recognize situations in which there is a sure win. No such real-world game has never been shown undecidable for the moment. With a poetic formulation, maybe there are "real" games such that gods play better than computers[6] - what is known is that there are (artificial) such games.

For the EXP-complete games cited above (go, chess, shogi, draughts...) we can see that the fact that most games or problems rely on a horizon much shorter than the exponential case has a strong impact: problems should be downgraded to a AP-completeness (equivalently, PSPACE-completeness, but the easy proof is by the use of alternating Turing machines) if we consider that usual games have a linear length. Maybe, nonetheless, the game would be incredibly long if players were almost as strong as gods. We see also a strong impact of the limited length of the strategy: this is quite consistent with the success of direct policy search for very difficult problems, with compact representations for making this tractable (e.g. factorized MDP[145, 214]).

The case (*****) is interesting, because it contradicts some usually claimed results - only in appearance however. Consider a partially observable game, with a finite state space, but possibly unbounded number of steps, and games concluded by a win of one of the players (or possibly, an infinite loop, which is a draw). Then consider the usual formalization of decidability:

(UD): Existence of an almost sure win
Instance: a position. Question: is there a strategy for winning with probability 1 whatever may be the decisions of the opponent ?

This problem is termed (UD) for "usual definition". The problem is decidable, without restriction on the length of games, if there are two players. The main drawback of this formalization of decidability in games is that it is useless for choosing how to play. In fully observable games, deciding this question is enough for playing optimally; not in partially observable games. On the other hand, the advantage is that this question is usually decidable, except in some decentralized cases[127]. (UD) simplifies things by two aspects:

- there's no probability, as we only deal with sure win; we only consider non-determinism (i.e. arbitrary choice by the opponent).

[6] Humans, according to the so-called Church-Turing Hypothesis, are in the same category as computers.

- the opponent does not need any observability or recall, as he can just play randomly - his goal is just to win with non-zero probability.

A more difficult question is the following:

Value function approximation
Instance: a position. Question: is there a strategy for ensuring a win with probability $\geq \zeta$?

It has been shown in [162] that, for any fixed $\zeta \in]0,1[$, this harder question (and many variants of it) is undecidable in stochastic one-player games. We'll see that this holds also for deterministic (which means here: non-randomized) two-player games, with unlimited horizon. Consider a deterministic partially observable two-player game with finite state space, and the value function approximation problem:

Problem (VF): Value function approximation for fixed $\frac{1}{2} > \delta > 0$.
Instance: a game such that • either there is a strategy for winning with probability $\geq \frac{1}{2} + \delta$, • or there's no strategy for winning with probability $\geq \frac{1}{2} - \delta$. Question: is there a strategy for ensuring a win with probability $\geq \frac{1}{2} + \delta$ whatever may be the decisions of the opponent ?

Then, we claim the following result for a game with finite state, partial observations, infinite horizon, two players, non-randomized transitions:

Undecidability Theorem for the approximation of value functions: *Problem (VF) is undecidable.*

Proof: We simulate a probabilistic finite automaton (PFA) in a deterministic game. The PFA for which the approximation is proved undecidable in [162] contain only rational probabilities, and the input word is the sequence of actions. The reduction is as follows:

- Each node of the PFA is a node of the game.
- Each decision of player 1 is a reading of a letter in the input tape of the finite automaton.
- The main trouble in the reduction, is how to encode random transitions: this encoding is discussed below.

For this encoding, we just have to simulate the random nodes thanks to the opponent. This is done as follows. Consider a node in which the probabilities are as follows:

Probability	n_1/N	n_2/N	...	n_k/N
Next node	t_1	t_2	...	t_k

with $\sum_{i=1}^{k} n_i = N$.

Then we simulate it as follows:

- Player 1 chooses (privately[7]) a number $a \in \{0,1,\ldots,N-1\}$. Player 1 knows that he is in one of the nodes used in the simulation of random nodes.
- Player 2 chooses (privately) a number $b \in \{0,1,\ldots,N-1\}$. Player 2 knows that he is in one of the nodes used in the simulation of random nodes.
- Let $c = ((a+b) \text{ modulo } N)+1$.
- Go to node t_c; both players observe c.

We claim the followings:

- If one of the player plays randomly and uniformly, then c is uniformly distributed in $\{1,2,\ldots,N\}$.
- Define z the probability of winning for player 1 if both players (i) play uniformly when choosing a and b (ii) have an optimal strategy for other cases. We will show that z is indeed the probability of winning, whenever we don't have the restriction (i) above.
- Let's assume, in order to get a contradiction, that player 1 has an advantage in playing something else than the uniform distribution, i.e. can reach a probability of winning higher than z. Then, player 2 can cancel this advantage by playing uniformly. Therefore, player 1 can't do better than a probability z of winning, whatever maybe his strategy.
- Let's assume, in order to get a contradiction, that player 2 has an advantage in playing something else than the uniform distribution. Then, player 1 can cancel this advantage by playing uniformly when choosing a. Therefore, player 2 can't reduce the probability of winning of a to anything smaller than z.
- Therefore, at least one of the player has the same results by playing uniformly.

As a consequence, the answer to problem (VF) above is the same as for the version of the game in which c is randomly chosen, uniformly in $\{1,2,\ldots,N\}$.

This is enough for simulating the PFA as in [162], and therefore it shows the undecidability of problem (VF). □

A close look at the proof shows that we don't need the full generality of games: player 2 does not need any observation, and he can indeed play randomly and uniformly. Therefore, we have the two following corollaries:

- Problem (VF) is undecidable also if the opponent is restricted to a uniform random player.

[7] Privately means that the other player does not see the chosen number. This is where we need a partially observable game.

- Problem (VF) is undecidable also if the opponent has no observation.

The main consequence of this is that, in the general case, two-player partially observable deterministic games, which are decidable for the usual definition (UD), have no computable optimal player. This is illustrated by the following construction, for some $\delta \in]0, \frac{1}{2}[$:

- Consider a game G with probability of winning (in optimal play) either $\geq \frac{1}{2} + \delta$ or $\leq \frac{1}{2} - \delta$.
- Consider the game G' in which the first action is as follows:
  - either you play G as the first player;
  - or you play G as the second player.

Optimal play (within precision δ) implies deciding Problem (VF). Therefore, optimal play is not computable (even with limited precision δ), when the horizon is not bounded.

We can also see, by checking the proof in [162], that solving (VF) implies solving the halting problem; therefore, in terms of Turing degrees, problem (VF) is an oracle for $0'$ (think of $0'$-hardness if you're not familiar with Turing degrees) - it is at least as hard as solving the halting problem.

5.3 Computational and human complexity, observed through the game of Go and vision

The human visual cortex is very impressive; vision provides many tasks for which humans are incredibly strong in front of computers (see e.g. Fig. 5.14 for a clear example suggested by J.-M. Jolion).

The game of Go (see Fig. 5.4) is a very ancient Asian game, supposed to exist since 4000 years, or 2600 years, depending on the sources (incidentally it is perhaps less old than Senet, the ancestor of Backgammon). It is known for having a particular pedagogical effect, and is a big part of culture in many countries. There are countries with TV channels dedicated to Go and the main tournaments have extremely big prizes.

Go can be played with handicap: handicap 7 (abbreviated H7) means that the weaker player (traditionally playing as black) plays 7 times before white plays. Levels in Go are evaluated on the kyu / dan scale:

- 20kyu (abbreviated 20k) is the level of a beginner.
- Then, increasing levels have decreasing numbers: 19kyu, 18kyu, until 1kyu (abbreviated 19k, 18k, ..., 1k).
- The first level above 1kyu is 1dan. Roughly speaking, 1kyu means 0dan, 2kyu means -1dan, ..., 20kyu means -19dan. These dans are termed "amateur" dans and are abbreviated 1d, 2d, 3d...

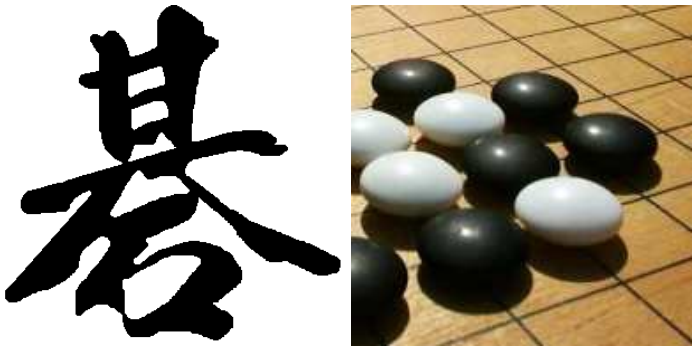


Figure 5.4: Left: the kanji of the game of Go. Right: a detail picture of a goban. E. Lasker, a world chess champion, said that "While the Baroque rules of chess could only have been created by humans, the rules of go are so elegant, organic, and rigorously logical that if intelligent life forms exist elsewhere in the universe, they almost certainly play go." Go is part of the "four arts": zither, board game, calligraphy and painting, since the Tang dynasty in China.

- The sixth or seventh amateur dan is roughly equivalent to the first professional dan. However, this depends on countries or even towns in some countries. Moreover, the professional dans (abbreviated 1p, 2p, 3p...) have the property that they never decrease: old players can be 9p whenever they do not have anymore the level of a professional player.
- The highest rank is 9p, except a few cases of 10p for honorific reasons.
- Some professional players are called "top pro". This means that they are 9p and have won recently one of the major open tournaments.

The amateur dans are supposed to be built so that when a player with level Ndan plays against a player with level Mdan, then the proper handicap is $N-M$ (but: $(N-M)/3$ for professional dans). Depending on the set of rules, a game of Go can be concluded by a draw or not. In some rare cases and for some rules (in particular the widely used Japanese rules), the game might never halt. Draws were considered as a good thing in the past; nowadays, due to tournaments and rankings, draws are made impossible by adapting the komi and by replaying games with a loop (see definition below).

The current best performances of computers are wins with H7 against a top pro, with H6 against a pro, with H2 against strong amateur players in 13x13, and with no handicap in 9x9.

In spite of simple rules (Go can be played correctly by a 4-years old child), Go has incredibly complicated tactics, mixed with important strategy concepts. It is often said than "Life is a game of Go in which rules have been made uselessly complicated". There are thousands of proverbs around Go and Go was among the four important arts for Chinese scholars: "qin" (now termed the game of Go), "qi" (a music instrument), "shu" (calligraphy) and "hua" (painting). According to Confucius (at least, the vision of Confucius usually accepted nowadays), Go is a better game than many other games,

but should not, nonetheless, take too much time. This is probably also true for computers: artificial intelligence learnt a lot by playing Go, but maybe we should not spend too much time on Go competitions.

We will here discuss some important families of Go problems for which the comparison with humans is particularly interesting. People who are not go players should not be afraid, this section is supposed to be readable by any reader.

5.3.1 The rules of the game: simple by induction and complex by deduction

It is amazing how the game of Go can be properly understood by 4-years old children, whilst being extremely complicated to formalize. In fact, it is much easier to teach the rules by playing a few games, than by explaining the rules. In fact, there are several sets of rules, which are usually equivalent, but one can find complicated cases in which it is not so obvious to know who has won for some set of rules.

This author prefers Chinese rules, because they are completely formalized (at least I believe there is a formalization which matches the intuitive idea of the rules), so that a computer can implement them; for Japanese rules and in some particular cases, it is still difficult to know who as won (even for humans), and some games have to be replayed differently in order to find a conclusion. See `http://senseis.xmp.net/?TrompTaylorRules` for pathological cases, and `http://senseis.xmp.net/?RulesOfGo` for a discussion on rules.

The rules are as follows (maybe you'll understand the rules faster on Fig. 5.5):

- The board is an $n \times n$ grid (usually, $n \in \{9, 13, 19\}$).
- Black plays against white. Black starts by putting a black stone, in one of the locations, then white plays by putting a white stone on an empty location, and so on.
- A **group** is a maximal connected set of stones (connected in 4-connectivity) of the same color. Each stone is in one and only one group (possibly limited to this stone).
- A **liberty** of a group is an empty location next to this group (in 4-connectivity).
- When black (resp. white) plays in the last liberty of a white (resp. black) group, then this white (resp. black) group **dies**; it is removed from the board, and it is allowed to play, in the future, at these locations. It is forbidden (in most rules) to reduce the liberty of our own groups to zero, except if by doing so we kill an opponent's group so that after the move the group has at least one liberty.
- ko-rule: it happens sometimes that a player captures a group limited to one stone, and the player might come back to the same situation at the next move, by capturing this stone again (see Fig. 5.9). This is forbidden by **ko**.
- superko-rule (not in all rules!): it is forbidden to come back at a board position which was already seen in the past. Depending on the version of the rule, it is

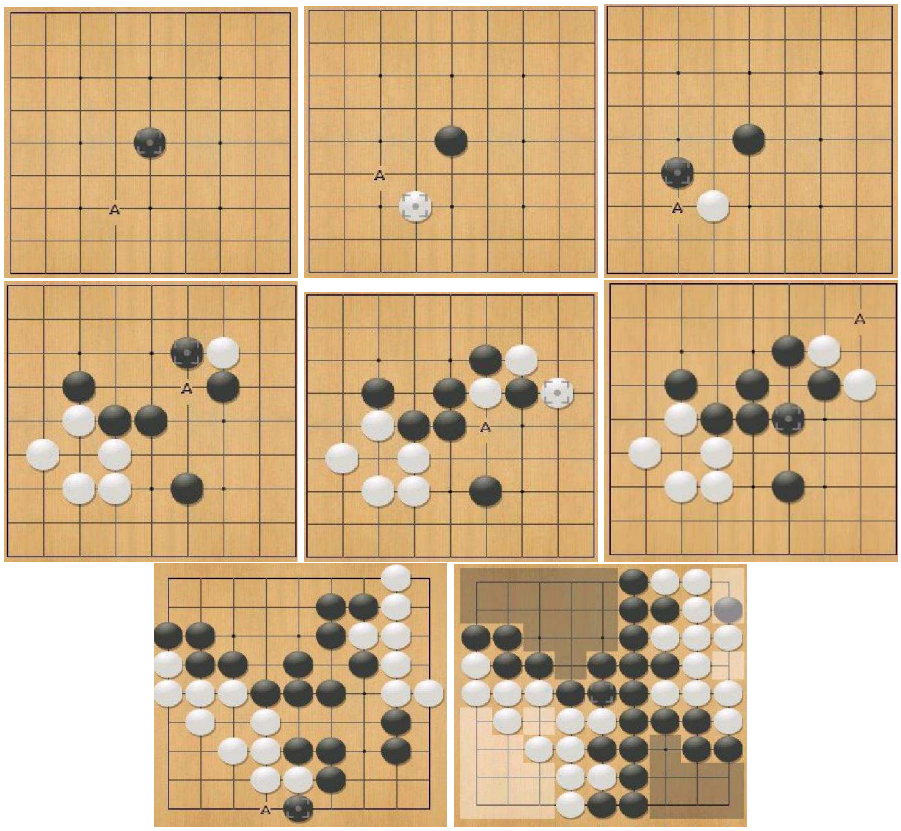


Figure 5.5: An example of 9x9 go game (not all intermediate situations are shown). Black starts, players play alternatively; black wins as its territory is bigger than the territory of white (white has a bonus of 7.5 because black starts, but this bonus is not sufficient here). The transition between the 5^{th} and the 6^{th} images shows a capture of a group of one white stone (by black playing in A).

forbidden to go to a same situation with the same player to play, or it is forbidden to go to a same situation with any player to play. This rule is not often involved in a game (yet, there are pro games with loops), but this is of crucial theoretical importance: with no superko rules, infinite games are possible.

- Instead of playing, a player can pass. If both players pass (3 passes are sometimes required), then the game is over. The score is then (in Chinese rules) for each player the sum of
  - the number of stones of his color on the board;
  - the number of locations surrounded by his stones;
  - for white, a **komi** is given as bonus: usually, 7.5 for Chinese rules in computer-Go.

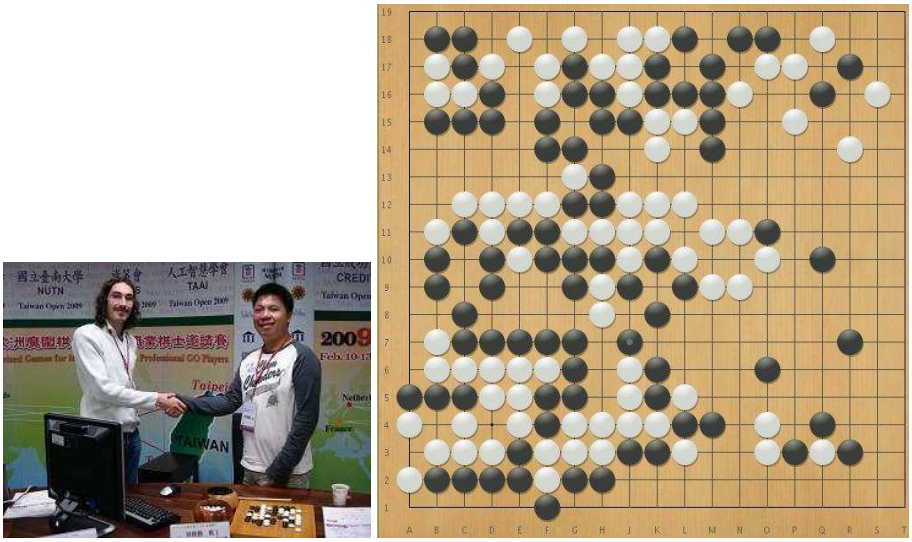


Figure 5.6: The first ever (and only) win of a computer against a top level human in 19x19 Go with handicap 7: MoGo won with H7 against a top professional player (C.-H. Chou, also known as Zhou Junxun, 9p, winner of the LG Cup 2007; game played in Taipei, 2009). The human operator on the left is Arpad Rimmel (Tao team, France); on the right the white group H4 is dead (and white groups in the neighborhood as well). In 9x9, MoGo won the first games against professional players in 2007; in 2009, Fuego (a Canadian program) won against a top player as white (the easiest side, in case of komi 7.5) and MoGoTW won against a top player as black, so that now computers have won even in the most difficult setting in 9x9 Go.

> Usually, players stop before the complete end of the game, as they guess the result without spending time for killing all the stones which will obviously die soon.

In Ponnuki-go (a simplified version), the first player which captures at least one stone has won. There is equality if both players pass[8]. Therefore, there's no ko, no superko, no komi, no count of points. There are also variants of the rules allowing suicide.

5.3.2 Best performances against humans

Go is scalable in the sense that it can be played in various sizes; the most standard sizes are 9x9, 13x13 and 19x19. Go is exactly solved until size 5x5. The best performances against humans are as follows:

- In 19x19 Go (the standard size), the current best performance of a computer against a top professional player is a win with H7 (Fig. 5.6). Zen performed a

[8]Pass is sometimes forbidden in Ponnuki, and in that case the first player who cannot play looses the game.

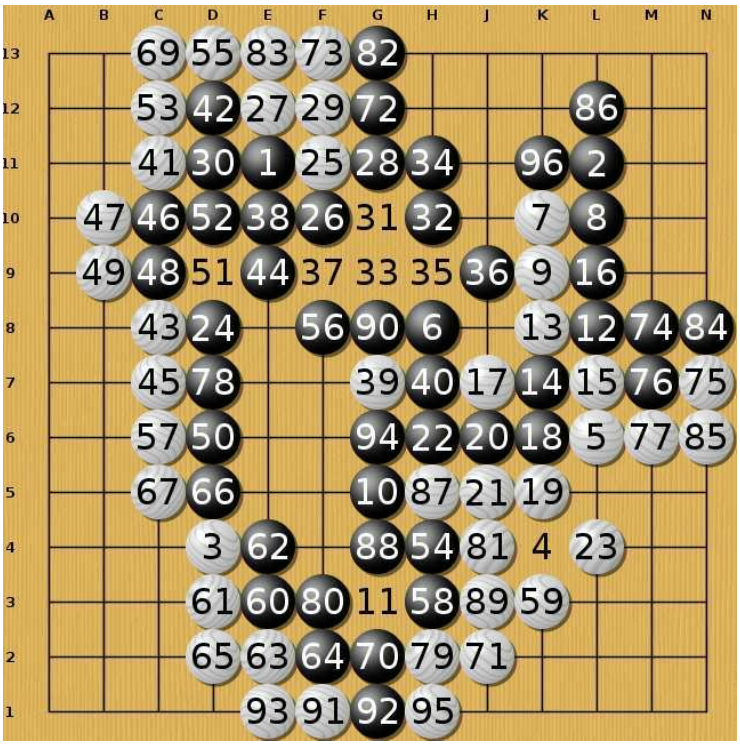


Figure 5.7: The first ever win of a computer against a 6D player with handicap 2 in 13x13 Go.

win with H6 against a 4P professional player in WCCI2010 (Barcelona).

- In 13x13, our program MoGo performed the first ever win against a 6D player (Shi-Jim Yen 6D) at WCCI2010, in Barcelona, with handicap 2. A few hours later, another program, Many Faces of Go, did the same, and then lost a rematch against a 6D; MoGo won a second game in the same conditions (win against a 6D player, Shang-Rong Tsai, in 13x13 with handicap 2). The first win is presented in Fig. 5.7. The second win of MoGo in the same conditions involves a nice ko-fight.

- In 9x9, MoGo was the first program which won against pro; it had several such wins. Fuego also won some games against pros. MoGoTW has won both as black and as white against a top professional player, and won several games as black against pro; no other program ever won a game as black against a professional human.

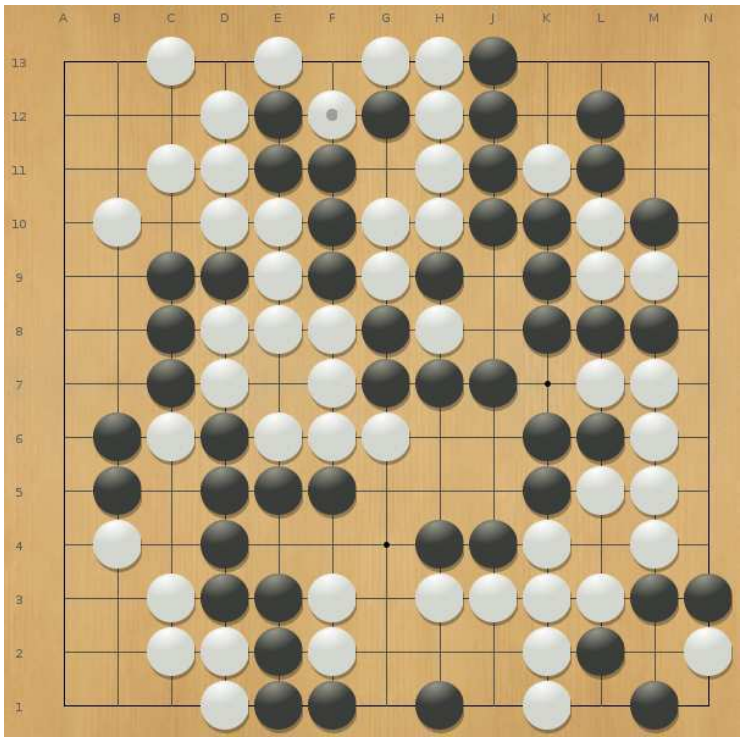


Figure 5.8: The second win of MoGo against a 6D player in 13x13 with handicap 2. At this point, MoGo (black) used a threat in B2; white can only win the ko by playing in the upper part (G11), otherwise black wins the game by killing the big white group H13; so white does not reply to the threat, MoGo captures the lower left corner and black (MoGo) wins. This shows that bots can sometimes play ko fights correctly. An example of ko-fight poorly played by a MCTS is shown in Fig. 5.15.

5.3.3 Ishi-no-shita and Nakade

To the best of our knowledge, the complexity of "Ishi-no-shita" (Fig. 5.10), i.e. tsume-gos in which captures and recaptures occur inside a first capture, is not known. This is an interesting question as it might be part of a more general question: which complexity classes are easy for humans and which are not ? To the best of our knowledge:

- The reading (solving) of Ishi-no-shita is very difficult and somehow unnatural for humans;
- Computers are not disturbed by the strange structure of these situations and are particularly strong in this case (according to `intelligentgo.org`).

Nakade are a particular case in which a player builds a group A inside an opponent group B; A will be killed, but the liberties of B will be reduced by the stones used for killing A so that B will be dead after a few iterations. Nakade situations are not

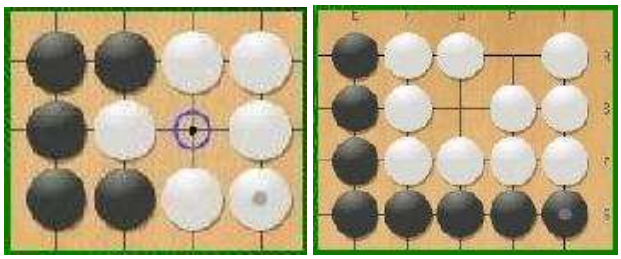


Figure 5.9: Left: An example of ko situation: if black captures the white stone by playing the circled location, then white is not allowed to immediately play the symmetric location - this avoids the simplest infinite loops. Right: an example of alive groups: it is not possible for black to kill the white group, due to the two empty surrounded locations (black could kill white if there was only one eye, by the "killing" rule, but with two eyes playing in one of the eyes is illegal (it's a suicide).)

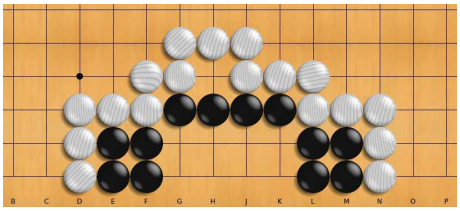


Figure 5.10: A beautiful ishi-no-shita from Denis Feldmann's web page; this problem published in Kido (a strong Japanese go magazine) in 1996 ends in a parallel double snapback. Black plays and saves some stones.

easily handled in MCTS unless they are the only fight (as usual, MCTS has troubles for mixing the solutions of several simultaneous fights).

5.3.4 Ladders

A ladder is a situation in which a group has one and only one liberty and repeatedly escapes by adding one liberty (see an example in Fig. 5.11). It has been shown in [77] that guessing if the group which tries to escape will escape or not is PSPACE-hard. There are examples of famous games played by computers in which programs made the mistake of playing bad ladders; nonetheless, this is rare and probably not the main weakness of programs in front of humans.

5.3.5 Semeais

A semeai is a fight between two groups: the white group can only live by killing the black one, and the black group can only live by killing the white group. Two examples are given in Fig. 5.12 and 5.13. They are easily solved by an amateur, whereas the strongest programs have difficulties for solving this: the reason for this is that strong

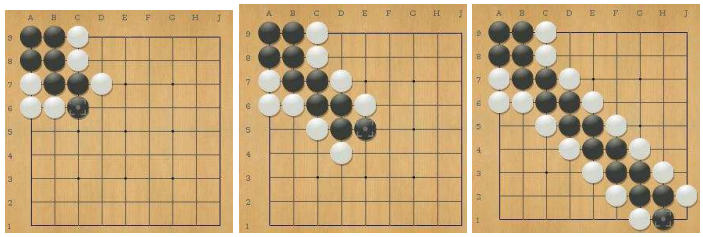


Figure 5.11: An example of ladder: white plays C5, black escapes with D6, white attacks with E6, and so on - at the end, the black group will necessarily die.

programs are based on MCTS, and MCTS only knows the rules through simulations; they don't know that all liberties of the black group are equivalent, and all liberties of the white group are equivalent, and therefore they will only know that the semeai is won by the first player attacking the opponent's group once they have simulated the $(m!)^2$ possible nodes, for a semeai with m liberties per group. This is probably the main limitation for MCTS programs in Go.

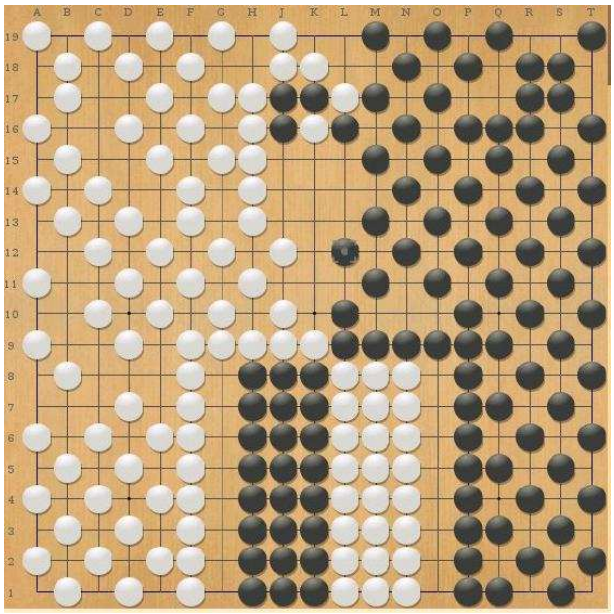


Figure 5.12: A semeai, white to play. White can kill the big black group J4 (and make its own group M4 live) just by filling its 8 liberties G1-G8. If white does not play in the semeai, black can do it and kill the white group: therefore, in this example, white must play G1 or G2 or ... or G8 and nothing else.

Semeais are particularly interesting situations as they cumulate the following properties:

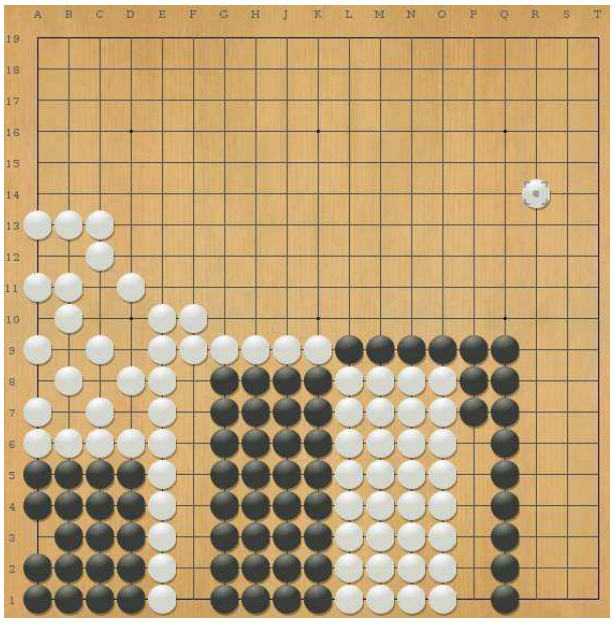


Figure 5.13: Another semeai. Black can kill the white group, but it's not urgent to play in the semeai: black is in advance of two liberties and should therefore play somewhere else. Playing P1 or P2 or ... or P6 is here a big mistake for black.

- The MCTS algorithms, which perform incredibly well in many games and in particular in Go when compared to other programs, are very weak (when compared to humans) on very simple and very typical semeais.
- However, what is difficult is not solving the semeai: it is to see its impact on the game. Implementing a semeai solver is possible, and it will predict correctly the answer to the following question in many cases: is it possible for white to ensure that the white group will live ? But the answer to this question is not as important as knowing whether it is worth trying to win it, or knowing what will be the effect of winning this semeai on other parts of the game. [36] clearly shows that current programs don't play semeais correctly.
- Analyzing a semeai involves some "geometric" elements; this suggests that humans, who are very strong for recognising shapes, are particularly strong for semeais thanks to this "geometry" ability (for an example of human visual skills, see Fig. 5.14). Humans guess, on a semeai, that some liberties are similar and know that it is therefore useless to analyze permutations of sequences which are obviously equivalent to the initial sequence. This idea of "grouping" similar actions is, in the humble opinion of this author, a key for solving semeais - yet, nobody successfully solved this.

It is interesting to point out that other games difficult for computers often involve

visual or spatial elements; typically, connection games are difficult for computers in front of humans (see the case of Havannah later - interestingly, connectivity was cited as one of the difficult problems for artificial intelligence in the very early papers around neural networks [167]).

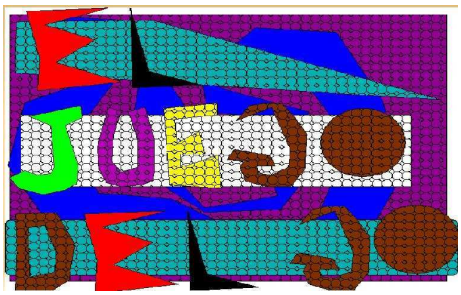


Figure 5.14: An example of task which is easily solved by a human and not by a computer: people who speak Spanish can easily read this, computers (Spanish or not...) don't. Thanks to J.-M. Jolion for the idea of this example, twelve years ago.

5.3.6 Small yose and big yose

The small yose consists in choosing between the last important locations of the goban which one must be played first; each location has several characteristics:

- the number of points you win by playing in it before your opponent;
- the fact that the opponent is forced to play around the same location immediately after your move ("sente" case) or not ("gote" case);
- possibly, a location forbids other locations for the opponent ('cutting') or ensures other possibilities for yourself.

It has been shown in [248] that this is PSPACE-complete, by a proper formalization of "small yose", and a reduction to a sum of local games. This was one of the first cases (subcases of the general Go game) in which humans were outperformed by humans: it is more combinatorial than visual. On the other hand, programs are weak in the so-called "big yose"[9]: they don't guess which shapes will lead to a better situation for the later "small yose".

5.3.7 Ko fights

When a ko occurs (see Fig. 5.9, left), a player cannot kill a stone *A* because it is not allowed to come back at the same situation. In order to capture the stone, the player must play a move elsewhere in the board, say in location *B*, at a place in which the

[9]This author is grateful to Li Yue, a strong amateur player who accepted to play (and won) against MoGo with handicap 6 and pointed out these elements.

other player must reply locally to *B* (or will suffer a big loss of territory); this forces the opponent to play around *B* instead of securing the stone *A*. Thanks to the threat around *B*, the stone *A* can be captured as this does not go back to an earlier situation.

This can become much more complicated when multiple kos are involved.

An important property of ko fights is that they make Go EXPTIME-complete; precisely, Go with Ko and without superko (i.e. Japanese rules) is EXPTIME-complete[194]. [10] It is often said that MCTS algorithms are not very strong at ko-fights; for example, MoGo lost an opportunity of winning with H9 against a 5P player in Paris in 2008 (Fig. 5.15); nonetheless MoGo successfully played a ko-fight against a 6D player in 13x13 with H2, as shown in Fig. 5.8.

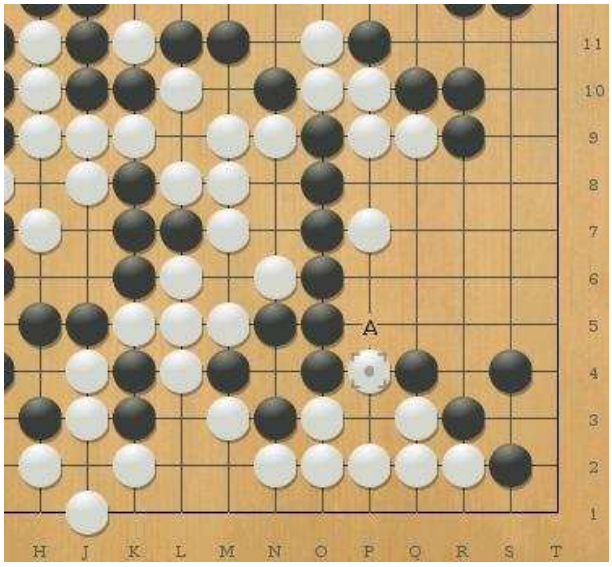


Figure 5.15: MoGo as black played P5 against Catalin Taranu (5p) and lost in spite of 9 handicap stones. The game was good for black until this move; with Q5 instead of P5, there was no ko fight and MoGo would probably have won.

It is often said that MCTS algorithms are weak in ko fights; the game lost by MoGo in 2008 with handicap 9 against Catalin Taranu (5p) confirmed this (see Fig. 5.15). Nonetheless, after some corrections, this is seemingly less and less true nowadays: ko fights are probably not the main weakness of MCTS algorithms now[11].

5.3.8 Go openings

In a game, the opening is the part which involves the biggest amount of long term understanding. The only solutions for finding openings in Go are (i) handcrafting (a not

[10] It is not proved, however, that Go without ko is not EXPTIME-complete; it is also shown that it is PSPACE-difficult.

[11] This author is grateful to Li Yue for pointing out an example of good behavior of MoGo in a ko-fight when playing (and loosing) against her with handicap 6.

very satisfactory solution, yet it is in some sense used by human players who learn by watching and learning games) (ii) self-built opening book (similar to a human player who learns by playing against himself) (iii) opening books learnt by playing against strong players (this is not handcrafting, and this is more satisfactory). Computers with none of these tools often play bad opening moves (see Fig. 5.16), suggesting that even in 9x9 Go, computers have a lack of strategical understanding.

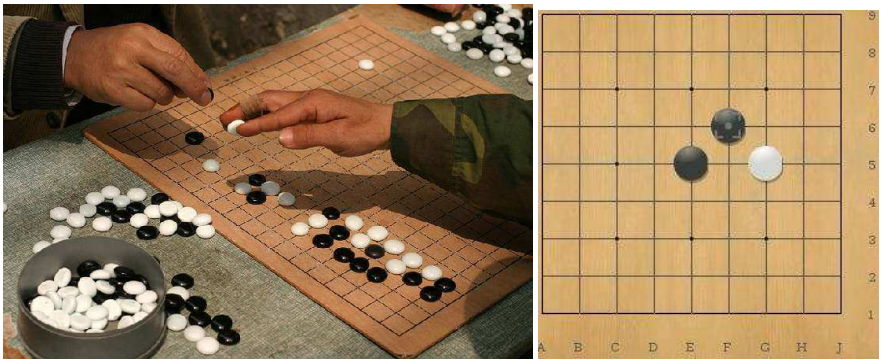


Figure 5.16: Left: a Go opening in 19x19 Go (with the traditional stone handling, picture from Wikipedia). Right: a known bad black opening in 9x9 Go with komi 7.5, which is often considered by computers (analyzed by MoGo during many years of computation in its self-built opening book, and played by Fuego in a game lost against C.-H. Hsou 9p).

5.3.9 Phantom Go and partially observable cases

As briefly discussed earlier, the complexity classes of partially observable (PO) games are usually very high. However, this is for the worst case on partially observable games: what about more natural PO cases ? A wide family of PO games is the family of phantom games: for example, in phantom-Go, players are not informed of the moves chosen by the opponent; they just know (i) when their own stones are captured (ii) when they have proposed an illegal move (then, they should choose another move). Each game can be converted into a “phantom” version. An interesting feature of those games is that algorithms performing best for this are essentially tuned heuristics (often using a Monte-Carlo sampling of possible hidden states as a key feature); this is somehow in accordance with [2] which suggests to use direct policy search (i.e.: defining a parametric family of strategies and optimize the parameters by non-linear stochastic optimization) for these problems with huge complexity and for which approaches based on function values cannot work. Whereas the control of a single plane (in take off, landing, or standard flight), with little unavailable information, is an easy task, the control of a flight of drones in front of unobservable enemies is hard (Fig. 5.17).

Figure 5.17: Top: controlling a single plane in observable “quiet” settings is easy for optimal control, and there are also good results for controlling a car in simple environments (see the Darpa challenge, in which a car must reach a point B from a point A without driver), but not in a town; similarly, a bot can walk in easy environments, but bots are still far from the flexibility of humans or animals. Bottom: controlling a flight of drones against unobservable enemies is difficult (here a Kzo drone (left) and its launching (right), from armyrecognition.com). Interestingly, this kind of problem is quite close to so-called ”real-time strategy games”.

5.4 Summarizing all this

We’d like to give a brief, partial, moderately precise, overview of all the stuff above in this chapter around one-player or two-player games. Roughly, we conclude as follows:

Partially observable games are much more complex and often solved by heuristic methods. Many natural questions in PO cases are indeed undecidable.

MCTS is a very strong tool in observable games, when no evaluation function is known.

When nothing works (e.g. difficult PO case), the best solution is often the design of a parametric policy to be optimized by direct policy search.

The computational complexity is not a good measure of "human" complexity or of the difference between humans and computers.

Computational complexity provides tools for generating difficult positions of a given game.

Chapter 6

From Dynamic Programming and Alpha-Beta to Monte-Carlo Tree Search (MCTS)[1]

The first section below introduces MCTS in a slow manner, comparing it to the state of the art and to the historical tools. Then, a more formal presentation is given (section 6.2), with an analysis based on the memory requirements in MCTS - based on more realistic formulas than in analysis using the UCB-like terms for exploration.

6.1 A pedestrian introduction to Monte-Carlo Tree Search

Minimax is a naive algorithm for games; it consists in computing recursively the value of a max-node as the max of the values of its sons, and the value of a min-node as the min of the values of its sons. As exploring the whole tree of possible futures is infeasible, the value at a fixed depth is approximated by a function termed the **evaluation function** (which approximates the value function). The evaluation function is usually designed by experts of the game, when experts can formalize their knowledge. Roughly speaking, Minimax is analogous, in games, to dynamic programming for one-player games (the category "one player games" include planning).

Alpha-beta is the most classical algorithm for games. It is essentially a fast form of Minimax: useless branches are pruned. When only a small prior knowledge is available, Alpha-Beta is "only" twice faster than Minimax (more precisely if the exploration is made in random order, the speed-up is two); see Fig. 6.1 for an illustration of the algorithm.

Modern versions of alpha-beta are often much more "depth-first" and anytime than

[1]This chapter is based on many things I learnt when I was in Artelys (`http://www.artelys.com`) and in joint works with all my coauthors around games.

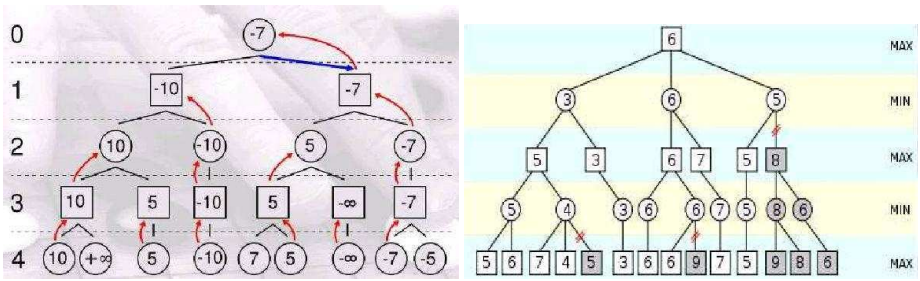


Figure 6.1: Left: minimax (see Alg. 1). Right: alpha-beta (see Alg. 2), which is equivalent but faster by cutoffs: the branches which will provably provide a reward lower than what another branch has already produced are stopped.

traditional versions (see e.g. iterative deepening [144], and singular extensions[3]). MCTS is, in some sense, an extremal case in this direction.

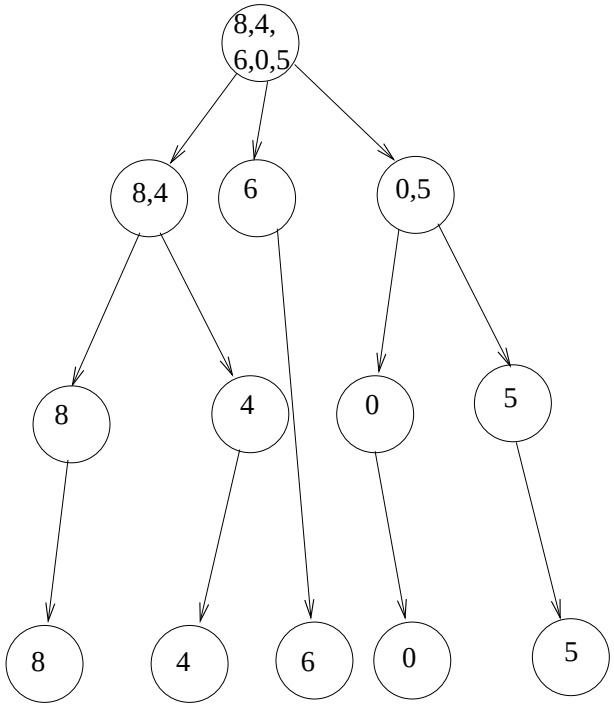


Figure 6.2: Example of MCTS tree. Each circle is a state, the current tree (in memory) is this set of states. Each circle contains a family of numbers: for example, (8,4,6,0,5) in the root node means that 5 simulations have crossed the root, and the rewards associated to these simulations were 8,4,6,0 and 5 respectively.

MCTS is briefly sketched as follows (see Fig. 6.2). The main data structure is

a tree[2], and the program is a loop which simulates one game (starting at the current simulation) at each iteration in the loop. The tree grows up at each simulation. At first, there is only one node in the tree, which represents the current situation. At each iteration:

- **Simulation step.** (a more formal presentation, by pseudo-code, can be found in Alg. 3) A game is simulated from the current situation until the game is over (we will see later how moves are chosen in these simulations). When we perform a simulation step, we have a tree of situations in memory, the root of which is the current situation; the simulated game crosses nodes of the tree, and then situations out of the tree: there are therefore two parts in the simulation step, namely **(i) simulation in the tree (ii) simulation out of the tree**. Depending on implementations, a given number (usually one, the first out of the tree) of these situations out of the tree are added in the tree, with edges reflecting transitions.

- **Update step.** Then, statistics are updated: in each node, we keep track of each result of a game crossing the situation represented by this node. In the figure, there has been 5 simulations; the root has been visited in all 5 simulations, with rewards 8, 4, 6, 0, 5. The nodes below the root have been seen twice, once, and twice respectively: there might be some legal moves which have not (not yet) been explored at all.

The main point is how to perform the simulations. In each situation, the simulated move is chosen as follows for a max node (a similar algorithm with negated rewards is used for min nodes):

- For each legal move, a score is computed. For example, in the Upper Confidence Tree [143] version of MCTS, the score for a transition $a \to b$ (from node a to node b) corresponding to a move m is the sum of:
  - The average reward for this move (a default value can be chosen for non-simulated moves);
  - $k\sqrt{\log(n_a)/n_b}$ where n_a is the number of simulations through a and n_b is the number of simulations through b; and k is a parameter (to be tuned). Some arbitrary choice can be made if $n_a = 0$ or $n_b = 0$.

  For example, in Fig. 6.2, the score of the transition from the node $8,4,6,0,5$ to the node $8,4$ is
  $$\frac{8+4}{2} + k\sqrt{\frac{\log(5)}{2}}.$$
  It is usually preferred to multiply k by the variance of the rewards; this is termed "tuned-UCT" [11]:
  $$\text{average reward} + k\hat{\sigma}(rewards)\sqrt{\log(n_a)/n_b}$$

[2]We here present MCTS in the case of a tree, but working on a DAG (directed acyclic graph) is very similar - using trees just makes the writing simpler.

but it's essentially useless in MCTS for Go; this comes from other frameworks than MCTS (namely stochastic bandits[148]), and might be relevant for MCTS in noisy environments. There are many other similar formulas [148, 13], termed "bandit" formulas.

- Then, the move with maximum score (ties are randomly broken) is simulated.

For random nodes (if the problem is stochastic), the simulated move is chosen randomly according to the distribution of possible moves.

In the case of Go, as many improvements have been proposed, the formula is much more complicated, and incidentally the constant k (ensuring exploration of weakly explored nodes) is zero or close to zero. Roughly speaking, the formula of the score, for a max node, is in usual optimized implementations for the game of Go (and probably for many deterministic two-player zero-sum games) the weighted sum of three terms:

- **Off-line value:** a value function based on patterns and/or expert rules.
- **On-line value:** the average reward.
- **Transient values:** transients values are estimates of the quality of a move, based on previous simulations *including those which did not use this move in this situation*; the most well known transient value is based on "Amaf" (all moves as first): it's a value function based on permutations of simulations [44, 112]; roughly, for a max node, instead of the average of the rewards of simulations containing the transition from a to b thanks to move m (as in the on-line version), we consider the average reward of simulations crossing a and containing action m in *any node later than a*. "Amaf" values within MCTS algorithms are termed "Rave" values (Rapid Action Value Estimate).

The coefficients of these three terms verify the following property[152]:

- When n_b and n_a are small, then the first term (off-line value) is the most important;
- When n_b increases, then transient values become more important;
- When $n_b \to \infty$, the on-line value has weight going to 1 and other values go to 0.

A typical formula might be

$$\text{maximize } c_{offline} V_{offline} + c_{transient} \widehat{\hat{V}} + c_{online} \hat{V} + c_{exploration} \tag{6.1}$$

where $\hat{V}$ is the average reward for simulations containing the considered move in the given situation (we'll give more formal definitions in section 6.2), $V_{offline}$ is some a priori evaluation of the move (e.g. by patterns[72] or rules[152]) and $\widehat{\hat{V}}$ is the transient

value. Constants can be chosen e.g. as follows:

$$c_{offline} = k_{offline}/(2+\log(n_b))$$
$$c_{transient} = K/(K+n_a)$$
$$c_{online} = n_a/(K+n_a)$$
$$c_{exploration} = k\sqrt{\frac{\log(n_a)}{n_b}}$$

In our program MoGo (dedicated to Go, which is a binary deterministic game), $c_{exploration} = k = 0$, but in stochastic cases $c_{exploration} = 0$ would probably be a bad idea.

The first term ($V_{offline}$) needs some work in terms of implementation; the third one is obviously central in the algorithm; the second one is easy to code and quite efficient for many applications far from Go (see [221, 5, 100]). Preliminary works ([61]) suggest that MCTS is also quite efficient in stock managements (Fig. 6.3).

Figure 6.3: Controlling a single production unit with little uncertainty is easy for computers. Yet, controlling many plants in front of big uncertainties requires a lot of handcrafting. This problem is typically a case in which MCTS techniques might provide big improvements in the future.

As summaries are fun, let's summarize:

MCTS essentially consists in
(i) implementing a simulator with a default randomized policy
(ii) introducing a bias in the policy thanks to formulas like Eq. 6.1 (full-featured MCTS) or Eq. 6.4 (UCT).

It performs very well in Go, in General Game Playing, and in various high-dimensional problems in particular when the horizon is not too large and when no good evaluation function is available.
There are extensions in continuous cases and/or partially observable problems (with one player only).

6.2 Null exploration constants and frugality analysis

The UCB formula (Eq. 6.4 below) is often presented as the explanation for the success of MCTS, usually justified by the game of Go and some other games. We will here present another point of view: MCTS works much better (in Go and in many other deterministic cases) without the UCB term $+\sqrt{\log(\dots)/\dots}$; and some other formula are both (i) theoretically more properly funded for binary-valued games (ii) empirically more efficient. In particular, these formula save up a lot of memory (we term this property "frugality") - an important requirement in MCTS, in which the tree structure is often heuristically reduced (by removing parts of the tree which are not often used).

A fully observable game[3] is (in this section) a finite[4] set of nodes, organized as a tree with a root. Each node n is of one of the following types:

- max node (nodes in which the max player chooses the next state among descendants);
- min node (nodes in which the min player chooses the next state among descendants);
- terminal node; then, the node n is equipped with a reward $Reward(n) \in [-1,1]$;
- random node; then, the node is equipped with a probability distribution on its descendants.

In all cases, we note $D(n)$ the set of the children of node n. We will consider algorithms which perform simulations; the first simulation is s_1, the second simulation is s_2, etc; each simulation is a legal game. The index of simulation s_i is, by definition, i. Each simulation is, formally, a path in the game from the current situation until a leaf (i.e. until a game over). Each node n is equipped with:

- Possibly, some side information $I(n)$.
- A father $F(n)$, which is the father node of n; this is not defined for the root.
- A (unknown) value $V(n)$, that we want to approximate; this value is the Bellman value; it is known since [19] that $V(n)$, equal to the expected value if both players play optimally, is well defined.
- For each $t \in \{0,1,2,3,4,\dots\}$,
  - $n_t(n) \in \{0,1,2,\dots\}$ is the number of simulations with index in $1,2,\dots,t-1$ including node n, possibly plus some constant $K_1(n)$ [72, 63, 152]:

$$n_t(n) = \#\{i < t; n \in s_i\} + K_1(n). \tag{6.2}$$

[3] With one player, a game becomes a puzzle, or a planning task, or a control task. This presentation is not limited to games in the traditional sense.

[4] Handling the infinite case is possible thanks to progressive widening; instead of considering the maximum score over all children, consider the maximum score over the $g(m)$ first children of the father node f where m is the number of simulations of f. $g(m)$ can be e.g. m^β for some $\beta \in [\frac{1}{4}, \frac{1}{2}]$ ([241, 73]).

– $w_t(n) \in \mathbb{R}$ is the sum of the rewards of the simulations with index in $1,2,\ldots,t-1$ including node n, possibly plus some constant $K_2(n)$ (possibly taking into account expert knowledge or offline values, possibly using $I(n)$):

$$w_t(n) = \sum_{i<t; n \in s_i} reward(s_i) + K_2(n). \tag{6.3}$$

– If n is not the root, $score_t(n) = score(w_t(n), n_t(n), n_t(F(n)), I(n), t)$.

We then consider MCTS algorithms as in Algorithm 3.

Input: a game.
Possibly initialize $w_t(n)$ and $n_t(n)$ to some arbitrary values $K_2(n)$ and $K_1(n)$ (possibly using expert knowledge).
for $t = 0,1,2,3,\ldots$ (until no time left) **do**
  $s_t \leftarrow ()$ // empty simulation
  $s = root(game)$
  while s is not terminal **do**
    $s_t \leftarrow s_t.s$ // s is added to the simulation
    switch s **do**
      case max *node*
        $s \leftarrow \arg\max_{n \in D(s)} score(n)$
      end
      case min *node*
        $s \leftarrow \arg\min_{n \in D(s)} score(n)$
      end
      case *random node*
        $s \leftarrow$ randomly drawn node according to distribution at s
      end
    end
  end while
  $s_t \leftarrow s_t.s$ // s is added to the simulation
  Updates the statistics (n_t, w_t) in all nodes concerned by simulation s_t.
end for
$decision = \arg\max_{n \in D(root)} n_t(n)$ // decision step
Output: *decision*, *i.e.* the node in which to move.

Algorithm 3: A framework of Monte-Carlo Tree Search. All usual implementations fall in this schema depending on the *score* function. The decision step is sometimes a bit different, without strong influence on the results. The algorithm here chooses only the move at the root; it is supposed to be run at other nodes later by ad hoc restriction of the game (i.e. this algorithm is run each time we have to take a decision, with the game restricted (the root is the current situation).

UCT is the special case of Alg. 3 with the special formula

$$score_t(s) = \hat{V}_t(s) + \sqrt{\log(n_t(F(s)))/n_t(s)} \text{ if } F(s) \text{ is a } \max node \quad (6.4)$$

$$score_t(s) = \hat{V}_t(s) - \sqrt{\log(n_t(F(s)))/n_t(s)} \text{ if } F(s) \text{ is a } \min node \quad (6.5)$$

where $\hat{V}_t(s) = w_t(s)/n_t(s)$. (constants or other terms are often placed before $\log(i)$; this does not matter for us here); i.e. it uses UCB [148] as a score function. It clearly provides consistency for a finite set of actions per node; in the sense that asymptotically, the most often sampled actions will be optimal actions. Its variants like "progressive widening" [72], "progressive unpruning" [62], are consistent as well. Our purpose is precisely to consider cases different from UCT, and in particular the frugal versions - those which do not visit all the tree.

In [26] we have shown the following consistency result, which has the advantage of covering frugal versions as well, in particular those which are used in real-world implementations.

Theorem (Consistency of Monte-Carlo Tree Search.) *Consider a MCTS Algorithm as in Algo. 6. Assume that for all sons s, s' of a* max *node:*

$$n_t(s) \to \infty, n_t(s') \to \infty, \liminf \hat{V}_t(s) > \limsup \hat{V}_t(s') \Rightarrow n_t(s') = o(n_t(s)) \quad (6.6)$$

$$n_t(F(s)) \to \infty \text{ and } \liminf \hat{V}_t(F(s)) < V(F(s)) \Rightarrow n_t(s) \to \infty \quad (6.7)$$

Assume the same equations 6.6 and 6.7, with $\liminf$ *replaced by* $\limsup$ *and* $<$ *replaced by* $>$*, for* min *nodes. Then, almost surely, there exists t_0 such that*

$$\forall t \geq t_0, \arg\max_{n \in D(root)} n_t(n) \subseteq \arg\max_{n \in D(root)} V(n). \quad (6.8)$$

Proof: See [26].□

The theorem above can be applied with UCB-like bandit formulas, but also with other formulas, much closer to what is used in efficient implementations. For example, we can consider a score as follows:

$$score_t(s) = \hat{V}_t(s) = \frac{w_t(s)}{n_t(s)} \text{ with } K_1 > 0 \text{ and } K_2 > 0. \quad (6.9)$$

Eq. 6.9 is the special case of Eq. 6.4 with $k = 0$. The advantage of Eq. 6.9 is not consistency; consistency was already ensured by Eq. 6.4 (UCB-like formulas). Importantly, Eq. 6.9 could be modified in order to take into account RAVE values (the theorem above works with RAVE values as well), heuristic information or other improvements - the important point is the regularization of the winning rate in order to avoid scores like 0 - in such cases, we have both frugality and consistency.

[26] gives more details on formula which do or do not provide consistency and/or frugality; the main idea is that, **using regularization of the winning rate (even as simple as in Eq. 6.9), we can get rid of exploration terms which imply exploration of the whole tree - we can have both frugality (no exploration of the whole tree) and consistency (convergence to optimal moves).** This is very consistent with presentations in [175], where many important MCTS codes were discussed and were

based on a zero exploration constant. Incidentally, when UCT is referred to the basis of MCTS research, it must be pointed out that for deterministic binary games, which are the main application of MCTS, the typically UCT formula, i.e. UCB, is usually not used. Nonetheless, we strongly believe that UCB is a really important formula for other applications of bandits, and that UCT is very efficient, as presented in [143], for problems which have a stochastic component - incidentally, the derivation of UCB-like formulas is meaningless in noise-free problems.

Chapter 7

On the generality of MCTS: other impressive applications with partial observability and continuous action spaces[1]

The game of Go and other difficult games, including the framework of General Game Playing, are the main testbeds for Monte-Carlo Tree Search; they have shown the extreme efficiency of these methods for some applications. Nonetheless, there are other important applications. This author has been personally convinced of the strength of MCTS when testing it on expensive optimization (section 7.1). The application to active learning (section 7.2) is also convincing to me, and directly industrial applications have also been proposed (section 7.3). Our personal feeling is that MCTS is a great candidate for applications with high dimensionality of the state space, not too many time steps, and lack of stable evaluation function; future works on MCTS will show us its limitations (long horizons, big expert knowledge available, strong need for exploration, unknown transition function ?). This chapter is devoted to original applications of MCTS, including partially observable problems; these applications are very difficult, as shown by the many unsuccessful hours this author has spent on them with other techniques. They have been the main motivation, for me, for spending more time on it - they show that MCTS is important far beyond games.

A main strength of MCTS is how easy it is to implement it on a simulator. Once you have a simulator, which simulates what happens with a given policy, plus a (heuristic) policy, then implementing MCTS can be done very quickly just by the UCT formula.

If you have a simulator and a policy, you can do some Monte-Carlo simulations; you can plot some indicators such as the value at risk, i.e. the curve $p \mapsto Q_p$ where Q_p is defined by $P(\textit{reward} \geq Q_p) = p$. The exact probability P is not known but can be approximated by the results of the Monte-Carlo simulations. Then, if you implement

[1] This chapter is based on joint works with A. Auger, R. Gaudel, S. Gelly, P. Rolet, M. Sebag, F. Teytaud.

the UCT formula in order to bias the policy towards the optimum, then you'll see the value-at-risk provably converge to the value-at-risk of the optimal (on average) policy; this is

- User-friendly: you see the results of your simulations, during the process.
- Anytime: you can stop when you want.
- Easy to develop: the simulator and the heuristic are the expert knowledge, and all you have to implement is (i) the archiving of pairs (states,rewards) (ii) the UCT formula.
- Versatile: you can use the Value-At-Risk instead of the expected value.

We'll discuss briefly below some applications.

7.1 Monte-Carlo Tree Search for Expensive Optimization

Optimization is presented in details in chapter 10. Optimization can also be rephrased as a MDP:

- Choosing the next point x_{n+1} to be visited is making a decision;
- Then, the fitness value $y_{n+1} = f(x_{n+1})$ is chosen randomly (according to a distribution of probability on the possible fitness functions f, conditionally to past observations):
$$y_{n+1} \sim \mathscr{L}\left(f(x_{n+1}) | \forall i \leq n, f(x_i) = y_i\right). \tag{7.1}$$
- The new state s_{n+1} is then the list of visited points, and their fitness values (if the fitness values are available):
$$s_{n+1} = (s_n, x_{n+1}, y_{n+1}); \tag{7.2}$$
- The distance (or log-distance) to the optimum, or some other criterion, is then the opposite of the reward function:
$$\textit{reward} \sim \mathscr{L}\left(-\log \|x_N - \arg\min f\|^2 | \forall i \leq N-1, f(x_i) = y_i\right). \tag{7.3}$$

The prior knowledge required for implementing this is therefore a distribution of probability on possible fitness functions (this means that we consider a MDP with max nodes, in which we choose decisions, and random nodes, in which fitness values are drawn). Another possibility is to consider a worst case on fitness functions (this means that we consider a MDP with max nodes for our decisions, and min nodes for the choice of fitness functions).

Solving such a MDP involves huge computation times, and the implementation is a bit difficult because expensive optimization (as well as active learning) is usually not

formulated as a MDP; and therefore this is only worth the candle when the computation cost of the optimization function is so big that implementing an algorithm involving a lot of work plus an hour of computation per iteration is not ridiculous. Algorithms for solving such MDPs are discussed in section 13, as well as some other related tools for this problem.

The same examples of applications can be provided, for expensive optimization, as those of active learning in section 7.2. Other examples of expensive optimization from the OMD project (`http://omd2.wikispaces.com/`) include the design of rockets, planes, cars; when building a prototype is involved, the cost is sufficiently big for justifying such an elaborate approach as (i) rephrasing the problem as a MDP (ii) implementing MCTS (iii) apply it with big computational time. However, at the date of this document, these applications were handled with Gaussian processes or evolutionary algorithms, but MCTS has not been tested.

Fig. 7.1, from [196], shows the result on the classical sphere benchmark; the theoretical limit is known for this benchmark, and we could therefore check that, with a big but feasible computational power we can be close to the optimum. Importantly, the algorithm can run with other prior informations as well - yet, more tests on this would be helpful.

7.2 Monte-Carlo Tree Search for Active Learning

Active learning is defined in chapter 16; we here just show how it can be rephrased as a MDP, and therefore solved by MCTS. Active learning can be rephrased as a MDP:

- choosing the next question (i.e. a point to be labelled) to the oracle is making a decision;
- the observation corresponding to the question is the label of this point (for a randomly drawn target function); it is randomly drawn conditionally to the past observations, and this assumes that we have a distribution of probability on possible target functions;
- the next state is then the sequence of questions to the oracle, with their answers;
- the reward is then a measure of the quality of the learnt function.

This implies that we have access to a distribution of probability on possible target functions; max nodes are nodes in which we make decisions, and random nodes are nodes in which we randomly draw the labels[2]. Under these conditions, the MCTS approach is just optimal in the sense that for the given distribution on target functions, and on average, it will provide the optimal result asymptotically in the computational power.

This detailed rephrasing is not trivial and details can be found in [195]. This has been experimentally tested; the results are very good as soon as (i) the model[3] is right

[2] Importantly, the labels must be drawn randomly, with conditioning; everything is conditionally to known observations.

[3] The model here is the distribution of fitness functions.

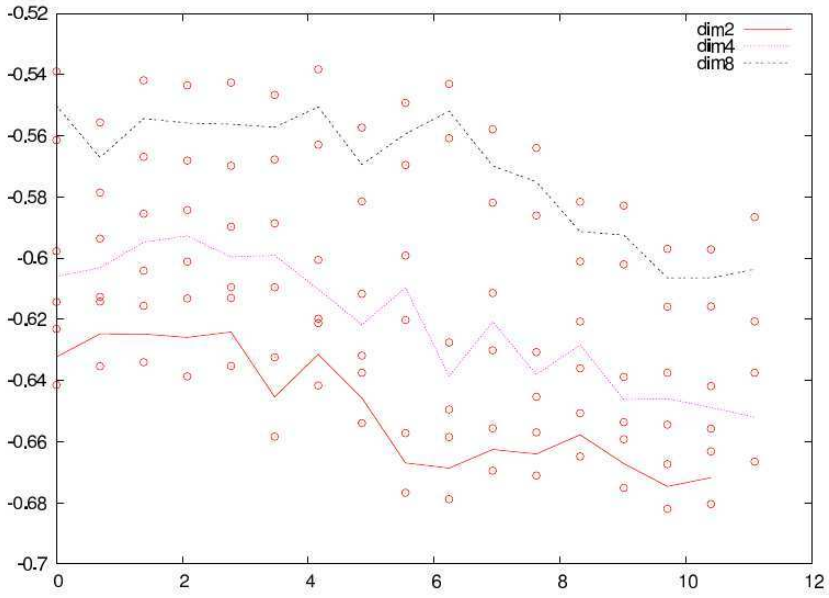


Figure 7.1: Results of expensive optimization by UCT (a specific form of MCTS, suitable for noisy cases) for horizon $N = 50$ with various dimensions - the goal of thes experiments (from [196]) is to show that MCTS can reach a nearly optimal behavior for optimization with $N = 50$ iterations. Error bars are $1/3$ of the standard deviation. The abscissa is the $\log_2$ of the number of simulations, the ordinate is the average of $\frac{d}{N}\log(||x_N \arg\min f||)$, supposed to reach $\log(2) = 0.69315$ asymptotically in N for an optimal algorithm (mathematical proof in [104] for comparison-based algorithms). The standard deviations are big (due to the huge computational cost of the algorithm), but we can nonetheless clearly show that we are close to optimality - in spite of many efforts we could not have such results with dynamic programming.

(ii) the computational power is not a limit. (ii) is ensured if the cost of labelling one example is huge; this is certainly the case if, for example,

- with x the architecture of a plane, $f(x)$ is the performance of the plane of architecture x in test flights;
- with x the architecture of a car, $f(x)$ is the performance of a car of architecture x in crash test (then, the cost of computing $f(x)$ includes the building of a prototype and its destruction against a wall!).

Unfortunately, MCTS has not yet been tested in these cases, but only on artificial active learning testbeds (Fig. 7.2). However, we'll see real applications in section 7.3 (far from active learning).

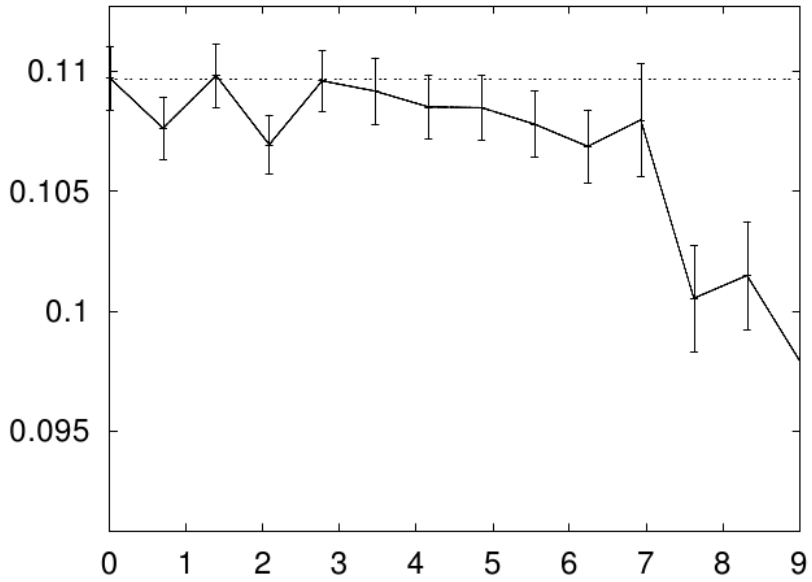


Figure 7.2: Performance of BAAL (Bandit-Based Active Learning[195]), which is UCT (a specific form of MCTS, suitable for noisy cases) applied to active learning in dimension 8: the abscissa is the $\log_2$ of the number of simulations of UCT, the ordinate is the generalization error, on a classical testbed, namely a random linear separator on a hypersphere. The dashed line is the performance of Query By Committee, a state of the art algorithm for active learning[105]. Query-By-committee is also used as a Monte-Carlo part of UCT - this is why BAAL is equivalent to QBC when only 1 simulation is performed. Provably, BAAL is equivalent to QBC for 1 simulation, and converges to optimal active learning as the number of simulations goes to infinity. More on this in [195].

7.3 Yet other applications, including industrial applications

An application to optimization on graphs, including an application to library performance tuning, was presented in [81]. The sequence of decisions is the decomposition of a big matrix, and the loss (negated reward) is the computation time of the calculus performed using the matrix. This shows the efficiency of UCT-like algorithms, in front of dynamic programming techniques in a real-world setting. This is a directly industrial application.

An application to feature selection, i.e. the problem of finding relevant variables in a dataset, has been proposed in [108].

There are various MCTS projects in the world, for various applications:

- In France and Taiwan, the IOMCA project[4] is aimed at (i) improving MCTS technologies (ii) extend its field of application (iii) compare it to classical methodologies (iv) test in on stock management, in particular in the ecologically and economically crucial problem of power management.
- Simon Lucas, Peter Cowling and Cameron Browne take care of an EPSRC project around games and beyond[161] (United Kingdom).

[172] shows an application to classical planning tasks; UCT was easy to use for these tasks and extremely competitive.

[4] http://www.lri.fr/~teytaud/iomca.pdf

Chapter 8

Parallel Monte-Carlo Tree Search[1]

This chapter, based on [36], will briefly summarize the parallelization of Monte-Carlo Tree Search. MCTS is presented in Alg. 3; essentially it consists in performing plenty of simulations, each of them updating a memory of nodes, each node representing a possible future situation and each node being equipped with statistics on rewards; the simulations are biased by the memory of nodes, as well as the memory of nodes is modified by the results of the simulations.

8.1 Multi-core parallelization

Multi-core parallelization is based on shared memory: there is only one memory, and all cores perform their own simulations and update the same memory. The resulting algorithm is summarized in Alg. 4; it is quite natural and is a kind of folklore knowledge. The main difficulty is that the updates of the statistics are performed concurrently, in a same hashtable[2] keeping all statistics; therefore, we need mutexes in order to ensure that no two cores write at the same address at the same time. The use of lockless hash-tables for accelerating the simulations has been proposed in [74, 98] for removing mutexes. On the other hand, for slow simulations (simulations with more expertise or with more learning), which perform best, the naive version is already quite efficient.

Amdahl's law and Gustafson's law, and their link with slow simulations

Amdahl's law[130] is an ultimate upper bound on the speed-up of parallel algorithms, depending on the percentage α of time spent in critical (non parallel) sections of the

[1] This chapter is based on joint works with V. Danjean, S. Gelly, J.-B. Hoock, Y. Kalemkarian, T. Hérault, A. Rimmel, F. Teytaud, Z. Yu.

[2] Many optimized implementations use a hashtable in order to avoid duplicate computations in case of two paths leading to the same situation (transposition tables).

Input: a game.
Possibly initialize $w_t(n)$ and $n_t(n)$ to some arbitrary values $K_2(n), K_1(n)$ (possibly using expert knowledge).
for $t = 0, 1, 2, 3, \ldots$ (until no time left) **do**
  on the first core with nothing to do
  $s_t \leftarrow ()$ // empty simulation
  $s = root(game)$
  while s is not terminal **do**
    $s_t \leftarrow s_t.s$ // s is added to the simulation
    switch s **do**
      case max *node*
        $s \leftarrow \arg\max_{n \in D(s)} score(n)$
      end
      case min *node*
        $s \leftarrow \arg\min_{n \in D(s)} score(n)$
      end
      case *random node*
        $s \leftarrow$ randomly drawn node according to distribution at s
      end
    end
  end while
  $s_t \leftarrow s_t.s$ // s is added to the simulation
  Updates the statistics in all nodes concerned by simulation s_t (see section 6.2).
end for
$decision = \arg\max_{n \in D(root)} n_t(n)$ // decision rule
Output: *decision*, *i.e.* the node in which to move.

Algorithm 4: The multi-core (multithreaded) version of MCTS. Compare to Alg. 3.

code. If the number of computation units is p, then the maximum speed-up is

$$1/(\alpha + (1-\alpha)/p).$$

The derivation of Amdahl's law is as follows:

- Let T be the computation time required by a single computation unit for doing all the job.
- The part $1-\alpha$ of the computation time will need time at least $T(1-\alpha)/p$.
- The other part, namely $T\alpha$, can't be parallelized.
- The overall time is therefore at least $T' = T(1-\alpha)/p + T\alpha = T(\alpha + (1-\alpha)/p)$.

- The speed-up is then $T/T' = 1/(\alpha + (1-\alpha)/p)$.

When $p \to \infty$, we get a limit speed-up at most $1/\alpha$. Gustafson's law[121] is a version of Amdahl's law which takes into account the fact that in some problems, $\alpha \to 0$ as bigger problems are considered. In our case, $\alpha \to 0$ means that simulations are slower, so that the critical section (updates of the tree) becomes negligible. We below test if the slow simulations are sufficiently slow to see an effect of Gustafson's law.

"Fillboard" is a modification of the Monte-Carlo part of the algorithm which strongly improves the results in 19x19 Go[152], at least when the overall number of simulations is large enough. This modification makes the algorithm slower, but nonetheless We tested two cases (i) without fillboard (ii) with fillboard (slower). Both are tested both in 9x9 and 19x19. Results are presented in Table 8.1.

Number of cores	Rate per core	Number of cores	Rate per core
19x19 with fast sims		9x9 with fast sims	
1	4226 sims/s	1	14757 sims/s
2	3572 sims/s	2	11211 sims/s
4	3520 sims/s	4	9519 sims/s
8	3528 sims/s	8	9963 sims/s
19x19 with slow sims (fillboard)		9x9 with slow sims (fillboard)	
1	3744 sims/s (-11.4 %)	1	13760 sims/s (-6.7 %)
2	3192 sims/s (-10.6 %)	2	10612 sims/s (-5.3 %)
4	3108 sims/s (-11.7 %)	4	9143 sims/s (-4.0 %)
8	3135 sims/s (-11.1 %)	8	9571 sims/s (-3.9 %)

Table 8.1: We show the computation cost of fillboard depending on the number of cores; the cost is the decrease in rate (between parenthesis). In 9x9 it decreases when the number of cores is bigger, and we'll see that the same phenomenon occurs in 19x19 with bigger numbers of cores in next experiments. These results are obtained on 8-cores AMD 2.6 GHz.

We see that the cost of expensive simulations is smaller for larger numbers of cores: it decreases from 6.7% to 3.9 % in 9x9. On the other hand in 19x19 the loss remains constant. We then tested on a machine with much more cores, a Power6 with 32 cores and hyperthreading. Results are given in Tables 8.2 and 8.3.

We see that the limit in terms of number of simulations per core is the same for fast and for slow simulations (compare with section above): the ultimate limit in terms of simulations per core is the same with slower simulations. This means that we are in Gustafson's regime. We conclude essentially that the speed-up is very good in terms of simulations per second, at least for machines with good bandwidth like Power-6 machines.

Nb of threads	Nb of simulations per second	Nb of simulations per second.thread
19x19 board, fast sims		
1	2456.96	2456.96
4	9515.59	2378.9
7	16336	2333.72
10	21846.6	2184.66
13	26696.6	2053.59
16	31578.3	1973.65
19	34858.3	1834.65
22	38708.5	1759.48
25	41197.1	1647.89
28	43156.3	1541.3
31	42492	1370.71
34	42475.8	1249.29
37	40864.1	1104.44
40	43437.6	1085.94
43	41785.4	971.753
46	41431.5	900.684
49	40255.4	821.539
52	43286.7	832.436
55	40434.4	735.172
58	40862.1	704.518
61	42249	692.606
64	41823.6	653.49

Nb of threads	Nb of simulations per second	Nb of simulations per second.thread
9x9 board, fast sims		
1	8236.61	8236.61
4	27883.5	6970.88
7	48609.8	6944.25
10	58385.1	5838.51
13	66869.2	5143.78
16	76966.6	4810.41
19	78957.8	4155.67
22	83167.2	3780.33
25	82172.1	3286.88
28	78113.6	2789.77
31	77237.4	2491.53
34	71657.6	2107.58
37	75225.7	2033.13
40	69419.1	1735.48
43	68011.4	1581.66
46	58986.6	1282.32
49	65165.6	1329.91
52	55104	1059.69
55	53243.1	968.056
58	54080.6	932.424
61	51873.8	850.39
64	50229.1	784.829

Table 8.2: Simulations per second, in 9x9 and 19x19. We see that hyperthreading seemingly does not improve the results (the rate is never better than with 32 threads). We see also that the speed-up is limited for these very fast simulations; however, for more efficient simulations (which are heavier, i.e. slower), results are much better (Table 8.3).

8.2 Message-passing parallelization

The extension to message-passing parallelization is crucial, as clusters provide much bigger computational powers than multi-core machines. The message-passing parallelization is less straightforward and therefore several solutions have been proposed, solutions to be compared experimentally. This topic is somehow controversial, as different papers provided different conclusions; we here give our experimental results.

The various published techniques for the parallelization of MCTS are as follows:

- *Fast tree parallelization* consists in simulating the multi-core process on a cluster; there's still only one tree in memory, on the master, and slaves (i) compute the Monte-Carlo part (ii) send the results to the master for updates. This is sensitive to Amdahl's law, and is quite expensive in terms of communication when RAVE values are used[130, 109].

- *Slow tree parallelization* consists in having one tree on each computation node,

Nb of threads	Nb of simulations	
	per second	per second.core
19x19 board, slow sims (fillboard)		
1	1616.3	1616.3
4	6502.34	1625.58
7	10795.2	1542.17
10	15284.6	1528.46
13	18455.6	1419.66
16	23202	1450.12
19	24644.8	1297.1
22	27066.8	1230.31
25	30841.9	1233.67
28	33485	1195.89
31	36729.4	1184.82
34	37218.3	1094.65
37	40093.4	1083.61
40	41659.6	1041.49
43	44212.6	1028.2
46	43952.8	955.496
49	47414.2	967.637
52	47088.7	905.551
55	45634.1	829.71
58	44843.4	773.162
61	42099.7	690.159
64	43812	684.562

Table 8.3: Number of simulations per second on a 32-cores Power-6 (4.7 GHz, hyper-threaded, hence the test until 64 threads). The main conclusion here is that the rate (sims/s) is the same for these slow simulations than with fast simulations (see Table 8.2). Incidentally, we see that with Power6, even with 32 cores, the efficiency is quite good.

and to synchronize these trees slowly, i.e. not at each simulation but with frequency e.g. three times per second [109]. The synchronization is not on the whole tree; it is typically performed as follows:

- Select all the nodes with
  - at least 5% of the total number of simulations of the root;
  - depth at most d (e.g. $d = 3$);
- Average the number of wins and the number of simulations for each of these nodes.

This can be computed recursively (from the root), using commands like *MPI_AllReduce* which have a cost logarithmic in the number of nodes. A special case is **slow root parallelization:** this is slow tree parallelization, but with depth at most $d = 0$; this means that only the root is considered.

Configuration of game	Winning rate in 9x9	Winning rate in 19x19
32 against 1	75.85 $\pm$ 2.49 %	95.10$\pm$01.37 %
32 against 2	66.30 $\pm$ 2.82 %	82.38$\pm$02.74 %
32 against 4	62.63 $\pm$ 2.88 %	73.49$\pm$03.42 %
32 against 8	59.64 $\pm$ 2.93 %	63.07$\pm$04.23 %
32 against 16	52.00 $\pm$ 3.01 %	63.15$\pm$05.53 %
32 against 32	48.91 $\pm$ 3.00 %	48.00$\pm$09.99 %

Table 8.4: Experiments showing the speed-up of "slow-tree parallelization" in 9x9 and 19x19 Go. We see that a plateau is reached somewhere between 8 and 16 machines in 9x9, whereas the improvement is regular in 19x19 and consistent with a linear speed-up - a 63% success rate is equivalent to a speed-up 2, therefore the results sill show a speed-up 2 between 16 and 32 machines in 19x19. Experiments were reproduced with different parameters without significant difference; in this table, the delay between two calls to the "share" functions is 0.05s, and x is set to 5%. The numbers with high numbers of machines will be confirmed in Table 8.5.

- *Voting schemes*. This is a special case of tree parallelization advocated in [64], that we will term here for the sake of comparison with other techniques above **very slow root parallelization:** this is slow root parallelization, but with frequency $f = 1/t$ with t the time per move: the averaging is only performed at the end of the thinking time. There's no communication during the thinking time, and the drawback is that consequently there's no load balancing.

It is usually considered that fast tree parallelization does not perform well; we will consider only other parallelizations. We present in Table 8.4 the very good results we have in 19x19 and the moderately good results we have in 9x9 for slow tree parallelization.

We can compare **slow root parallelization to the "voting scheme" termed very slow root parallelization:** with 40 machines and 2 seconds per move in 9x9 and 19x19, the slow root parallelization wins clearly against the version with very slow root parallelization, as shown by Table 8.5. with a frequency $f = 1/0.35$ against the very slow

Framework	Success rate against voting schemes
9x9 Go	63.6 % $\pm$ 4.6 %
19x19 Go	94 % $\pm$ 3.2 %

Table 8.5: The very good success rate of slow tree parallelization versus very slow tree parallelization. The weakness of voting schemes appears clearly, in particular for the case in which huge speed-ups are possible, namely 19x19.

root parallelization. As a rule of thumb, it is seemingly good to have a frequency such that at least 6 averagings are performed; 3 per second is a stable solution as games have usually more than 2 seconds per move; with a reasonable cluster 3 times per second is a negligible communication cost.

We now compare **slow tree parallelization** with depth $d = 1$, to the case $d = 0$ (slow root parallelization) advocated in [54]. Results are as follows and show that $d = 0$ is a not so bad approximation:

Time per move	Winning rate of slow-tree-parallelization (depth=1) against slow-root-parallelization
2	50.1 ± 1.1 %
4	51.4 ± 1.5 %
8	52.3 ± 1 %
16	51.5 ± 1 %

These experiments are performed with 40 machines. The results are significant (when averaged) but very moderate.

8.3 Conclusions on the parallelization

We revisited scalability and parallelism in MCTS. Parallelizing MCTS involves two different questions:

- **Parallelization.** Can be simulate a MCTS run which needs a computation time T on a single computer, with time $T' << T$ on a large number of processors ?
- **Scalability.** Is it worth parallelizing MCTS, in the sense: is a MCTS with huge computation time and/or computational power (large number of processors) much more efficient than a traditional MCTS with reasonnable time on a single computer.

8.3.1 Parallelization

Multicore-parallelization is a well known parallelization of MCTS; the straightforward implementations work quite well (with nonetheless the limitation that MCTS strength does not scale so well with computation time - this limitation, emphasized above, holds for all parallelizations so we do not discuss it further from now on). Hyperthreading does not seem to provide improvements in MCTS; our results with more than 32 threads on the 32 hyperthreaded cores of the 4.7 GHz power-6 were at best equivalent to 32 threads. With a smaller number of cores, i.e. 6, some authors (Hiroshi Yamashita, on the computer-go mailing list) reported positive results for hyperthreading.

Several parallelizations of MCTS on clusters have been proposed. We clearly conclude that communications during the thinking time are necessary for optimal performance; voting schemes ("very" slow root parallelization) don't perform so well. In particular, slow tree parallelization wins with probability 94 % against very slow root parallelization in 19x19, showing that the slow tree parallelization from [109] or the slow root parallelization from [54] are probably the state of the art. Slow tree parallelization performs only moderately better than slow root parallelization when MCTS is used for choosing a single move, suggesting that slow root parallelization (which is equal to slow tree parallelization simplified to depth$= 0$) is sufficient in some cases for good speed-up - when MCTS is applied for proposing a strategy (as in e.g. [12] for opening books), tree parallelization naturally becomes much better.

$N =$ number of simulations	Success rate of 2N simulations against N simulations in 9x9 Go	Success rate of 2N simulations against N simulations in 19x19 Go
1 000	71.1 ± 0.1 %	90.5 ± 0.3 %
4 000	68.7 ± 0.2 %	84.5 ± 0.3 %
16 000	66.5 ± 0.9 %	80.2 ± 0.4 %
256 000	61.0 ± 0.2 %	58.5 ± 1.7 %

Table 8.6: Scalability of MCTS for the game of Go. These results show a decrease of scalability as computational power increases.

8.3.2 Scalability: is it worth parallelizing MCTS ?

Table 8.6 shows that, for the game of Go, both in 9x9 and 19x19, whenever we have a huge computation time (and the parallelization can't do better than emulating a huge computation time), the results are not so good, and in particular $2N$ simulations do not perform so well in front of N simulations when N becomes large. Section 5.3.5 shows an example of situation which is not solved, in spite of huge computational power, and whereas it is extremely simple for human players (even beginners). Similar results were obtained in the Havannah game in [36].

The scalability of MCTS has often been emphasized as a strength of these methods; we'll see below that when the computation time is already huge, then doubling it has a much smaller effect than when it is small. This completes results proposed by Hideki Kato[141] or the scalability study `http://cgos.boardspace.net/study/index.html`; the scalability study was stopped at 524288 simulations, and shows a concave curve for the ELO rating in a framework including different opponents; Hideki's results show a limited efficiency, when computational power goes to infinity, against a non-MCTS algorithm. Seemingly, there are clear limitations to the scalability of MCTS; even with huge computational power, some particular cases can't be solved. We also show that the limited speed-up exists in 19x19 Go as well, and not with much more computational time than in 9x9 Go. In particular, cases involving visual elements (like big yose) and cases involving human sophisticated techniques around liberties (like semeais) are not properly solved by MCTS, as well as situations involving multiple unfinished fights. Our experiments also show that the situation is similar in Havannah with good simulations.

The main limitation of MCTS is clearly the bias, and for some situations (as those proposed in Figs 5.12 and 5.13) introducing a bias in the score formula is not sufficient; even discarding simulations which are not consistent with a tactical solver is not efficient for semeai situations or situations in which liberty counting is crucial.

8.3.3 Parallelization and scalability: conclusion

The main lessons in our work about the parallelization of Monte-Carlo Tree Search are as follows. First conclusion is quite good for us as we have had a good idea. Complicated ideas sometimes don't survive. We've seen incredibly complicated ideas around

how to develop parallel Monte-Carlo Tree Search for message passing machines. We decided not to work on these complicated ideas, and this decision was a good idea. We took time to think about the problem and designed a very simple solution which worked perfectly well, by simple averaging of statistics. The speed-up is huge, for the application to the Asian game of Go. When we have seen the speed-up, we believed (erroneously) that we were ready for reaching the level of professional players in 19x19 Go with big clusters.

Here comes the second main lesson, which is less positive. Unfortunately, parallelization reduces the variance of Monte-Carlo estimates, but not the bias. And, unfortunately, whenever Monte-Carlo Tree Search is asymptotically unbiased, it is very clear that there are situations in which MCTS is highly biased unless the computational power is incredibly high - more than what we could do even if we had all the cluster in the world connected with a perfect network with no latency and infinite speed. This bias shows a big limitation to the parallelization of MCTS, at least for applications in which there are big biases.

Chapter 9

Monte-Carlo Tree Search and Upper Confidence Trees: Limitations and Current Trends[1]

It is very impressive how techniques developed around the game of Go can be applied for completely different applications or games, and mimic some human behaviors in front of games:

- The **"amaf" [44] or "rave" [112] values**, i.e. the idea, within MCTS, of simulating preferably, in situation *b*, moves which are good at a situation *a*. These values, related , which originated in Go, were successfully applied in other games [221], [5]. This is a kind of permutations; considering permutations is quite natural for humans.
- **The "biasing" by learnt values** [72, 62] (using supervised machine learning) which looks like the idea of local "patterns" by humans is also probably quite general, even if it was born (for MCTS) in the case of Go. There are, however, no application of this idea out of Go - yet, the technique has been applied without any expert knowledge and should therefore be widely applicable.
- The **construction of opening books by self-play** is also quite natural for human players: if a professional player wants to test an opening, then he can simulate several games from this opening and conclude[12]; the fact of testing it against strong humans and use their moves is also quite natural. This is after all quite related to the general field of imitation learning.

This chapter is about recent trends, not yet as widely accepted as those above, for improving MCTS.

[1] This chapter is based on joint works with S. Gelly, J.-B. Hoock, A. Rimmel, F. Teytaud.

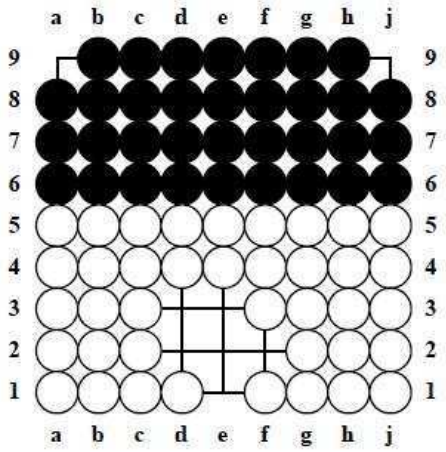


Figure 9.1: Black must kill the white group for winning; this is possible by playing in the middle of the "flower-six" empty space. This is an example of Nakade.

Divide-and-conquer. A main direction for further research around MCTS methods is the learning of the Monte-Carlo part; after all, MCTS current techniques only learn the short term, *i.e.* the tree part - the Monte-Carlo simulations don't change and don't use tactical solvers. Humans predict the issue of local fights, and include this knowledge in their reasoning; we made many trials around this idea, without success. See [53].

Partial observability. A main limitation is that MCTS has been mainly applied in the completely observable case:

- With one player only, partially observable problems are much simpler; we can rewrite the problem so that it becomes a completely observable stochastic Markov Decision Process. This is the principle used in our applications of MCTS to non-linear optimization, active learning, noisy optimization[195, 15, 196].
- In the case of two-player games with "significant" partial observability, it seems that the most efficient techniques strongly rely on heavy tuning: direct policy search, or direct policy search combined with simple tree search or with Monte-Carlo information on what are possible "states", given some observations. It is too difficult to sample consistently (i.e. taking into account the opponent's strategy) the unknown part in e.g. phantom-go[50, 52]: this important case is therefore handled by essentially (and somehow disappointingly) heuristic algorithms. Importantly, this case is essentially undecidable, as shown in the undecidability theorem (section 5.2.2).

Research trends for learning and/or including function values. We briefly outline a few promising research directions in Monte-Carlo Tree Search:

- [160] shows the mixing of MCTS with techniques using a value function: the Monte-Carlo evaluation is replaced by an evaluation function after some random

steps. This is a real progress in Monte-Carlo Tree Search, as it combines this recent technique with well known stuff - the resulting algorithm performed very well in competitions and this kind of techniques might provide huge improvements in the future.

- MCTS is based on the idea of biasing the simple Monte-Carlo Search for the top part of the tree (close to the root). How is it possible to improve bottom parts of the tree, namely improving the Monte-Carlo part instead of the tree part ? A promising idea is **nested Monte-Carlo** [51], presented in Algorithm 5. This algorithm provided some world records in a one-player game termed "Morpion solitaire".

nested(position,level)
while not end of the game do **do**
  if level=1 **then**
    $move = \arg\max_m monteCarlo(play(position, m))$
  else
    $move = \arg\max_m nested(play(position, m), level - 1)$
  end if
  position = play(position, move)
end while
return score

Algorithm 5: Nested Monte-Carlo evaluation. With $level = 1$, this function just computes a Monte-Carlo simulation and returns a reward; with $level > 1$, a recursive call improves the quality, but at the price of a big computational cost. This algorithm performs well in some cases, but never improved the results in the case of Go. $play(position, m)$ is the next location when playing m is position $position$ and this procedure returns a reward obtained by a nested-Monte-Carlo simulation.

- Another form of learning consists in **guessing, from past simulations, good and bad moves in the Monte-Carlo part**. This is a kind of Graal in MCTS research; some published works are [190, 91, 90, 134, 100] (see in Fig. 9.1 an application of [91]). The methodologies published there are not tested in a general case yet. These results are non-negligible, but do not solve the main issues like the semeais or life-and-death problems.

- As well as in other game or control algorithms, using **structured representations and/or factored actions and/or factored states** is appealing[145, 214]. It is nonetheless unfortunately known theoretically that succinct representations do not make things faster (see section 5.2.2), and in the framework of MCTS positive results were not yet published for factored actions/states.

Part III

Optimization

Chapter 10

Introduction: terminology of optimization[1]

Given an objective function f, optimization is the research of x in a given search domain so that $f(x)$ is small. There are plenty of techniques for doing this, depending on what we know about f. Artificial intelligence usually considers the case in which f has plenty of drawbacks: not smooth, with no known gradient, and possibly computing f takes a lot of time. By the way, f is not necessarily a program: it might involve an industrial process; then, computing f costs money. This idea of refusing the case of simple functions f is consistent with the increasing power of machines: we work on more realistic models because we have better machines than in the past. As well as MCTS was probably useless with weak old hardware, evolutionary algorithms were probably not useful when only linear optimization was tractable. Today, either we consider linear optimization, or quadratic optimization, in really huge dimension, or we have to consider complicated functions.

We here define some vocabulary:

- The **search space** is the domain in which we have elements, the quality of which can be evaluated.
- **Individuals** are elements of the search space.
- A **constraint** is a function which says if a point is in the search space. Constraints can be handled by simply setting the fitness value at $+\infty$ (in minimization), but in many cases there are much better techniques; this is not discussed here (see e.g. [66]).
- The **fitness function**, a.k.a the **objective function**, is the function which evaluates the quality of a point in the search space.

[1]This part benefited from collaborations with A. Auger, J.-B. Hoock, P. Rolet, N. Sokolovska, F. Teytaud, M. Schoenauer, E. Vasquez. It also benefited from discussions with D. Arnold, H.-G. Beyer.

- We are in **minimization** if we look for a point at which the fitness value is minimal, and in **maximization** if we look for a point at which the fitness value is maximal.
- The **horizon**, or the **budget**, is the number of fitness evaluations which are allowed. Be careful with the terminology in case of direct policy search; you can have two distinct horizons, namely the horizon of the control problem (how many time steps are considered in the problem) and the horizon of the optimization of the policy (how many iterations of the optimization algorithm).
- Optimization is **parallel** if several fitness values can be computed simultaneously. This parallelism can be due to the use of parallel computers, or, if the computation of the fitness value involves the physical construction of a prototype, the simultaneous building and testing of several prototypes.
- The **time budget** is the number of iterations which can be performed (or a time measure). In the parallel case, this is very different from the horizon - in the sequential case it is equivalent. Please note that we here neglected the internal cost of the optimizer; this is not always a valid assumption.
- Optimization is **noisy** if the result provided by the fitness function is not the same at each time we evaluate it. This has a very strong effect on optimization algorithms.
- A **local optimum** is a point at which the fitness is better than in a neighborhood of this point. It is termed **local minimum** or **local maximum** in minimization or maximization respectively.
- A **global optimum** (global minimum, global maximum) is a point with fitness better than all other points.
- A **plateau** is a part of the search space at which the fitness is constant. **Premature convergence** occurs when an algorithm stagnates on a small part of the search space (e.g. a plateau) whereas there are better values elsewhere.
- Optimization is termed **uni-modal** if there is only one local optimum. Optimization is **convex** if the fitness function (or its expected value in the noisy case) is convex. Optimization is **quasi-convex** if the level sets of the fitness functions are convex (or, in maximization, complement of convex sets).
- Optimization is termed **linear** if the objective function is linear (with linear constraints), **quadratic** if the objective function is quadratic (with linear constraints, or quadratic constraints in some cases), and **non-linear** in other cases. Linear and quadratic optimizations are not discussed here.
- Optimization is termed **black-box** if no gradient, Hessian, or important side information is available: only fitness values are available. Optimization is termed **comparison-based** if only comparisons between individuals are available, or if only comparisons between individuals are used.

- Optimization is termed **multimodal** when there are multiple local optima, and/or multiple global optima. Optimization is termed **global** if we want the algorithm to avoid local minima which are not global minima (in minimization).

- **Multiobjective optimization** is the case in which we have several objective functions simultaneously. This involves specific techniques, not to be developed further here (see [82] for more on this).

Chapter 11 will be devoted to some examples of applications. The reader familiar with the basic notions of optimization or with already many applications in mind can move to next chapters directly.

Chapter 12 will be devoted to recall a few important optimization algorithms; gradient descent, BFGS, evolutionary algorithms.

Chapter 13 will be devoted to optimal optimization, *i.e.* some ideas around what are the optimal algorithms for solving an optimization task. This might be uninteresting for readers essentially interested in applications, or for readers interested only in algorithms which are easy to develop and use; yet, some of the algorithms cited in this chapter have already been implemented, applied, and provide real improvements if your optimization problem is sufficiently important and expensive (in the sense that evaluating the quality of a solution takes a lot of time).

We will then consider parallel optimization, and what I like around that is that there are not so many people who investigated this. This will be done in chapter 14. Noisy optimization will be considered as well. Multimodal optimization is not discussed a lot in these pages; the main usual tricks for multimodal cases are (i) increasing the population size in an evolutionary algorithm (ii) multiple (possibly quasi-random) restarts[2] (iii) random diversification (i.e. some points are randomly sampled uniformly in the search space), (iv) "murder" operators[207] which discard points which are too close to previously found local optima. "Islands" are sometimes considered as well; we will not discuss this further here.

[2]Multiple restarts consist in running several times the same optimization algorithm, possibly with different randomized (or quasi-randomized) restarts.

Chapter 11

Examples of optimization problems

We will give here several examples of optimization problems, with their specificities. This will emphasize the importance of techniques presented in the sequel. Also, these problems are important tools for machine learning, games, control.

11.1 Direct policy search

Consider a strategy in an economical and ecological environment. You can make decisions $d_1, \ldots, d_H$ at different time steps $1, 2, \ldots, H$. At each time step you are in some unknown state: x_1 at time step 1, x_2 at time step 2, ... When you make decision d_i at time step i whereas you are in state x_i, you reach the state $x_{i+1} = f(x_i, d_i)$, and you get an observation $o_i = o(x_i)$ (but you don't observe x_i).

This means that, if you model the decision method by function u, the variables are

related as follows:

$$x_1 = initial\ state \quad // \text{ time step 1} \tag{11.1}$$
$$o_1 = o(x_1) \tag{11.2}$$
$$d_1 = u(o_1) \tag{11.3}$$
$$x_2 = f(x_1, d_1) \quad // \text{ time step 2} \tag{11.4}$$
$$o_2 = o(x_2) \tag{11.5}$$
$$d_2 = u(o_1, o_2) \tag{11.6}$$
$$x_3 = f(x_2, d_2) \quad // \text{ time step 3} \tag{11.7}$$
$$o_3 = o(x_3) \tag{11.8}$$
$$d_3 = u(o_1, o_2, o_3) \tag{11.9}$$
$$\ldots \tag{11.10}$$
$$x_k = f(x_{k-1}, d_{k-1}) \quad // \text{ time step k} \tag{11.11}$$
$$o_k = o(x_3) \tag{11.12}$$
$$d_k = u(o_1, o_2, o_3, \ldots, o_{k-1}) \tag{11.13}$$
$$\ldots \tag{11.14}$$
$$x_H = f(x_{H-1}, d_{H-1}) \quad // \text{ time step H} \tag{11.15}$$
$$reward = R(x_H) \tag{11.16}$$

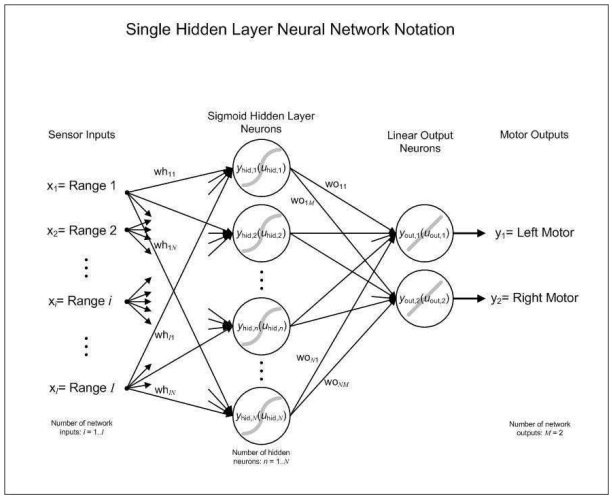


Figure 11.1: A neural controller. It takes as inputs the sensors (the observations), and outputs some decisions (the motor signals, in a robotic framework). Its parameters are the weights of the neurons. Picture from the evolutionary robotics webpage. From `http://www.nelsonrobotics.org/evolutionary_robotics_web/`.

The reward (Fig. 6.3) then depends on (i) the decision function u (ii) the initial state

and possibly (iii) the random seed, if the problem is randomized. If u is a parametric function (e.g. a neural network, as in Fig. 11.1), parametrized by some θ, then we might want to find θ such that the reward is as high as possible. How to find θ, in particular if f, R or u are too complicated for an analytic solving ? The theoretical ultimate limits for such problems are discussed in chapter 13.

If the initial state is known and deterministic, then this is an optimization problem with deterministic fitness functions; it is nonetheless very difficult to compute the derivative of the reward as a function of u. On the other hand, it is often possible to parallelize several evaluations. This has a big impact on the computational cost, it provides really big improvements, and it is discussed in chapter 14.

When the initial state is randomized and its distribution is given, then the average reward for decision function u (we can also call it a strategy u) can be evaluated as follows:

Naive evaluation method for strategy u:
Randomly select an initial state. Compute the reward by Eqs 11.1-11.16 (i.e. by stochastic simulation).

The result is stochastic. What are good algorithms for this case, and to which extent can we improve the results by the use of parallel machines ? This is discussed in chapter 14.2. Many classical algorithms (even evolutionary algorithms, which are usually supposed to be robust) are unstable in that case.

11.2 Evolutionary robotics

An important case of direct policy search (see above the definition of direct policy search) is evolutionary robotics[89]: evolutionary robotics is the optimization of a policy (equivalently, the optimization of a controller) for a robot by evolutionary algorithms. Evolutionary algorithms are often used for this as they are quite convenient and easy to use: in many cases, a success in robotics is due to good sensors, and to a good representation, and secondly to the robustness of the optimization algorithm used for building the controller, rather than to the convergence rate of the optimization algorithm: therefore, evolutionary algorithms are quite efficient in this context, and in particular the simplest evolutionary algorithms. In evolutionary robotics one can optimize parameters, but also sometimes the structure of the controller[119, 118]. Sometimes, the evolutionary algorithm, instead of evolving directly a controller or a robot or a structure, evolves the "construction guide"; this is an extremal case in which the genotype is very different from the phenotype[86].

Some particularly nice examples are emphasized below (see Fig. 11.2). Importantly, a recent research trend is the inclusion of diversity criteria (curiosity), where the diversity is not estimated on the individuals but on the simulation results (e.g. final state at a fixed time); this kind of result is both practically relevant for constructing machines

which really learn and philosophically interesting[89]. Another important progress was in terms of representation: by the use of central pattern generators (CPG), directly inspired by biology, very stable robots can be designed[136]. The idea is to have control signals sent by oscillating systems with negative gain; changing direction is possible by sending an ad hoc signal to actuators in order to briefly perturbate the normal cycle, and when a sudden noise perturbates the system (e.g. when the robot is suddenly pushed) the negative gain corrects the trouble and the system goes back to the normal cycle.

In the Symbrion project, several robots with a same controller are suppose to interact for solving problems, such as building a bridge or crossing a river. **Swarm intelligence** is the case in which each robot has small capabilities (as for ants or bees).

11.3 Structure optimization: is operation research good for society ?

Structure optimization can refer to different things[177]:

- optimization of the physical structure of a mechanical element, *e.g.* aircraft structure, various parts of a car, turbines, robots, bacteriae.
- optimization of the structure of a company, *i.e.* optimization of the decomposition of the structure into substructures with their individual goals and sizes.
- optimization of the organization of the work inside a company/service; *e.g.* a planning of the recyclable waste collecting, or specification of a strategy for taxis so that delays before services are reduced.

The first is often cited, as it is often crucial for companies which have enough money for taking care of this. Nonetheless, other elements often also have a huge impact; all these points are obviously directly related to optimization; sometimes linear or quadratic, and often expensive optimization. Here is a list of applications of operation research (OR) cited by ThinkOR.org:

- evacuation planning;
- cancer therapy;
- acquisition prioritization;
- dispatching service vehicles;
- delay management in public transportation;
- design of a house for disabled persons;
- hub location in cargo applications;
- production resetting optimization;

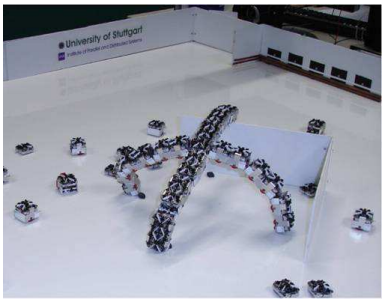
- optimization of the collection and disposal of recyclable waste.

This author has been impressed several times by the extent to which it is possible, by clean mathematics/computer science, to improve greatly the efficiency of an industrial process. Often the main limit is not technical, but economical or political - operation research companies are often too expensive for possible users who are afraid of bad results, and free or low price applications realized by people in academia are not encouraged by research institutes which usually consider that if a work is made for free then it must be a bad work. Often OR is developed during wars, as the effect is immediate and more motivating than health care, pollution reduction or things like that - this author strongly believes that developing the culture of applications made for free in an academic world with stable fundings would be much more beneficial for society than developing researches with short term fundings which increase the quantity of administrative tasks and imply competition based on showing off and at the end kills the motivation.

11.4 Program optimization, a.k.a genetic programming (GP)

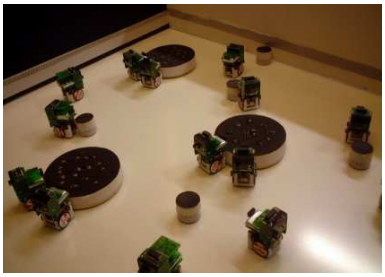
In many cases, designing an optimal algorithm for a given task is extremely difficult. For example, what is the fastest algorithm for sorting 7 items ? The solution proposed by Koza, Benett, Andre and Keane (1999) and shown in Fig. 11.3 involves 16 steps, i.e. less than what was proposed in a earlier patent, and with equal performance as the solution by Floyd and Knutt (1973). This was performed by **genetic programming**: optimization of the structure of a program by evolutionary algorithms. Please note however that in this widely cited example, humans found the solution earlier.

Many important examples of strong realizations of genetic programming come from quantum computing. Quantum programming is so difficult for humans that computers are relatively strong. A human competitive program was also derived by a genetic programming by Andre and Teller in soccer; the program performed well (average score) in front of human-developed controllers (Fig. 11.3).

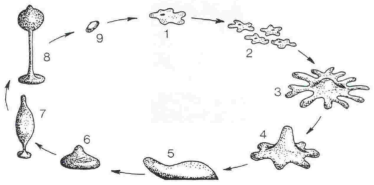


Symbrion project...

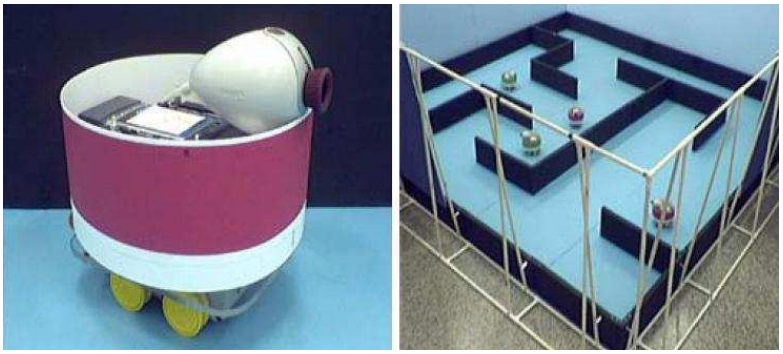
...vs nature (photo from WordPress).



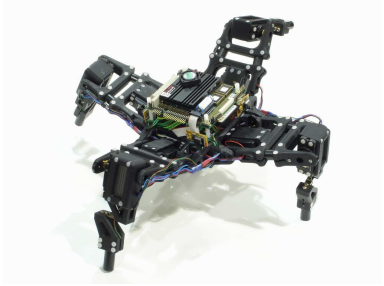
Alice micro-robot...



...and cooperation in nature (at stage 8 of Dictyostelium discoideum (amoeba), the building of a high colony is only possible thanks to the suicide of plenty of individuals in the stem) (web page of Reinhard Eichelbeck)



The EvBot project ([174]).



← A robot which optimizes its behavior on the fly in case of body damage (resiliency), by Josh Bongard.

Figure 11.2: Examples of evolutionary robotics.

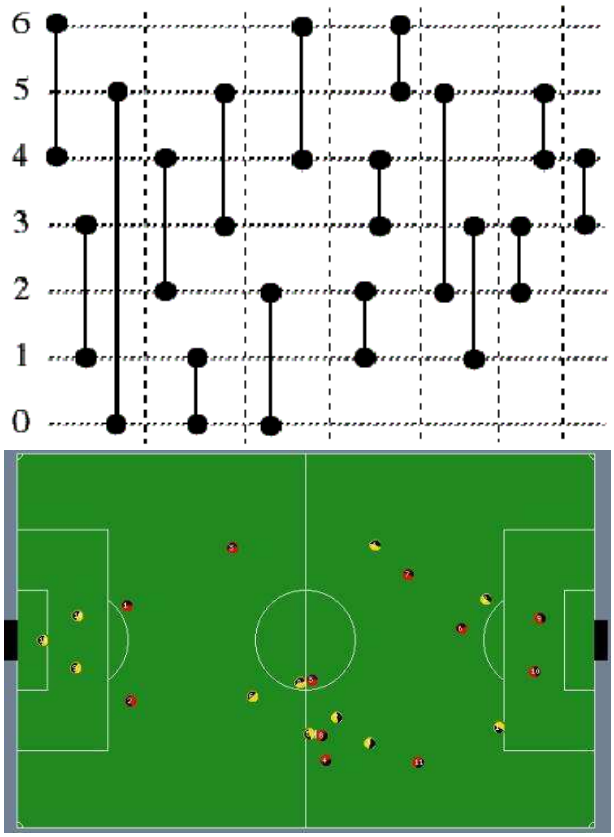


Figure 11.3: Top: a sorting algorithm in 16 steps for 7 items - each vertical line represents a comparison (possibly with exchange). Bottom: a simulated soccer game in which genetic programming performed reasonably well (Andre and Teller, RoboCup 97) in front of handcrafted programs.

Chapter 12

Some important algorithms for optimization[1]

This chapter is devoted to presenting a few optimization algorithms. We consider an objective function f, and we look for x such that $f(x)$ is as small as possible[2]; sometimes, there's one and only one x^* such that $\forall x \neq x^*, f(x) > f(x^*)$. Usually, optimization algorithms provide iterations: $x_1,\ldots,x_n$ are approximations of the unknown optimum x^*. Some natural questions are then:

- Does $f(x_n) \rightarrow f(x^*)$? Or does $\inf_{i\in[[1,n]]} f(x_i) \rightarrow_{n\rightarrow\infty} f(x^*)$? (please note that this question is not clearly defined for noisy objective functions)
- In the noisy case, does $\mathbb{E}f(x_n) \rightarrow_{n\rightarrow\infty} \mathbb{E}f(x^*)$?
- Does $\limsup_{n\rightarrow\infty} \frac{1}{n}\log(||x_n - x^*||) < 0$? (this is **linear convergence**) . If yes, then the quantity
$$\exp(\limsup_{n\rightarrow\infty} \frac{1}{n}\log(\|x_n - x^*\|))$$
is termed the **convergence rate**.
- Does $\limsup_{n\rightarrow\infty} \frac{1}{n}\log(||x_n - x^*||) \rightarrow -\infty$? (this is **superlinear convergence**)
- Does $\limsup_{n\rightarrow\infty} \frac{||x_n - x^*||}{||x_{n-1} - x^*||^q} < \infty$ with $x_n \rightarrow x^*$? (this is **order-q** convergence, **quadratic convergence** if $q = 2$)[3]

[1] This chapter uses many valuable discussions with A. Auger, M. Schoenauer, N. Bredèche.

[2] We here assume minimization. Maximization is obviously quite similar.

[3] Please note that the condition $x_n \rightarrow x^*$ is necessary, as the first condition might occur whenever $x_n \not\rightarrow x^*$. Also, this definition of convergence of order q is misleading if there are episodically values of n such that x_n is bad, whenever we have an overall very good convergence (for example, consider $x_n = x^* + 1/2^n$ which quadratically converges to x^*; if $x_n = x^* + 42$ for n an exponent of 4, and $x_n = x^* + 1/2^n$ otherwise, then intuitively we would like to consider that there is fast convergence anyway; other definitions can be proposed for that.

The **runtime** for reaching precision ε with probability $1-\delta$ in a family F of fitness functions is usually n minimal such that for all fitness in F, with probability at least $1-\delta$, $||x_n - x^*|| \leq \varepsilon$.

The order q of convergence, or the runtime, is not necessarily a good measure of performance. For example, an algorithm might spend a huge computational power for choosing each x_n; this has no impact on the order of convergence. The runtime and the order of convergence are only an approximation of a real performance measure. This approximation which is valid when the computation time of the fitness is large. Also, take care of what is x_n: if x_n is the n^{th} evaluated point for an algorithm, whereas it is the $(\lambda n)^{th}$ evaluated point for another algorithm (e.g. if λ fitness values are used for guiding the search or for evaluating the gradient), the comparison is unfair.

Let's present now some classical algorithms, presented with more details in [34, 28].

12.1 Newton's algorithm

x_0 is arbitrary. For choosing x_{n+1}, consider the following quadratic approximation of f:

$$\hat{f}(x) = f(x_n) + \nabla f(x_n) \times (x - x_n) + \frac{1}{2}(x - x_n)^t \times Hf(x_n) \times (x - x_n) \tag{12.1}$$

where

$\nabla f(z)$ is the gradient of f at z,
$Hf(z)$ is the Hessian of f at z (i.e. the second order derivative)
u^t is the transpose of vector u.

It is somehow natural to choose x_{n+1} minimizing the quadratic model of Eq. 12.1. This leads to Newton's algorithm:

$$x_{n+1} = x_n - Hf(x_n)^{-1} \nabla f(x_n).$$

The inverse of the Hessian is not necessary for computing x_{n+1}: it is sufficient to solve the linear system

$$Hf(x_n)(x_{n+1} - x_n) = -\nabla f(x_n) \tag{12.2}$$

Newton's algorithm	
Advantages	Quadratic convergence. Conceptually simple. Able to tackle very high-dimensional problems.
Drawbacks	Requires the Hessian and the gradient. Might find local minima (instead of global minima). Might even find local maxima instead of minima.

The lack of robustness is very clear even on quasi-convex but non smooth fitness functions such as $x \mapsto \sqrt{\sqrt{||x||}}$. The bad results on multimodal functions are also a clear weakness, possibly more or less solved by multiple restarts. We'll see that evolutionary algorithms are much more robust from this point of view. There are various improvements of this algorithm; however, as BFGS is much more widely used than Newton's algorithm, we will now switch to BFGS, after a fast introduction to gradient descent.

12.2 Gradient descent and stochastic gradient descent

The Hessian is usually very difficult to compute. We'll see that Quasi-Newton is based on the idea of approximating the Hessian; in some sense, the gradient descent is an extremal case consisting in considering a constant diagonal matrix as an approximation of the Hessian. Consider, instead of Eq. 12.2, the following equation:

$$x_{n+1} = x_n - k\nabla f(x_n).$$

Optimize k (this is a one-dimensional optimization): this is gradient descent. You can even get rid of the optimization:

$$x_{n+1} = x_n - \frac{1}{n}\nabla f(x_n). \tag{12.3}$$

This algorithm is not extremely fast, but it is quite convenient in some cases, and in particular the stochastic case. Consider that instead of having access to f and ∇f, you have only access to random values f_n and ∇f_n, with the property that the f_i's are independently and identically distributed, and $\mathbb{E}f_i = f, \mathbb{E}\nabla f_i = \nabla f$: this stochastic optimization case[4] is extremely frequent in machine learning. In this case, Eq. 12.3 can be just replaced by Eq. 12.4:

$$x_{n+1} = x_n - \frac{1}{n}\nabla f_n(x_n). \tag{12.4}$$

($\frac{1}{n}$ is often replaced by other series so that the sum is infinite, and the term goes to 0).

It has been shown in particular that this algorithm very often outperforms the sophisticated algorithms proposed for Support Vector Machines[35].

[4]We here use "stochastic optimization" in the sense that the fitness function is stochastic; the same words are sometimes used when the optimization algorithm is stochastic.

Gradient descent and stochastic gradient descent	
Advantages	Able to tackle reasonably high-dimensional problems Handles efficiently the stochastic case Easy to implement
Drawbacks	Slower (per iteration, not always per second) than BFGS Requires the gradient Might find local minima (but, interestingly, not so often in the stochastic case...)

12.3 Quasi-Newton algorithms and BFGS

Computing the Hessian as in Eq. 12.1 for Newton's algorithm is usually very expensive; therefore, it is natural to try to estimate it instead of computing it exactly. This is the principle of quasi-Newton algorithms. With H_n the approximation at step n, we might want to use the following equation:

$$H_n(x_{n+1} - x_n) = -\nabla f(x_n) \tag{12.5}$$

Following the BFGS form of Quasi-Newton [43, 102, 115, 212] (named after the authors of [43, 102, 115, 212]), let's define:

H_{n+1} is the approximation of the Hessian used instead of the Hessian in Eq. 12.5

$\nabla_n = \nabla f(x_n)$ gradient of the fitness function

$\delta_n = x_{n+1} - x_n$ column vector of the step

$\delta'_n = \nabla_{n+1} - \nabla_n$ column vector of the difference of gradient

$u_n = \delta_n/(\delta_n^t \delta'_n) - H_n \delta'_n/(\delta'_n H_n \delta_n)$

It is natural to consider the evolution of the gradient in order to estimate the Hessian. This is done as follows:

$$H_{n+1} = H_n + \frac{\delta_n \delta_n^t}{\delta_n^t \delta'_n} - \frac{(H_n \delta'_n)(\delta'^t_n H_n)}{\delta'^t_n H_n \delta'_n} + (\delta_n^t H_n \delta_n) u_n u_n^t.$$

H_0 is usually initialized at the identity matrix. We can also get rid of the index n for clarity:

$$H_{n+1} = H + \frac{\delta \delta^t}{\delta^t \delta'} - \frac{(H\delta')(\delta'^t H)}{\delta'^t H \delta'} + (\delta^t H \delta) u u^t.$$

The convergence to local maxima (instead of local minima) can be avoided thanks to e.g. Armijo's rule (or variants [218]), or by line minimization; instead of Eq. 12.5, use Eq. 12.6:

$$H_n(x_{n+1} - x_n) = -k_n \nabla f(x_n) \tag{12.6}$$

where k_n is optimized so that x_{n+1} is a local minimum; this is an optimization in dimension 1. Usually, it is not necessary in BFGS to have a very good optimizer for

this one-dimensional optimization; a few steps of quadratic approximations should be enough (e.g. BFGS itself, in dimension 1), or some dichotomy-based research; see Brent's algorithm [40] for more.

The Hessian is not necessary in BFGS, but the gradient is. There are variants of this algorithm, the most important one being probably the limited memory BFGS[251, 49]. We will summarize the advantages and drawbacks of BFGS as follows: :

BFGS algorithm	
Advantages	Quadratic convergence (often limited due to machine precision - yet, usually the fastest in high dimension).
Drawbacks	Requires the gradient (unless the gradient is computed by finite differences) Might find local minima (instead of global minima) Might be very slow in (highly) ill-conditioned cases or non-differentiable cases

We will now switch to other algorithms, which are both simpler and more robust - but slower, in particular in high dimension.

12.4 The $(1+1)$ evolution strategy with one-fifth rule

The general principle of an **evolutionary algorithm** is presented in Alg. 6. The parent population size μ and the offspring size λ are parameters of the algorithm.

Generate a set of λ individuals, termed the population.
while Time left > 0 **do**
  Evaluate each individual in the population, i.e. compute its fitness value.
  Define the set of parents as the μ best individuals.
  Generate λ individuals by cross-over and/or mutations of the parents $\rightarrow$ this is the new population.
end while

Algorithm 6: A high-level view of an evolutionary algorithm.

In some cases, we can distinguish the genotype and the phenotype. This is shown in Alg. 7.

Generate a set of λ individuals, termed the population.
while Time left > 0 **do**
  Evaluate each individual x in the population, i.e. build its phenotype $p(x)$ and compute its fitness value $f(p(x))$.
  Define the set of parents as the μ best individuals.
  Generate λ individuals by cross-over and/or mutations of the parents.
end while

Algorithm 7: A high-level view of an evolutionary algorithm, with distinct phenotype and genotype. The important point is that $p(x)$ involves a significant transformation from x.

A **cross-over** is any mapping which proposes a new individual as a (usually stochastic) function of a given finite set of individuals (at least 2). An example is given in Alg. 8. A **mutation** is any mapping which proposes a new individual as a

Cross-over between $x \in \{0,1\}^d$ **and** $y \in \{0,1\}^d$**.**
for $i \in [[1,d]]$ **do**
  if $random < \frac{1}{2}$ **then**
    $z_i \leftarrow y_i$
  else
    $z_i \leftarrow x_i$
  end if
  Return z**.**
end for

Algorithm 8: A simple cross-over between individual x and individual y.

(usually stochastic) function of a given individual. A simple mutation operator in the case of a search space $\{0,1\}^d$ is proposed in Alg. 9.

In the continuous case $[0,1]^d$, a usual choice is $z \leftarrow x + \sigma N$ where N is a standard Gaussian vector in dimension d and σ is the step-size, to be adapted as discussed later (e.g. one-fifth rule).

Consider x_0 arbitrary, as well as some $\sigma_0 > 0$. Then, define

$$\begin{aligned} x'_n &= x_n + \sigma_n N \text{where } N \text{ is an independent Gaussian} \\ x_{n+1} &= x'_n \text{ if } f(x'_n) < f(x_n), \text{ and then } \sigma_{n+1} = 2\sigma_n \\ &= x_n \text{ otherwise, and then } \sigma_{n+1} = 2^{-\frac{1}{4}}\sigma_n \end{aligned}$$

Mutation of an individual $x \in \{0,1\}^d$**.**
for $i \in [[1,d]]$ **do**
  if $random < \frac{1}{2}$ **then**
    $z_i \leftarrow x_i$
  else
    $z_i \leftarrow 1 - x_i$
  end if
  Return z**.**
end for

Algorithm 9: A simple example of mutation operator.

This very simple algorithm has linear convergence on many fitness functions, and will avoid some easily avoidable local minima. σ_0 should be chosen greater than the distance to the global optimum. The convergence rate is $1 - \Theta(1/d)$, where d is the dimension of the search space. In spite of its simplicity, it's not so easy to outperform this algorithm, and many practitioners like it.

$(1+1)$-ES with one-fifth rule	
Advantages	Linear convergence Easy to adapt to a new problem
Drawbacks	Not as fast as BFGS Not able to handle high dimensionality*

The fact that adapting the algorithm to a new problem is easy is not negligible as an advantage. For example, in many cases, the fitness function is computed much faster when only 1 or few coordinates are removed: this is easy to modify in a $(1+1)$-ES, and not in many algorithms.

(*) Handling high-dimensionality with the $(1+1)$-ES is possible when many variables have a negligible impact[31]. Importantly, many algorithms do not have this ability.

12.5 Including covariances

The main drawback of the $(1+1)$-ES is that it cannot handle highly ill-conditioned quadratic functions (see Fig. 12.1); as a consequence, some evolutionary algorithms have been defined for this case. We will here focus on algorithms using the self-adaptation principle, a beautiful and widely applicable principle that one can summarize roughly as follows:

Self adaptation
Choose your offspring by random mutations, and then reinforce the efficient forms of mutations.

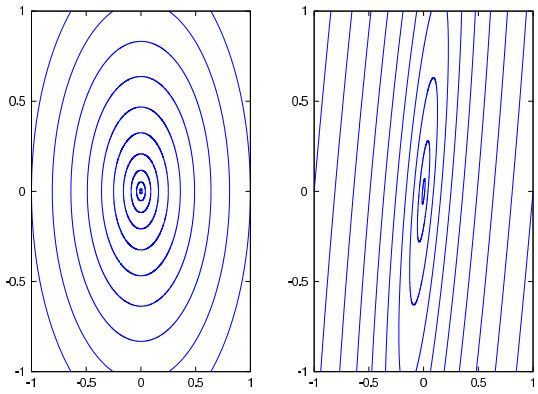


Figure 12.1: Left: level-sets of a well-conditioned function. Right: level-sets of a ill-conditioned function.

For example, if offspring are chosen by randomly drawing Gaussian mutations, then the efficient mutations (which provided successes in the past) should have a higher probability of being generated in the future. The Covariance Matrix Self-Adaptation Revisited (CMSA [32]) algorithm provides a solution for this. The algorithm, presented in Alg. 10, is based on the following ideas:

- The current search distribution (*i.e.* the probability distribution used for sampling new iterates) is a Gaussian random variable centered at y, with covariance $\sigma^{avg} \times C$ where σ^{avg} is a real number and C a covariance matrix.

- When new points are sampled, each of them has a modified step-size (randomly chosen around σ^{avg} - the offspring will have different mutation strengths).

- The search distribution is modified as follows:

  - y is moved to the mass center of the μ best points;
  - σ^{avg} is moved to the average (possibly log-average) value of the step-sizes of selected offspring (therefore, best step-sizes should be reinforced in the future);
  - the covariance matrix C is "averaged" with the covariance of selected points, therefore better directions will be reinforced in the future.

An important modification of CMSA (Alg. 10) is $\mu = \min(dimension, \lambda/4)$; for λ large, this provides a big improvement. This also shows the strength of a rigorous mathematical analysis as this modification was derived by complexity analysis (see Fig. 14.2 for more on this).

We summarize the advantages and drawbacks of this algorithm as follows:

Initialize $\sigma^{avg} \in \mathbb{R}$, $y \in \mathbb{R}^N$, C. Usually $\mu = \lambda/4$.
while Halting criterion not fulfilled **do**
  for $i = 1..\lambda$ **do**
    $\sigma_i = \sigma^{avg} e^{\tau N_i(0,1)}$ // random modification of the step-size
    $s_i = \sqrt{C}\sigma_i N_i(0, Id)$ //
    $z_i = \sigma_i s_i$
    $y_i = y + z_i$
    $f_i = f(y_i)$
  end for
  Sort the individuals by increasing fitness; $f_{(1)} < f_{(2)} < \cdots < f_{(\lambda)}$.
  $z^{avg} = \frac{1}{\mu}\sum_{i=1}^{\mu} z_{(i)}$
  $s^{avg} = \frac{1}{\mu}\sum_{i=1}^{\mu} s_{(i)}$
  $\sigma^{avg} = \frac{1}{\mu}\sum_{i=1}^{\mu} \sigma_{(i)}$
  $y = y + z^{avg}$
  $C = (1 - \frac{1}{\tau_C})C + \frac{1}{\tau_C} ss^t$
end while

Algorithm 10: Covariance Matrix self-adaptation. τ is equal to $1/\sqrt{N}$. The initial covariance matrix C is the identity matrix. The time constant τ_C is equal to $1 + \frac{N(N+1)}{\frac{\lambda}{2}}$. $\sqrt{C}$ denotes the square root of a matrix in the Choleski sense.

The CMSA optimization algorithm	
Advantages	Easily readable More parallel than CSA[32] No gradient required
Drawbacks	Slow in large dimension (like most algorithms for ill-conditioned problems)

12.6 Discrete optimization algorithms

A main strength of evolutionary algorithms is their ability to handle discrete optimization. We will here focus on optimization in $\{0,1\}^N$, but tools are often the same (except in the particular case of genetic programming) for other discrete search spaces. An example is given in Alg. 11. The complexity bounds provided in chapter 14 also hold in the discrete case; the automatic parallelization (by speculative parallelization) proposed in section 15.2 also works in the discrete case.

Randomly draw λ individuals $x_1, \ldots, x_\lambda$ in the search space $\{0,1\}^N$.
while Time left > 0 **do**
  Let $x'_1, \ldots, x'_\mu$ be the μ best individuals among these λ points.
  for $i = 1, 2, \ldots, \lambda'$ **do**
    Let $j = (i \text{ modulo } \mu) + 1$
    $x_i = randomMutation(x'_j)$
  end for
  for $i = \lambda' + 1, \lambda' + 2, \ldots, \lambda$ **do**
    j, k are randomly drawn in $\{1, \ldots, \mu\}$
    $x_i = crossOver(x_j, x_k)$
  end for
end while

Algorithm 11: A possibly discrete evolutionary algorithms (which can be used also in the continuous case). j and k are sometimes uniformly drawn in $\{1, \ldots, \mu\}$, and sometimes $j = i$ with probability decreasing as i increases (variants are roulette wheel selection, truncation selection, fitness proportional selection; we do not develop this here). $c = crossOver(a, b)$ can be defined in many manners; a simple solution is $c_e = a_e$ or $c_e = b_e$ with equal probability and independently for each $e \in \{1, \ldots, N\}$; other variants are one-point crossover or two-points crossover. The mutation is usually $c = randomMutation(a)$ with $c_e = 1 - a_e$ (with probability $1/N$) and $c_e = a_e$ with probability $1 - 1/N$, independently for each $e \in \{1, \ldots, d\}$.

Chapter 13

The ultimate limits of optimization of expensive fitness functions: optimal algorithms[1]

The No Free Lunch (NFL) theorem is usually interpreted as "on average, all algorithms perform equally well", with the corollary "there's no optimal algorithms". Nonetheless, this is only for a finite domain and a finite codomain (and not in continuous domains, see [15]), and only for uniform averaging or some other very strange distribution [243, 92]; as a consequence, it makes sense to look for good algorithms, and even for optimal algorithms, in some sense mathematically specified below (section 13.1). We'll consider practical algorithms, based on tuning optimizers, in section 13.2, before switching to formally optimal (but not very practical) algorithms in section 13.3, including their more practical approximations like EGO and IAGO.

There's quite a big quantity of purely experimental works, or purely applied works, in evolutionary algorithms. We are therefore proud that some algorithms discussed here are theoretical, are justified by mathematical analysis only, and are not tested on real world problems. Various tools and notions discussed in this chapter (surrogate models, compromise between exploration and exploitation) are nonetheless important and useful, for philosophical reasons[88], and also for understanding what we do, and even for designing optimization algorithms.

13.1 Optimal optimization algorithms

What is an optimal algorithm ? The efficiency of an algorithm depends on the fitness function: therefore, we must specify which fitness function is considered for defining the optimality. If there's only one fitness function, then the optimal algorithm is trivial: it is the algorithm which does not compute anything and outputs immediately the optimum value. Obviously, this optimal algorithm is perfect for one fitness function,

[1]This chapter is based on joint works with A. Auger, P. Rolet, M. Sebag, F. Teytaud, E. Vasquez.

and makes no sense for traditional optimization. We see that we must consider a family of fitness functions. Then, we must also consider a criterion: for example, we can consider, on average on fitness functions (this implies that we have a distribution on our family of fitness functions), and on average on the internal random parts of the algorithm, and after 10 minutes, the expected fitness value of the best visited point. This becomes a well defined quantity, for a given hardware and technical conditions of the run. As this measure is a bit unstable (depends on the hardware, software, on the load of the machine...), it is usually preferred to consider a more abstract and idealized criterion: the expected fitness value ot the best visited point after a fixed number of function evaluations. This provides a first drawback of many testbeds: only the number of evaluations is considered, whereas in many cases this will completely change the ranking of the various algorithms. This is particularly important as in many cases algorithms getting the first ranks cannot be used above dimension 30 due to their huge internal computational cost. The approximation of neglecting the internal cost of the optimization algorithm however makes sense when considering **expensive** optimization problems, *i.e.* problems in which most of the cost is in the evaluation of the fitness function.

The log**-effect.** Another modification is often applied: instead of considering the expected fitness value, people consider the expected logarithm of the fitness value, or the expected logarithm of the distance to the optimum - something with a log. What is the consequence of this choice ? It implies that when an algorithm misses the optimum with probability 95 %, but has a very good precision with probability 5%, it will have a much better score for such criteria than an algorithm which is slower but finds the optimum with very high probability . This is certainly not satisfactory for robust optimization, and essentially justified by the fact that it provides nicer graphs: as many algorithms are linear in log-scale, comparisons are more visible with this log-representation.

Using testbeds for optimization of algorithms	
Advantages	Easy to use. Can include real world fitness functions (but usually does not). Can use fitness functions related to the target problem.
Drawbacks	Usually based on artificial fitness functions. Computationally expensive. Misleading criteria (see the log-effect). Internal cost of the algorithm often not taken into account. Dimensionality chosen so that experimentations are easy. Overfitting: when benchmarks are public, optimizers are carefully tuned specifically for them.

We have defined a criterion of optimality (but we emphasize that many variants of this criterion can be defined), now among which family of optimization algorithms are we going to look for an optimal algorithm ?

A first approach consists in considering a parametric family of optimization algorithms, and then to tune them for a given distribution of fitness functions. This will be discussed in section 13.2. A second approach consists in mathematically deriving an optimal optimization algorithm; we'll see that this is mathematically feasible, but computationally extremely hard; nonetheless, for expensive optimization it might make sense (section 13.3); we'll see that approximate versions (third approach) of this mathematical target can be derived and applied industrially.

Figure 13.1 (from [234]) briefly presents these three approaches for approaching optimal optimization algorithms.

13.2 Tuning optimization algorithm

Consider the fitness function $x \mapsto f(x)$. An criterion of quality for an optimization algorithm applied to f might be $\mathbb{E}_r f(x_{500}^f)$, where

- x_{500}^f is the 500^{th} iterate of your optimization algorithm (or the best of the 500 first iterates) when the optimization algorithm is applied on f,
- $\mathbb{E}_r$ is the expectation operator on the random part of the optimization algorithm.

When f is a parametric family of fitness functions, e.g. $x \mapsto f(\theta, x)$ for some parameter θ, and if you have a distribution on θ, then you might consider

$$\mathbb{E}_\theta \mathbb{E}_r f(\theta, x_{500}^{f(\theta,.),r}));$$

and if you prefer, or if you need beautiful graphs for your next publication to be accepted,

$$\mathbb{E}_\theta \mathbb{E}_r \log f(\theta, x_{500}^{f(\theta,.),r})) \text{ assuming that } \inf f(\theta,.) = 0.$$

For many ES, this provides nice linear curves. When you have a function f and don't know how to extend it to a distribution of fitness functions, usual solutions are:

- random translations of f:
$$f(\theta, x) = f(x - \theta)$$
with θ of the same dimension as x;
- random rotations of f:
$$f(\theta, x) = f(\theta.x)$$
where θ is a rotation matrix on the search space;
- random noise on f:
$$f(\theta, x) = f(x) + \sum_i sin(x_i - \theta_i)^2.$$

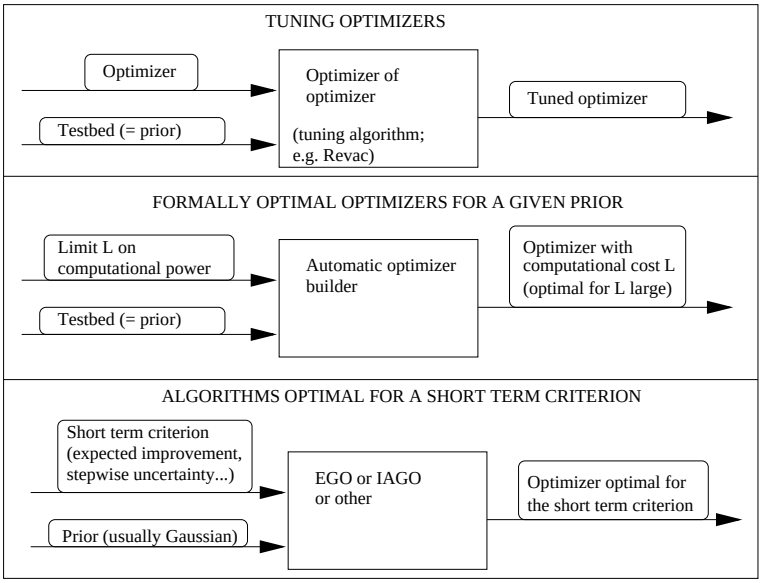


Figure 13.1: Three approaches for optimization given a prior. The first approach [96, 173, 215] consists in optimizing the parameters of an OA (this assumes that the OA has parameters); this approach is limited to a given structure of OA. It is easy to use, well understood, immediately operational and efficient in practice. The second approach [111, 196, 15] consists in considering the problem specified by Eqs. 7.1-7.3 and solve it as a Markov decision process; this approach is mathematically appealing as it provides a provably optimal algorithm; unfortunately, the approach is computationally very expensive and might be difficult to use in practice. The third approach considers an approximate criterion, e.g. Eqs. 13.8-13.10, and a prior, e.g. it assumes that the fitness function is drawn according to a Gaussian process, and proposes an algorithm which is optimal (within the limit of the computational cost) for this criterion. This third approach is optimal for a "local" criterion only, and can therefore not be proved optimal for the complete optimization run, but it is by far simpler than the second approach.

If your optimization algorithm has a parameter p, then your criterion becomes a function of p, i.e.:

$$p \mapsto \mathbb{E}_r \mathbb{E}_\theta f(\theta, x_{500}^{f(\theta.),r,p})).$$

Then you can optimize p. If the set of functions is finite and if r lies in a finite space, then this function can be computed exactly for a given value p; usually, this is not possible, and you can only have a noisy measurement of the function above. Therefore, tuning an optimization algorithm is usually a noisy optimization problem. You can use for it your favorite optimization algorithm, or use one of published algorithms for it [173, 215, 96], and possibly get a very strong optimizer for a given testbed; the question is now to which extent an optimization algorithm extensively tuned for a given family

of problems will be good for another. Either you precisely know the family of problems from which your target problem will be drawn (but, except in competitions, this is usually not the case), or you might be disappointed. This issue is termed overfitting.

13.3 Mathematically optimal algorithms and their approximations

The solution emphasized above, essentially parameter tuning, is clear and efficient; nonetheless, it might be disappointing: you obviously cannot reach optimality with such an approach as the structure of the algorithm is fixed. Is there something more that we could do ? In particular, given a prior and a criterion, can we derive an optimal optimization algorithm ? This is indeed possible at least for some criteria. This is developed in section 7.1. However, this is not very practical: it provides an algorithm which is optimal in the sense that for a given distribution of fitness functions, the algorithm will be optimal for a given prior (e.g. the squared distance between the proposed approximation of the optimum and the real location of the optimum) after a given number of time steps. However, this criterion might be a bad criterion: finding a very good solutions in 20 iterations is a good thing... except if you need so much time for generating each iterate that a simple (1+1)-ES algorithm is indeed faster (even if it needs 20000 iterations for the same precision...). The reader interested in the mathematical effort of optimal optimizers is referred to section 7.1; we will here consider some more practical approximations. Nonetheless, the computational cost of the techniques, even in these "more practical" sections, is huge; therefore, they should be considered only for very expensive problems, for which spending a very long time on each iterate is worth the candle. Some such examples are proposed in section 7.1.

13.3.1 Surrogate models

Consider a distribution of probability on the fitness function f, and consider that you have already observed

$$\begin{aligned} f(x_1) &= y_1 \\ f(x_2) &= y_2 \\ \cdots &= \ldots \\ f(x_n) &= y_n \end{aligned}$$

How to choose x_{n+1} ? A simple intuitive solution consists in the following:

$$\hat{f} = \arg\max_f P(f(x_1) = y_1, f(x_2) = y_2, \ldots, f(x_n) = y_n | f); \tag{13.1}$$

$$x_{n+1} = \arg\min_x \hat{f}(x). \tag{13.2}$$

Eq. 13.1 mean that we search for the maximum likelihood of f conditionally to observations; this $\hat{f}$, termed surrogate model, is an approximation of the unknown fitness function. Then, in Eq. 13.2, we pick up the minimum of the surrogate model. Eq. 13.1

can be replaced by other tools than maximum likelihood; for example, empirical risk minimization, possibly with regularization (see section 16). This is obviously centered on exploitation of already known good parts of the domain; it might suffer from convergence to a local minimum or premature convergence; a simple solution consists in adding some random exploration (e.g. randomly sample the domain one iterate over 10 and switch to this point if it's better than your current iterate...). Some more subtle techniques will be emphasized below.

13.3.2 Taking uncertainty into account

Maximum likelihood as in Eq. 13.1 reduces all the information to an approximation of the fitness function. Indeed, what is known from theoretical analysis (as discussed in section 7.1) is that for optimality, the information we need is the distribution of the fitness function, conditionally to observations. How can we take this into account ?

Let's consider something more complicated than Eq. 13.1 and Eq. 13.2:

$$\hat{f} = \mathscr{L}(f | f(x_1) = y_1, f(x_2) = y_2, \dots, f(x_n) = y_n); \tag{13.3}$$

$$x_{n+1} = \arg\min_x \mathbb{E}\hat{f}(x) - t\sqrt{\mathbb{E}(\hat{f}(x) - \mathbb{E}\hat{f}(x))^2}. \tag{13.4}$$

for some constant t. $\hat{f}$ is now a conditional distribution; it is the probability distribution of f, *after conditioning to observations*. This is obviously a complicated notion, that is not easy to plug into an optimization algorithm. Nonetheless, for specific distributions, it is possible to do this analytically (e.g. Gaussian processes[140, 240]).

If you don't want to model your problem with a Gaussian process, or if you don't want to study the maths for computing Eq. 13.3 for a Gaussian process, a natural tool for sampling $\hat{f}$ consists in using *bootstrap replicates*.

Bootstrap replicates for conditional distributions: imagine that you have a sample of n points $x_1, \dots, x_n$, with their (possibly noisy) labels $y_1, \dots, y_n$. You might just pick up the maximum likelihood

$$\hat{f} = \arg\max_f \prod_{i=1}^{n} P(y_i = f(x_i)),$$

that we will sum up as

$$\hat{f} = maxiLikelihood(x_1, \dots, x_n, y_1, \dots, y_n). \tag{13.5}$$

But you might prefer to sample several $\hat{f}$ conditionally to $\prod_i P(y_i = f(x_i))$ or even conditionally to $P(f)\prod_i P(y_i = f(x_i))$. This is certainly useful for Bayesian statistics (and, in our particular case, for computing $\mathscr{L}(f|\dots)$ in Eq. 13.3). If you sample several such $\hat{f}$ (let's call them $\hat{f}_1, \dots, \hat{f}_k$), then an approximate distribution for $\mathscr{L}(f|\dots)$ is just the uniform distribution on the $\hat{f}_i$. Each $\hat{f}_i$ can be sampled as follows:

- For each $j \in [[1, N]]$, draw, with replacement, a sample (x_{z_j}, y_{z_j}) from the sample of the $(x_k, y_k)_{k \in [[1,N]]}$; this is just drawing z_j uniformly in $[[1, N]]$.
- Then, $\hat{f}_i$ is $maxiLikelihood(x_{z_1}, \dots, x_{z_n}, y_{z_1}, \dots, y_{z_n})$.

It is known (see e.g. [95, 237]) that in many cases (but not all) this provides a good estimator of $\hat{f}$ conditionally to $\prod_i P(y_i = f(x_i))$ (a prior $P_0(f)$ can be included as well, by including it in Eq. 13.5).

Formally, the algorithm consists in approximating $\hat{f}$ by a uniform distribution on

$$\hat{f}_1, \dots, \hat{f}_k$$

where the f_i's are k bootstrap replicates of the maximum likelihood estimator, (or of the empirical risk minimizer or anything you want), i.e. $\forall i \in [[1,k]]$,

$$\hat{f}_i = \arg\max_{f \in F} \prod_{j \in [[1,n]]} P(f(x_{z_{i,j}}) = y_{z_{i,j}}),$$

where for each $(i,j) \in [[1,k]] \times [[1,n]]$, $z_{i,j}$ is uniformly drawn in $[[1,N]]$.

The $\hat{f}_i$ are now simple functions, and not distributions on functions, and are therefore much easier to implement in a computer.

Expected improvement

What is the idea behind Eq. 13.4 ? Instead of considering the best value on average (i.e. $\mathbb{E}\hat{f}(x)$), we consider the best confidence bound for the estimate of $f(x)$. We don't consider what is expected, but what is expected in lucky cases; we increase exploration. When a part of the domain is very precisely known, the variance term $\mathbb{E}(\hat{f}(x) - \mathbb{E}\hat{f}(x))^2)$ becomes very small and other parts of the domain, with more uncertainty, are preferred. Yet, there is an arbitrary constant t. Is there something more natural that could be considered ? A natural solution has been proposed by [140], as follows:

$$\hat{f} = \mathscr{L}(f | f(x_1) = y_1, f(x_2) = y_2, \dots, f(x_n) = y_n); \tag{13.6}$$

$$x_{n+1} = \arg\max_x \mathbb{E} \max(0, \inf_i y_i - \hat{f}(x)). \tag{13.7}$$

Eq. 13.6 is the same as Eq. 13.3; but Eq. 13.7 is different from Eq. 13.4 and in particular has no free parameter. Intuitively, it is the expectation of the improvement: $\inf_i y_i - \hat{f}(x)$ is the improvement over the best already seen point. This solution is therefore termed **expected improvement**. It is more expensive (in terms of computational power per iterate) than just a surrogate model, but it is also much faster (in terms of performance for a given number of iterates) in many cases - it is a really operational approach for expensive optimization, and it naturally handles global optimization - it avoids for sure local minima.

Informational approach to global optimization

Eq. 13.7 is intuitively appealing, but it is not a candidate for exactly reaching mathematical optimality. It is clearly optimal for a "one-step ahead" expected improvement: this is greedy. Is there a solution beyond this ? Such a tool has been proposed in [240], under the name **informational approach to global optimization** (IAGO). The idea is

the following:

$$\hat{f} = \mathscr{L}(f | f(x_1) = y_1, f(x_2) = y_2, \ldots, f(x_n) = y_n); \quad (13.8)$$

$$x_{n+1} = \arg\min_x \mathbb{E}_{y=\hat{f}(x)} \mathbb{E}_{\hat{f}_{x,y}} || \arg\min \hat{f}_{x,y} - \mathbb{E}_{\hat{f}_{x,y}} \arg\min \hat{f}_{x,y} ||^2 | y = \hat{f}(x); \quad (13.9)$$

$$\hat{f}_{x,y} = \mathscr{L}(f | f(x_1) = y_1, \ldots, f(x_n) = y_n, f(x) = y) \quad (13.10)$$

Ouch! This is complicated. What means Eq. 13.10 ? It is the distribution of f, conditionally to past observations, but also conditionally to x and y.

Therefore,

$$\mathbb{E}_{\hat{f}_{x,y}} || \arg\min \hat{f}_{x,y} - \mathbb{E}_{\hat{f}_{x,y}} \arg\min \hat{f}_{x,y} ||^2$$

is the variance of the optimum of f if we assume that we observe $f(x) = y$. Then

$$\mathbb{E}_{y=f(x)} || \arg\min \hat{f}_{x,y} - \mathbb{E}_{\hat{f}_{x,y}} \arg\min \hat{f}_{x,y} ||^2$$

is the expected variance of the optimum of f if we choose x. Therefore Eq. 13.9 consists in choosing x such that the expected variance of the optimum will be as small as possible *after the observation of y*, on average on this observation y. Eq. 13.9 can therefore be rewritten, for short:

$$x_{n+1} = \arg\min_x \mathbb{E}_{y=\hat{f}(x)} Var_{\hat{f}_{x,y}} \arg\min \hat{f}_{x,y}.$$

Compared to Eq. 13.7, Eq. 13.9 is "one step ahead": we consider the uncertainty after one more step. This is clearly more appealing, more efficient in terms of performance for a fixed number of iterations, but it is also much more expensive.

Can we do better and consider k steps ahead ? Or even k equal to the horizon, so that the algorithm is provably optimal ? The answer is positive and proposed in [15]; however, the computational cost becomes extremely big. The interested reader can see more on this (and applications) in section 7.1.

13.4 Conclusion on optimal optimization

Surrogate models are a partial answer to the problem of expensive optimization; they require a tuning of exploration parameters (for choosing to which extent we trust the surrogate model). In order to solve these issues, the research trend on parameter free optimization algorithms which provide (provably) very good results for a fixed number of fitness evaluations or a fixed number of population evaluations is an active field - these algorithms require, however, some prior knowledge of the fitness function. Unfortunately, these algorithms are usually moderately practical, as they are very complicated to implement, and involve a huge computation time internally to the algorithm. IAGO might be a good compromise between tractability and efficiency for applications in expensive optimization; MCTS-based approaches using the POMDP formulation of optimization (see section 7.1 for more on this) are mathematically appealing as they reach optimality (asymptotically in the computational power) but are for the moment a pure research algorithm.

Chapter 14

Complexity of optimization and complexity of parallel optimization[1]

We here consider the complexity of optimization algorithms, and in particular comparison-based optimization algorithms. Results in this chapter summarize [231], [110], [104, 198, 197, 225, 224] and contains also some new ideas (novelty will be discussed in part V).

The main originality of the work presented here is the use of the branching factor and its finiteness. This approach provided realistic tight bounds for many algorithms, recovering and extending existing results (including, unfortunately, bounds published later).

14.1 Deterministic fitness functions

This chapter discusses the advantages (robustness) and drawbacks (slowness) of algorithms searching the optimum by comparisons between fitness values only. The results are mathematical proofs, but practical implications in terms of speed-up for algorithms applied on parallel machines are presented, as well as practical hints for tuning parallel optimization algorithms and on the feasibility of some specific forms of optimization. In all the chapter, $[[a,b]] = \{a, a+1, \ldots, b\}$.

14.1.1 Introduction: comparison-based algorithms and their robustness

There are several important families of optimization algorithms in the literature: Newton-like algorithms, using the Hessian (i.e. the second order derivative); gradient descent, using the gradient (i.e. the first order derivative); and algorithms using

[1] This chapter is based on joint works with R. Coulom, H. Fournier, P. Rolet, N. Sokolovska, F. Teytaud.

the objective function values (also termed the fitness values) only. There are particular cases to be emphasized.

First, Quasi-Newton methods (in particular BFGS [43, 102, 115, 212]) are, formally, in the family of algorithms using the gradient: they build, internally, an approximation of the Hessian, but they never request the Hessian.

Second, some algorithms use less than the fitness values: these algorithms are *comparison-based* algorithms. They include direct search methods [71], and most evolutionary algorithms [189, 30]. This choice of using only a small part of the available information is justified by the followings:

- sometimes it is just the only available information. For example, when optimizing a strategy for two-player games; then, one has only access to a limited information, i.e. the comparison between two strategies;
- whenever more information is available, the robustness is better with comparisons only; this was explained in [16, 244, 17] and was formally established in [111]. Essentially, [111] shows that when considering the worst case among compositions of a given family of fitness functions with increasing mappings (i.e. when considering that the fitness function f might be replaced by $g \circ f$ for some increasing $g : \mathbb{R} \to \mathbb{R}$), then optimality is necessarily reached by a comparison-based algorithm.

Algorithms using comparisons are now widely established and this chapter is devoted essentially to these algorithms (however, the tools used in the proofs can be applied in other cases). In particular, we will consider the following families of algorithms (more formally presented in [230]):

- Selection-based non-elitist (μ, λ) evolution strategies (SB-(μ, λ)-ES). These algorithms, at each generation, generate λ points (also termed individuals) in the search space, and the fitness function must inform the algorithm of which μ of these λ points are the μ best individuals (we have to take care of ties here): these μ points are termed the *selected set*.
- Selection-based elitist $(\mu + \lambda)$ evolution strategies (SB-$(\mu + \lambda)$-ES). These algorithms, at each generation, generate λ points (also termed individuals) in the domain of the optimization, and the fitness function must inform the algorithm of which μ of the union of (i) these λ points and (ii) the μ points selected at the previous generation, are the μ best individuals (we have to take care of ties here).
- Full ranking versions of the algorithms above, i.e. FR-(μ, λ)-ES and FR-$(\mu + \lambda)$-ES; in these cases, the optimization is informed of which μ points are the best, and also of the complete ranking of these μ points.

It is intuitively quite natural that comparison-based algorithms are slower, as they have less information for guiding the search (in particular, it will be difficult for evolution strategies to be as fast as surrogate models like NEWUOA[186]). This is the price to pay for the increased robustness, shown in [111] and well known and widely asserted in evolutionary algorithms. In this chapter, we will:

- introduce, in section 14.1.2, the important notion of branching factor [231]; this notion is central in the understanding of comparison-based algorithms, in parallelization, and it is also important beyond the scope of this chapter;
- show the complexity bounds derived from this notion (section 14.1.3);
- show the computational cost associated to some real-world algorithms (section 14.1.4), which is often significantly different from the theoretical optimum;
- give the implications of these complexity bounds (section 14.1.5).

14.1.2 The branching factor

We present below a simplified version of [230]; please refer to [230] for formal details and detailed proofs. We consider a (μ,λ)-ES (the same reasoning holds for $(\mu+\lambda)$-ES - see [230] for details and more generality). (μ,λ)-ES are as in Alg. 12.

One iteration of a (μ,λ)-ES, for internal state s and fitness function f
Compute individuals $x_1,\ldots,x_\lambda$ as a function of the internal STATE.
Compute their fitness values $y_1,\ldots,y_\lambda$ with $y_i = f(x_i)$.
Consider S the permutation of $[[1,\lambda]]$ uniquely determined by: $y_{S(i)} < y_{S(i+1)}$
switch S **do**
 case S_1
  $s = updateFormula1(s,x_1,\ldots,x_\lambda)$.break;
 end
 case S_2
  $s = updateFormula2(s,x_1,\ldots,x_\lambda)$.break;
 end
 case S_3
  $s = updateFormula3(s,x_1,\ldots,x_\lambda)$.break;
 end
 ... **case** S_K
  $s = updateFormulaK(s,x_1,\ldots,x_\lambda)$.break;
 end
 otherwise
  No other case should never be raised.
 end
end

Algorithm 12: One iteration of a (μ,λ)-ES (simplified by the assumption that there's no tie). We here assume that all the y_j's are distinct. Each S_i is a fixed permutation of $[[1,\lambda]]$; therefore $K = \lambda!$.

Of course, Algorithm 12 does not mean that the algorithm must absolutely be written under this form in order to be under the scope of the bounds in this chapter; it must

only be *equivalent* to an algorithm written under this form. In particular, most evolutionary algorithms (at least (μ,λ)-ES and $(\mu+\lambda)$-ES) can be rewritten under this form.

The constant K (i.e. the number of branches in the SWITCH) is the branching factor. More precisely, the minimum number K such that the algorithm is still equivalent to the initial version should be considered; therefore, we can consider a rewriting of the algorithm as presented in Algorithm 13 - several cases are grouped into only one update formula in order to reduce the branching factor. For example, in the case of SB algorithms, K is at most $\binom{\lambda}{\mu}$: there are only $K=\binom{\lambda}{\mu}$ different update formula, one for each possible selected set. In the case of FR algorithm, there are $K=\binom{\lambda}{\mu}\mu!$ different update formulas at most.

One iteration of a SB-(μ,λ)-ES, for internal state s and fitness function f:
Compute individuals $x_1,\ldots,x_\lambda$ as a function of the internal state.
Compute their fitness values $y_1,\ldots,y_\lambda$ with $y_i=f(x_i)$.
Consider S the permutation of $[[1,\lambda]]$ uniquely determined by:
$y_{S(i)}<y_{S(i+1)}$
switch S **do**
 case S_1 ;
 case S_2 ;
 case S_3
  $s=updateFormula1(s,x_1,\ldots,x_\lambda)$.break;
 end
 case S_4 ;
 case S_5
  $s=updateFormula2(s,x_1,\ldots,x_\lambda)$.break;
 end
 ...
 case S_{L-2} ;
 case S_{L-1} ;
 case S_L
  $s=updateFormulaK(s,x_1,\ldots,x_\lambda)$.break;
 end
 otherwise
  No other case should never be raised.
 end
end

Algorithm 13: A rewriting of Algorithm 12, by grouping cases leading to the same update formula. Please note that the number of cases leading to the same update formula does not need to be the same for all update formulas; the important number is only the total number of update formula.

We have seen how to upper bound the branching factor, by rewriting the Algorithm in order to group the cases of the SWITCH. This uses the fact that the formula is the same for several cases, for example all rankings of the λ points which lead to the same selected set.

However, a second tool can be used for removing some branches: removing cases which are not possible because there's no fitness function which leads to this permutation S. At first view, all permutations are possible; however, some permutations are very unlikely; e.g. it is very unlikely that the crosses with circles are selected in Fig. 14.1. The essential principle in [230], for improving bounds in [231] is to reduce the branching factor accordingly, thanks to assumptions on the set of fitness functions.

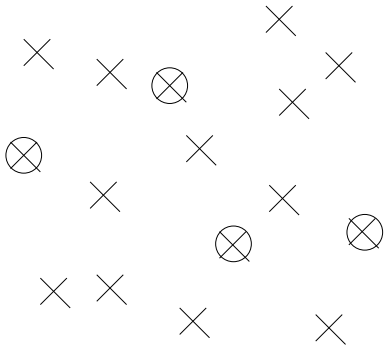


Figure 14.1: Unlikely selected set. The individuals are the crosses, with circles for the selected individuals.

14.1.3 Complexity bounds

We express bounds in terms of the convergence ratio. The convergence ratio is defined in [230] as

$$CR_\varepsilon = \frac{\log N(\varepsilon)}{d n_\varepsilon}, \tag{14.1}$$

where

- n_ε is the number of iterations necessary for ensuring that with probability at least $\frac{1}{2}$, the algorithm has an estimate of the location of the optimum with precision ε for the euclidean norm;
- d is the dimension of the search space;
- $N(\varepsilon)$ is the minimum number k such that there are at least k points in the domain with pairwise distance at least 2ε. Typically, $N(\varepsilon)$ is equal to the cardinal of the search space if it is finite and if ε is small enough, and $N(\varepsilon) = \Theta(\frac{1}{\varepsilon^d})$ if the search space is an open subset of $\mathbb{R}^d$.

The constant $\frac{1}{2}$ is arbitrary and very similar results are derived for a confidence $1-\delta$ (see [230] for more details).

In the continuous case, this quantity is related to convergence rate through the formula

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

The advantage of CR_ε is that it is inversely proportional to the computational cost for a given precision, and parallel speed-ups can be expressed by divisions between various CR_ε.

Combining combinatorial arguments on the branching factor and geometrical tricks (using VC-dimension), [231, 230] get bounds provided in Table 14.1 on the convergence ratio.

14.1.4 The limited speed-up of many real-world algorithms

There are a lot of different methods for the *updateFormula* function. The internal state to be updated contains usually, at least, (i) the step-size σ and (ii) the mean of a Gaussian distribution, to be used for generating new points. The most difficult issue is usually the update of σ ; σ is the standard deviation of the Gaussian distribution used in Alg. 14.

$\sigma = \sigma_0$
$n = 0$
while halting criterion not reached **do**
  $x_{n,1}, x_{n,2}, \ldots, x_{n,\lambda} = Gaussian(x_n, \sigma_n)$
  $\forall i \in [[1, \lambda]]$, $y_{n,i} = f(x_{n,i})$
  $(x_{n+1}, \sigma_{n+1}) = updateFormula(x_{n,1}, x_{n,2}, \ldots, x_{n,\lambda}, y_{n,1}, y_{n,2}, \ldots, y_{n,\lambda})$
end while

Algorithm 14: A typical evolutionary algorithm in the continuous domain.

Three of the most widely known methods for updating σ are the one-fifth rule [189], the self-adaptation (SA) [189, 210] and the cumulative step-size adaptation (CSA) [126]. We have seen in section 14.1 that the optimal speed-up, for λ sufficiently large, is $\theta(\log(\lambda))$; we'll see now that these methods do not reach the optimal speed-up, at least under their usual specifications.

We will use $\eta^* = \sigma_{n+1}/\sigma_n$. The important point is that the log-convergence rate is lower bounded by $\mathbb{E}\log\eta^*$. Therefore, if we can build an absolute lower bound $\mathbb{E}\log\eta^* = \log(\Omega(1))$, this provides an absolute upper bound on the convergence ratio, or if you prefer a lower bound on the convergence rate, whereas we know that the convergence ratio should be $\Theta(\log(\lambda))$ for well designed algorithms - equivalently, the convergence rate should be $\exp(-\Theta(\log(\lambda))$.

We will now see, in sections below, that some well known algorithms have $\mathbb{E}(\log\eta^* | x_n, \sigma_n) \geq C$ for some $C > -\infty$ and independent of λ. As the $O(\log\lambda)$ is provably tight, this shows that improvements are possible here: the difference between the $\Theta(\log\lambda)$ and and $O(1)$ in algorithms below show that the algorithms cited below (which cover most of evolutionary algorithms in the continuous domains) are not optimal.

The one-fifth rule

The one-fifth rule is a very common rule for the update of σ. The idea is to increase σ if the probability of success p is greater than $\frac{1}{5}$, and to decrease it otherwise. The probability of success is the probability that an offspring is better than his parent (Formally, in case of minimization, there is success for the i^{th} offspring if $f(x_{n,i}) < f(x_n)$).

An usual implementation is

- $p \leq \frac{1}{5} \Rightarrow \eta^* = K_1 \in]0,1[$, corresponding to the decreasing case, we want $\sigma_{n+1} < \sigma_n$.
- $p > \frac{1}{5} \Rightarrow \eta^* = K_2 > 1$, corresponding to the increasing case, σ_{n+1} must be greater than σ_n.

In one line, this is

$$\hat{p} \leq 1/5 \Rightarrow \eta^* = K_1 \in]0,1[\text{ and } \hat{p} > 1/5 \Rightarrow \eta^* = K_2 > 1 \tag{14.2}$$

It's easy to see that, in the first case, $\eta^* \geq K_1 > 0$ and in the second case $\eta^* > 1$. Therefore, there exists a constant C such that $\mathbb{E} \log \eta^* > C$.

The one-fifth rule can also be expressed as

$$\eta^* = K_3^{(\hat{p}-1/5)} \text{ for some } K_3 > 1. \tag{14.3}$$

here also, it's easy to see that $\eta^* \geq K_3^{-\frac{1}{5}} > 0$, therefore the same conclusion, namely $\mathbb{E} \log \eta^* \geq C > -\infty$ holds, for some C independent of λ.

As a consequence, the one-fifth rule does not have the optimal speed-up $\log(\lambda)$ with its usual parametrization. Increasing K_3 (as a function of λ) might solve this; we will not develop this here.

Self-adaptation

Self-adaptation is another well known algorithm for choosing the step-size (Algorithm 15).

Initialize $\sigma^{avg} \in \mathbb{R}$, $x_0 \in \mathbb{R}^d$.
while We have time **do**
  for $i = 1..\lambda$ **do**
    $\sigma_i = \sigma^{avg} e^{\tau \mathcal{N}_i(0,1)}$
    $z_i = \sigma_i \mathcal{N}_i(0, Id)$
    $x_{n,i} = x_n + z_i$
    $f_i = f(x_{n,i})$
  end for
  Sort the individuals by increasing fitness; $f_{(1)} < f_{(2)} < \cdots < f_{(\lambda)}$.
  $z^{avg} = \frac{1}{\mu} \sum_{i=1}^{\mu} z_{(i)}$
  $\sigma^{avg} = \frac{1}{\mu} \sum_{i=1}^{\mu} \sigma_{(i)}$
  $x_{n+1} = x_n + z^{avg}$
end while

Algorithm 15: Self-adaptation algorithm. τ usually depends on the dimension only. As well as for the one-fifth rule and cumulative step-size adaptation, the speed-up is $\Theta(1)$ independently of λ.

η^* is an average between log-normal random variables. Unfortunately, if $\mu = \lfloor \lambda/4 \rfloor$, then even if the μ selected mutations correspond to the smaller values of σ, $\log(\sigma)$ is, on average, decreased by $-\log(\tau Q_{\frac{1}{4}} \mathcal{N})$, where $Q_{\frac{1}{4}} \mathcal{N}$ is the average of the first quartile of the standard Gaussian variable. One can show, as well as for the one-fifth rule, that $\mathbb{E} \log(\eta^*) \geq C > -\infty$. Therefore, SA does not have the optimal speed-up $\log(\lambda)$ with its usual parametrization. Increasing τ (as a function of λ) might solve this.

Cumulative step-size adaptation

A third method for updating the step size is the cumulative step-size adaptation (CSA). The idea of this method is to look at the path followed by the algorithm, and to compare its length to the expected length under random selection, and to increase σ if the first path is greater than the second one, and decrease σ in the other case.

We formalize an iteration of CSA in dimension d as follows; we don't have to assume anything on χ_d and p_c except assumptions 14.9 and 14.10:

$$\sigma_{n+1} = \sigma_n \exp\left(\left(\frac{||p_c||}{\chi_d} - 1\right) \cdot \frac{c_\sigma}{d_\sigma}\right) \quad (14.4)$$

$$\sum_{i=1}^{\mu} w_i = 1 \text{ choose your weights, provided the sum is 1} \quad (14.5)$$

$$\mu_{eff} = \frac{1}{\sum_{i=1}^{\mu}(w_i^2)} \quad (14.6)$$

$$d_\sigma = 1 + 2\max(0, \sqrt{\frac{\mu_{eff}-1}{d+1}} - 1) \quad (14.7)$$

$$c_\sigma = \frac{\mu_{eff}+2}{d+\mu_{eff}+3} \quad (14.8)$$

$$\chi_d > 0 \quad (14.9)$$

$$||p_c|| \geq 0. \quad (14.10)$$

($||.||$ does not have to be a norm, we just need Eq. 14.10) These assumptions, to the best of our knowledge, hold in all current implementations of CSA. They do not completely specify the algorithm, but are sufficient for our purpose - all algorithms matching these equations are covered by our result.

One can easily show that Eqs. 14.5-14.10 imply that $\forall\lambda, \mathbb{E}(\log\eta^*|x_n,\sigma_n) \geq -1$; for this algorithm also, we see that $\exists C, \forall\lambda, \mathbb{E}\log\eta^* \geq C > -\infty$. CSA does not have the optimal speed-up $\log(\lambda)$ with its usual parametrization. Increasing c_σ/d_σ might solve this.

14.1.5 Implications

These results have several implications on practice.

Changing usual algorithms for λ large.

The first consequence, around parallelism, is implied by the combination of section 14.1.3 (which shows complexity bounds) and section 14.1.4 (which shows the speed-up of usual algorithms like cumulative step-size adaptation, the one-fifth rule, and self-adaptation). The results show that these three rules, as usually parametrized, cannot reach the logarithmic speed-up $\Theta(\log(\lambda))$ for λ large, and have even a bounded speed-up $\Theta(1)$. However, this might be easy to modify by adapting constants; for example, increasing the log-normal mutation strength as a function of λ, for the self-adaptation of σ, might solve this issue. Also, modifying $\frac{c_\sigma}{d_\sigma}$ as a function of λ, for CSA, might solve this issue for CSA. As an illustration, we show in Fig. 14.2 the great improvement provided by the reduction of μ (in order to avoid the weakness pointed out in section 14.1.4) on the most recent SA variant. This is certainly an example of theory which has a direct impact on practice, with more than 100 % speed-up on our graph, increasing as the number of processors increases, with only one line of code modified in SA. Other such applications are discussed in [224] - the same modification has an impact on many algorithms.

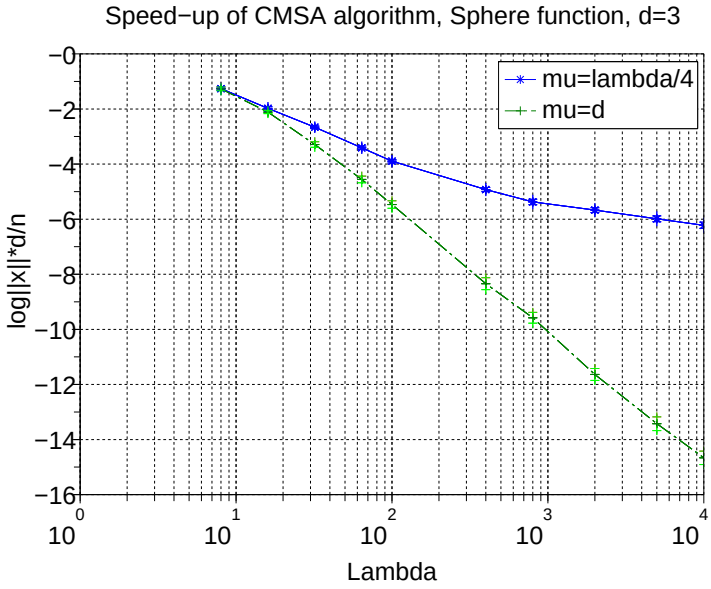


Figure 14.2: Example of the limited speed-up of real-world algorithms. The Covariance Matrix Self-Adapatation (CMSA) Evolution Strategy is an algorithm using the Self-Adaptation rule combined with a full covariance matrix. This experiment is done in dimension 3, and we look at the distance to the optimum normalized by the dimension, divided by the number of generations of the algorithm (the lower the result, the better; this is a normalized convergence rate). With usual initialization, we have a selection ratio $\frac{\mu}{\lambda}$ equals to $\frac{1}{4}$. As we can note, using a smaller selection ratio (here the selection ratio is equal to $\min(d, \lfloor \lambda/4 \rfloor)/\lambda$ is a much better choice. With this improvement we can reach the theoretical logarithmic speed-up.

Choice of the algorithm, given a number λ of processors. Let's consider the choice of an algorithm, as a function of λ; this is the case in which λ is equal to the number of computing units available. New machines have an increasing number of cores, clusters or grids have thousands of cores, and as "jobs" submitted on grids must sometimes be grouped the value of λ can be huge, beyond tenths of thousands.

It is known [30] that the evolution strategies don't all have the same speed-up. If λ is small in front of the dimension, $(\mu/\mu, \lambda)$-ES can reach a linear speed-up, whereas $(1, \lambda)$-ES have only logarithmic speed-up. If λ is small or of the same order as the dimension, this suggests that $(\mu/\mu, \lambda)$ is better than $(1, \lambda)$.

For λ large, [230] (summarized in Table 14.1) has shown that the theoretical speed-up is $\Theta(\log(\lambda))$ for both algorithms (namely $(\mu/\mu, \lambda)$ and $(1, \lambda)$), at least for good parametrizations of these families of algorithms. However, as shown in section 14.1.4, most usual $(\mu/\mu, \lambda)$ evolution strategies have limited speed-up $\Theta(1)$, and therefore their speed-up is much worse than $(1, \lambda)$-ES which reaches $\Theta(\log(\lambda))$! Should we

deduce from this that $\mu = 1$ is better when λ is large ? In fact, choosing μ linear as a function of λ (roughly one forth in many papers) is not a good idea for algorithms based on recombination by averaging. Maybe $\mu = \min(d, \lambda/4)$ could be a good idea; but this will not be sufficient (except maybe for SA ?). Our results are independent of μ in CSA, therefore changing μ in CSA is not sufficient for ensuring $\log(\lambda)$ speed-up in CSA, but preliminary investigations suggests that this formula for μ, combined with the modifications suggested above on $\frac{c_\sigma}{d_\sigma}$, might give good results.

This suggests that a lot of work remains around the case of λ larger than the dimension. In particular [222] has shown that EMNA [150] is, for λ large, much faster than most usual algorithms, and [220] has shown that some algorithms could be improved by a factor 8 by combining tricks dedicated to λ large.

Implication for the structure of the set of solutions. An assumption of [231], and in all bounds summarized in Table 14.1, is that there is one and only one optimum. The results can be extended to finitely many optima, but not to the general case of complex sets of optima. If there are useless variables, the set of solutions is not a point but a linear subspace. This strongly reduces the complexity; roughly, the dimension of the search space in Table 14.1 is replaced by the codimension of the set of solutions, *i.e.* the dimension *minus* the dimension of the set of optima. This implies that evolutionary algorithms might be able to solve problems in very high dimension, provided that the set of solutions has a large dimensionality also - this is in particular the case when there are variables with no impact of the fitness function, a case which often occurs in reality. Testing evolutionary algorithms in high dimension (> 10000), but with plenty of useless variables, instead of low dimension as usually done in research papers, might be interesting.

Implications for multi-objective algorithms. An original application of the branching factor and of bounds derived with it is [226]. We have seen in Eq. 14.1 that the convergence ratio depends on the packing numbers (*i.e.* $N(\varepsilon)$); this means that the number of fitness evaluations strongly depends on the packing number, *i.e.*, for a precision ε, the number of disjoint balls of radius ε that can be put in the domain. Unfortunately in the multi-objective case, when the number of conflicting objectives is large, then the packing number is huge. Thanks to this principle, [226] shows that, even if we restrict our attention to problems with Lipschitzian Pareto fronts, finding the optimal Pareto front with precision ε and confidence $1-\delta$ for the supremum norm requires a number of fitness evaluations $\Omega(1/\varepsilon^{d-1}) + \log_2(1-\delta)$

- if the algorithm is based on Pareto-comparisons between individuals (i.e. we only know which points dominate which points);
- and if the number of objectives[2] is d.

This is close to the efficiency of the random search in the fitness space[3] (The runtime of random search in fitness space is $O(1/\varepsilon^d)$, neglecting logarithmic factors and dependencies in d); this means that all comparison-based algorithms require, if they want to be significantly faster than random search, either:

[2] Importantly, the result is based on the fact that those objectives can all be conflicting[41].

[3] Random search in the fitness space consists in randomly drawing points uniformly in the fitness space, and then discard and redraw those which are on the infeasible side of the Pareto front.

- some feedback from the human user;
- or some more information on the fitness (see e.g. informed operators [201]);
- or moderately many conflicting objectives (see in particular [42]).

14.1.6 Conclusions on the complexity of deterministic optimization algorithms

Some published papers consider the speed-up between an algorithm with a huge population size on k processors and the same population size on $l >> k$ processors. Unfortunately, a great improvement for a fixed population size is pointless if this population size is useless, and if the same convergence rates can be recovered with a population divided by 50.

We have summarized theoretical complexity bounds for classes of algorithms. We have shown that many real-world algorithms are far from these complexity bounds, when λ is large. This suggests several modifications for real-world algorithms, easy to implement and which both provably (see Section 14.1.4 compared to bounds in section 14.1.3) and experimentally (see Fig. 14.2) greatly improve the results for λ large. Take these elements into account if you have a large population size.

14.2 Stochastic fitness functions: adaptive noisy optimization

This section, directly extracted from a joint work with R. Coulom, P. Rolet and N. Sokolovska[76], exhibits lower and upper bounds on runtimes for expensive noisy optimization problems. Runtimes are expressed in terms of number of fitness evaluations. Fitnesses considered are monotonic transformations of the *sphere* function. The analysis focuses on the common case of fitness functions quadratic in the distance to the optimum in the neighbourhood of this optimum—it is nonetheless also valid for any monotonic polynomial of degree $p > 2$, and we recall other results. Upper bounds are derived via a bandit-based estimation of distribution algorithm that relies on *Bernstein races* called R-EDA. It is known that the algorithm is consistent even in non-differentiable cases. Here we show that: (i) if the variance of the noise decreases to 0 around the optimum, it can perform optimally for quadratic transformations of the norm to the optimum, (ii) otherwise, it provides a slower convergence rate than the one exhibited empirically by an algorithm called Quadratic Logistic Regression (QLR) based on surrogate models—although QLR requires a probabilistic prior on the fitness class.

14.2.1 Introduction

The following work deals with expensive noisy optimization. Noisy means that the result of a fitness evaluation at a given point is a random variable, whose probability

distribution depends only on the location of the point—this noise model will be detailed in Section 14.2.2, as well as the class of fitnesses that we address in this chapter. Expensive means that each fitness call is considered costly: for example, evaluating the fitness of an individual might involve the building and testing of a prototype, or hours of simulations on a computer or on a grid. Therefore, an expensive optimization algorithm's performance is measured by the number of fitness calls required to find the optimum with a given precision, rather than considering the computational time required by the algorithm to function.

A practical example of such a framework is searching for the parameters of an algorithm that minimize its probability of failure: a fitness call is then a Bernouilli random variable, resulting for a given parameter vector in a success with probability p and in a failure with probability $1-p$. Each fitness call implies running the algorithm, and as such is quite costly in time.

State of the art

Using evolutionary algorithms and Estimation of Distribution algorithms (EDAs) to deal with noisy fitnesses is a topic that has been substantially discussed in the literature. Notably, many question the idea that repeatedly evaluating the same points in order to average values and decrease noise variance is effective, as compared to, for instance, simply increasing the population size [101, 30, 122, 6, 8]. A simple rule, for evolutionary algorithms, consists in using statistical tests, and to repeat the evaluation of each point in the population until the μ best points in the population are found with confidence at least 95%; we'll see variants of this idea in this chapter, and we'll see also that things are not so simple as knowing exactly which μ points are the best might be extremely expensive if the μ^{th} point and the μ^{th} point have very close fitness values.

A brief survey can be found in [197], where it has been shown that averaging can be efficient when used in the framework of *multi-armed bandits* (following ideas of [128]) and *races*. Specifically, it is proved that an EDA using Bernstein races to choose the number of evaluations of a given point reaches an optimal convergence rate for some noise models.

When dealing with noisy optimization, it is important to distinguish cases in which the variance of the noise decreases to zero near the optimum—which we will refer to as the *small noise* assumption hereafter—and cases where it does not—*large noise* assumption (see section 14.2.2 for more details on noise settings). Small noise has been tackled in [139] for a quite restricted noise model. [7] has then shown that in case of large noise, all usual step-size adaptation rules diverge or stop converging: the usual behavior of evolutionary algorithms for models with large noise is that they stop converging as they get too close to the optimum, and then keep a residual error, with a step-size which does not decrease to zero.

[197] and [198] tackle cases of large noise, but only for linear transformations of the distance to the optimum—excluding the common case of quadratic (or higher-order polynomial) fitness functions. Furthermore, classical algorithms for noise handling such as Uncertainty Handling for Covariance Matrix Adaptation (UH-CMA), empirically quite efficient for small noise, are unfortunately not yet stable enough to deal with large noise cases: for the Scaled-Translated sphere (STS) model presented below,

UH-CMA does not converge. Consistently with these results, [229] has shown that fast convergence involves a number of evaluations running to infinity with the number of iterations. This was further developed in [197, 198] with both lower bounds and algorithms reaching the bound in many cases. However, the natural case of fitnesses that are quadratic in the distance to the optimum was not covered. In the following, we show that:

- the Estimation of Distribution Algorithm defined in [198] and recalled in Algorithm 18, based on a Race (termed R-EDA) has good theoretical guarantees (e.g. outperforming UH-CMA), for both small noise and large noise scenarios;
- for $p = 2$, R-EDA is empirically outperformed in case of large noise by surrogate models such as Quadratic Logistic Regression (QLR), that fits a quadratic model using a Bayesian prior;
- R-EDA also converges at a controlled rate for polynomial functions of the distance to the optimum.

Note that R-EDA has first been used in [198], and has not been modified for this work: all positive properties of R-EDA are preserved, in particular the convergence in many difficult cases, including optimality for fitnesses f such that $f(x) = c + \Theta(\|x - x^*\|)$, i.e. functions that behave similarly to a translated sphere function in the neighborhood of the optimum x^*.

14.2.2 Framework

In this section, our framework for expensive noisy optimization is introduced.

Parameters: N, number of fitness evaluations; t, unknown element of X.
θ: random state of the nature $\in [0,1]^{\mathbb{N}}$; each coordinate θ_i for $i \in \{1,2,\dots\}$ is uniformly distributed in $[0,1]$.
for $n \in [[0, N-1]]$ **do**
  $x_{n+1}^{t,\theta} = Opt(x_1^{t,\theta}, \dots, x_n^{t,\theta}, y_1^{t,\theta}, \dots, y_n^{t,\theta})$
  $y_{n+1}^{t,\theta} = (f(x_{n+1}^{t,\theta}, t) < \theta_{n+1})?1 : 0$ // Return noisy fitness $\sim \mathcal{B}(f(x_{n+1}^{t,\theta}, t))$
end for
$Loss(t, \theta, Opt) = d(t, x_N^{t,\theta})$

Algorithm 16: Noisy optimization framework. Opt is an optimization algorithm taking as input a sequence of visited points and their binary, noisy fitness values. It outputs a new point to be visited, looking for points x of the domain such that $f(x,t)$ is as small as possible. The algorithm Opt is successful on target f parametrized by t and random noise θ if $Loss(t, \theta, Opt)$ is small.

The optimization framework is described in Algorithm 16. This is a black-box optimization framework: the algorithm can request the fitness values at any chosen point, and no other information on the fitness function is available. We consider a fitness function f parametrized by the (unknown) location of its optimum, t. The noise is accounted for by a random variable $\theta \in [0,1]^{\mathbb{N}}$; each coordinate θ_i for $i \in \{1,2,\dots\}$ is

uniformly distributed in $[0,1]$. The goal is to find the optimum t of $f(.,t)$, by observing noisy measurement of f at x_i. Measurements are random variables $F_t(x_i)$ with law $\mathscr{L}$ in $[0,1]$. They satisfy $\mathbb{E}[F_t(x_i)] = f(x_i,t)$. For the proof of the lower bound, the law of random variable $F_t(x_i)$ is Bernouilli, with parameter $f(x_i,t)$ as shown in Algorithm 16. This fits applications based on highly noisy optimization, such as games: let x be a parameter of a game strategy, that we wish to set at its best value; a noisy observation is a game against a baseline, resulting either in a win or in a loss; the aim is to find the value of x maximizing the probability of winning. Usual viability problems or binary control problems tackled by direct policy search also involve this kind of optimization.

We are interested in the number of requests needed for an optimization algorithm to find optimum t with precision ε and confidence $1-\delta$; $\varepsilon = \|x_n - t\|$ is the Euclidian distance between t and the output x_n of the algorithm after n fitness calls. This chapter focuses on fitnesses of the form $(x,t) \mapsto c + \lambda ||x-t||^p$, referred to as the Scaled-Translated sphere (STS) model. It is more general than the STS model of [198] which addresses only $p = 1$. In the following, t is not handled stochastically, i.e. the lower bounds are not computed in expectation w.r.t. all the possible fitness functions yielded by different values of t. Rather, we will consider the worst case on t. Therefore the only random variable in this framework is θ, accounting for noise in fitness measurements, and all probability / expectation operators are w.r.t. θ. For simplicity, we considered only deterministic optimization algorithms; the extension to stochastic algorithms is straightforward by including a random seed of the algorithm in θ.

In the following, $\tilde{O}$ means that logarithmic factors in ε are neglected.

Races

The algorithm used to prove upper bounds on convergence rates is based on Bernstein confidence bounds. It is a variation of the well-known Hoeffding bounds [131] (aimed at quantifying the discrepancy between an empirical mean and an expectation for bounded random variables), which takes variances into account [65, 23, 24]. It is therefore tighter in some settings. A detailed survey of Hoeffding, Chernoff and Bernstein bounds is beyond the scope of this chapter; we will only present the Bernstein bound, within its application to *races*. A *race* between two or more random variables aims at distinguishing with high confidence random variables with better expectation from those with worse expectation. Algorithm 17 is a *Bernstein race* applied to distinct points x_i of a domain X—the 3 random variables are $F_t(x_i)$, the goal is to find a good point and a bad point such that we are confident that the good one is closer to the optimum than the bad one.

It is crucial in this situation to ensure that there exist i, j such that $f(x_i,t) \neq f(x_j,t)$, otherwise the race will last very long, and the output will be meaningless. At the end of the race, $3T$ evaluations have been performed, therefore T is called the halting time. Intuitively, the closer the points x_i are in terms of fitness value, the larger T will be. This is formalized below.

The reason why δ' is used in Algorithm 17 as the confidence parameter instead of δ will appear later on (the notation δ is needed elsewhere).

Let us define $\Delta = \sup\{\mathbb{E}F_t(x_1), \mathbb{E}F_t(x_2), \mathbb{E}F_t(x_3)\} - \inf\{\mathbb{E}F_t(x_1), \mathbb{E}F_t(x_2), \mathbb{E}F_t(x_3)\}$.

Bernstein(x_1, x_2, x_3, δ')
$T = 0$
repeat
$T \leftarrow T + 1$
Evaluate the fitness of points x_1, x_2, x_3 once, *i.e.* evaluate the noisy fitness at each of these points.
Evaluate the precision:

$$\varepsilon_{(T)} = 3\log\left(\frac{3\pi^2 T^2}{6\delta'}\right)/T + \max_i \hat{\sigma}_i \sqrt{2\log\left(\frac{3\pi^2 T^2}{6\delta'}\right)/T}. \tag{14.11}$$

until Two points (*good* ,*bad*) satisfy $\hat{f}(bad) - \hat{f}(good) \geq 2\varepsilon$ — **return** $(good, bad)$

Algorithm 17: Bernstein race between 3 points. Eq. 14.11 is Bernstein's inequality to compute the precision for empirical estimates (see e.g. [87, p124]); $\hat{\sigma}_i$ is the empirical estimate of the standard deviation of point x_i's associated random variable $F_t(x_i)$ (it is 0 in the first iteration, which does not alter the algorithm's correctness); $\hat{f}(x)$ is the average of the fitness measurements at x. $\mathscr{B}(a)$ denotes a Bernoulli random law with parameter a.

It is known [169] that if $\Delta > 0$ and if we consider a fixed number of arms[4],

- with probability $1 - \delta'$, the Bernstein race is consistent: $\mathbb{E}F_t(good) < \mathbb{E}F_t(bad)$;
- the Bernstein race halts almost surely, and with probability at least $1 - \delta'$, the halting time T verifies

$$T \leq K\log\left(\frac{1}{\delta'\Delta}\right)/\Delta^2, \tag{14.12}$$

  where K is a universal constant;
- if, in addition,

$$\Delta \geq C\sup\{\mathbb{E}F_t(x_1), \mathbb{E}F_t(x_2), \mathbb{E}F_t(x_3)\}, \tag{14.13}$$

  then the Bernstein race halts almost surely, and with probability at least $1 - \delta'$, the halting time T verifies

$$T \leq K'\log\left(\frac{1}{\delta'\Delta}\right)/\Delta, \tag{14.14}$$

  where K' depends on C only.

The interested reader is referred to [169] and other references for more information.

[4] We here consider 3 arms only, but more general cases can be handled with a logarithmic dependency (see e.g. [169]).

14.2.3 Lower bound

This section describes a general lower bound derived in [197], and concludes with the application of this bound to the STS model.

Let us consider a domain X, a function $f : X \times X \to \mathbb{R}$, and define

$$d(t,t') = \sup_{x \in X} |f(x,t) - f(x,t')|$$

for t and t' in X. In all the chapter, $B(n,p)$ is a binomial random variable (sum of n independent Bernoulli variables of parameter p).

Theorem 1 *For any optimization algorithm Opt, let $N \in \mathbb{N}^*$ (a number of points visited), $\varepsilon_0 > 0$, $0 < \varepsilon < \varepsilon_0$, $D \in \mathbb{N}^*$, $\delta \in]0,1[$. We assume:*

- $H(\varepsilon_0, D)$: $\forall \varepsilon_1 < \varepsilon_0 \; \exists (t_1, \ldots, t_D) \in X^D, \forall (i,j) \in [[1,D]]^2, i \neq j \Rightarrow d(t_i,t_j) = \varepsilon_1$ *(generalized dimension)*
- $H_{PAC}(\varepsilon, N, \delta)$: $\forall t, P(d(x_N^{t,\theta}, t) < \varepsilon/2) \geq 1 - \delta$.

Then, if $\delta < 1/2D$,

$$P(B(N,\varepsilon) \geq \lceil \log_2(D) \rceil) \geq 1 - D\delta. \tag{14.15}$$

The lower bound is related to a topological property of space X: a number D is taken such that for any distance $\varepsilon < \varepsilon_0$, D equidistant points of X can be found (assumption $H(\varepsilon_0, D)$). This is closely related to the dimension of X: for instance, in $\mathbb{R}^d$, the maximum number of such equidistant points is $d+1$.

The theorem states that if an optimization algorithm is able to find the optimum at precision ε with probability $1-\delta$ in N fitness calls (i.e. the algorithm satisfies assumption $H_{PAC}(\varepsilon, N, \delta)$), then N is necessarily large; the theorem explicitly gives a lower bound on N. Indeed, Eq. 14.15 implies a clearer expression of the lower bound (using Chebyshev inequality):

$$N = \Omega(\log_2(D)/\varepsilon) \tag{14.16}$$

for fixed D, where N is the number of iterations required to reach precision ε with confidence $1-\delta$ for $\delta < 1/2D$. The theorem holds for any monotonic transformation of the sphere function. However, the distance d is not the same for different classes of fitnesses. As mentioned earlier, we are interested in the Scaled-Translated sphere model ($(x,t) \mapsto c + \lambda \|x-t\|^p$ with optimum t).

Corollary 2 *Under the conditions of Theorem 1, for any optimization algorithm learning a fitness of the STS model, if ε_N is the quantile $1-\delta$ of the Euclidean distance to the optimum after N fitness calls and if $p \geq 1$, then $\varepsilon_N = \Omega(log(D)/N)$.*

As stated in [197], the lower bound for $p = 1$ is straightforward, since in this case it is clear that $d(t,t') = \|t-t'\|$. Moreover, in the general STS model, we can show that for any $p \geq 2$, $d(t,t') = \Theta(||t-t'||)$, which validates the above corollary. The lower bound of the corollary is tight for $p = 1$ (see [197]). We will see that it is also tight if $p = 2$ for $c = 0$—in this case, both QLR and R-EDA reach this dependency.

14.2.4 Upper bounds

Upper bounds on the convergence rate for the STS model will now be presented, using Algorithm 18 along with a Bernstein race. In the model restricted to $p = 1$, upper bounds for small noise (i.e. $c = 0$) have been derived in [197], and upper bounds for large noise (i.e. $c > 0$) have been derived in [198]. In both cases, the bounds match the lower bound. This is why we focus on $p \geq 2$, which includes the case $p = 2$ that often appears in practice. In this section, the optimum will be referred to as x^*, and $f(x, x^*)$ will be noted $f(x)$ for short.

$n \leftarrow 0$
while True **do**
  $c = \arg\max_i (x_n^+)_i - (x_n^-)_i$ // Pick the coordinate with highest uncertainty
  $\delta_n^{\max} = (x_n^+)_c - (x_n^-)_c$
  for $i \in [[1,3]]$ **do**
    ${x'}_n^i \leftarrow \frac{1}{2}(x_n^- + x_n^+)$ // Consider the middle point
    $({x'}_n^i)_c \leftarrow (x_n^-)_c + \frac{i-1}{2}(x_n^+ - x_n^-)_c$ //The c^{th} coordinate may take 3 $\neq$ values
  end for
  $(good_n, bad_n) = Bernstein({x'}_n^1, {x'}_n^2, {x'}_n^3, \frac{6\delta}{\pi^2(n+1)^2})$.
  // A good and a bad point
  Let H_n be the halfspace
  $\{x \in \mathbb{R}^D; ||x - good_n|| \leq ||x - bad_n||\}$
  Split the domain: $[x_{n+1}^-, x_{n+1}^+] = H_n \cap [x_n^-, x_n^+]$
  $n \leftarrow n + 1$
end while

Algorithm 18: R-EDA: algorithm for optimizing noisy fitness functions. *Bernstein* denotes a Bernstein race, as defined in Algorithm 17. The initial domain is $[x_0^-, x_0^+] \in \mathbb{R}^D$, δ is the confidence parameter.

Sketch of Algorithm 18. We will use Algorithm 18 for showing the upper bounds. It proceeds by iteratively splitting the domain in two (not necessarily equal) halves, and retaining the one that most probably contains the optimum. At iteration n, from the n_{th} domain $[x_n^-, x_n^+]$, the $(n+1)_{th}$ domain $[x_{n+1}^-, x_{n+1}^+]$ is obtained by:

- Finding the coordinate c such that $\delta_n^{max} = (x_n^+)_c - (x_n^-)_c$ is maximal;
- Selecting three regularly spaced points along this coordinate (see Figure 14.3);
- Repeatingly assessing those 3 points until we have confidence that the optimum is closer to one point ${x'}_n^i$ than to another ${x'}_n^j$ (by Bernstein race);
- Splitting the domain by the hyperplane in the middle of these points and normal to the line they define, and keeping only the side of the domain containing ${x'}_n^i$.

It is important to notice that three points selected at each iteration are necessarily distinct. A key element in proving upper bounds with this algorithm is that the fitness monotonic in the distance to the optimum ($\|a - x^*\| > \|b - x^*\| \Rightarrow f(a) > f(b)$), and

it also has spherical symmetry ($\|a-x^*\| = \|b-x^*\| \Rightarrow f(a) = f(b)$). Consequently, it is guaranteed that when choosing three points as in Algorithm 18, at least one of them will have an expected fitness that is different from two others. That is why the race will output a consistent result with high probability.

For simplicity, it is assumed that the initial domain is a hyperrectangle. Consequently, at any iteration n, the halfspace H_n is a hyper-rectangle, whose largest axis' length δ_n^{max} (defined in Algorithm 18) satisfies $\delta_n^{max} \leq \frac{3}{4}^{\lfloor n/D \rfloor}$. The straightforward proof of this fact is given in [197], where R-EDA first appears.

The following lemma will be used for the upper bound. A similar lemma was published in [197], but it only applied to $p = 1$. Notations are those introduced in Algorithm 18.

Lemma 3 (The conditions of the Bernstein race are met) *Assume that* $x^* \in [x_n^-, x_n^+]$ *and* $p \geq 2$. *Then*

$$\max_{(i,j)\in[[1,3]]^2} f(x'^j_n) - f(x'^i_n) \geq 2\left(\frac{\delta^n_{max}}{2}\right)^p. \tag{14.17}$$

Theorem 4 (Upper bounds for the STS model) *Consider the STS model, and a fixed dimension D. The number of evaluations requested by R-EDA (Algorithm 18) to reach precision* ε *with probability at least* $1-\delta$ *is* $\tilde{O}(\frac{\log(1/\delta)}{\varepsilon^{2p}})$.

Proof of Theorem 4. First, note that at iteration n, ε is upper bounded by $\|x_n^- - x_n^+\|$. Eq. 14.17 (shown in Lemma 3) ensures that $\Delta_n = \Omega(\|x_n^+ - x_n^-\|^p)$ (Δ_n is defined by Eq. 14.2.4). Therefore, applying the concentration inequality, presented as Eq. 14.12, the number of evaluations in the n^{th} iteration is at most

$$\tilde{O}\left(\log\left(\frac{6\delta}{\pi^2(n+1)^2}\right)/\|x_n^- - x_n^+\|^{2p}\right). \tag{14.18}$$

Now, let us consider the number $N(\varepsilon)$ of iterations before a precision ε is reached. Eq. 14.2.4 shows that there is a constant $k < 1$ such that $\varepsilon \leq \|x_n^+ - x_n^-\| \leq Ck^{N(\varepsilon)}$. Injecting this in Eq. 14.18 shows that the cost (the number of evaluations) in the last call to the Bernstein race is

$$Bound_{last}(\varepsilon) = \tilde{O}\left(-\log\left(\frac{6\delta}{\pi^2(N(\varepsilon)+1)^2}\right)/\varepsilon^{2p}\right). \tag{14.19}$$

Since $N(\varepsilon) = O(\log(1/\varepsilon))$, $Bound_{last} = O(\log(\log(1/\varepsilon)/\delta))/\varepsilon^{2p}$. For a fixed dimension D, the cost of the $(N(\varepsilon)-i)^{th}$ iteration is $O(\lceil Bound_{last}/(k')^i \rceil)$ because the algorithm ensures that after D iterations, $\|x_n^+ - x_n^-\|$ decreases by at least $3/4$ (see Eq. 14.2.4). The sum of the costs for $N(\varepsilon)$ iterations is the sum of $O(Bound_{last}/(k')^i)$ for $i \in [[0, N(\varepsilon)-1]]$, that is $O(Bound_{last}/(1-k')) = O(Bound_{last})$ (plus $O(N(\varepsilon))$ for the rounding associated to the $\lceil ... \rceil$). The overall cost is therefore $O(Bound_{last} + \log(1/\varepsilon))$, yielding the expected result. □

Proof of Lemma 3. Let $\bar{x_n^*}$ be the projection of x^* on the line on which $x_n'^1, x_n'^2, x_n'^3$ lie. The result will now be proved for $(\bar{x_n^*})_c \in [(x_n'^1)_c, (x_n'^2)_c]$. The proof for the case $(\bar{x_n^*})_c \in [(x_n'^2)_c, (x_n'^3)_c]$ is symmetric (see Figure 14.3).

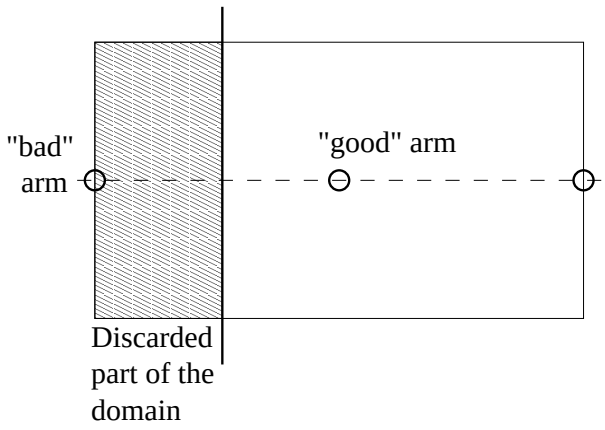


Figure 14.3: The large rectangle is the domain $[x_n^-, x_n^+]$. The three circles are arms x'^1_n, x'^2_n, x'^3_n; the left arm is the "bad" arm, whereas the arm in the center is the "good" arm, *i.e.* the one which proved to be closer to the optimum than the left arm, with confidence $1-6\delta/(\pi^2 n^2)$.

First of all, we have

$$\Delta_n \doteq \max_{i,j\in[[1,3]]^2} f(x'^i_n) - f(x'^j_n) \geq f(x'^3_n) - f(x'^2_n).$$

By Pythagora's theorem, $\forall i \in [[1,3]], ||x'^i - x^*||^2 = ||x'^i_n - \bar{x^*_n}||^2 + ||\bar{x^*_n} - x^*||^2$. Thus,

$$\Delta_n \geq \left(\sqrt{||x'^3_n - \bar{x^*_n}||^2 + ||\bar{x^*_n} - x^*||^2}\right)^p - \left(\sqrt{||x'^2_n - \bar{x^*_n}||^2 + ||\bar{x^*_n} - x^*||^2}\right)^p.$$

Note that $||x'^3 - \bar{x^*_n}|| = ||x'^2_n - \bar{x^*_n}|| + \delta^{\max}_n/2$. Define $d = ||\bar{x^*_n} - x^*||^2$ and $a = ||x'^3 - \bar{x^*_n}||$. Then, observing that $\delta^{\max}_n \geq a \geq \delta^{\max}_n/2$, we have

$$\begin{aligned}
\Delta_n &\geq \left(\sqrt{a^2+d}\right)^p - \left(\sqrt{(a-\delta^{\max}_n/2)^2+d}\right)^p \\
&\geq a^p \left(\left(\sqrt{1+d/a^2}\right)^p - \left(\sqrt{\left(1-\frac{\delta^{\max}_n}{2a}\right)^2 + \frac{d}{a^2}}\right)^p\right) \\
&\geq \left(\frac{\delta^{\max}_n}{2}\right)^p \left((\sqrt{1+d/a^2})^p - (\sqrt{\frac{1}{4}+\frac{d}{a^2}})^p\right). \qquad (14.20)
\end{aligned}$$

By setting $u = d/a^2$, it is clear that Δ_n is greater than the minimum of $u \mapsto (\sqrt{1+u})^p - (\sqrt{1/4+u})^p$ on the interval $[0, D]$ (since $\sqrt{d} = ||\bar{x^*_n} - x^*|| \leq \sqrt{D}\delta^{\max}_n/2$). This function is non-decreasing for $p \geq 2$, and therefore its minimum is its value in 0, which is, for

all $p \geq 2$, at least $\frac{1}{2}$; injecting in Equation 14.20 yields $\Delta_n \geq 2\left(\frac{\delta_n^{\max}}{2}\right)^p$, as stated by Eq. 14.17. □

Theorem 4 can be modified to use the small noise assumption, i.e. the case $c = 0$. We then get a Bernstein's type rate, as follows:

Theorem 5 (Upper bounds for the STS model with small noise) *Consider the STS model, and a fixed dimension D. Assume additionally that $c = 0$, i.e. the scaled sphere model. The number of evaluations requested by R-EDA (Algorithm 18) to reach precision ε with probability at least $1-\delta$ is $\tilde{O}(\frac{\log(1/\delta)}{\varepsilon^p})$.*

Proof of Theorem 5. The variance of a Bernoulli random variable is always upper bounded by its expectation. The case $c = 0$ implies that the expectation is upper bounded by the square of the distance to the optimum. Therefore, Eq. 14.13 holds. Thanks to Eq. 14.13, we can then use Eq. 14.14 instead of Eq. 14.12 in the proof of Theorem 4. This yields the expected result. □

Note that this analysis is not limited to fitnesses that are exactly described by $f(x) = c + ||x - x^*||^p$, but apply to any monotonic transformation of the sphere function that has a Taylor expansion of degree p around its optimum.

14.2.5 Experiments

In this section, we illustrate results of our experiments with an algorithm without surrogate models, UH-CMA, introduced in [125], and an algorithm with surrogate models, QLR (based on Quadratic Logistic Regression).

Experimental results for UH-CMA—optimization without surrogate models

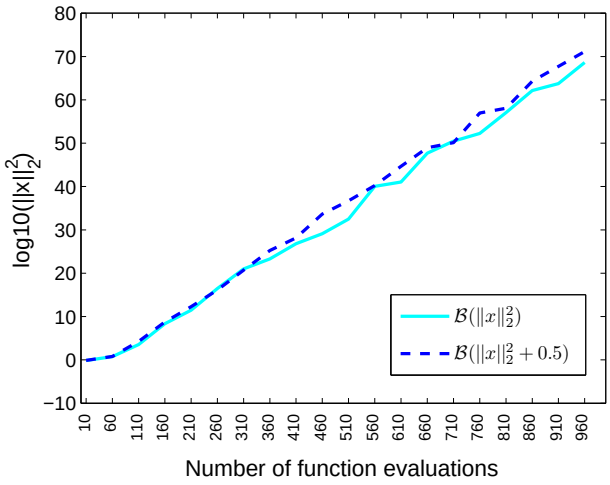
UH-CMA has been developed with intensive testing on the BBOB challenge [14], which includes mild models of noise. See [123] for the source code used in these experiments. The optimization domain is $\mathbb{R}^2$. Let $\mathscr{B}(q)$ denote a Bernoulli distribution of parameter q, $\mathscr{N}(\mu, \sigma^2)$ denote a Gaussian distribution centered on μ with variance σ^2, and $\mathscr{U}(I)$ denote a uniform distribution on interval I. UH-CMA was tested on four different noisy fitnesses: 1) $\|x\|^2(1+\mathscr{N}(0,0.1))$; 2) $\|x\|^2 + \mathscr{U}([0,1])$; 3) $\mathscr{B}(\|x\|^2)$; 4) $\mathscr{B}(\|x\|^2 + 0.5)$.

The initial values required by UH-CMA to start the search were sampled from $\mathscr{U}([0,1]^2)$. The convergence (and divergence) of UH-CMA—illustrated on Figure 14.4—is known to be log-linear.

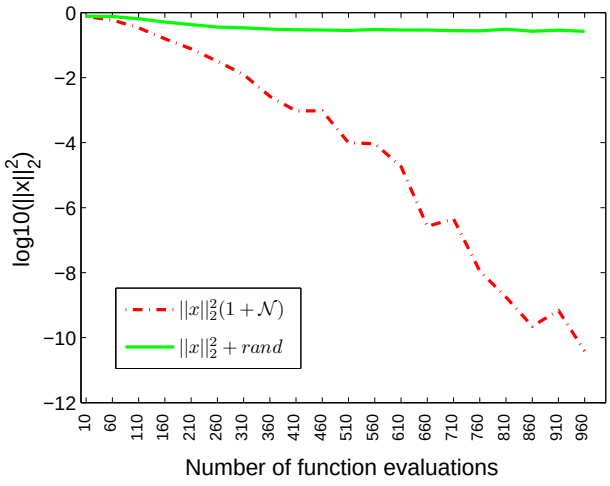
For $||x||^2(1+\mathscr{N}(0,0.1))$, the algorithm converges efficiently: the precision decreases exponentially as the number of iterations increases. For $||x||^2 + \mathscr{U}([0,1])$, the precision stops improving after a few hundred iterations. For $\mathscr{B}(\|x\|^2)$ and $\mathscr{B}(\|x\|^2 + 0.5)$ we observed divergence.

Let us point out that by adding some specific rules for averaging multiple fitness evaluations depending on the step-size, specifically for each fitness function, it is possible to obtain much better rates [124]. However, the rates remain worse than those reached by QLR, as shown in the following section.

Experiments with QLR—optimization with surrogate models



divergence for Bernoulli models



no convergence for $||x||_2^2 + \mathcal{N}$

Figure 14.4: Optimization with UH-CMA (as downloaded at `http://www.lri.fr/~hansen/cmaesintro.html` at the time of [76]), $\log 10$ is the logarithmic function to the base 10. There's a good behavior on $||x||_2^2 \times (1+\mathcal{N})$ (with $\mathcal{N}$ the standard Gaussian noise).

QLR is based on a Bayesian quadratic logistic regression. It samples regions of the search space with maximum variance of the posterior probability, i.e. regions with high variance conditionally to past observations. This is a key difference w.r.t. algorithms

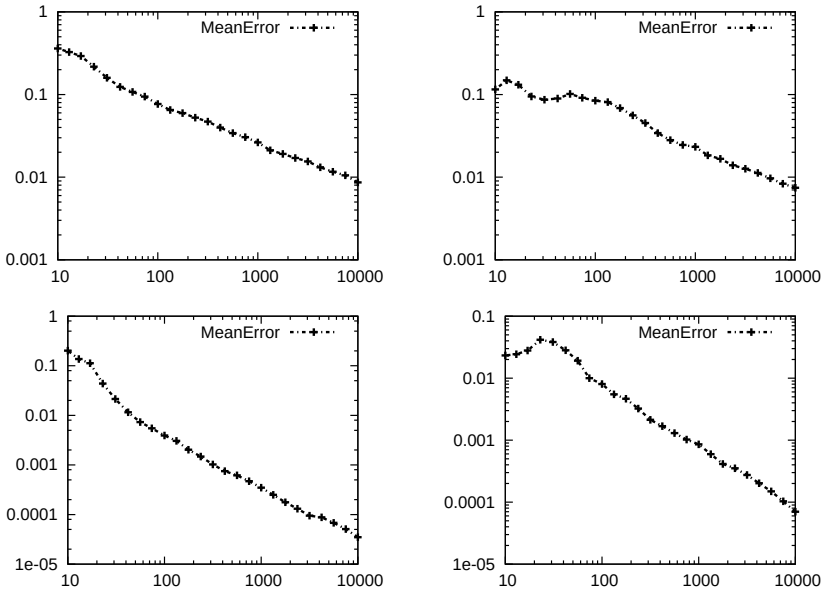


Figure 14.5: Convergence rate of QLR in various cases. On the X-axis: the number of evaluations; on the Y-axis: $\mathbb{E}f(x_n) - \mathbb{E}f(x^*)$. Both are in log-scale to emphasize the exponent. The noisy fitnesses tested are of the form $B(||x||^p + c)$ (top: $p = 1$, bottom: $p = 2$, left: $c = 0$, right: $c = 1/2$). There's convergence in all cases (with exponent $\frac{1}{2}$ (top) and 1 (bottom)); see more in the text.

without surrogate models, which tend to sample points close to the optimum. QLR is fully described by [99, 59, 142] (design of experiments for quadratic logistic model), [206] (active learning for logistic regression). See [75] for the code we used here, specifically tailored to binary noisy fitnesses.

QLR was tested on fitnesses of the form $B(||x||^p + c)$, for p in $\{1,2\}$ and c in $\{0,1/2\}$. The search space is $\mathbb{R}^2$. Figure 14.5 shows the experimental results:

Top left (p=1, c=0): QLR converges on $x \mapsto B(||x - x^*||)$, but with a suboptimal exponent $\frac{1}{2}$ (the slope of the curve is $-\frac{1}{2}$ in log-scale), i.e. $\mathbb{E}f(x_n) - \mathbb{E}f(x^*) \simeq \Theta(1/\sqrt{n})$. R-EDA reaches a better $1/n$ in this case;

Top right (p=1, c=1/2): QLR converges with optimal exponent $1/\sqrt{n}$ also reached by R-EDA;

Bottom left (p=2, c=0): QLR reaches $\mathbb{E}f(x_n) - \mathbb{E}f(x^*) \simeq \Theta(1/n)$ as well as R-EDA;

Bottom right (p=2, c=1/2): QLR still reaches $\mathbb{E}f(x_n) - \mathbb{E}f(x^*) \simeq \Theta(1/n)$ whereas R-EDA only reaches $1/\sqrt{n}$.

14.2.6 Conclusion on the complexity of stochastic fitness functions

The convergence rates for R-EDA (see [198]) and QLR are as follows:

f	$\|\|x_n - x^*\|\|$ for R-EDA	Known lower-bound	$\|\|x_n - x^*\|\|$ for QLR ($p=2$)
$\lambda\|\|x-x^*\|\|$	$\tilde{O}(1/n)$	$\Omega(1/n)$	$\simeq 1/\sqrt{n}$
$\lambda\|\|x-x^*\|\|+c$	$\tilde{O}(1/\sqrt{n})$	$\Omega(1/n)$	$\simeq 1/\sqrt{n}$
$g(\|\|x-x^*\|\|)$	$o(1)$	–	–
$\lambda\|\|x-x^*\|\|^p+c$	$\tilde{O}(1/n^{1/2p})$	$\Omega(1/n)$	$\simeq 1/\sqrt{n}$
$\lambda\|\|x-x^*\|\|^p$	$\tilde{O}(1/n^{1/p})$	$\Omega(1/n)$	$\simeq 1/\sqrt{n}$
$\lambda\|\|x-x^*\|\|^2$	$\tilde{O}(1/\sqrt{n})$	$\Omega(1/n)$	$\simeq 1/\sqrt{n}$

Convergence rates are given for minimization; the fitness at point x is the Bernoulli random variable $\mathscr{B}(f(x))$ with parameter $\min(1,\max(0,f(x)))$, x_n is the approximation of the optimum after n fitness evaluations, x^* is the optimum, $c>0$, and g is some increasing mapping.

For the rightmost column, it is important to point out that we tested QLR *without* knowledge of the parameter p, so that the comparison with other algorithms is fair. In particular, there is a single algorithm, R-EDA, which provably realizes the upper bounds above; a better algorithm should be better for all cases simultaneously without problem-specific parametrization.

The original results of this chapter are presented by three last rows and the rightmost column; in particular we have shown:

- The upper and lower bounds for an exponent $p>1$;
- For $p=1$ and $c=0$, QLR is not optimal; R-EDA reaches (provably) $\tilde{O}(1/n)$ whereas QLR has convergence $1/\sqrt{n}$. By construction, it is probably difficult for QLR to do better than $1/\sqrt{n}$;
- For $p=1$ and $c>0$, QLR and R-EDA perform equivalently ($1/\sqrt{n}$); the lower bound does not match the upper bound. For R-EDA we have a mathematical proof and for QLR empirical evidence.
- For $p=2$ and $c=0$, QLR and R-EDA perform equivalently ($1/\sqrt{n}$); the lower bound does not match the upper bound. For R-EDA we have a mathematical proof and for QLR empirical evidence.
- For $p=2$ and $c>0$, QLR (empirically) performs better than the proved upper bound and worse than the proved lower bound.

There is therefore still room for improvements.

Results for QLR and for UH-CMA are empirical, based on current versions of the algorithms. The available implementations of UH-CMA cope quite well with small noise situations, but as soon as the variance does not go to zero sufficiently fast they do not succeed.

R-EDA is efficient in many cases, yet its theoretical convergence rates are sub-optimal in the case $B(c+||x-x^*||^2)$, more relevant from a practical point of view. However, R-EDA is not limited to Bernoulli-like fitness functions, whereas QLR is.

This is why QLR is more efficient in the case $B(c+||x-x^*||^2)$ for $c>0$. UH-CMA does not converge in such cases, what demonstrates that algorithms tailored for small noise models do not easily extend to models with large noise. However, UH-CMA is the only algorithm with log-linear precision as a function of the number of iterations in the easy case $||x-x^*||^2(1+\mathcal{N})$.

Given the convergence rate table above, one can see that lower bounds for $p>1$ or $c>0$ are not tight. A relevant further work would be either to find out how to reach these bounds, or to prove lower bounds achieving tightness—which seems more likely, given that the current lower bounds are quite optimistic.

14.3 Summary on the complexity of optimization

Due to the lack of widely accepted model of multimodal optimization (see however [45] in the noisy case and [208] in the deterministic case), we mainly have complexity results about the optimization of monomodal functions. In the non-noisy case, it is shown that algorithms based on fitness comparisons have at best a linear convergence rate, with a rate scaling linearly as a function of the population size (i.e. linearly as a function of the number of processors if the population is evaluated in parallel, one individual per processor), and logarithmically beyond. Noisy optimization leads to slower rates, and can benefit from much more processes.

The next chapter will be devoted to practical results around parallelization. BFGS-like algorithms can reach superlinear convergence (at the price of the computation of gradients), and Newton-methods can converge quadratically (at the price of the computation of a Hessian); these results are however only in the case of an infinite precision on real numbers which in practice does not hold.

Framework	SB- (μ,λ) -ES	SB- $(\mu+\lambda)$ -ES	FR- (μ,λ) -ES	FR- $(\mu+\lambda)$ -ES
General case	$\frac{1}{d}\left(\lambda - \frac{1}{2}\log(2\pi\lambda)\right)$	$\frac{1}{d}\left(\log\left(\binom{\lambda}{\mu}\right)\right)$	$\left(\lambda - \frac{1}{2}\log(2\pi\lambda)\right)$ $\times\frac{1}{d}\log(\mu!)$	$\left(\log\binom{\lambda}{\mu}\right)$ $\times\frac{1}{d}\log(\mu!)$
General case, $\mu = 1$	$\frac{1}{d}$ $\times\log(\lambda)$	$\frac{1}{d}$ $\times\log(\lambda+1)$	$\frac{1}{d}$ $\times\log(\lambda)$	$\frac{1}{d}$ $\times\log(\lambda+1)$
VC-dim. V	$\frac{V}{d}\log(\lambda)$	$\frac{V}{d}\log(\lambda+\mu)$	$\frac{V}{d}(4\mu+\log(\lambda))$	$\frac{V}{d}(4\mu+\log(\lambda))$
Quadratic	$O(d\log\lambda)$	$O(d\log(\lambda+\mu))$	$O(d(\mu+\log\lambda))$	$O(d(\mu+\log\lambda))$
Sphere	$(1+\frac{1}{d})\log(\lambda)$	$\log(\mu+\lambda)(1+\frac{1}{d})$	$2\log(\lambda)$	$O(\mu+\log(\lambda))$
Sphere, $\lambda = 2d$			$\Omega(1)$	$\Omega(1)$

Table 14.1: Optimization of deterministic fitness functions: upper bound on the convergence ratio; also some lower bounds on the convergence ratio for $\lambda = 2d$ for the sphere function, in the last row - these lower bounds from [230] show that a linear speed-up can be achieved w.r.t. λ constant for $\lambda = 2d$ (by comparison with the first row). The first row is the general case [231]; it might be better than other rows (for λ small). The second row is when the level sets of fitness functions have VC-dimension V in $\mathbb{R}^d$ [230]. The third row is just the application of the second row to the case of convex quadratic functions ($V = \Theta(d^2)$). The fourth row is the special case of the sphere function [230]. The tightness of the $\log(\lambda)$ dependency will be shown in section 15.2.

Chapter 15

And parallel optimization in practice ? A case in which runtime analysis works[1]

It is usually considered that evolutionary algorithms are highly parallel. In fact, the theoretical speed-ups for parallel optimization shown in [230] are far better than empirical results; this suggests that evolutionary algorithms, for large numbers of processors, are not so efficient, at least for the usual implementations (and parametrization) of many evolution strategies. In this chapter, we show (i) that in many cases automatic parallelization provably provides better results than the standard parallelization consisting in simply increasing the population size λ (ii) that automatic parallelization of evolutionary algorithms has optimal asymptotic speed-up (iii) a simple modification which improves the speed-up of several evolution strategies and estimation of distribution algorithms.

15.1 Introduction to parallel optimization

Chapter 14 has summarized the state of the art for complexity lower bounds in evolutionary algorithms (EA) and parallel EA, derived in previous chapters. Section 15.2 shows how an optimal speed-up for parallel EA can be reached; this is an automatic construction of a parallel algorithm with asymptotically optimal speed-up. Importantly, we have shown in section 14.1.4 that this speed-up is not, by far, reached by existing algorithms. Section 15.3 shows experimentally the efficiency of the automatic parallelization proposed in section 15.2, and also of some other modified (much more convenient) parallelizations of EA. Section 15.4 concludes.

[1] This chapter is based on collaborations with F. Teytaud.

15.2 Automatic parallelization

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 speculative parallelization. We here show that this simple approach can be applied to EA.

As already pointed out in [231], most EA (in particular, all algorithms based on comparisons of fitness values) can be rewritten as follows:

$$(x_{n\lambda+1}^{O_1,O_2},\ldots,x_{(n+1)\lambda}^{O_1,O_2}) = O_1(\theta, I_n) \quad \text{(generation)} \tag{15.1}$$

$$\forall i \in [[n\lambda+1,(n+1)\lambda]], y_i = f(x_i^{O_1,O_2}) \quad \text{(fitness)} \tag{15.2}$$

$$g_n^{O_1,O_2} = g(y_{n\lambda+1},\ldots,y_{(n+1)\lambda}) \quad \text{(selection)} \tag{15.3}$$

$$I_{n+1} = O_2(I_n, \theta, g_n^{O_1,O_2}), \quad \text{(update)} \tag{15.4}$$

for some fixed O_1, O_2, I_0, some random variable θ, and g with values in a set of cardinal 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 extracted from the fitness function, 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 k^{th} 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 will 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 will 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 EA, g_n is a discrete information (typically the ranking of the individuals or the indices of selected individuals), which is the only information that the algorithm extracts from the fitness function. In the FR case and $\mu = \lambda$, for example g_n is $(\text{SIGN}(y_{n\lambda+i} - y_{n\lambda+j})_{(i,j)\in[[1,\lambda]]^2})$ where $\text{SIGN}(t) = 1$ for $t \geq 0$ and $\text{SIGN}(t) = -1$ 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 cardinal K with usually K efficiently bounded. Important examples are:

- (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 $K \leq \lambda$.

The notion of branching factor, and bounds above on the branching factor, have been used in [231, 230] for proving results shown in Table 14.1; we will use it here for proving lower bounds on the parallelization of EA; the lower the branching factor, the better the speed-up.

We will 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

$$\forall\theta, \forall n \geq 0, \forall i \in [[1,\lambda]], x^{O'_1,O'_2}_{n\lambda'+i} = {x^{O_1,O_2}}_{nD\lambda+i}. \tag{15.5}$$

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. (Automatic parallelization of EA - simulation of EA by parallel EA.) *Consider a λ-optimization algorithm (O_1, O_2) as in Eqs 15.1-15.4 with branching factor K, and consider λ' such that for some $D \geq 1$:*

$$\lambda \frac{K^D - 1}{K-1} = \lambda'. \tag{15.6}$$

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

Remark: The speed-up is therefore $D = \frac{\log(1+\frac{\lambda'}{\lambda}(K-1))}{\log(K)}$.

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. 15.5 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^{O_1,O_2}_{nD+i} for $i \in [[0, D-1]]$, i.e.

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

over all fitness functions f and for a fixed value of θ. It has cardinal 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^{O_1,O_2}_i 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

$$\lambda(1+K^1+\dots+K^{D-1}) = \lambda(K^D-1)/(K-1) = \lambda', \tag{15.7}$$

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. □

Importantly, section 14.1.4 has shown that this speed-up is not reached by usual real world algorithms; we will propose simple modifications with a big impact on this.

15.3 Experimental speed-up

In this section, we experimentally evaluate speed-ups.

The theory above proves that the automatic parallelization reaches $\log(\lambda)$, which is asymptotically optimal within a constant factor, but there are algorithms for which the automatic parallelization works only for λ very large, in particular when the full ranking of selected individuals is used (because in this case the branching factor K is much bigger). Therefore, we will provide other tricks for ensuring the $\log(\lambda)$ correction. In this section we propose hand-crafted "$\log(\lambda)$" corrections, i.e. tricks for having speed-up $\log(\lambda)$. A simple solution is as follows in continuous domains.

If the speed-up is bounded, then σ decreases at a constant rate on a multiplicative scale, *i.e.* σ is divided by, at most, a fixed constant, independently of λ. If we want to reach the "$\log(\lambda)$" speed-up, then we must substract $\log(\lambda)$ to $\log(\sigma)$; *i.e.* divide σ by an exponent of λ. We will here apply $\sigma \leftarrow \sigma / \max(1, (\zeta\lambda)^{1/N})$ for some value of ζ.

We consider EMNA, CMA-ES and SA-ES. CMA-ES is particularly interesting, because it is a FR-(μ,λ)-ES, and therefore has a big branching factor $K = \lambda!/(\lambda-\mu)!$, and therefore the automatic parallelization becomes efficient only for huge numbers of processors. More precisely, the numerical application for the automatic parallelization is as follows:

- In the case $N = 2$, $\lambda = 6$ (default suggested value in CMA3.24 for $N = 2$), the speed-up is $D = 2$ only with $\lambda' \geq 4326$ (whereas with equal weights, as for EMNA, we get $\lambda' = 126$);
- In the case $N = 10$, leading to $\lambda = 10$, we get a speed-up 2 with $\lambda' = 302410$ (whereas equal weights as in EMNA lead to a more reasonable 2530).

We see that for CMA the automatic parallelization provides a limited speed-up unless the number of processors is huge (in spite of its asymptotic optimality with a speed-up $\Theta(\lambda)$), and even for EMNA we might hope better results. We will propose below specific parallelizations, better than both (i) the standard parallelization with $\lambda = \lambda'$ and (ii) the automatic parallelization.

Parallelization of EMNA

We present results of the isotropic EMNA (the step-size is a constant, used in all directions), on the sphere function. The presented numbers are the mean progress of the log of the distance to the optimum, multiplied by the dimension[2], estimated with the following experimental conditions:

- Column "baseline": the standard EMNA algorithm from [150], with $\mu = \lambda/4$;
- Column "+QR": EMNA, plus the quasi-random (low discrepancy) mutations as defined in [228] (quasi-random mutations with scrambled Halton sequence, showing, by the way, that the effect of quasi-random on the convergence rate is very strong when λ is large;
- Column "+log(λ)": the same as "+QR", except that we add the log(λ) correction, i.e. we modify σ according to formula $\sigma \leftarrow \sigma / \max(1, (0.15\lambda)^{1/N})$;
- Column "+weighting": the same as "+log(λ), except that we apply the reweighting as in [223].

In all cases the initial step-size is $\sigma = 1$ and the initial point is randomly drawn on the unit sphere with radius $\sqrt{N}$ with N the dimension. The 3 following columns provide the p-value of the comparison between a column and the previous column; the significance is very high. Then, the last column presents the normalized convergence rate of the algorithm with *QR* and reweighting, but without the log(λ)-correction; with this column, we can check that the improvement is due to the log(λ) modification and not to the combination QR+reweighting.

This is detailed in Algorithm 19. The results are then presented in Table 15.1.

[2]It is known that the log-distance to the optimum decreases linearly with the dimension; therefore we multiply the results by the dimension in order to have homogeneous results for various dimensions. Following the theoretical analysis in [230], we expect an improvement as the dimension increases, which is confirmed experimentally here.

Initialize $\sigma \in \mathbb{R}$, $y \in \mathbb{R}^N$.
while Halting criterion not fulfilled **do**
  for $l = 1..\lambda$ **do**
    $z_l = \sigma N_l(0, Id)$
    $y_l = y + z_l$
    $f_l = f(y_l)$
  end for
  if "Reweighting" version **then**
    Let $w(i) = 1/density(x_i)$ // with *density* the density of the
    // distribution used for generating the offspring.
  else
    Let $w(i) = 1$
  end if
  Sort the indices by increasing fitness; $f_{(1)} < f_{(2)} < \cdots < f_{(\lambda)}$.
  $z^{avg} = \frac{1}{\sum_i^\mu w_{(i)}} \sum_{i=1}^{\mu} w_{(i)} z_{(i)}$
  $\sigma = \sqrt{\frac{\sum_{i=1}^{\mu} w(i) ||z_{(i)} - z^{avg}||^2}{\sum_{i=1}^{\mu} w(i) \times N}}$
  if $\log(\lambda)$ version **then**
    $\sigma = \sigma / \max(1, (0.15\lambda)^{1/N})$.
  end if
  $y = y + z^{avg}$
end while

Algorithm 19: The EMNA algorithm with weighted averages. N_l is a Gaussian random variable, or a Gaussian quasi-random variable as in [228] for "QR" versions.

Interestingly, the $\log(\lambda)$ correction is not efficient if we do not apply the reweighting trick from [223]. This is somehow natural, as the $\log(\lambda)$ correction strongly increases the risk of premature convergence, which is reduced by the reweighting.

Parallelization of CMA-ES

We here compare the performance of the standard CMA algorithm with λ equals to the number of processors, and the same CMA but with the $\log(\lambda)$-correction encoded as follows:

$$\sigma = \sigma / \max(1, (\zeta\lambda)^{1/N}). \tag{15.8}$$

We consider f as the best fitness found by the algorithm after a fixed number of evaluations. We report the mean of $\frac{N \cdot \log(f)}{\#evaluations}$ and the mean of $\log(f)$ in table 15.2. The number of function evaluations is $100N^2$. Following [32], we experiment two sizes of population, $\lambda = 8N$ and $\lambda = 8N^2$. If the dimension is small (2) we almost have a speed-up of 2 whatever 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 ($\lambda = 8N^2$). All the parameters are those recommended in CMA 3.24. The initial step-size sigma is 1, and the initial point is randomly chosen according to a standard Gaussian distribution. We show a clear, yet moderate, improvement.

Parallelization of CMSA

The CMSA algorithm is presented in Alg. 10; a very simple and efficient modification is just setting $\mu = \min(\lceil \lambda/4 \rceil, dimension)$ instead of $\mu = \lceil \lambda/4 \rceil$. Results have already been presented in section 14.1.5 and show a very clear and convincing improvement - with just a one-line modification of the code.

15.4 Conclusion on the practical parallelization of optimization

First, we have shown in section 15.2 that theoretical bounds in [230] (recalled in chapter 14.1) are tight for their dependencies in λ.

Second, we have shown in section 14.1.4 that many current algorithms do not match this tight dependency.

The tightness is shown by an explicit construction of a parallel version of EA, which can readily be applied also for direct search methods [71] as well; thanks to this explicit construction, we provide an automatic parallelization with efficient results, outperforming standard parallelization in several cases (see section 15.2); the speed-up of the automatic parallelization, vs just increasing λ, can reach incredible high values thanks to (i) the poor behavior of classical algorithms for λ large and (ii) the good behavior of the automatic parallelization for these algorithms with limited branching factor.

However, the automatic parallelization is far from optimal (for the constant) and it is by far too complicated; therefore, we propose in section 15.3 other "$\log(\lambda)$" corrections, with good empirical results. The improvement for EMNA can reach 280 % in dimension 2 ($\lambda = 6000$), 248% in dimension 3 ($\lambda = 9000$), 100% in dimension 20 ($\lambda = 60000$); for CMA in dimension 2, 10 and 30, the speed-up is moderate with $\lambda = 8N$ and around 100% for $\lambda = 8N^2$. However, whereas the automatic parallelization is a kind of "free lunch" parallelization (it's faster than the sequential algorithm with exactly the same result, as shown by the simulation theorem), we feel that with this rule we might have more premature convergence with CMA with $\lambda = 8N^2$.

A first further work is the design of rules for EA which would be simpler than the automatic parallelization, and as efficient, without risk of premature convergence.

A second further work consists in finding a modification of the automatic parallelization for taking into account the fact that some branches are more likely than others; this is quite straightforward thanks to results in [230]. However, the automatic parallelization proposes very complicated algorithms, and therefore it makes no sense to develop this idea before having found a simplification.

15.5 And parallel optimization for noisy fitness functions ?

The case of noisy fitness functions is quite different. The optimization algorithms proposed in section 14.2 and which reach the $\varepsilon \simeq 1/\sqrt{n}$ convergence rate (ε is the distance to the optimum, n is the number of fitness evaluations) have a number of fitness evaluations per iteration running to infinity. This means that whatever may be your number of processors, the simple parallelization consisting in evaluating the population in parallel has a linear speed-up if the target precision is sufficiently small (we here neglect communication costs, a very reasonable assumption here as for ε sufficiently small the computational power spent on each computation unit will grow to infinity and the communication will remain constant. This leads to the non-surprising conclusion that noisy optimization (which is much slower than deterministic optimization) is extremely parallel.

15.6 Summarizing parallel optimization in practice

Following our tradition of writing succinct summaries, we conclude as follows:

- Optimization can be parallelized:
  - at the level of one fitness evaluation (i.e. the fitness or its gradient is parallelized),
  - at the level of evaluations (for population-based algorithms: one individual per processor),
  - by speculative parallelization (for algorithms with branches).
- Evolutionary algorithms are naturally parallel by distributing the fitness evaluations (second case above), but monomodal algorithms are usually not tuned for the case of large populations, which is the natural framework for many processors; we proposed above simple rules for strongly improving this case (including reweighting, forcing the decrease of σ, limiting the selection ratio).
- Noisy optimization and multimodal optimization are easily highly parallel by parallelization of the fitness evaluations.

Dimension, lambda	Baseline	+QR	+log(λ)	+weight	P-value for +QR	P-value for +log(λ)	P-value for +weight	QR+weight but no log(λ)
2,20	-1.61	-1.91	-0.66	-2.43	0.00	1	0	-2.02
2,60	-2.04	-2.13	-0.27	-3.95	0.00	1	0	-2.17
2,200	-2.17	-2.27	-0.17	-5.31	6e-16	1	0	-2.16
2,600	-2.22	-2.27	-0.14	-6.44	4e-15	1	0	-2.27
2,2000	-2.22	-2.38	-0.13	-7.68	0	1	0	-2.32
2,6000	-2.33	-2.51	-0.13	-8.85	0	1	0	-2.38
3,30	-2.09	-2.49	-0.69	-1.67	0.00	1	0	
3,90	-2.43	-2.52	-0.28	-4.58	2e-05	1	0	
3,300	-2.53	-2.59	-0.21	-6.02	0.00	1	0	
3,900	-2.57	-2.71	-0.17	-7.20	5e-09	1	0	
3,3000	-2.65	-2.87	-0.16	-8.52	0	1	0	
3,9000	-2.77	-2.94	-0.15	-9.63	3e-16	1	0	
5,50	-2.72	-2.96	-0.54	-3.28	1e-12	1	0	-2.72
5,150	-3.02	-3.09	-0.42	-5.60	0.00	1	0	-2.85
5,500	-3.08	-3.26	-0.31	-6.97	2e-14	1	0	-3.00
5,1500	-3.22	-3.41	-0.26	-8.19	1e-12	1	0	-3.17
5,5000	-3.35	-3.63	-0.22	-9.56	0	1	0	-3.32
5,15000	-3.53	-3.74	-0.20	-10.84	1e-15	1	0	-3.53
20,200	-5.56	-5.89	-2.52	-2.24	1e-09	1	0.74	-3.30
20,600	-6.05	-6.55	-1.86	-7.57	0	1	0	-4.83
20,2000	-6.81	-7.17	-1.44	-11.27	1e-13	1	0	-6.29
20,6000	-7.25	-7.73	-1.17	-12.98	0	1	0	-6.75
20,20000	-7.71	-8.03	-0.99	-14.62	0	1	0	-7.36
20,60000	-7.93	-8.09	-0.87	-16.17	1e-08	1	0	-7.96
40,400	-8.36	-8.83	-5.35	-1.31	3e-09	1	1	
40,1200	-9.27	-9.54	-4.33	-2.94	8e-05	1	0.97	
40,4000	-10.00	-10.15	-3.47	-8.25	3e-05	1	0	
40,12000	-10.38	-10.48	-2.88	-16.30	0.01	1	0	

Table 15.1: Table of convergence rates for EMNA. We see that (i) QR works very well (ii) reweighting does not always improve the results (it has been published as a tool against premature convergence and not as a tool for fastening EMNA) (iii) the log(λ) correction greatly improves the results, but only if reweighting is applied; this is somewhat natural, as, without reweighting, the log(λ) correction increases the risk of premature convergence.

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

Table 15.2: Comparison between CMA and CMA with $\log(\lambda)$-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 λ correction (Eq. 15.8) $0.4^{1/2}$ in dimension 2, 1 in dimension 10, $1.3^{1/30}$ in dimension 30. In all cases the λ-correction provides an improvement. Whereas in the case of EMNA we could use the same constant in all cases and the results were very stable as a function of the constant, with CMA we had to modify the constant ζ as a function of the dimension in order to get good results. Also, we have here moderately good results, whereas the improvement was better for EMNA and very important for CMSA.

Part IV

Machine Learning Tools

Chapter 16

Introduction to Machine Learning tools [1]

Machine Learning is the field of computer science devoted to the analysis of programs or robots which learn from examples.

In 1955, Alex Lewyt predicted that "Nuclear powered vacuum cleaners will probably be a reality within 10 years." Early predictions in artificial intelligence were nearly that wrong. **Artificial intelligence** is the design of programs or machines which have some forms of intelligence. The **Turing test** is a test consisting in discussing with a human: if it is not possible, for a human, to guess whether he is discussing with a computer or with a human, then the machine has passed the Turing test. For the moment, if machines had some success when compared to humans for some highly specialized forms of intelligence (for example, playing chess, or playing 9x9 Go, or piloting), they are far from success in the Turing test, or even just for a few minutes standing up in front of journalists (Fig. 16.1). For example, autopilots are better than humans in most situations (in particular, they use less fuel than humans for piloting a plane), they are nonetheless less flexible than humans for difficult and new situations (as in the example of planes in icy environments, in which autopilots are subject to fatal errors, as illustrated by various crashes in aviation history). Also, missiles in modern military aircrafts are extremely autonomous and able of making decisions on a set of possible targets, but they don't replace humans in the plane (Fig. 16.1).

Supervised learning is the automatic building of an application $\hat{f}$, close to an unknown function f, thanks to examples $(x_1, y_1), \ldots, (x_n, y_n)$ where it is assumed that

- either $\forall i \in [[1,n]], f(x_i) = y_i$ (supervised noise-free learning);
- or $\forall i \in [[1,n]], f(x_i) = \mathbb{E}y_i|x_i$ (supervised noisy learning with no bias);

[1] This part, benefiting from collaborations with S. Gelly, J. Mary, P. Rolet and M. Sebag, is devoted to supervised machine learning. Supervised machine learning is useful in many applications, and it is in particular used as a module for many other chapters of this document: surrogate models, approximation of Bellman values, imitation learning. We will first present the terminology of machine learning (chapter 16) and some main tools. The further chapters will be devoted to parallel active learning (chapter 17).

Figure 16.1: Top: A french Mirage 2000-5. Autopilots in missiles are highly autonomous; even the choice of priority among targets is automatically performed by the plane in many modern planes. Bottom: the "HRP-4C" robot, which made mistakes in her facial expressions during her first show in front of journalists in March 2009.

- or other formalizations such that $\forall i \in [[1,n]], f(x_i) \simeq y_i$.

f is sometimes referred to as the **oracle** or **target function** (when y-values are in $\mathbb{R}$) or **target concept** (when y-values are in some discrete set). When a supervised learning algorithm proposes a function $\hat{f}$ (termed a classifier, or predictor) from a given family H, then this H is termed **the family of models** or the model of the learning algorithm. When a supervised learning algorithm proposes a function $\hat{f}$ which has the form of a finite tree with simple functions at each node, as in Fig. 16.3, then this algorithm is termed a **decision tree**.

Some algorithms for supervised learning (which are cited without much details here) are based on the idea of using a given algorithm multiple times on modified versions of the original, complete dataset:

- **Bagging** consists in averaging multiple predictors learnt on randomly drawn subsets of the dataset[39]; these subsets are bootstrap replicates of the original dataset; see section 13.3.2 for more on this. Roughly, a bootstrap replicate

of a dataset of N items is a family of N individuals randomly drawn, with replacement, in this dataset; this means that (at least in most cases) some items are present more than once (and others are missing).

- **Subagging** consists in averaging multiple predictors learnt on randomly drawn small subsets of the dataset[46]. It is similar to bagging, except that in subagging bootstrap replicates are smaller than the complete dataset; for example, we randomly draw $M = N/20$ items in a dataset of N items.
- **Boosting** consists in averaging multiple predictors learnt on weighted samples of the dataset[106]. Individuals which are poorly predicted have their weights increased.
- **Random subspace** consists in averaging multiple predictors learnt on projections of the dataset on randomly chosen subspaces.[180]

These methods are quite convenient for parallelization.

Artificial intelligence for learning policies is a complicated field (see Part II); moreover, in many cases, it is far weaker than humans. Therefore, it is appealing to use a human as a policy, to log its observations and actions and then to learn a policy by supervised learning. This replaces a control problem, or a gaming problem, by supervised learning. This method is termed **imitation learning**[204].

Sigmoidal functions are non-constant and non-decreasing mappings from $\mathbb{R}$ to $\mathbb{R}$, with a finite limit in $-\infty$ and $+\infty$; usual sigmoidal functions f verify $f(x) = -f(-x)$, have a derivative f', and verify $\lim_{x\to\infty} f'(x) = 0$. When a function can be written with only $+$, $\times$, tanh (or other sigmoidal functions) and real parameters, then it is termed a **neural network** (see Fig. 16.4). The first layer of a neural network is also termed the input layer: there's one neuron in the input layer per dimension. The second layer is made of neurons such that there is an edge from the input layer to these neurons. The third layer is defined similarly, and so on. The output layer is made of all neurons whose output is the output of the neural network: there might be output neurons in the first layer even if there are 20 layers. The notion of layer is less useful when there are cycles in the graph of the neural network.

In some cases, Gaussian functions are allowed in a neural network; then, we can use $\exp(-||. - .||^2/K)$ in the mathematical writing of the function; in many cases, the Gaussian functions are only allowed in the first layer of hidden nodes. More generally, neurons with $g(-||. - .||^2)$ for some function g decreasing to 0 (i.e. $\lim_{x\to\infty} g(x) = 0$ and $x > y \Rightarrow g(x) < g(y)$) are sometimes allowed and these nets are then termed "radial basis function"(RBF) networks. There are neurons not so far from being radial basis neurons in animals and humans.

The parameters are termed the **weights** of the neural net. If the underlying graph is acyclic then the neural network is termed **feedforward**. When all the weights are chosen randomly by the algorithm, independently of the x_i's and y_i's, except those in the output layer, and if the neural net is sparse, then the algorithm is termed **reservoir computing**.

When the x_i's in supervised learning are chosen randomly and independently in their domain, then the learning is termed **passive**. When the learning algorithm is allowed to choose x_i as a function of $x_1, \ldots, x_{i-1}, y_1, \ldots, y_{i-1}$, then the learning is termed

active. It has been shown a long time ago that active learning was extremely efficient when learning is performed in a small family of models; in other cases, active learning is not so strong and might not be worth the candle[227].

16.1 Supervised learning and loss functions

Supervised learning is the following task:

- Inputs: $(x_1,y_1),\dots,(x_n,y_n)$ (usually assumed to be sampled according to some probability distribution (x,y)). Each (x_i,y_i) is termed an **example**. y_i is often termed a **label**.
- Outputs: f
- Goal:
$$L_n = \mathbb{E}||f(x)-y||^2 \text{ as small as possibly} \tag{16.1}$$

L_n is termed the **generalization error**. We here consider $||.||^2$, but many other norms or dissimilarity measures can be considered; this is termed the **loss function**. In an **unsupervised setting**, there's no y_i's, only the x_i's are available, and we look for a function f so that the $f(x_i)$ keep some structure of the x_i's. Some examples are as follows:

- we minimize $\sum_{i,j}(||f(x_i)-f(x_j)||-||x_i-x_j||)^2$ (we aim at preserving the distances), for f constrained to be in a set of functions which reduce the dimension;
- we minimize $\sum_i(f(x_i)-x_i)^2$ with f constrained to have values in a finite set (e.g. in K-**means algorithms** f has values in a set of cardinal K).

Let's come back to supervised learning. Many results can be proved for various families of functions, but for clarity we will consider only the quadratic loss function (Eq. 16.1). The problem is ill posed in the sense that without assumption on the distribution, we cannot reach the minimum of Eq. 16.1. Nonetheless, many things are possible. In this short overview, we will not give a lot of details and refer to [87, 238] for more. A classical and simple algorithm for this is k nearest neighbours:

$$f(x) = \frac{1}{k}\sum_{i=1}^{k} y_{(x,i)}$$

where (x,i) is the index of the i^{th} closest to x point among the x_i (break ties randomly).

It is possible (see [87] and references therein) to ensure that

- $L_n \to 0$ almost surely, independently of the distribution of (x,y) provided that y is a deterministic function of x (with k nearest neighbours with k fixed or going to ∞ sufficiently slowly as a function of n, but also for structural risk minimization and some other approaches);
- $L_n \to \inf_f \mathbb{E}||f(x)-y||^2$ almost surely (for example for $k(n)$-nearest neighbours, with $k(n)\to\infty$ sufficiently slowly.

Empirical risk minimization in a family F of functions consists in selecting $\hat{f} \in F$ minimizing the **empirical risk**, i.e.

$$\hat{f} = \arg\min_f \frac{1}{n} \sum_{i=1}^{n} ||f(x_i) - y_i||^2. \tag{16.2}$$

If f^* minimizing $\mathbb{E}||f(x) - y||^2$ lies in a given set F with finite VC-dimension, then it is known that **empirical risk minimization** has the same good properties as above, plus convergence rates which do not depend on the distribution. In many cases, it is more efficient to minimize a regularized term:

$$\hat{f} = \arg\min_f \frac{1}{n} \sum_{i=1}^{n} ||f(x_i) - y_i||^2 + C(f,n) \tag{16.3}$$

where $C(f,n)$ is some complexity measure, typically the sum of the squared parameters when f is parametric ($\forall f \in F, \exists \theta, f = f_\theta$; θ is the parameter of f), multiplied by some parameter depending on n:

$$C(f_\theta, n) = k(n) \sum_{i=1}^{|\theta|} \theta_i^2.$$

Minimizing the regularized empirical risk is often termed **regularized empirical risk minimization** or sometimes **structural risk minimization**. $k(n)$ is usually chosen by **hold out** or **cross-validation** instead of a fixed value depending on n:

- in hold-out, a random subset of the available examples (termed the **test set**[2]) is not used in the regularized empirical risk minimization. One regularized risk minimization is performed for each value of $k(n)$ in a finite set. The generalization error is then evaluated on the examples which have not been used in the empirical minimization (the best $k(n)$ is kept for minimizing Eq. 16.3 on the whole set);
- in cross-validation, the hold-out is performed for several partitions of the data set into a test set and a learning set, and the results are averaged (over those random partitions);

(there are other forms of cross-validation, e.g. k-fold cross validation, or bootstrap, which are not discussed here).

Usual families of functions for empirical risk minimization are neural networks or decision trees (see above). However, choosing the structure of a neural network (the underlying graph) and the parameters of the neurons is not an easy task. For a single neuron, we know which output should be associated with which input: therefore, a simple linear regression can be performed; or techniques like Hebbian rules can be applied (not discussed further here - Hebbian rules consist in reinforcing weights which are often activated simultaneously). For the general case we have a credit assignment problem: how to know which connections are good (and therefore should be reinforced by increasing their weights), and which connections are bad (and therefore should have

[2]The other examples are termed the **learning set**.

their weights decreased, or even should be removed) ? This is the so-called **credit assignment problem**.

We have presented supervised learning. We have discussed empirical risk minimization as a reasonable tool for choosing the parameters of a function in supervised learning. Now, we'll see how to actually perform risk minimization (or its variants) - which is basically an optimization problem.

16.2 The credit assignment problem and backpropagation

Consider a neural network as in Fig. 16.5. Then the question is: how to minimize the regularized empirical risk ? We now present algorithms for this setting, but we'll see that the same technique can be used for other things than empirical risk minimization, and in particular for direct policy search when the policy is a neural network.

[202] proposed the backpropagation for computing the gradient of a neural network. More precisely, consider that you want to minimize some criterion. Typically, the empirical risk

$$\frac{1}{n}\sum_{i=1}^{n} ||f(x_i)-y_i||^2.$$

Usually it is not difficult to know the gradient of the criterion with respect to the outputs of your neural networks (however, this might be uneasy for control applications, like direct policy search - except by unfolding over a given number of iterations). For empirical risk as above, this is

$$\delta_{output} = -2\frac{1}{n}\sum_i (f(x_i)-y_i).$$

This formula is obtained by deriving the empirical risk above (Eq. 16.2). This is intuitively an "error signal"; in gradient descent we would like to "push" the neural network in this direction. For this, we have to propagate the error signals until all the weights. How can we propagate the signal, backwards through a neuron ? The output of the neuron is $o = g(\sum_{i=1}^{k} w_i x_i + w_0)$. How to compute δw_j for some j, and also δx_m so that we propagate the error signal to previous layers ? This is a simple calculus of derivative:

$$\frac{\partial o}{\partial x_m} = g'(\sum w_i x_i + w_0) \times w_m;$$

$$\frac{\partial o}{\partial w_m} = g'(\sum w_i x_i + w_0) \times x_m.$$

Or, in more concise notations:

$$\frac{\partial o}{\partial x_m} = g' \times w_m;$$

$$\frac{\partial o}{\partial w_m} = g' \times x_m.$$

Therefore, the error signals can be computed as follows:

$$\delta x_m = g' \times w_m \times \delta_o \text{ (this propagates the error signals to other neurons)}$$

$$\text{and } \delta w_m = g' \times x_m \times \delta_o \text{ (this propagates the error signals to weights).}$$

We can therefore summarize the propagation of error signals in a feedforward neural network as follows:

- the error signal for an output neuron is

  $$\delta_o = -2\frac{1}{n}\sum_i (f(x_i) - y_i).$$

  As this is usually applied for one example at a time only (the error signal is then propagated for each example in turn), this leads to

  $$\delta_o = -2(f(x_i) - y_i).$$

- the error signal for other neurons is

  $$\delta o = \sum_{o';o\to o'} g' \times w_{o,o'} \times \delta_{o'}$$

  (where o' loops on neurons which have o in their inputs, and $w_{o,o'}$ is the weight of the connection between o and o')

- when the error signal reaches a neuron, then the error signal can be sent to the weights:

  $$\delta_w = g' \times x \times \delta_o$$

  where g' is the derivative of the output, o is the output of the neuron, x is the value connected to the neuron with weight w (x can be an input or the output of another weight, this does not matter).

These equations are the backpropagation and can be used directly for feedforward neural network. The weights can then be adapted by stochastic gradient descent, i.e.

$$w \leftarrow w + \frac{1}{n}{\delta_w}^n$$

(or other functions of n than $\frac{1}{n}$ can be considered, e.g. $1/\sqrt{n}$, $1/(n_0+n)$...) where ${\delta_w}^n$ is the error signal for the n^{th} example (or n^{th} time step in a control application). When a recurrent neural network is used, then the neural network can be unfolded in order to account for several time steps; and backpropagation can be used as well. This is termed **back-propagation through time** (BPTT). Nonetheless, it must be pointed out that backpropagation for many layers is not comfortable, unstable, slow, and in particular when time is involved in the problem; time implies, after unfolding of the neural controller, very big networks. Also, for a control application, this implies the derivation of the plant to be controlled. Direct policy search with an optimization algorithm with no gradient might be more convenient, and not really less efficient. Using genetic algorithms can be interesting as well, as it might allow to evolve the structure of the controller, and not only the weights[119, 118].

As a summary around backpropagation:

- Backpropagation is the application of gradient descent to minimize loss functions in neural networks.
- Backpropagation has the advantage of scaling well with the number of examples, much more than SVM (but not with the number of layers or the dimension).
- Backpropagation is slow and unstable when there are many layers or when the network is unfolded for a discrete time control problem.
- Recent techniques (section 16.5) for credit assignment problems consist in learning each layer, one at a time, by unsupervised learning, and then to learn the output layer by supervised learning and then optimize the complete network by backpropagation (see deep networks in section 16.5). Other specialized techniques, with very impressive results in some visual recognition tasks, are convolutional neural networks[151, 250].
- Other recent techniques consist in having one big hidden layer only (possibly with cycles, in control problems), and to learn only the output weights (see reservoir computing and random kitchen sinks in section 16.4).
- For control problems, direct policy search by gradient-free techniques is often simpler and more efficient than backpropagation.

Neural networks for direct policy search

Essentially the approach above provides a tool for computing the gradient of a neural network with respect to its output, its inputs, its weights. This can directly be used in a stochastic gradient descent, in a quasi-Newton, in BFGS (see index), and so on. However, gradients are often difficult to evaluate, and in many cases direct policy search will be performed by gradient free algorithms.

When using direct policy search with a neural network as a controller:

- Unrolling the neural network and the plant on many levels and computing the derivative of all this is possible, but tedious and unstable.
- Directly optimizing the coefficients by (i) Monte-Carlo estimation of the loss, (ii) optimization of this estimated loss without derivative or by finite-differences, is a simple and stable idea (see section 14.2 for more on the optimization of stochastic fitness functions).
- Using reservoir computing strongly reduces the dimensionality of neural net based direct policy search.

16.3 Avoiding local minima: support vector machines

Generalized linear regression is a tool for supervised learning; it consists in learning parameters $w_1,\ldots,w_k$ such that

$$f(x) = \sum_{i=1}^{k} w_i K_i(x) + b.$$

In the case of support vector machines, k is equal to the number n of examples; if the examples are $(x_1,y_1),\ldots,(x_n,y_n)$, $K_i(x) = K(x_i,x)$. The usual regularized loss function is then

$$\underbrace{\frac{1}{n}\sum_{i=1}^{n}\left|\left(\sum_{j=1}^{n} w_j K(x_j,x_i)+b\right)-y_j\right|_{\varepsilon}}_{empirical\ loss} + \underbrace{C\sum_{i,j} w_i w_j K(x_i,x_j)}_{regularization}$$

where $|.|_\varepsilon$ is the ε-insensitive loss function $|x|_\varepsilon = \max(0,|x|-\varepsilon)$.

In the case of binary classification i.e. $y_i \in \{-1,1\}$, the loss function above is usually replaced by

$$\underbrace{\frac{1}{n}\sum_{i=1}^{n}\max(y_i\left(\sum_{j=1}^{n} w_j K(x_j,x_i)+b\right)-1,0)}_{empirical\ loss} + \underbrace{C\sum_{i,j} w_i w_j K(x_i,x_j)}_{regularization}.$$

Support vector machines:

- are the most classical tool for supervised learning;
- have no local minima;
- can use (highly ill-conditioned) quadratic programming;
- are very (very) slow for big numbers of examples;
- are more compliant with high-dimension than back-propagation neural networks;
- are less effective than convolution networks for building complex internal representations[151, 250].

16.4 Simple efficient algorithms: random kitchen sinks

Random kitchen Sinks (RKS) are a now classical learning algorithm; it's a form of reservoir computing. It is so simple that we can present it just by copy pasting a few lines from [188]; see Alg. 20. The main result in [188] shows that, for any distribution

Input: A dataset $\{x_i, y_i\}_{i=1\dots m}$ of m points, a bounded feature function $|\phi(x;w)| \leq 1$, K, a scalar C, and a probability distribution $p(w)$ on the parameters of ϕ.
Output: A function $\hat{f}(x) = \sum_{k=1}^{K} \phi(x;w_k)\alpha_k$.
Draw $w_1, \dots, w_K$ iid from p.
Featurize the input: $z_i \leftarrow [\phi(x_i;w_1), \dots, \phi(x_i;w_K)]^\top$.
With w fixed, solve the empirical risk minimization problem

$$\begin{aligned} \underset{\alpha\in\mathbb{R}^K}{\text{minimize}} \quad & \frac{1}{m}\sum_{i=1}^{m} c\left(\alpha^\top z_i,\ y_i\right) \\ \text{s.t.} \quad & \|\alpha\|_\infty \leq C/K. \end{aligned}$$

Algorithm 20: Random kitchen sinks. The constraint (last line) is usually removed, and usually $c(a,b) = ||a-b||^2$ in regression (or any other classical loss function). A simple example is $\phi(a,b) = cos(b_0 + b_1.a_1 + b_2.a_2 + \cdots + b_d.a_d)$.

p, the generalization error of RKS is upper bounded by

- the minimum reachable error for functions that can be written as $\int \alpha(w)\phi(.,w)dw$ with $\alpha(w) = O(p(w))$,

- plus $\sqrt{\log(\delta)} \times O(1/\sqrt{n} + 1/\sqrt{K})$,

with probability $1 - \delta$.

As a summary:

- RKS are a very comfortable and user-friendly algorithm.
- RKS are often as efficient as other more complicated algorithms.
- RKS have a memory-consuming representation (scaling linearly with the product of the dimension and of the number K of kitchen sinks).

16.5 Deep networks

It is known that many functions are much more properly represented by neural networks with several hidden layers, instead of only one or two layers; this is a weakness for random kitchen sinks, support vector machines, and also for backpropagation neural networks as backpropagation becomes very slow for more than two hidden layers.

As a consequence, people have tried to design multilayered neural networks, with a learning algorithm different from backpropagation (see a survey in [20, 9]). A particularly important recent family of algorithms consists in learning each layer in an unsupervised manner, and then to finalize the learning by some classical supervised learning algorithm (e.g. backpropagation).

Deep networks:

- are a much more compact representation than one-layered neural networks in many cases;
- are often trained by unsupervised learning first, one layer at a time, and (possibly) afterwards by classical supervised learning (supervised learning is mandatory for the output layer);
- can have loops in particular when used as controllers.

16.6 Big datasets and parallel machine learning

What about machine learning for big datasets ?

There has been a lot of work on supervised learning for big datasets. A long time ago, it was classical to claim that the best algorithm for SVM was using the quadratic

nature of the problem. Now, we often see more stochastic gradient descent, a generalist algorithm[69] (whereas 10 years ago it was a blaspheme to propose something like that). There are also papers recalling that for big datasets, other algorithms are much better than SVM[156].

Some simple facts for supervised learning of big datasets:

- Decision trees are very fast;
- Quadratic programming does not always outperform stochastic gradient descent;
- Linear SVM are much faster than Gaussian SVM for big numbers of examples.
- Parallelization matters, with little programmation work (see below).

On the parallel side, many algorithms have been designed specifically; parallel mixture of SVMs[70], Cascade SVM[117], with moderately good results. It was then pointed out that the main tools for parallel machine learning might be the most simple ones:[55] suggests in particular that parallelizing the cross validation for choosing the parameters is extremely easy, simple, and efficient; and might make other parallelizations less worth the candle. Other aspects than cross-validation that can be easily parallelized are discussed below:

Some machine learning tools which are easy to parallelize:

- Cross-validation for choosing parameters (easy and efficient parallelization).
- Bagging and a fortiori subagging.
- Random subspaces.

Also stochastic gradient descent can be parallelized by averaging the computation of the gradient on multiple examples.[84]

Figure 16.2: Various cyborgs, from top to bottom and left to right: a cyborg by PlasticPals, named Simon, based on imitation learning; the two following robots, based on human (remote) pilots, are maybe less elegant but more efficient: a cyborg setting up a small bomb for destroying a dangerous packet; a medical robot for surgery. The next robot is here for helping disabled people; the last robot picks up irradiated bodies (4 last images from botson.com).

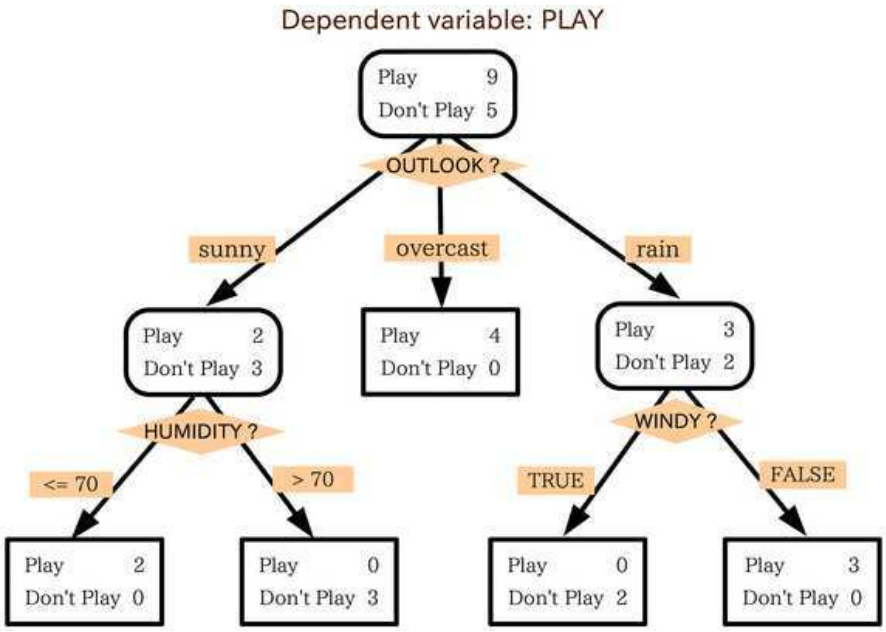


Figure 16.3: An example of decision tree, from the ”Monk” website (`http://gautam.lis.illinois.edu/monkmiddleware/public/analytics/decisiontree.html`).

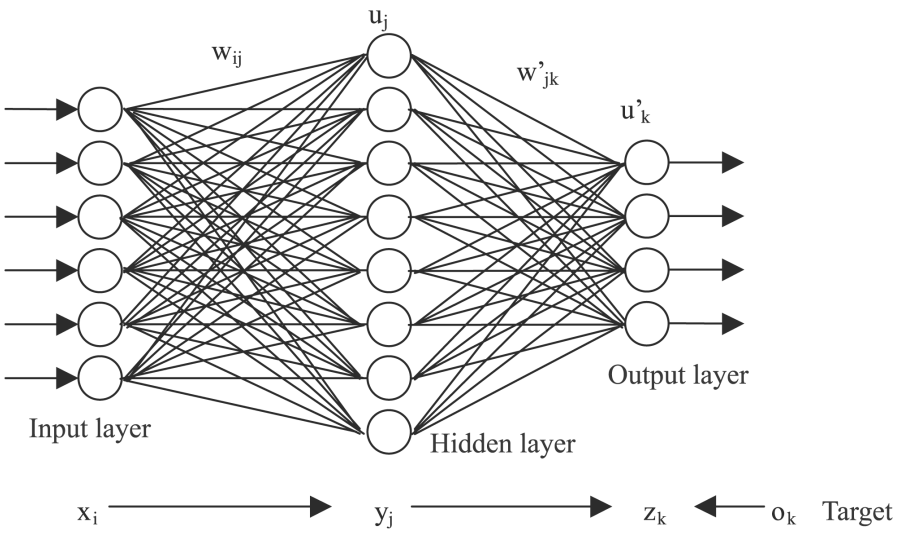


Figure 16.4: An example of neural network, from the EmeraldInsight (`http://emeraldinsight.com`) website. The output is $W' \times tanh(W \times X)$, where X in the input vector and where $tanh(v_1, \ldots, v_k) = (tanh(v_1), tanh(v_2), \ldots, tanh(v_k))$. The coefficients in the matrix W and the matrix W' are termed the weights (output weights for W'). The neural network can be based on a non-acyclic graph, and can be sparse.

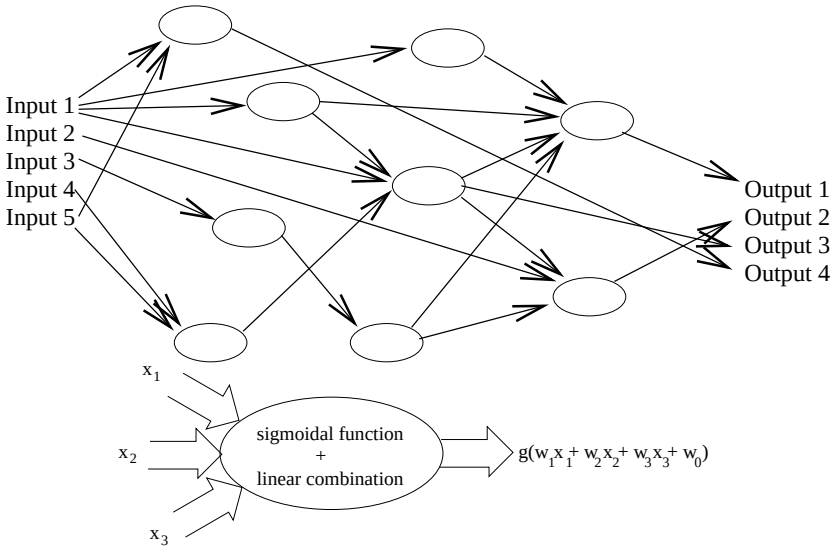


Figure 16.5: Top: a neural network. Bottom: a neuron. A neural network is a graph, with each node equipped with one bias w_0 plus one weight w_i for each of its input edge. This neural network is feedforward (no cycle), but this is not necessary; also, the output of a neuron can be the input of several other neurons without trouble.

Chapter 17

Parallel Active Supervised Learning [1]

Active learning [4] is a branch of statistical Machine Learning in which the learning algorithm is allowed to choose the examples. It is particularly suited to settings where labeling instances is costly. This chapter analyzes the speed-up of batch (parallel) active learning compared to sequential active learning (where instances are chosen 1 by 1). The contributions reside in proving lower and upper bounds on the possible gain, and illustrating them by experimenting on usual active learning algorithms. Roughly speaking, the speed-up is asymptotically logarithmic in the batch size λ (i.e. when $\lambda \to \infty$). However, for some classes of functions with finite VC-dimension V, a linear speed-up can be achieved until a batch size of V—practically speaking, this means that parallelizing computations on an expensive-to-learn problem, suited to active learning, is very beneficial until V processors, and less interesting (yet still bringing improvement) afterwards.

The main limitation of this chapter is that all results are based on the assumption of exact model.

17.1 Introduction

Active learning [4] is a Machine Learning setting in which the learning algorithm is allowed to choose the examples. Batch active learning [132, 217, 133] is the particular case of active learning in which the algorithm must choose λ examples at each iteration. Active learning is particularly efficient if the oracle labelling the examples is expensive; batch active learning comes into play when it can be called on each computation unit of a parallel machine. This chapter provides rigorous bounds on the number of iterations before a given precision is reached for batch active learning in binary classification, in particular as a function of λ. This model of complexity, based on the number of iterations only, is relevant to cases in which almost all the cost is in the calls to the

[1] This chapter is based on a collaboration with M. Sebag and P. Rolet.

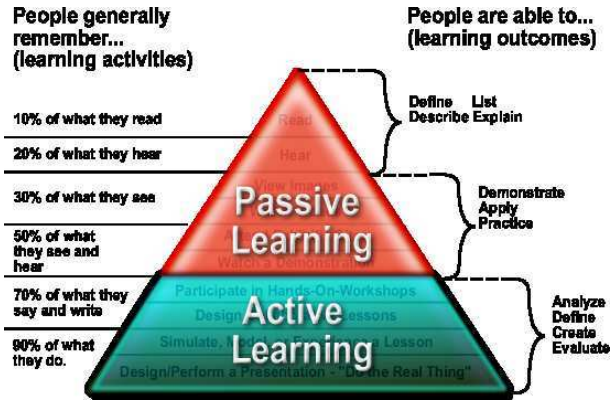


Figure 17.1: Active Learning, far from computer science, as explained on the EduTechnorama website: we keep in memory what we actively do much more than what we passively see. In machine learning, the active part is in the choice of examples.

oracle function (*expensive* oracle), and that at least λ computation units are available. The internal cost of the learning algorithm is not taken into account. Obviously, under this assumption, passive learning has a linear speed-up, in the sense that for passive learning all the points can be generated simultaneously since it does not make any difference with the case in which points are generated sequentially. We here investigate to which extent such a good speed-up can be recovered for active learning. The chapter is organized as follows:

- Section 17.2 presents the framework and notations, so that complexity bounds can be properly formalized.
- Section 17.3 shows bounds for batch active learning using covering and packing numbers. Results include lower and upper bounds on the speed-up of batch learning, seen as a parallel algorithm.
- Section 17.4 presents some experiments; these experiments are aimed at comparing predicted speed-ups (for optimal algorithms) to the speed-ups of simple or usual algorithms.
- Section 17.5 concludes.

State of the art

The query learning models introduced by [4] can be viewed as the first attempts of the learning algorithm to directly interact with the oracle. Another early work ([146]) establishes a lower bound for the instance size in any AL classification setting, logarithmic in the ε-packing number of the hypothesis space $\mathscr{F}$—the number of ε-radius balls needed to cover $\mathscr{F}$ (see section 17.2 below).

Classification

[67] devised a heuristic for error-free learning (i.e., in the realizable setting) of a binary classifier, considering a large pool of unlabeled examples and selecting the best example to be labeled in each time step (pool-based adaptive sampling). Considering the set of hypotheses compatible with the available examples (the *version space* (VS) defined in [168]), the selected examples were meant to prune the version space.

[105] analyzed another algorithm based on a Bayesian prior on the hypothesis space called *Query-by-committee* (QBC) from [211]; it directly reduces the VS volume. A related research direction focuses on *error reduction*, meant as the expected generalization error improvement brought by an instance. Many criteria reflecting various measures of the expected error reduction have been proposed ([68, 138, 200, 154, 80]), with sometimes encouraging results in, for instance, pharmaceutical industry ([242]). Specific algorithms and methods have been developed active learning in linear and kernel spaces, either heuristically ([209]) or theoretically grounded ([56, 80, 18]).

On the theoretical side, [105] related the efficiency of QBC to a statistical criterion called *Information Gain*, measuring how efficiently the VS can be divided. [78] shown that with a Bayesian prior, greedily choosing examples that most evenly divide the VS is an almost optimal AL strategy.

Dasgupta also studied the non-Bayesian setting ([79]), deriving upper and lower complexity bounds based on a criterion called *splitting index*.

Batch active learning has received less attention. [132] assesses the information brought by batches of examples via a criterion based on Fisher information matrix reduction. [120] seeks sets of examples with low uncertainty; they phrase this as an optimization problem (NP-hard), and devise a method to find an acceptable approximation of the solution. Both works provide empirical evidence of the soundness of their strategies. However, they do not provide any formal proof guaranteeing their behavior. Further, we are not aware of any theoretical study of the speed-up of batch Active Learning over sequential Active Learning , in terms of sample complexity bounds.

17.2 Framework

In all the chapter, log refers to the logarithm with basis 2. If $x \in [0,\infty[^d$, then we note $[0,x] = \{a \in \mathbb{R}^d; \forall i, 0 \leq a_i \leq x_i\}$. The *speedup* of a parallel algorithm $\mathscr{A}_\lambda$ over its sequential counterpart A is the ratio of A's complexity on $\mathscr{A}_\lambda$'s complexity.

Only deterministic algorithms are considered here; the lower bounds can be extended, nonetheless, to stochastic cases within logarithmic dependencies on the risk δ [2] and simulation results (Theorem 7) can also be extended to the stochastic case. The framework of batch active learning is presented in Algo. 21. A batch active learning algorithm $\mathscr{A}_\lambda$ is thus defined by the triplet $(learn_\lambda, generate_\lambda, update_\lambda)$. Let D be a domain with measure 1, and $f : D \to \{0,1\}$ be the unknown oracle, supposed to be deterministic and to belong to some set $\mathscr{F} \subset \{0,1\}^D$. We assume that considered

[2] Precisely, the sample complexity is multiplied by $\log(1-\delta)$ if we request that the algorithm finds the solution with probability $1-\delta$.

concept class $\mathscr{F}$ has a finite VC-dimension V. It is common to consider finite VC-dimension in active learning settings since the improvement over passive learning is potentially much bigger in this case (sample complexity $N = \Theta(V \log(1/\varepsilon))$, see for instance [79, 105]).

$I_0 = initial\ state$
$n \leftarrow 0$
while true **do**
  $f_n \leftarrow learn_\lambda(I_n)$
  $(x_{n\lambda+1}, \ldots, x_{n\lambda+\lambda}) = generate_\lambda(I_n)$
  for $i \in [[1, \lambda]]$ **do**
    $y_{n\lambda+i} = f(x_{n\lambda+i})$
  end for
  $I_{n+1} \leftarrow update_\lambda(I_n, x_1, \ldots, x_{n\lambda}, y_1, y_{n\lambda})$
  $n \leftarrow n+1$
end while

Algorithm 21: Batch active learning algorithm. λ is the number of visited points per iteration.

For a given λ and a given algorithm A, it is said that the algorithm requires $N_\lambda^{\mathscr{A}}()$ iterations[3]if

$$N_\lambda(\varepsilon) = \sup_{f \in F} \min\{n; ||f_n - f|| \leq \varepsilon\}$$

with $||.||$ the L_1 norm. In the sequel, for some of our results, the following equation will be assumed:

$$pack_{\mathscr{F}}(\varepsilon) \geq (M/\varepsilon)^{(C \times V)} \tag{17.1}$$

for constants C and M. It states that we have log-packing numbers at least $-CV\log(\varepsilon)$. $pack_{\mathscr{F}}(\varepsilon)$[4] is the maximum number of points in $\mathscr{F}$ with pairwise distance at least 2ε for the L_1 norm. The product $C \times V$ in Eq. 17.1 stems from many results emphasizing some constant C, and the VC-dimension V [79], such as, for example, the well-known case of homogeneous linear separators (linear separators with the origin at the boundary) of the sphere with homogeneous distribution[105, 80, 18] (V is equal to the dimension of the domain in this case).

17.3 Covering numbers and batch active learning

Eq. 17.1 has the following consequence (see [147, 239]):

$$N_1(\varepsilon) \geq \lceil CV\log(M/\varepsilon) \rceil. \tag{17.2}$$

[3]noted $N_\lambda()$ unless the context requires an explicit A.
[4]hereafter noted $pack(\varepsilon)$ for short

Eq. 17.2 (lower bound on the sample complexity of AL) has the following consequence (lower bound on the sample complexity of batch AL):

$$N_\lambda(\varepsilon) \geq \lceil CV \log(M/\varepsilon)/\lambda \rceil \tag{17.3}$$

Eq. 17.3 is the ultimate limit for batch active learning; it is the case of a linear speed-up. The following explores the extent to which it can be approached, depending on λ. In a parallel setting, in which λ calls to the oracle are performed in parallel, Eq. 17.3 refers to a linear speed-up for the parallel (i.e. batch) form of active learning. It will be convenient to note

$$L_\lambda(\varepsilon) = \inf_{all\ algorithms} N_\lambda(\varepsilon)$$

the infimum of $N_\lambda(\varepsilon)$.

The first contribution of this work is the following extension of the classical bound 17.2:

Theorem 6 (Lower bound for batch AL) *If $\mathscr{F}$ has packing number*

$$pack(\varepsilon) \geq (M/\varepsilon)^{C \cdot V}, \tag{17.4}$$

then the following holds:

$$L_\lambda(\varepsilon) \geq CV \log(M/\varepsilon)/\log(K) \tag{17.5}$$

where $K = \lambda^V$ if $V \geq 3$ ($K = \lambda^V + 1$ if $V \leq 2$), i.e.

$$L_\lambda(\varepsilon) \geq C \log(M/\varepsilon)/(\log(\lambda)). \tag{17.6}$$

Remark. K is an upper bound on the number of possible classifications of λ points, given a class of function with VC-dimension V. $K = \lambda^V$ stems from Sauer's lemma (see [203]).

Proof: Consider an algorithm realizing $L_\lambda(\varepsilon)$.

There is one possible value $x_1, \ldots, x_\lambda$ for this algorithm, independently of f:

$$(x_1, \ldots, x_\lambda) = generate(I_0).$$

Thanks to the finiteness of the VC-dimension and to Sauer's lemma, there are at most K possible values for $y_1, \ldots, y_\lambda$; therefore there are at most K possible values for I_1 (since the algorithm is assumed to be deterministic).

Similarly, for each possible value of I_1, there are at most K possible values for I_2; therefore the total number of possible values for I_2 is at most K^2.

By induction, there are at most K^i possible values for I_i. After $L_\lambda(\varepsilon)$ iterations, each possible state $I_{L_\lambda(\varepsilon)}$ corresponds to a function learned with $\lambda L_\lambda(\varepsilon)$ examples. Since the algorithm realizes the bound $L_\lambda(\varepsilon)$, for any 2 oracle functions distant of ε or more, the algorithm must have 2 different states. Thus, the final number of states is at least as big as the packing number: $K^{L_\lambda(\varepsilon)} \geq pack(\varepsilon)$. As a consequence,

$$L_\lambda(\varepsilon) \geq \log(pack(\varepsilon))/\log(K). \tag{17.7}$$

Eqs. 17.7 and 17.4 yield the expected result. □

The following result shows that this bound is tight, at least asymptotically ($\lambda \to \infty$).

Theorem 7 (Upper bound for batch AL) *Consider the batch AL framework (Algo. 21). If $\mathscr{F}$ has VC-dimension V, then the following holds for all $D \geq 1$:*

$$L_{\lambda'}(\varepsilon) \leq \lceil L_{\lambda}(\varepsilon)/D \rceil \tag{17.8}$$

where

$$\lambda' = \lambda \frac{K^D - 1}{K - 1} \tag{17.9}$$

and $K = \lambda^V + 1$.

Remark. Eq. 17.8 leads to

$$L_{\lambda'}(\varepsilon) \leq \lceil L_{\lambda}(\varepsilon)/\Omega(\log(\lambda)) \rceil \tag{17.10}$$

for fixed V and λ; this is a logarithmic speed-up.

Proof: The proof exhibits an algorithm realizing Eq. 17.8. Consider an algorithm $\mathscr{A}_\lambda = (learn_\lambda, generate_\lambda, update_\lambda)$ realizing $L_\lambda(\varepsilon)$ and consider some $D \geq 1$. Define $\lambda' = \lambda \frac{K^D-1}{K-1}$. Consider, then, another algorithm $\mathscr{A}_{\lambda'} = (learn_{\lambda'}, generate_{\lambda'}, update_{\lambda'})$ which generates λ' points by simulating Aon D steps; if the $\mathscr{A}_\lambda$ has internal state I_n, then $\mathscr{A}_{\lambda'}$ has n^{th} internal state $I'_n = I_{Dn}$. At each iteration:

- $generate_{\lambda'}$ simulates the K^D possible internal paths

$$I_{Dn}, I_{Dn+1}, \ldots, I_{Dn+D}. \tag{17.11}$$

  and generates for each iteration all the λ' possible visited points (note that λ' as in Eq. 17.9 is enough) for internal states in Eq. 17.11;

- the target f is computed at these λ' points in order to see which path (among the K^D possible paths) is the good one;

- $update_{\lambda'}$ is the result of $update_\lambda$ for the path selected in $generate_{\lambda'}$.

- the output of $learn_{\lambda'}$ is the output of $learn_\lambda$ on all points visited in the selected path.[5] □

Eqs. 17.6 and 17.8 show that $\log(\lambda)$ is the optimal speed-up when no assumption on λ are made: theorem 7 shows that in all cases, a logarithmic speed-up is achievable and theorem 6 shows that we cannot do much better for λ large.

The remaining question is what happens for moderate values of λ, and in particular how many processors we need for removing the dependency in V. We now show that $\lambda = V$ leads to a nearly linear speed-up, for some families $\mathscr{F}$; and this removes the dependency in V in runtimes - this means that we can break the curse of dimensionality, with V processors, whereas using more processors (than V) will only provide a logarithmic speed-up.

[5] Please note that a big part of the points for which the target value has been computed is discarded. This is necessary for the formal proof of the simulation result. For real-world applications, we guess that applying the learning algorithms on all points might be much better, within constant factors however.

Theorem 8 (Linear speed-up until $\lambda = V$) *Consider $\mathscr{F}_V = \{[0,x], x \in [0,1]^V\}$; $VC-dim(\mathscr{F}) = V$. Then, for some $M > 0, M' > 0$,*

$$\exists C > 0, \forall V, \exists \varepsilon_0, \forall \varepsilon < \varepsilon_0, L_1^{\mathscr{F}_V}(\varepsilon) \geq CV \log(M/\varepsilon) \tag{17.12}$$

and

$$\exists C'; \forall V, \exists \varepsilon_0, \forall \varepsilon < \varepsilon_0, L_V^{\mathscr{F}_V}(\varepsilon) \leq C' \log(M'/\varepsilon). \tag{17.13}$$

Eqs. 17.12 and 17.13 state the linear speed-up for batch active learning with $\lambda = V$ for this family of fitness functions (within the constants C and C').

Proof: We first show that Eq. 17.14 holds:

$$\exists C > 0, \forall V, \exists \varepsilon_0, \forall \varepsilon < \varepsilon_0, pack_{\mathscr{F}}^V(\varepsilon) \geq M/\varepsilon^{(C \times V)}. \tag{17.14}$$

Eq. 17.14 is a version of Eq. 17.1 modified for considering only ε small; it is weaker than Eq. 17.1 and sufficient for our purpose.

Eq. 17.14 is proved as follows:

- For x and y in $[\frac{1}{2}, 1]^V$, the L^1 distance between $[0,x]$ and $[0,y]$ is lower bounded by $\Theta(||x-y||)$.
- Therefore, the packing number of the $\{[0,x]; x \in [0,1]^d\}$ is $\Theta(1/\varepsilon^V)$ and is therefore $f(1/\varepsilon^{V/2})$.
- This shows Eq. 17.14 for $C = \frac{1}{2}$.

Then, Eq. 17.14 classically leads to Eq. 17.15 (this is analogous to the proof of Eq. 17.2 from Eq. 17.1, see section 17.2):

$$\exists C > 0, \forall V, \exists \varepsilon_0, \forall \varepsilon < \varepsilon_0, N_1(\varepsilon) \geq \lceil CV \log(1/\varepsilon) \rceil. \tag{17.15}$$

Eq. 17.15 is the first part of the theorem (Eq. 17.12). Let us now show Eq. 17.13, by considering the following algorithm (described at iteration n):

- $generate_\lambda$ sets the j^{th} coordinate of $x_{n\lambda+i}$, for $j \neq i$ and $(i,j) \in [[1,\lambda]]^2$ to 0, and chooses the i^{th} coordinate of $x_{n\lambda+i}$ for $i \in [[1,\lambda]]$ as follows [6]:

$$(x_{n\lambda+i})_i = \frac{1}{2}\left(\min_{n' \leq n}\{(x'_{n'\lambda+i})_i | y'_{n'\lambda+i} = 0\} + \max_{n' \leq n}\{(x'_{n'\lambda+i})_i | y'_{n'\lambda+i} = 1\}\right). \tag{17.16}$$

- $learn_\lambda$ selects any function $f_n \in \mathscr{F}_V$ which is consistent with $x_1, \ldots, x_{n\lambda}$.

At a given iteration n each point $x_{n,i}$ of the batch of size λ makes sure that the domain will be halved along the ith coordinate. Thus, after N iterations, it is known that the target oracle/classifier is in a square of edge size 2^{-N}. As a consequence, precision ε is reached in at most $\Theta(\log(1/\varepsilon))$ iterations, which shows Eq. 17.13. □

This theorem shows that, at least for $\mathscr{F}$ as above, we can have a linear speed-up until $\lambda = V$; this is the tightness of Eq. 17.3 for $\lambda \leq V$—similarly to the tightness of Eq. 17.6 (*i.e.* logarithmic speed-up) shown by Eq. 17.10 for λ large.

[6] In Eq. 17.16, if no point $x'_{n'\lambda+i}$ has been labeled as 0, the minimum is set to 0; equivalently, if no point has been labeled as 1, the maximum is set to 1.

17.4 Experiments

We have formally proved both lower and upper bounds on batch AL. The following illustrates whether some simple or usual algorithms match these bounds.

17.4.1 Experiments with naive AL

We here experiment a simple batch AL algorithm for $\mathscr{F} = \{[0,x]; x \in [0,1]^V\}$ (VC-dimension V). The new sample(s) $x_{n\lambda+1}, \ldots, x_{(n+1)\lambda}$ are λ points randomly drawn in v where

$$v = \{x \in [[0,1]^V; \forall j \in [[1,n\lambda]] y_j = 0 \Rightarrow \neg(x_j \leq x)\}$$

$$\cap \{x \in [[0,1]^V; \forall j \in [[1,n\lambda]] y_j = 1 \Rightarrow x_j \leq x\}.$$

This means that we randomly sample the version space.

We plot the inverse of the number of iterations for reaching precision $0.003D^2$, depending on λ; this means that the ordinate is the rate. Results are presented in Fig. 17.2. Runtimes are averaged over 33 runs.

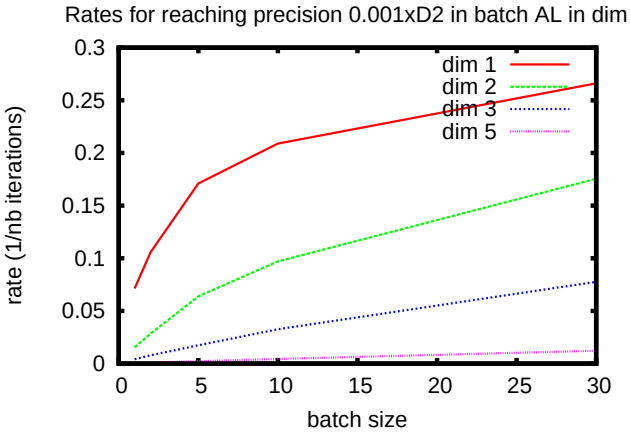


Figure 17.2: Speed-up of batch AL for a simple AL algorithm (see text). We see that the speed-up is consistent with theory; in particular, the larger the dimension and the better the parallelization. Nonetheless this algorithm might be easier to parallelize than a good AL algorithm as max-uncertainty (see section 17.4.2).

17.4.2 Experiments with max-uncertainty

This part of the experiments is concerned with a straightforward adaptation of a good, classical active learning heuristic that we call *Maximum Uncertainty* to the batch setting. The idea behind *Maximum Uncertainty* is to select the examples that split the version space, the space of all possible functions consistent with examples observed

so far, the most evenly. It has been studied empirically and theoretically algorithms enforcing this criterion have been proposed[211, 78].

Experiments learn homogeneous linear separators of $\mathbb{R}^d$, where examples lie on the hypersphere $\mathbb{S}^{d-1}$ for dimensions $d = 2,4,6,8$. This setting has been widely studied for sequential active learning[18, 80, 105] and thus fitted to a speed-up analysis for the batch setting.

In such a setting, for $d > 2$, an infinite number of points of the hypersphere maximize uncertainty given previously witnessed instances—whereas if $d = 2$, the maximum is unique. Thus, a possibly good batch strategy may consist in selecting λ of those points maximizing uncertainty, at each iteration.

Batch sizes are $\lambda = 1,2,4,6,8,12,16,20,24,32,40,48,64$. Precision is set to $0.0001 * (d/2)^4$. For each (d,λ), the number of iteration is averaged over 160 experiments.

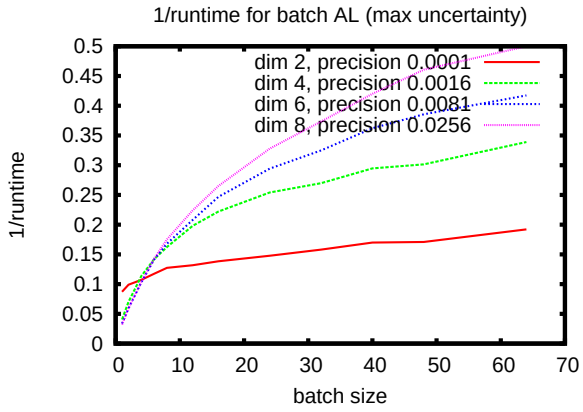


Figure 17.3: Speed-up of batch maximum uncertainty, without diversity preservation. We see that the parallelization works better in high dimension: the speed-up is nearly linear until a larger number of processors.

17.5 Summary and discussion

This chapter shows that batch active learning exhibits:

- a linear speed-up until $\lambda = V$ for some families of target functions;
- a speed-up at least logarithmic in all cases;
- and a logarithmic speed-up at most for λ large.

Please note that the logarithmic speed-up is a simulation result. The point is not to analyze the convergence rate of active learning in general, but to emphasize that *any* active learning algorithm can be transformed into a batch active learning algorithm (with λ computation units) which simulates it with speedup D, with D logarithmic as a function of λ.

All proofs have been made for deterministic algorithms. Their extensions to stochastic cases, however, is straightforward.

Experiments have been performed only in moderate dimension and for easy families of functions; the extension of the experiments to bigger dimensions and to other families of functions, is a possible further work.

Part V

Conclusion

This document covers several fields of artificial intelligence, including discrete time control (a somehow extremely general form of decision making), optimization (an essential part of many algorithms) and supervised learning. We will here try to extract the main lessons of the last years, in these fields, from a high-level point of view.

Monte-Carlo Tree Search is incredibly efficient in some cases in which all other techniques fail. It's not better than dynamic programming (or than retrograde analysis, in games) in small dimension, but it's much more able to handle high dimension. It's extremely easy to code (unless you try to have a very fast and optimized version for competitions). Very importantly, it is able to tackle problems with little or no structure. In some sense, it is in **a general direction of AI, namely working on more accurate models, with techniques which do not rely on a strongly simplified model of the world**; this direction is made possible by the increased computational power. It is often said that MCTS would make no sense with computers as in the 80's. Some weaknesses are however very clear; the algorithm has no extrapolation from one branch to another, i.e. no (or almost no) extrapolation from one analyzed state to newly visited ones[7]. The fact that the algorithm performs much better than other approaches in spite of such weaknesses is interesting. The application to Go is impressive, but we must point out some facts:

- contrarily to what is often said, the use of the UCB formula in MCTS is not important for Go. Indeed, it's not the best formula, and if you use it you must modify the formula deeply by reducing the exploration constant to almost zero (as discussed by many people in [175]);
- it was necessary to strongly adapt the Monte-Carlo part, in a very complicated way (not yet clearly understood).

A main trouble is that MCTS relies on the capacity to simulate; we need a model. There are some works for performing Monte-Carlo simulations without models[103]. Also, the partially observable case, which leads to much higher complexity classes, remains moderately explored, in particular with the point of view of consistency; see however [195, 15].

Essentially, tree search methods (alpha-beta and dynamic programming, or MCTS) provide great results in observable cases; in some randomized cases, the evaluation function becomes more crucial than the tree search; a tree search with depth 1 is often enough (as in backgammon). In partially observable cases, Monte-Carlo simulations, conditionally to observations, provides a good set of features for developing heuristics by policy search (possibly optimized by evolutionary or coevolutionary algorithms).

Machine Learning is an old field in which you can find plenty of useful algorithms, most of them able to outperform all the others if you carefully choose your testbeds. There are problems for which none of them works, whereas they are easy for humans, as shown by the many failed attempts for learning life and death problems in Go, even very simple ones. A basic conclusion is that supervised learning can help in

[7] "To generalize is to be an idiot", according to William Blake; but he was answering Reynolds: "disposition to abstractions, to generalising and classification, is the great glory of the human mind", which is about art. For making decisions, we have to generalize

some cases but extrapolation techniques are still far below human capabilities in many cases in which we would need it. Nonetheless, there are many interesting algorithms, their variety showing that things are not over, by far; SVM are a standard choice for supervised machine learning, and are strong for high-dimensional cases, but they often fail for large databases; decision trees have plenty of advantages of readability, but are very suboptimal for law dimensional sets with separations involving all variables; classical neural networks with backpropagation are strongly improved in some cases by the use of random weights or unsupervised layers (deep networks, reservoir computing). But more than algorithms, machine learning provides concepts: supervised learning, unsupervised learning, reinforcement learning, policy search, belief states.

Evolutionary algorithms. I've often been said that evolutionary algorithms are only for people who are not able to code serious algorithms like BFGS. This opinion is unfortunately often dominant. Nonetheless, the general idea above, that AI should tackle unstructured problems, can be applied here: evolutionary algorithms require neither convexity nor differentiability. Evolutionary algorithms are great tools and people who don't like them nonetheless often use them in desperate situations, e.g. for problems involving both continuous and discrete variables, or for discrete problems with complicated structures that can be encoded only in a crossover or randomized mutation operator.

Classical evolutionary algorithms in continuous domains were to the best of my knowledge all weak when the population size is large, compared to the new algorithms proposed here. The difference, however, is all in a log for the monomodal case; the linear speed-up until a number of processors roughly linear as a function of the dimension was already reached - we improve the speed-up from $\Theta(\min(d,\lambda))$ to $\Theta(\min(d\log(\lambda),\lambda))$. The case of multimodal fitness functions is probably much more parallel.

At the intersection of evolutionary optimization, discrete time decision making in uncertain environments, supervised machine learning, the main point in this work remains the idea of working on tools which do not rely on strong assumptions on the problems to tackle.

What is new in this work

As it is often said currently in France that researchers are somehow useless, it might be worth mentioning the main new results obtained recently in France around the topics analyzed in this document. This part will also survey our contributions. I consider as new (at least at the time of their first presentation) the following elements:

- Parallel versions of the MCTS algorithms. The algorithm can be used far from Go. I point out that the version provided by Tristan Cazenave, also in France, is simpler than our algorithm, and as efficient for many (but not all) cases. Incidentally, most of the research around MCTS, which is a scientific revolution, appeared in Paris, Orsay, Lille - French towns.
- The clear understanding of the semeai weakness. This is not a small boring issue: semeais are only Go positions, but a general solution to the weakness of MCTS

shown by semeais would be interesting far from Go. This open problem (how to play Go correctly, including semeai situations) is a crucial AI open problem. It also shows simple things which are beyond computers' abilities (see section 5.3.5).

- The bounds on optimization and parallel optimization are tighter and more general than previously existing bounds. We could deduce from these bounds algorithms which are much better than existing algorithms when the number of computation units is large. This bridges a gap, in my humble opinion, between theory and practice: it provides new bounds, which are tight, and leads to modifications in real-world algorithms which lead to better convergence rates[224].

- The bounds on noisy optimization are better than published bounds. The version of UH-CMA taking into account the requirement of cancelling the residual error by increasing the number of evaluations per stage to infinity is the first one with good asymptotic behavior in terms of the dependency of the precision as a function of the number of evaluations - a main further work is the analysis of the dependency in the dimension. Another crucial point is the weakness of evolutionary algorithms, when they don't use surrogate models, for noisy optimization problems with strong models of noise; this does not appear in the results of the BBOB challenge, as the BBOB challenge is restricted so that it contains only fitness functions which are reasonably well solved by classical evolution strategies, but we have clearly seen that on other noise models there's really a trouble in e.g. UH-CMA. The quadratic logistic regression by R. Coulom performs much better for binary fitness values: evolutionary algorithm, under their current form, are not able to sample the most informative points instead of focusing on the currently best points: there's a subtle issue here which is not only the exploration/exploitation dilemma. A natural solution for things like that are tools like IAGO (but at the price of a huge computational cost) or surrogate models. QLR has its own weaknesses, and noisy optimization is a very open field for further research.

- The bounds and algorithms proposed in this document for active learning are tighter than existing bounds. The parallel active learning analysis is new.

- This is not in the main topics of this document (but see Table 15.1 with very clear positive results for quasi-random numbers) We proposed the use of Quasi-Monte-Carlo methods in active learning[233], for mutations in evolutionary algorithms[232], for restarts[208]. A main conclusion is that, at least with good Quasi-Random sequences, we are rarely disappointed when replacing random by Quasi-Random.

- Some other main parts of my research are not cited here, including global optimization. Essentially, these results show the significant improvement that we can have by using quasi-random restarts instead of random restarts, and also show that some sophisticated algorithms are in fact not better than restarts. A deep problem around multimodal optimization is that we have little understanding of

what is a good model of multimodal optimization problem, leading to contradictory published results; also, some simple baselines have not been used in some published tests, whereas they seemingly perform as well as more sophisticated techniques in many cases.

Further research

In my humble opinion, the main directions for future research around topics discussed in this document are:

- including the use of value function into MCTS algorithms (as done in [160] for Amazons);
- sharing information between nodes in MCTS algorithms, more than in Rave; some preliminary and interesting works around that are [91, 191, 190, 134, 90]. As pointed out by M. Müller in [175], introducing some "divide-and-conquer" ideas into MCTS might be a key for strong future improvements. Factorized states/actions might be a key for further research; taking inspiration from the works in reinforcement learning might be useful[145, 214].
- evaluating the dependency in the dimension in noisy (evolutionary) optimization.
- the speed-up of parallel optimization algorithms improved so much in the recent years that I guess many improvements are still possible - each publication provides huge improvements, and therefore this is probably a good direction of research. There are ultimate limits which are not so far away (see the log speed-up in chapter 14) but only in the monomodal case.
- globalization of optimization algorithms is, in my humble opinion, necessary, and not yet fully analyzed in the evolutionary community. We need good models of multimodal problems for setting the basis for further work on this, and we have to keep quasi-random restarts in the comparison.

Appendix A

Sociological and biological elements

What is inspired by, or related to, biology or sociology in algorithms above, and can we compare humans, animals and computers ?

Computers are often compared to nature. This provides nice pictures, as in Fig. A.1. Games are a nice tool for visualizing the differences between humans and computers, as discussed in section A.2; some cases show the clear superiority of humans (are there cases of animal superiority over computers in mind sports ?). Neural networks are obviously related to biology (section A.3). Nature is the biggest evolutionary algorithm experimentation (section A.4).

A.1 Games, humans and economy

Obviously, the title of this section is by far too ambitious for the few paragraphs below. We will just outline a few things about games and their interpretation for humans, in the special case in which no limit on the computational power of humans is involved - this means that we consider games in which the mathematical solving is possible and easy, but are nonetheless interesting for what they say about who we are.

Games have been widely used as a model for the economic world. A widely used model is that humans have rational actions, maximizing their own reward; this is in particular used in works emphasizing the market economy as the solution for everything (this is often attributed to Adam Smith in [216]; yet, Adam Smith himself had discussed a lot the limitations of this approach and therefore the weakness of this model should not be considered as a mistake by Adam Smith, who was indeed strongly in favor of public services, in spite of the fact that he is usually cited by people who want the destruction of public services). Nonetheless:

- there are plenty of examples of games for which the “rational” solution, in the sense of the solution which arises if everyone is egocentric and maximizes only his own reward, is a disaster, whereas a simple collaboration makes the solution

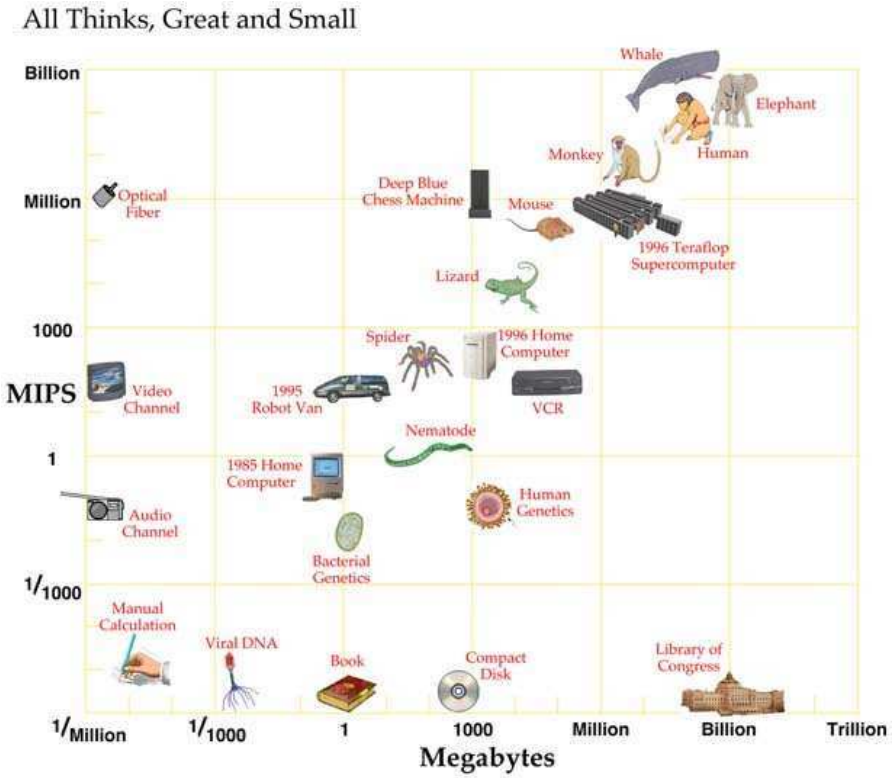


Figure A.1: Computers and animals on the same graph. X-axis: megabytes of memory. Y-axis: millions of instructions per second. Source: H. Moravec (1998). The main weakness of this graph is that networks are not taken into account; the networks of the human or animal brains are very impressive.

much better. A very nice example is **Rosenthal's centipede game**[199]: there are 100 rounds (0,1,2,...,99). The pot is made of p_t euros at round t. At the beginning, there's no money in the pot ($p_0 = 0$); the player (whose turn it is to play) chooses between:

- **Stealing** half of the pot, plus one; i.e. his reward is $p_t/2+1$; the other player takes the other half minus 1, i.e. $p_t/2-1$. The game is over.
- **Cooperating**: in that case (and except at the last round) the pot is increased by 2: $p_{t+1} = p_t + 2$. The other player becomes to play.

After the 100 rounds, if no player has stolen the pot, then both players earn half of the pot. Informally, the game is therefore as follows:

- Alice plays first; the pot contains 0 euro. She can "steal" 1 euro (and Bob earns nothing), and then the game stops.

- If Alice does not steal, the pot contains 2 euros; Bob can steal 2 euros (and Alice earns nothing), and then the game stops.
- If Bob does not steal, the pot contains 4 euros; Alice can steal 3 euros (and Bob earns 1 euro), and then the game stops.
- If Alice does not steal, the pot contains 6 euros; Bob can steal 4 euros (and Alice earns 2 euros), and then the game stops.
- ...
- If Alice (resp. Bob) does not steal, the pot contains $2k$ euros; Bob (resp. Alice) can steal $k+1$ euros (and Alice (resp. Bob) earns $k-1$ euros), and then the game stops.
- ...
- If Alice does not steal, the pot contains 198 euros; Alice can steal 100 euros (and Bob earns 98 euros), and then the game stops.
- Otherwise, both players earn 99 euros.

One can immediately check that at the last round, Bob earns more money if he steals (he then earns 100 euros instead of 99). Therefore Bob should steal, and Alice will earn nothing. Knowing this, Alice should steal if the game reaches the 98^{th} round... Knowing this, Bob should steal if the game reaches the 97^{th} round, and so on. By induction, we see that the "best" (the "rational") choice is always to steal. The consequence is that Alice earns 1 euro, Bob earns nothing, whereas the trivial solution with complete cooperation gives 99 euros to both. Humans are usually somewhere between this (dirty) "rationality" and full cooperation; as the number of rounds increases, humans become closer to "rationality". Experiments show that humans tend to steal if the amounts are big, and not if the amounts are small; humans are seemingly more gentle, less egocentric, for small things than for big things... Nonetheless, for the ultimatum game (discussed below), [178] shows that very high stakes lead to equilibrated decisions in which players have an equal income; things are therefore not so simple.

- There are many other cases in which humans choose a non-optimal solution (non-optimal from a "rational" point of view), because they have a sense of dignity, politeness, which leads to much better overall solutions than "rational solutions". Counter-examples to rational behaviors in an egocentric world arise immediately for parents taking care of their children, or taking care of their old parents; but a more surprising example is the so-called **ultimatum game**. In the ultimate game, a player A is given 100 \$. A must propose a deal to the player B: he should propose some amount x, and b should either accept or reject. If B accepts, B earns x euros, and A earns $100-x$; if B rejects, they don't earn anything. In the "rational agents" theory, A should propose 1 euro to B, B should accept and win 1 euro, and A would win 99 euros - in reality, B usually accepts only if he gets at least 20 euros (and often at least 40 euros). This happens even if the game is anonymous and if players will never meet again. This is true for most cultures, but there are cultures in which people propose less, and cultures in which people propose more than in our occidental civilizations[179].

These results are mathematically simple, but they show that "rationality" (as discussed above, this term "rational" might be irrelevant due to its usual non-mathematical meaning...) is clearly not a good solution, and does not take into account the possibility of cooperation. As a summary,

> Humans usually cooperate much more than what is predicted in the so-called "rational" (which means selfish) model.
>
> Cooperative biases which deviate from the rational model lead to better overall rewards.
>
> They cooperate less for important things than for small ones.

A.2 Games and interaction in the animal and human world

Why are games important for research in computer science ?

Humans, computer & games. Games are fun for humans, and they are also fun for animals. This is usually used as an argument for justifying research in games: if games are important for animals, they are probably also important for computers. I prefer to emphasize that pointing out games in the animal world is fun and philosophically/ethologically/psychologically interesting. Nonetheless, there are serious reasons for considering that games are really important for computers, as well as for humans or animals:

- We use games for comparing artificial intelligence systems. I find that funny that without our work on Go, we would never have had the opportunity of testing Monte-Carlo Tree Search on industrial applications. After all, this is not so far from choosing a sexual partner from the result of a fight between mooses.
- We use games for understanding weaknesses of computers (as in the example from Fig. 5.14). An important weakness of Monte-Carlo Tree Search is shown in section 5.3.5: this weakness is termed Semeai, and thanks to this weakness in Go we could understand a general limitation of the MCTS approach.

Animals and games. Some examples of animals who like games are known:

- many animals (e.g. cats) play during childhood, mimicking hunting and battles.
- macaques can play with snowballs, and learn that game by social interaction[94].
- games can cross species; eared seals sometimes play with ocean sunfishes.

Collaboration between animals. Some examples of collaborative activities between animals (see [166] for more details):

- Dictyostelium Discoideum (amoeba) are able of suicide for building fruiting bodies (the stalk in Fig. 11.2, right, is made of dead cells)[93].
- in groups of muskoxen, the strongest will be in the frontline in an attack.
- great tits can take care of orphans.
- penguins form groups when there is a cold wind, and the group rotates so that the penguins who suffer from the wind are regularly changed.
- many species can get grouped for protection against predators; also, predators can establish bands for attacking more efficiently some preys by a sophisticated methodology (pack-hunting strategies). Pack hunting strategies are quite elaborate, require sophisticated planning abilities (see the complexity of decentralized POMDP[127]), and it's conjectured that dinosaurs were already able of pack-hunting[181].

Games and wars. The fact that games can be used in order to train, or for replacing war, is also true for humans. For the case of the game of Go, there are legends according to which some wars have been replaced by Go games. Also, ritualized fights (which are a kind of game, more or less dangerous depending on animals) exist among animals for avoiding real fights. This exists both among humans and among animals, and can be used for biasing the choice of sexual partners[157, 153, 83][1].

A.3 Neural networks

One might find inspiration in biology for **neural networks**; for example, sparsity as in reservoir computing, the importance of unsupervised learning in recent deep networks, the use of recurrent neural networks. Mathematics and empirism might be considered as a more comfortable framework for inspiring neural networks: neural networks are essentially a parametric family of functions in which derivatives are easy to compute and which empirically don't need too many parameters for approximating complex landscapes. See Fig. A.1 for a (user-friendly) comparison between natural neural networks and computers.

A recent improvement in robotics which was directly inspired by biology is **central pattern generators**; this is briefly sketched in section 11.2 and shows that even from lampreys we have many things to learn. Also, **anticipatory behaviors** (with architectures dedicated to anticipation) [184] have their roots in biology; the idea is to separate prediction and decision; and to use predictions in decision rules.

Backpropagation, or time-consuming iterations of learning algorithms, have sometimes be **related to dreams**; it is believed that all mammals and birds have dreams[247] (do dinosaurs dream ?), with a decomposition in states related to the structure of human dreams. The very effective **convolutional neural networks** [151, 250] are sometimes related to biological networks and their DNA succinct representation.

Radial basis functions are also quite related to biological input layers.

[1] "Involvement in wrestling had a significant positive effect on mens number of offspring and a marginally significant effect on polygyny, controlling for age, body condition and socioeconomic status." [157]

A.4 Evolutionary optimization, Nash equilibria and real life

In this section we'll see (very briefly) three classical topics in the analysis of natural evolution, and their counterparts in artificial evolution.

A.4.1 Is the result of evolutionary optimization optimal ?

Coevolution as often simulated on computers is quite analogous to dynamic game theory, and is a reasonable model for biological coevolution. Fictitious play[193] is a formal algorithm that can be considered as dynamic game theory: each individual, in turn, is optimized in front of the other individuals. It is known that fictitious play does not necessarily optimize the reward (in particular when cooperation is involved) but provides solutions close to nature. This has been used for explaining various suboptimalities in the animal world:

- **Suboptimality of the size of bands of female lions**; often of size 3, whereas the optimal size is 2 (depending on the preys) - roughly, 3 is evolutionary stable, whereas 2 is best on average for the lions;

- **Existence of cheating behaviors in various animal species** (e.g. birds), who don't look for food and just wait for another bird to find something and then try to steal something, whereas the optimal strategy consists in searching (all together) and sharing the rewards; interestingly, in that case, mathematical game theory can predict the proportion of cheaters in a population ([57] and references therein).

- A less impressive but philosophically interesting example is the case of **competition between males for females or of females for males** - it is limited in some (but not all) species, so that fights are "ritualized" so that they don't lead to murders. Also, **males often kill children of other males** (e.g. white bears), what is clearly suboptimal for the community, but efficient at the individual level (incidentally, female lions protect their children, but if they can't avoid the murder of their children will try to have new children with the killer).

For more on this, the interested reader is referred to [58, 113, 107, 57]. As a summary,

> Many animals have a collectively suboptimal behavior.
>
> Mathematical models of (co-)evolution predict this behavior (sometimes quantitatively).

A.4.2 Diversity in nature and diversity in computer evolution

The diversity mechanism (i) as a tool for robustness in front of dynamic environments (ii) as a necessity for optimal joint work (collaboration) is sometimes emphasized in some really funny (yet serious, but not fully admitted) works [85]:

- Cases in which specificities which are dangerous at the individual level are efficient at the cooperative (global) level: schizophrenia and anorexia are examples of disease which might be (or have been) helpful for populations, whenever at the individual level they are dangerous.
- Sickle-cell disease might be useful in some cases ; populations with such a disease are more robust in front of paludism. This is an example in which a disease might be correlated with a good property; a disease might sometimes be here due to its efficiency in some variants of the environments (i.e. due to a dynamic environment).
- Homosexuality among male geese [47] has also been shown as a good tool for protecting children in case of hostile environment (see [47] for understanding how homosexuality among male geese can be useful for reproduction in hostile environments).

A related point is that nature provides aging and a delay before reproducing ability, what has sometimes been emphasized in artificial evolutionary algorithms as well[135, 48].

A.4.3 Sex is useful

The use of (μ, λ)-strategies instead of $(1, \lambda)$ is directly related to the existence of sex (defined as: a new individual is a combination/mutation of at least two individuals): it improves convergence rates. This was already shown in [29] and generalize to high scale cases in [224]. Please note, however, that what is good is not the fact that there are two different sexes (what might be good for other reasons, like diversity), but the fact that two different persons can share their DNA (this is also done in some paramecia, without having two different sexes). Also, this does not explain why, in most sexual reproduction, there are two partners involved; a key point might be diversity. In nature, "merging" DNA from two different species is (by definition of species) impossible; this is consistent with cross-over operators, in evolutionary algorithms, limited to individuals which are close to each other.[158].

Yet, some particular cases must be emphasized (these particular cases remind us the use of migrations between artificial "islands" in artificial evolution); there are in nature surprising examples of "merging" between different species. Humans contain mitochondria, which were seemingly incorporated from archeobacteria in eucaryotic cells[97]. Therefore we contain DNA from proteobacteria. This is an example of "merge" between different species, termed endosymbiosis. Chloroplasts are another example of endosymbiosis (between plant cells and cyanobacteria).

There are many recent works aimed at explaining psychology or various aspects of animals' life by evolutionary aspects[249, 85].

As a summary,

Sharing DNA is crucial, both in mathematical models and in the real world[29]. This does not imply the existence of two (or more) different sexes.

Most simple mathematical models encourage high-scale DNA sharing with more than 2 partners ($\mu > 1$), but this result is independent of logistic constraints and it assumes that efficient mutations frequently occur - the discrepancy with the usual schema of 1 or 2 partners in nature might come from this.

The existence of different sexes (males and females) might be good for population diversity and can be seen as a more or less collaborative game between males and females[159].

A crucial difference between computational coevolution and biological coevolution is that in biology there's no mecanism for optimizing the overall behavior (this implies suboptimal behavior), except when several groups compete for resources (selection at the level of groups). The importance of group selection in the biological world is controversial.

Another difference (between artificial evolution targeted to engineering and natural evolution) is that on a non-parallel (or a limitedly parallel) computer a huge population has an immediate negative consequence on the overall efficiency (nature is an unlimited computer for species which have nearly infinite room to conquer around them).

Bibliography

[1] *Reinforcement Learning and Dynamic Programming using Function Approximators*. Taylor & Francis CRC Press, 2010.

[2] C. Amato, D. Bernstein, and S. Zilberstein. Optimizing fixed-size stochastic controllers for POMDPs and decentralized POMDPs. In *AAMAS*, 2009.

[3] T. S. Anantharaman, M. Campbell, and F. hsiung Hsu. Singular extensions: Adding selectivity to brute-force searching. *Artif. Intell.*, 43(1):99–109, 1990.

[4] D. Angluin. Queries and concept learning. *Mach. Learn.*, 2(4):319–342, 1988.

[5] B. Arneson, R. Hayward, and P. Henderson. Mohex wins hex tournament. *ICGA journal*, pages 114–116, 2009.

[6] D. V. Arnold and H.-G. Beyer. Efficiency and mutation strength adaptation of the (mu/mui,lambda)-es in a noisy environment. In M. S. et al., editor, *Parallel Problem Solving from Nature*, volume 1917 of *LNCS*, pages 39–48. springer, 2000.

[7] D. V. Arnold and H.-G. Beyer. A general noise model and its effects on evolution strategy performance. *IEEE Transactions on Evolutionary Computation*, 10(4):380–391, 2006.

[8] D. V. Arnold and H. georg Beyer. Evolution strategies with cumulative step length adaptation on the noisy parabolic ridge. Technical report, 2006.

[9] L. Arnold, H. Paugam Moisy, and M. Sebag. Unsupervised Layer-Wise Model Selection in Deep Neural Networks. In *ECAI 2010 19th European Conference on Artificial Intelligence (ECAI'10)*, Lisbon Portugal, 08 2010.

[10] K. Astrom. Optimal control of Markov decision processes with incomplete state estimation. *Journal of Mathematical Analysis and Applications*, 10:174–205, 1965.

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

[12] P. Audouard, G. Chaslot, J.-B. Hoock, J. Perez, A. Rimmel, and O. Teytaud. Grid coevolution for adaptive simulations; application to the building of opening books in the game of Go. In *Proceedings of EvoGames*, pages 323–332. Springer, 2009.

[13] P. Auer. Using confidence bounds for exploitation-exploration trade-offs. *The Journal of Machine Learning Research*, 3:397–422, 2003.

[14] A. Auger, S. Finck, N. Hansen, and R. Ros. BBOB 2009: Comparison Tables of All Algorithms on All Noisy Functions. Technical Report RT-0384, INRIA, 04 2010.

[15] A. Auger and O. Teytaud. Continuous lunches are free plus the design of optimal optimization algorithms. *Algorithmica*, Accepted.

[16] T. Bäck, F. Hoffmeister, and H.-P. Schwefel. Extended selection mechanisms in genetic algorithms. In R. K. Belew and L. B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, San Mateo, CA, 1991. Morgan Kaufmann Publishers.

[17] J. E. Baker. Reducing bias and inefficiency in the selection algorithm. In *Proceedings of the Second International Conference on Genetic Algorithms on Genetic algorithms and their application*, pages 14–21, Mahwah, NJ, USA, 1987. Lawrence Erlbaum Associates, Inc.

[18] M.-F. Balcan, A. Broder, and T. Zhang. Margin based active learning. In *Proc. of the 20 th Conference on Learning Theory*, 2007.

[19] R. Bellman. *Dynamic Programming*. Princeton Univ. Press, 1957.

[20] Y. Bengio and Y. LeCun. *Scaling learning algorithms towards AI*. MIT Press, 2007.

[21] H. Berliner. Backgammon program beats world champ. *ACM SIGART Bulletin*, 69:6–9, 1980.

[22] D. S. Bernstein, R. Givan, N. Immerman, and S. Zilberstein. The complexity of decentralized control of markov decision processes. In *Mathematics of Operations Research*, page 2002, 2000.

[23] S. Bernstein. On a modification of chebyshev's inequality and of the error formula of laplace. *Original publication: Ann. Sci. Inst. Sav. Ukraine, Sect. Math. 1*, 3(1):38–49, 1924.

[24] S. Bernstein. *The Theory of Probabilities*. Gastehizdat Publishing House, Moscow, 1946.

[25] V. Berthier, H. Doghmen, and O. Teytaud. Consistency Modifications for Automatically Tuned Monte-Carlo Tree Search. In *Proceedings of Lion4*, page 14, 2010.

[26] V. Berthier, H. Doghmen, and O. Teytaud. Consistency Modications for Automatically Tuned Monte-Carlo Tree Search. In *Lion4*, page 14 p., venice Italy, 2010.

[27] D. Bertsekas and J. Tsitsiklis. *Neuro-dynamic Programming*. Athena Scientific, 1996.

[28] D. P. Bertsekas and D. P. Bertsekas. *Nonlinear Programming*. Athena Scientific, 2nd edition, September 1999.

[29] H.-G. Beyer. Toward a theory of evolution strategies: On the benefit of sex - the $(\mu/\mu,\lambda)$-theory. *Evolutionary Computation*, 3(1):81–111, 1995.

[30] H.-G. Beyer. *The Theory of Evolutions Strategies*. Springer, Heidelberg, 2001.

[31] H.-G. Beyer. Personal communication, 2008.

[32] H.-G. Beyer and B. Sendhoff. Covariance matrix adaptation revisited - the CMSA evolution strategy. In G. Rudolph, T. Jansen, S. M. Lucas, C. Poloni, and N. Beume, editors, *Proceedings of PPSN*, pages 123–132, 2008.

[33] D. Billings. *Algorithms and Assessment in Computer Poker*. PhD thesis, Univ. Alberta, 2006.

[34] C. Bishop. Curvature-driven smoothing: a learning algorithm for feedforward networks. *IEEE Transactions on Neural Networks*, 4(5):882–884, 1993.

[35] L. Bottou and O. Bousquet. The tradeoffs of large scale learning. In J. C. Platt, D. Koller, Y. Singer, and S. T. Roweis, editors, *NIPS*. MIT Press, 2007.

[36] A. Bourki, G. Chaslot, M. Coulm, V. Danjean, H. Doghmen, J.-B. Hoock, T. Herault, A. Rimmel, F. Teytaud, O. Teytaud, P. Vayssiere, , and Z. Yu. Scalability and parallelization of monte-carlo tree search. In *Proceedings of The International Conference on Computers and Games*, 2010.

[37] C. Boutilier, editor. *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, 2009.

[38] M. Boutin. Les jeux de pions en france dans les années 1900 et leurs liens avec les jeux étrangers. linvention dun jeu singulier : lattaque. In *Proceedings of BGA'2010*, 2010.

[39] L. Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996.

[40] R. P. Brent. *Algorithms for Minimization without Derivatives*. Prentice-Hall, Englewood Cliffs, N.J., 1973.

[41] D. Brockhoff and E. Zitzler. Dimensionality Reduction in Multiobjective Optimization with (Partial) Dominance Structure Preservation: Generalized Minimum Objective Subset Problems. TIK Report 247, Computer Engineering and Networks Laboratory (TIK), ETH Zurich, Apr. 2006.

[42] D. Brockhoff and E. Zitzler. Objective reduction in evolutionary multiobjective optimization: Theory and applications. *Evol. Comput.*, 17(2):135–166, 2009.

[43] C. G. Broyden. The convergence of a class of double-rank minimization algorithms 2. *The New Algorithm. J. of the Inst. for Math. and Applications*, 6:222–231, 1970.

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

[45] S. Bubeck. *Bandits games and clustering foundations*. PhD thesis, Université Lille 1, 2010.

[46] Bühlmann and B. Yu. Analyzing bagging. *Annals of Statistics*, 30:927–961, 2002.

[47] D. Buiscan. *Ethologie comparée*. Hachette, 1995.

[48] J. A. Bullinaria. The effect of learning on life history evolution. In *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 222–229, New York, NY, USA, 2007. ACM.

[49] R. Byrd, P. Lu, J. Nocedal, and C.Zhu. A limited memory algorithm for bound constrained optimization. *SIAM J. Scientific Computing, vol.16, no.5*, 1995.

[50] T. Cazenave. A phantom-go program. In *ACG*, pages 120–125, 2006.

[51] T. Cazenave. Nested monte-carlo search. In Boutilier [37], pages 456–461.

[52] T. Cazenave and J. Borsboom. Golois wins phantom go tournament. *ICGA Journal*, 30(3):165–166, 2007.

[53] T. Cazenave and B. Helmstetter. Combining tactical search and monte-carlo in the game of go. *IEEE CIG 2005*, pages 171–175, 2005.

[54] T. Cazenave and N. Jouandeau. On the parallelization of UCT. In *Proceedings of CGW07*, pages 93–101, 2007.

[55] S. Celis and D. R. Musicant. Weka-parallel: Machine learning in parallel. Technical report, Carleton College, CS TR, 2002.

[56] N. Cesa-bianchi, A. Conconi, and C. Gentile. Learning probabilistic linear-threshold classifiers via selective sampling. In *In Proc. 16th COLT*, pages 373–386. Springer, 2003.

[57] F. Cezilly, L.-A. Giraldeau, and G. Théraulaz. *Les sociétés animales: lions, fourmis et ouistitis*. Le Collège de la Cité, 2006.

[58] F. Cézilly, F. Thomas, V. Médoc, and M. Perrot-Minnot. Host-manipulation by parasites with complex life cycles: adaptive or not? *Trends Parasitol*, 2010.

[59] K. Chaloner. Bayesian design for estimating the turning point of a quadratic regression. *Communications in Statistics—Theory and Methods*, 18(4):1385–1400, 1989.

[60] A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer. Alternation. *J. ACM*, 28(1):114–133, 1981.

[61] G. Chaslot, S. de Jong, J.-T. Saito, and J. Uiterwijk. Monte-carlo tree search in production management problems. In *Proceedings of BNAIC*, 2006.

[62] G. Chaslot, J.-T. Saito, B. Bouzy, J. W. H. M. Uiterwijk, and H. J. van den Herik. Monte-Carlo Strategies for Computer Go. In P.-Y. Schobbens, W. Vanhoof, and G. Schwanen, editors, *Proceedings of the 18th BeNeLux Conference on Artificial Intelligence, Namur, Belgium*, pages 83–91, 2006.

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

[64] G. Chaslot, M. Winands, and H. van den Herik. Parallel Monte-Carlo Tree Search. In *Proceedings of the Conference on Computers and Games 2008 (CG 2008)*, pages 60–71, 2008.

[65] H. Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Annals of Math. Stat.*, 23:493–509, 1952.

[66] C. A. Coello Coello. Constraint-handling techniques used with evolutionary algorithms. In *GECCO '09: Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference*, pages 3089–3110, New York, NY, USA, 2009. ACM.

[67] D. Cohn, L. Atlas, and R. Ladner. Improving generalization with active learning. *Mach. Learn.*, 15(2):201–221, 1994.

[68] D. Cohn, Z. Ghahramani, and M. Jordan. Active Learning with Statistical Models. *Journal of Artificial Intelligence Research*, 4:129–145, 1996.

[69] R. Collobert and S. Bengio. A gentle hessian for efficient gradient descent. In *IEEE International Conference on Acoustic, Speech, and Signal Processing, ICASSP*, 2004.

[70] R. Collobert, S. Bengio, and Y. Bengio. A parallel mixture of svms for very large scale problems. *Neural Comput.*, 14(5):1105–1114, 2002.

[71] A. Conn, K. Scheinberg, and L. Toint. Recent progress in unconstrained nonlinear optimization without derivatives, 1997.

[72] R. Coulom. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. *In P. Ciancarini and H. J. van den Herik, editors, Proceedings of the 5th International Conference on Computers and Games, Turin, Italy*, 2006.

[73] R. Coulom. Computing elo ratings of move patterns in the game of go. In *Computer Games Workshop, Amsterdam, The Netherlands*, 2007.

[74] R. Coulom. Lockless hash table and other parallel search ideas, 2008. Post on the computer-go mailing list.

[75] R. Coulom. Source code for QLR, Mar. 2010. http://remi.coulom.free.fr/QLR/.

[76] R. Coulom, P. Rolet, N. Sokolovska, and O. Teytaud. Handling expensive optimization with large noise. Submitted.

[77] M. Crasmaru and J. Tromp. Ladders are PSPACE-complete. In *Computers and Games*, pages 241–249, 2000.

[78] S. Dasgupta. Coarse sample complexity bounds for active learning. In Y. Weiss, B. Schölkopf, and J. Platt, editors, *Advances in Neural Information Processing Systems 18*, pages 235–242. MIT Press, Cambridge, MA, 2006.

[79] S. Dasgupta. Coarse sample complexity bounds for active learning. In Y. Weiss, B. Schölkopf, and J. Platt, editors, *Advances in Neural Information Processing Systems 18*, pages 235–242. MIT Press, Cambridge, MA, 2006.

[80] S. Dasgupta, A. T. Kalai, and C. Monteleoni. Analysis of perceptron-based active learning. In *In COLT*, pages 249–263, 2005.

[81] F. De Mesmay, A. Rimmel, Y. Voronenko, and M. Püschel. Bandit-Based Optimization on Graphs with Application to Library Performance Tuning. In *ICML*, Montréal Canada, 2009.

[82] K. Deb and D. Kalyanmoy. *Multi-Objective Optimization Using Evolutionary Algorithms*. Wiley, 1 edition, June 2001.

[83] T. Decarvalho, P. Watson, and S. F. Scott. Costs increase as ritualized fighting progresses within and between phases in the sierra dome spider, neriene litigiosa. *animal behavior*, 63:473–482, 2004.

[84] O. Delalleau and Y. Bengio. Parallel stochastic gradient descent. In *CIAR summer school*, 2007.

[85] A. Demaret. *Ethologie et psychiatrie*. Mardaga, 1995.

[86] A. Devert, N. Bredeche, and M. Schoenauer. Blindbuilder: A new encoding to evolve lego-like structures. In P. Collet, M. Tomassini, M. Ebner, S. Gustafson, and A. Ekárt, editors, *EuroGP*, volume 3905 of *Lecture Notes in Computer Science*, pages 61–72. Springer, 2006.

[87] L. Devroye, L. Györfi, and G. Lugosi. *A probabilistic Theory of Pattern Recognition*. Springer, 1997.

[88] J. Dieudonné. *Pour l'honneur de l'esprit humain*. Hachette, 1987.

[89] S. Doncieux, J.-B. Mouret, and N. Bredeche. Exploring new horizons in evolutionary design of robots. In *IROS Workshop "Exploring New Horizons in Evolutionary Design of Robots"*, pages 5–12, Saint Louis, USA, october 2009.

[90] P. Drake. The last-good-reply policy for monte-carlo go. *ICGA Journal*, 32(4), 2009.

[91] P. Drake and Y.-P. Chen. Coevolving partial strategies for the game of go. In *International Conference on Genetic and Evolutionary Methods*. CSREA Press, 2008.

[92] S. Droste, T. Jansen, and I. Wegener. Perhaps Not a Free Lunch But At Least a Free Appetizer. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *Proceedings of theFirstGenetic and Evolutionary Computation Conference(GECCO '99)*, pages 833–839, San Francisco CA, 13–17 1999. Morgan Kaufmann Publishers, Inc.

[93] D. B. Dusenbery. *Life at Small Scale*. Freeman, New York, 1996 1996.

[94] G. Eaton. Snowball construction by a feral troop of japanese macaques (macaca fuscata) living under seminatural conditions. *Primates*, 13:411–14, 1972.

[95] B. Efron. Bootstrap methods: another look at the jackknife. *Annals of Statistics*, 7:1–26, 1979.

[96] A.-E. Eiben. Principled approach to tuning EA parameters. In *Tutorial of CEC*, 2009.

[97] V. V. Emelyanov. Mitochondrial connection to the origin of the eukaryotic cell. *European journal of biochemistry*, 270(8), 2003.

[98] M. Enzenberger and M. Müller. A lock-free multithreaded Monte-Carlo tree search algorithm. In *Proceedings of Advances in Computer Games 12*, 2009.

[99] E. Fackle Fornius. *Optimal Design of Experiments for the Quadratic Logistic Model*. PhD thesis, Department of Statistics, Stockholm University, 2008.

[100] H. Finnsson and Y. Björnsson. Simulation-based approach to general game playing. In *AAAI'08: Proceedings of the 23rd national conference on Artificial intelligence*, pages 259–264. AAAI Press, 2008.

[101] J. M. Fitzpatrick and J. J. Grefenstette. Genetic algorithms in noisy environments. *Machine Learning*, 3:101–120, 1988.

[102] R. Fletcher. A new approach to variable-metric algorithms. *Computer Journal*, 13:317–322, 1970.

[103] R. Fonteneau, S. Murphy, L. Wehenkel, and D. Ernst. Model-free monte carlo-like policy evaluation. In *Proceedings of The Thirteenth International Conference on Artificial Intelligence and Statistics (AISTATS 2010), JMLR W&CP Volume 9*, pages 217–224, 2010.

[104] H. Fournier and O. Teytaud. Lower bounds for comparison based evolution strategies using vc-dimension and sign patterns. *Algorithmica*, January 2010.

[105] Y. Freund, H. S. Seung, E. Shamir, and N. Tishby. Selective sampling using the query by committee algorithm. *Mach. Learn.*, 28(2-3):133–168, 1997.

[106] Y. Freund and R. Shapire. Experiments with a new boosting algorithm. In L. Saitta, editor, *Proceedings of the 13th International Conference on Machine Learning*, pages 148–156. Morgan Kaufmann, 1996.

[107] S. Garnier, C. Jost, R. Jeanson, J. Gautrais, M. Asadpour, G. Caprari, and G. Theraulaz. Aggregation behaviour as a source of collective decision in a group of cockroach-like-robots. In *ECAL*, pages 169–178, 2005.

[108] R. Gaudel and M. Sebag. Feature selection as a one-player game. In *Proceedings of ICML*, 2010.

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

[110] S. Gelly, S. Ruette, and O. Teytaud. Comparison-based algorithms: worst-case optimality, optimality w.r.t a bayesian prior, the intraclass-variance minimization in eda, and implementations with billiards. In *PPSN-BTP workshop*, 2006.

[111] S. Gelly, S. Ruette, and O. Teytaud. Comparison-based algorithms are robust and randomized algorithms are anytime. *Evolutionary Computation Journal (MIT Press), Special issue on bridging Theory and Practice*, 15(4):26p, 2007.

[112] S. Gelly and D. Silver. 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, 2007. ACM Press.

[113] L.-A. Giraldeau and T. Caraco. *Social Foraging Theory*. Princeton University Press, May 2000.

[114] K. Gödel. ber formal unentscheidbare stze der principia mathematica und verwandter systeme, i. *Monatshefte fr Mathematik und Physik*, 38:173–198, 1931.

[115] D. Goldfarb. A family of variable-metric algorithms derived by variational means. *Mathematics of Computation*, 24:23–26, 1970.

[116] J. Goldsmith and M. Mundhenk. Complexity issues in markov decision processes. In *Proc. of 13th Conference on Computational Complexity*, 1998.

[117] H. P. Graf, E. Cosatto, L. Bottou, I. Durdanovic, and V. Vapnik. Parallel support vector machines: The cascade svm. In *In Advances in Neural Information Processing Systems*, pages 521–528. MIT Press, 2005.

[118] F. Gruau. *Neural Network Synthesis using Cellular encoding and the Genetic Algorithm*. PhD thesis, Ecole Normale Superieure de Lyon, 1994.

[119] F. Gruau. Modular genetic neural networks for six-legged locomotion. In *Artificial Evolution*, pages 201–219, 1995.

[120] Y. Guo and D. Schuurmans. Discriminative batch mode active learning. In *Advances in Neural Information Processing Systems (NIPS)*, pages 593–600, Cambridge, MA, 2008. MIT Press.

[121] J. L. Gustafson. Reevaluating amdahl's law. *Commun. ACM*, 31(5):532–533, 1988.

[122] U. Hammel and T. Bck. Evolution strategies on noisy functions: How to improve convergence properties. In Y. Davidor, H.-P. Schwefel, and R. Mnner, editors, *Parallel Problem Solving From Nature*, volume 866 of *LNCS*, pages 159–168, Jerusalem, 9–14Oct. 1994. springer.

[123] N. Hansen. Source code for UH-CMA, June 2008. Version 3, http://www.lri.fr/ hansen/cmaesintro.html.

[124] N. Hansen. Personal communication. 2009.

[125] N. Hansen, A. Niederberger, L. Guzzella, and P. Koumoutsakos. A Method for Handling Uncertainty in Evolutionary Optimization with an Application to Feedback Control of Combustion. *IEEE Transactions on Evolutionary Computation*, 2009.

[126] N. Hansen and A. Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary Computation*, 11(1), 2003.

[127] R. A. Hearn and E. Demaine. *Games, Puzzles, and Computation*. AK Peters, 2009.

[128] V. Heidrich-Meisner and C. Igel. Hoeffding and bernstein races for selecting policies in evolutionary direct policy search. In *ICML '09: Proceedings of the 26th Annual International Conference on Machine Learning*, pages 401–408, New York, NY, USA, 2009. ACM.

[129] M. Helmert and B. Nebel. Principles of ai planning. Technical report, Albert-Ludwigs-Universitat Freiburg, 2006.

[130] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, 2008.

[131] W. Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58:13–30, 1963.

[132] S. C. H. Hoi, R. Jin, J. Zhu, and M. R. Lyu. Batch mode active learning and its application to medical image classification. In *ICML '06: Proceedings of the 23rd international conference on Machine learning*, pages 417–424, New York, NY, USA, 2006. ACM.

[133] S. C. H. Hoi, R. Jin, J. Zhu, and M. R. Lyu. Semisupervised svm batch mode active learning with applications to image retrieval. *ACM Trans. Inf. Syst.*, 27(3):1–29, 2009.

[134] J.-B. Hoock, C.-S. Lee, A. Rimmel, F. Teytaud, O. Teytaud, and M.-H. Wang. Intelligent agents for the game of go. *Computational Intelligence Magazine*, accepted.

[135] C. Horoba, T. Jansen, and C. Zarges. Maximal age in randomized search heuristics with aging. In *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 803–810, New York, NY, USA, 2009. ACM.

[136] A. J. Ijspeert and J. Kodjabachian. Evolution and development of a central pattern generator for the swimming of a lamprey. *Artificial Life*, 5(3):247–269, July 1999.

[137] D. C. Ince, editor. *Collected Works of A. M. Turing: Mechanical Intelligence*. North-Holland, Amsterdam, The Netherlands, 1992.

[138] V. S. Iyengar, C. Apte, and T. Zhang. Active learning using adaptive resampling. In *Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 91–98, 2000.

[139] M. Jebalia and A. Auger. On multiplicative noise models for stochastic search. In *Parallel Problem Solving From Nature*, dortmund Allemagne, 2008.

[140] D. R. Jones, M. Schonlau, and W. J. Welch. Efficient global optimization of expensive black-box functions. *J. of Global Optimization*, 13(4):455–492, 1998.

[141] H. Kato. Post on the computer-go mailing list, october, 2009.

[142] A. I. Khuri, B. Mukherjee, B. K. Sinha, and M. Ghosh. Design issues for generalized linear models: A review. *Statistical Science*, 21(3):376–399, 2006.

[143] L. Kocsis and C. Szepesvari. Bandit based Monte-Carlo planning. In *15th European Conference on Machine Learning (ECML)*, pages 282–293, 2006.

[144] R. E. Korf. Depth-first iterative-deepening: an optimal admissible tree search. *Artif. Intell.*, 27(1):97–109, 1985.

[145] O. Kozlova, O. Sigaud, P.-H. Wuillemin, and C. Meyer. Considering unseen states as impossible in factored reinforcement learning. In *ECML/PKDD (1)*, pages 721–735, 2009.

[146] S. R. Kulkarni, S. K. Mitter, and J. N. Tsitsiklis. Active learning using arbitrary binary valued queries. *Mach. Learn.*, 11(1):23–35, 1993.

[147] S. R. Kulkarni, S. K. Mitter, and J. N. Tsitsiklis. Active learning using arbitrary binary valued queries. *Mach. Learn.*, 11(1):23–35, 1993.

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

[149] T. J. Lambert III, M. A. Epelman, and R. L. Smith. A fictitious play approach to large-scale optimization. *Oper. Res.*, 53(3):477–489, 2005.

[150] P. Larranaga and J. A. Lozano. *Estimation of Distribution Algorithms. A New Tool for Evolutionary Computation*. Kluwer Academic Publishers, 2001.

[151] Y. LeCun and Y. Bengio. word-level training of a handwritten word recognizer based on convolutional neural networks. In IAPR, editor, *Proc. of the International Conference on Pattern Recognition*, volume II, pages 88–92, Jerusalem, October 1994. IEEE.

[152] C.-S. Lee, M.-H. Wang, G. Chaslot, J.-B. Hoock, A. Rimmel, O. Teytaud, S.-R. Tsai, S.-C. Hsu, and T.-P. Hong. The Computational Intelligence of MoGo Revealed in Taiwan's Computer Go Tournaments. *IEEE Transactions on Computational Intelligence and AI in games*, 2009.

[153] A. Lessa and S. M. F. M. de Souza. Broken noses for the gods: ritual battles in the atacama desert during the tiwanaku period. *Memrias do Instituto Oswaldo Cruz*, pages 133–138, 2006.

[154] M. Lindenbaum, S. Markovitch, and D. Rusakov. Selective sampling for nearest neighbor classifiers. *Machine Learning*, 54:125–152, 2004.

[155] M. L. Littman, A. R. Cassandra, and L. P. Kaelbling. Efficient dynamic-programming updates in partially observable markov decision processes. Technical report, Providence, RI, USA, 1995.

[156] X. Liu, L. O. Hall, and K. W. Bowyer. Comments on "a parallel mixture of svms for very large scale problems". *Neural Comput.*, 16(7):1345–1351, 2004.

[157] V. Llaurens, M. Raymond, and C. Faurie. Ritual fights and male reproductive success in a human population. *Evolutionary biology*, 22:1854–1859, 2009.

[158] T. Lodé. Genetic divergence without spatial isolation in polecat mustela putorius populations. *Evol. Biol.*, 14:228–236, 2001.

[159] T. Lodé. *La guerre des sexes chez les animaux, une histoire naturelle de la sexualité*. Odile Jacob, 2006.

[160] R. J. Lorentz. Amazons discover monte-carlo. In *CG '08: Proceedings of the 6th international conference on Computers and Games*, pages 13–24, Berlin, Heidelberg, 2008. Springer-Verlag.

[161] S. Lucas, P. Cowling, and C. Browne. Mcts for games and beyond: Upcoming epsrc project. In *MCTS: state of the art workshop, London*. Imperial College, London, 2010.

[162] O. Madani, S. Hanks, and A. Condon. On the undecidability of probabilistic planning and related stochastic optimization problems. *Artif. Intell.*, 147(1-2):5–34, 2003.

[163] M. Margenstern. Cellular automata and combinatoric tilings in hyperbolic spaces: a survey. In *DMTCS'03: Proceedings of the 4th international conference on Discrete mathematics and theoretical computer science*, pages 48–72, Berlin, Heidelberg, 2003. Springer-Verlag.

[164] D. Marks and J. Clarkson. Conservatism as non-bayesian performance: A reply to de swart. *Acta Psychologica*, 37-1:55–63, 1973.

[165] P. Massé. *Les Réserves et la Régulation de l'Avenir dans la vie Economique*. Herman, 1946.

[166] K. L. Matignon. *The Emotional Life of Animals*. Konecky, 2006.

[167] M.-L. Minsky and S. Papert. *Perceptrons: An Introduction to Computational Geometry*. MIT Press, Cambridge, 1969.

[168] T. Mitchell. Generalization as search. *Artificial Intelligence*, 18:203–226, 1982.

[169] V. Mnih, C. Szepesvári, and J.-Y. Audibert. Empirical Bernstein stopping. In *ICML '08: Proceedings of the 25th international conference on Machine learning*, pages 672–679, New York, NY, USA, 2008. ACM.

[170] H. Moravec. When will computer hardware match the human brain? *Journal of Transhumanism*, 1, March 1998.

[171] M. Mundhenk, J. Goldsmith, C. Lusena, and E. Allender. Complexity of finite-horizon markov decision process problems. *J. ACM*, 47(4):681–720, 2000.

[172] H. Nakhost and M. Müller. Monte-carlo exploration for deterministic planning. In Boutilier [37], pages 1766–1771.

[173] V. Nannen and A. E. Eiben. Relevance estimation and value calibration of evolutionary algorithm parameters. In *IJCAI'07: Proceedings of the 20th international joint conference on Artifical intelligence*, pages 975–980, San Francisco, CA, USA, 2007. Morgan Kaufmann Publishers Inc.

[174] A. Nelson, E. Grant, G. Barlow, and T. Henderson. A colony of robots using vision sensing and evolved neural controllers. In *Proceedings of IROS*, 2003.

[175] A. Network. Monte carlo tree search (mcts) workshop, 2010.

[176] J. V. Nunen. A set of successive approximation methods for discounted markovian decision problems. *Z. Oper. Res.*, 20:203–208, 1976.

[177] N. Omont. *La convergence des modularités structurelles et fonctionnelles des systèmes complexes*. PhD thesis, Université d'Evry-Val d'Essonne 2009., 2009.

[178] H. Oosterbeek, R. Sloof, and G. van de Kuilen. Cultural differences in ultimatum game experiments: Evidence from a meta-analysis. *Experimental Economics*, 7:171–188, 2004. 10.1023/B:EXEC.0000026978.14316.74.

[179] B. Paciotti. Interethnics contacts, 2010.

[180] P. Panov and S. Džeroski. Combining bagging and random subspaces to create better ensembles. In *IDA'07: Proceedings of the 7th international conference on Intelligent data analysis*, pages 118–129, Berlin, Heidelberg, 2007. Springer-Verlag.

[181] S. Parker. *Dinosaur pack-hunters*. Grolier educational, 2000.

[182] L. Péret and F. Garcia. On-line search for solving markov decision processes via heuristic sampling. In *Proceedings of ECAI'04*, pages 530–534, 2004.

[183] G. Peterson, J. Reif, and S. Azhar. Lower bounds for multiplayer non-cooperative games of incomplete information. *COMPUT. MATH. APPL*, 41:2001, 2001.

[184] G. Pezzulo, M. V. Butz, O. Sigaud, and G. Baldassarre, editors. *Anticipatory Behavior in Adaptive Learning Systems, From Psychological Theories to Artificial Cognitive Systems [4th Workshop on Anticipatory Behavior in Adaptive Learning Systems, ABiALS 2008, Munich, Germany, June 26-27, 2008]*, volume 5499 of *Lecture Notes in Computer Science*. Springer, 2009.

[185] L. D. Phillips and W. Edwards. Conservatism in a simple probability inference task. *Journal of Experimental Psychology*, 72(3):346–354, 1966.

[186] M. J. D. Powell. Developments of newuoa for minimization without derivatives. *IMA J Numer Anal*, pages drm047+, February 2008.

[187] M. L. Puterman and M. C. Shin. Modified policy iteration algorithms for discounted markov decision problems. *Management Science*, 24:1127–1137, 1978.

[188] A. Rahimi and B. Recht. Weighted sums of random kitchen sinks: Replacing minimization with randomization in learning. In *NIPS*, pages 1313–1320, 2008.

[189] I. Rechenberg. *Evolutionstrategie: Optimierung Technischer Systeme nach Prinzipien des Biologischen Evolution*. Fromman-Holzboog Verlag, Stuttgart, 1973.

[190] A. Rimmel and F. Teytaud. Multiple Overlapping Tiles for Contextual Monte Carlo Tree Search. In *Evostar*, Istanbul Turquie.

[191] A. Rimmel, F. Teytaud, and O. Teytaud. Biasing Monte-Carlo Simulations through RAVE Values. In *The International Conference on Computers and Games 2010*, Kanazawa Japon, 05 2010.

[192] J. Rintanen. Complexity of Planning with Partial Observability. In *Proceedings of ICAPS'03 Workshop on Planning under Uncertainty and Incomplete Information*, Trento, Italy, June 2003.

[193] J. Robinson. An iterative method of solving a game. *Annals of Mathematics*, 54:296–301, 1951.

[194] J. M. Robson. The complexity of go. In *IFIP Congress*, pages 413–417, 1983.

[195] P. Rolet, M. Sebag, and O. Teytaud. Optimal active learning through billiards and upper confidence trees in continous domains. In *Proceedings of the ECML conference*, 2009.

[196] P. Rolet, M. Sebag, and O. Teytaud. Optimal robust expensive optimization is tractable. In *Gecco 2009*, page 8 pages, Montréal Canada, 2009. ACM.

[197] P. Rolet and O. Teytaud. Bandit-based estimation of distribution algorithms for noisy optimization: Rigorous runtime analysis. In *Proceedings of Lion4 (accepted); presented in TRSH 2009 in Birmingham*, 2009.

[198] P. Rolet and O. Teytaud. Adaptive noisy optimization. In *EvoApplications (1)*, pages 592–601, 2010.

[199] R. Rosenthal. Games of perfect information, predatory pricing, and the chain store. *Journal of Economic Theory*, 25:92–100, 1981.

[200] N. Roy and A. McCallum. Toward optimal active learning through sampling estimation of error reduction. In *Proc. 18th International Conf. on Machine Learning*, pages 441–448. Morgan Kaufmann, San Francisco, CA, 2001.

[201] O. Rudenko and M. Schoenauer. Dominance based crossover operator for evolutionary multi-objective algorithms. *CoRR*, abs/cs/0505080, 2005.

[202] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representation by error propagation. In D. E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, volume 1, pages 318–362. Cambridge, MA: MIT Press, 1986.

[203] N. Sauer. On the density of families of sets. *J. Comb. Theory, Ser. A*, 13(1):145–147, 1972.

[204] S. Schaal, A. Ijspeert, and A. Billard. *computational approaches to motor learning by imitation*, pages 199–218. Number 1431. oxford university press, 2004.

[205] J. Schaeffer, N. Burch, Y. Bjornsson, A. Kishimoto, M. Muller, R. Lake, P. Lu, and S. Sutphen. Checkers is solved. *Science*, pages 1144079+, July 2007.

[206] A. I. Schein and L. H. Hungar. Active learning for logistic regression: An evaluation. *Machine Learning*, 68(3):235–265, 2007.

[207] M. Schoenauer, F. Teytaud, and O. Teytaud. A rigorous runtime analysis for quasi-random restarts and decreasing step-size. *submitted*, 2010.

[208] M. Schoenauer, F. Teytaud, and O. Teytaud. A rigorous runtime analysis for quasi-random restarts and decreasing step-size. Submitted.

[209] G. Schohn and D. Cohn. Less is more: Active learning with support vector machines. *Proceedings of the Seventeenth International Conference on Machine Learning*, 282:285–286, 2000.

[210] H.-P. Schwefel. Adaptive Mechanismen in der biologischen Evolution und ihr Einfluss auf die Evolutionsgeschwindigkeit. Interner Bericht der Arbeitsgruppe Bionik und Evolutionstechnik am Institut für Mess- und Regelungstechnik Re 215/3, Technische Universität Berlin, Juli 1974.

[211] H. S. Seung, M. Opper, and H. Sompolinsky. Query by committee. In *COLT '92: Proceedings of the fifth annual workshop on Computational learning theory*, pages 287–294, New York, NY, USA, 1992. ACM.

[212] D. F. Shanno. Conditioning of quasi-newton methods for function minimization. *Mathematics of Computation*, 24:647–656, 1970.

[213] O. Sigaud and O. Buffet, editors. *Markov Decision Processes and Artificial Intelligence*. Wiley - ISTE, 2010. ISBN: 978-1-84821-167-4.

[214] O. Sigaud, M. V. Butz, O. Kozlova, and C. Meyer. Anticipatory learning classifier systems and factored reinforcement learning. In *ABiALS*, pages 321–333, 2008.

[215] S. Smit and A. Eiben. Parameter tuning of evolutionary algorithms: Generalist vs. specialist. In C. Chio, S. Cagnoni, C. Cotta, M. Ebner, A. Ekárt, A. I. Esparcia-Alcazar, C.-K. Goh, J. J. Merelo, F. Neri, M. Preus, J. Togelius, and G. N. Yannakakis, editors, *Applications of Evolutionary Computation*, volume 6024, chapter 56, pages 542–551. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[216] A. Smith. *An Inquiry into the Nature and Causes of the Wealth of Nations*. 1776.

[217] M. Sugiyama and N. Rubens. A batch ensemble approach to active learning with model selection. *Neural Netw.*, 21(9):1278–1286, 2008.

[218] W. Sun, J. Han, and J. Sun. Global convergence of nonmonotone descent methods for unconstrained optimization problems. *J. Comput. Appl. Math.*, 146(1):89–98, 2002.

[219] G. Tesauro. Td-gammon, a self-teaching backgammon program, achieves master-level play. *Neural Comput.*, 6(2):215–219, 1994.

[220] F. Teytaud and O. Teytaud. Bias and variance in continuous eda. In *Proceedings of EA09*. springer LNCS, 2009.

[221] F. Teytaud and O. Teytaud. Creating an Upper-Confidence-Tree program for Havannah. In *ACG 12*, Pamplona Spain, 2009.

[222] F. Teytaud and O. Teytaud. On the parallel speed-up of Estimation of Multivariate Normal Algorithm and Evolution Strategies. In *Proceedings of EvoStar 2009*, page accepted, 2009.

[223] F. Teytaud and O. Teytaud. Why one must use reweighting in estimation of distribution algorithms. In *Proceedings of Gecco*, 2009.

[224] F. Teytaud and O. Teytaud. Log(lambda) Modifications for Optimal Parallelism. In *Parallel Problem Solving From Nature*, Krakow Pologne, 09 2010.

[225] F. Teytaud and O. Teytaud. *Solving optimization problems with comparisons only*. in press.

[226] O. Teytaud. On the hardness of offline multiobjective optimization. *Evolutionary Computation Journal*, 2007.

[227] O. Teytaud. On the almost optimality of blind active sampling. In F. d'Alché Buc, editor, *Conférence d'Apprentissage*, Porquerolles France, 2008.

[228] O. Teytaud. When does quasi-random work?. In G. Rudolph, T. Jansen, S. M. Lucas, C. Poloni, and N. Beume, editors, *PPSN*, volume 5199 of *Lecture Notes in Computer Science*, pages 325–336. Springer, 2008.

[229] O. Teytaud and A. Auger. On the adaptation of the noise level for stochastic optimization. In *IEEE Congress on Evolutionary Computation*, Singapour, 2007.

[230] O. Teytaud and H. Fournier. Lower bounds for evolution strategies using vc-dimensio n. In G. Rudolph, T. Jansen, S. M. Lucas, C. Poloni, and N. Beume, editors, *PPSN*, volume 5199 of *Lecture Notes in Computer Science*, pages 102–111. Springer, 2008.

[231] O. Teytaud and S. Gelly. General lower bounds for evolutionary computation. In *proceedings of PPSN*, 2006.

[232] O. Teytaud and S. Gelly. DCMA: yet another derandomization in covariance-matrix-adaptation. In *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 955–963, New York, NY, USA, 2007. ACM.

[233] O. Teytaud, S. Gelly, and J. Mary. Active learning in regression, with application to stochastic dynamic programming. In *ICINCO and CAP*, 2007.

[234] O. Teytaud and E. Vazquez. *Designing An Optimal Search Algorithm With Respect to Prior Information*. in press.

[235] A. Turing and J.-Y. Girard. *La machine de Turing*. Seuil, 1999.

[236] A. M. Turing. Computing machinery and intelligence. *Mind*, 59(236):433–460, 1950.

[237] A. V. D. Vaart and J. Wellner. *Weak Convergence and Empirical Processes*. Springer series in statistics, 1996.

[238] V. N. Vapnik. *The Nature of Statistical Learning*. Springer Verlag, 1995.

[239] M. Vidyasagar. *A Theory of Learning and Generalization*. Springer-Verlag, New York, New York, 1997.

[240] J. Villemonteix, E. Vazquez, and E. Walter. An informational approach to the global optimization of expensive-to-evaluate functions. *Journal of Global Optimization*, Submitted.

[241] Y. Wang, J.-Y. Audibert, and R. Munos. Algorithms for infinitely many-armed bandits. In *Advances in Neural Information Processing Systems*, volume 21, 2008.

[242] M. K. Warmuth, J. Liao, G. Rätsch, M. Mathieson, S. Putta, and C. Lemmen. Support vector machines for active learning in the drug discovery process. *Journal of Chemical Information Sciences*, 43:667–673, 2003.

[243] B. Weinberg and E.-G. Talbi. NFL theorem is unusable on structured classes of problems. In *Proceedings of the 2004 IEEE Congress on Evolutionary Computation*, pages 220–226, Portland, Oregon, 20-23 June 2004. IEEE Press.

[244] D. Whitley. The GENITOR algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, San Mateo, CA, 1989. Morgan Kaufman.

[245] Wikipedia. Alpha-beta pruning, 2010.

[246] Wikipedia. Computer backgammon, 2010.

[247] R. Wilkerson. The evolution of REM dreaming: New research includes all mammals. *Electric Dreams*, 10(1), 2003.

[248] D. Wolfe. Go endgames are pspace-hard. In *More Games of No Chance*, pages 125–136. Cambridge University Press, 2001.

[249] L. Workman, W. Reader, and (trad) Francoise Parot. *Psychologie évolutionniste une introduction*. De Boeck, 2007.

[250] Q. Wu, Y. LeCun, L. D. Jackel, and B. Jeng. on-line recognition of limited vocabulary chinese character using multiple convolutional neural networks. In *Proc. of the 1993 IEEE International Symposium on circuits and systems*, volume 4, pages 2435–2438. IEEE, 1993.

[251] C. Zhu, R. Byrd, P.Lu, and J. Nocedal. L-BFGS-B: a limited memory FORTRAN code for solving bound constrained optimization problems. *Technical Report, EECS Department, Northwestern University*, 1994.

Index