



Iterative optimization -Complexity and Efficiency are not antinomic

Maurice Clerc

► To cite this version:

Maurice Clerc. Iterative optimization -Complexity and Efficiency are not antinomic. 2024. hal-04487869

HAL Id: hal-04487869

<https://hal.science/hal-04487869v1>

Preprint submitted on 4 Mar 2024

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

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

Iterative optimization - Complexity and Efficiency are not antinomic

Maurice Clerc*

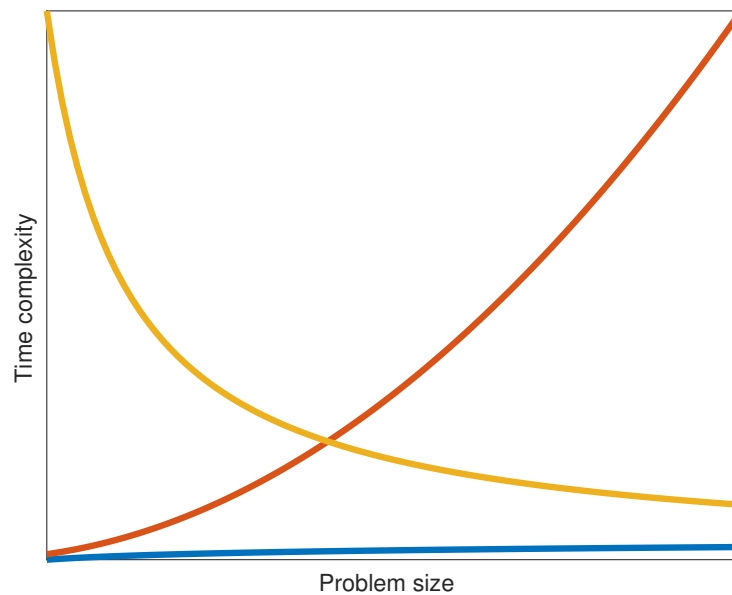
2024-03-03

Ce qui se conçoit bien s'énonce clairement

Nicolas Boileau, L'Art poétique, 1674

Abstract

When we compare two algorithms, complexity and efficiency are often thought of as almost the same, implying that low complexity indicates good efficiency. However, these two concepts are actually distinct. We provide theoretical reasons and a few examples to demonstrate that an algorithm can be both more complex and more efficient than another at the same time. We also address several common misconceptions about iterative optimization.



Thanks

To my colleague and friend Mohamed G. Omran for our fruitful discussions. Even though he doesn't agree with all my conclusions!

*Maurice.Clerc@WriteMe.com

1 Introduction

The user is asking, “Which algorithm, A or B , is the best for my optimization problem?”

Even if we say “better” means getting a better result without more effort in searching (usually by evaluating different points in the search area), finding the answer is still not easy.

Sometimes you can read something like “ B is better because its complexity is lower”. However, if not based on other measures such a claim is highly questionable, to say the least.

Here, we’ll provide just a few examples to illustrate that the connection between time complexity or even execution time and efficiency may not be straightforward. However, before we proceed, let’s accurately define the concepts we are working with.

2 Definitions

Some concepts like search space, landscape, population size, etc. are supposed to be well known by the reader. But some other have to be clarified or are even new.

Definition 1. Input and input size

We are examining an algorithm designed to locate the minimum of a numerical function within a specified search space.

Two methods can be employed to define its input. The first approach considers the input as the problem itself. In this scenario, to analyze the algorithm’s behavior as the input size increases, the problem must be scalable, typically achieved through increasing its dimension D (number of variables).

The second approach involves defining the input to include additional elements such as population size, maximum number of iterations, etc. In this case, to study the asymptotic behavior, the problem does not necessarily need to be scalable, as we can adjust these other elements.

We have chosen to adopt the first definition here, as the second one may lead to inconsistencies (see 1).

So the size of the problem takes into account all its components (search space, function, constraints) and nothing else. However, in practice, the last two are even often overlooked. This is because it is assumed that the computational time they induce is the same for all algorithms. So only the “volume” of the search space is referred to as the size of the problem. Moreover, to avoid some technical difficulties we may consider that the search space is discrete. So, finally, the problem size is simply the number of possible positions in it (but see the Remark 1). In what follows we call it n .

Definition 2. Solution

This is a position in the search space on which the value of the function is minimum or smaller than an user-defined threshold. In that case one can call it “acceptable position”. A discrete unimodal problem has just one solution. If a threshold is defined, a quasi-continuous unimodal problem has many solutions. And of course a multimodal problem, discrete or not, may have several solutions.

Definition 3. Execution time $T_{f,g}$

Or running/computing time. It is depending on the input size, on some parameters of the algorithm and on the function f and constraints g .

Typically we may have something like $T_{f,g}(n, N, t) = CnN^2t$ where n is the input size, N a population size, t a number of time steps (iterations) and C a constant.

When *comparing* two algorithms the time taken to sample a position and to evaluate the function and some constraints are assumed to be independent of the algorithm and can be ignored.

Similarly we usually don’t take implementation details into account. Particularly, when it comes to theoretical analyses, we assign a specific duration to each elementary operation. In the case of experimental comparisons, we assume, at the very least, that the algorithms are executed under the same conditions (same computer, same software environment). Note that, though, it implicitly assumes that both algorithms are perfectly coded. It is not always the case ...

In practice, when there is no ambiguity, the indices f and g are omitted and we simply write T . This is what we will do from now on.

Definition 4. Probabilistic execution time T_p

A shortened name for “execution time for a given probability of success”.

Execution time alone is not very meaningful. It has to be considered along the probability p to find a solution. Actually, for a given total search effort (number of evaluations, usually over several runs for a stochastic algorithm) comparing two algorithms is a bi-objective problem whose parameters are the execution time and the probability of success. An user may prefer an algorithm that is fast but not very successful or the contrary.

Assuming f and g are omitted the T_p notation means “execution time to reach a solution with a probability at least equal to p ”. For example, for some algorithms, deterministic or not, it is important to know that on some problems we have $T_\varepsilon(*) = \infty$, where $*$ means “any list of parameters”, no matter how small ε is, which simply means they never find a solution. In short it could be written $T_0(*) = \infty$.

More generally, for a stochastic algorithm on a given function we can often prove, at least experimentally, a relationship between the probability of success and the execution time. It is usually increasing but not always, in particular when some parameters are dynamically modified as functions of the current time step (see 5).

Definition 5. Space complexity

Total memory space taken by the algorithm with respect to the input size. Space complexity includes both Auxiliary space and space used by input. Auxiliary space is the extra space or temporary space used by an algorithm. We do not study the space complexity here, in particular for it is often possible to reduce it by recomputing some things instead to save/recall them. Of course it then increases the execution time.

Definition 6. Time complexity O

The time complexity is referring to the asymptotic behavior of the execution time when we consider an increasing problem size and keep everything else constant. In particular it means we can evaluate it by analyzing the execution time for just one iteration.

It is usually given thanks to the big- O notation, in which the input size is typically denoted n . So we may have $O(n)$, $O(\ln(n))$, etc. Do not confuse with execution/running/computing time. In particular we have $C \times O(*) = O(*)$ for any constant C , where $*$ means ‘any function of n ’, as this is not true for the execution time.

Using it in practice to compare algorithms may lead to wrong conclusions. If the computing time of the algorithm A is $10n$ and if the one of B is $10^{10}n$ they have the same time complexity but on any real problem A is largely faster. Any user would find it more efficient.

So it is important to know how the time complexity is calculated to avoid such situations.

Time complexity usually is evaluated by analyzing the execution time.

If we are unable to conduct such an analysis, it is always possible to execute the algorithm on a scalable problem and record the execution times for growing problem sizes. In practice, it is quite common to estimate the shape of the time complexity by dividing these running times by the number of iterations (see 7.1).

It does make sense because time complexity it is not itself an execution time. Its mathematical nature is a class of equivalence.

Let’s consider a time complexity with a formula of $O(r(n))$. We suppose that when n tends to infinity r tends to a function that characterizes a certain class of growth rate. This implies that r represents a member of an equivalence class defined by the following relation:

$$\text{Two functions } r_1 \text{ and } r_2 \text{ are equivalent if } \lim_{n \rightarrow \infty} \left(\frac{r_1(n)}{r_2(n)} \right) = 1$$

While it is possible for a strange optimization algorithm to have a time complexity such as $O(\sin(n))$, we do not consider here such functions for which the above definition of class of equivalence may be difficult to apply.

For more details about *asymptotic analysis* see for example [2] or simply the Wikipedia page about this topic.

Classical classes in optimization are: Constant, Linear, Logarithmique, Polynomial, etc. but there are less common ones like Inverse Ackerman, which grows very slowly or, on the contrary, Double exponential. For all of them r is non-decreasing but this is not mandatory. It could perfectly be decreasing (see 2).

To write symbolically that an algorithm A has a lower time complexity than an algorithm B we can write

$$A \underset{O}{\prec} B$$

If the complexities are the same

$$A \underset{O}{=} B$$

Definition 7. Probabilistic time complexity O_p

This is the time complexity when the algorithm has to reach a solution with a probability at least equal to p . It is defined by the asymptotic behavior of the probable execution time T_p .

Definition 8. Efficiency comparison

Algorithm A is considered more efficient (equivalent) than algorithm B for a given problem if it exhibits a better (equivalent) result (more generally a higher probability of success) for the same search effort (measured by the number of positions evaluated) ¹. We note it

$$A \succ B$$

or $A \succeq B$ in case of possibly equivalent results. This definition can be extended to a set of problems (a test data set) if A is more efficient (equivalent) in the majority of them when compared.

Definition 9. Asymptotic efficiency comparison

Algorithm A is considered asymptotically more efficient than (or equivalent to) algorithm B for a given problem if it exhibits a lower (or equal) probabilistic time complexity for a given probability of success. We note it

$$A \underset{O_p}{\succ} B$$

or $A \underset{O_p}{=} B$ in case of equivalence. As the previous one this definition can be extended to a set of problems.

Remark 1. For a (pseudo-)continuous search space the number of possible positions is depending on the ε -machine of the device on which you run the algorithm. If the search space is $[x_{min}, x_{max}]^D$ it would be $\left(\frac{x_{max}-x_{min}}{\varepsilon}\right)^D$. However when we just want to *compare* algorithms a more common measure is simply D .

Remark 2. The computing time studied here assumes that the algorithm is working on a special kind of computer called a deterministic Turing machine. This means it may not be accurate if the algorithm is running on a quantum machine or using stochastic algorithms. We're not discussing the first case here. In the second case, it's common to assess an average execution time for a specific problem (not to be confused with an average across different problems). Think of each run of a stochastic algorithm as if it were a run of a deterministic algorithm, based on a sequence of random numbers that could, in theory, be predecided and even used again.

¹A detailed "Result vs Search effort" approach can be found in [4], but here we use something simpler.

3 Misconceptions

Claim 1. The time complexity depends on the maximum number of iterations. Variant: it depends on the population size.

Not always a misconception, but this claim is valid only for special cases.

On one hand, according to the definition of time complexity, this assertion is clearly incorrect if the input of the algorithm is the problem and nothing else; time complexity depends solely on the parameter n , which represents then the size of the problem under consideration. On the other hand, claims like this (or similar ones) are so common in the literature that it may be beneficial to try to understand why.

The primary reason is likely the confusion between running time and time complexity – a confusion between standard numerical calculations and asymptotic calculations on equivalence classes.

Let's consider a classic algorithm for which the running time is $T = C \times D \times N \times \text{iter}_{\max}$ where C is a constant, D the dimension of the problem, N the population size and iter_{\max} the maximum number of iterations (or time steps) allowed to the algorithm.

As the time complexity must be $O(r(n))$ it is tempting to define $n = C \times D \times N \times \text{iter}_{\max}$. Writing this means that N and iter_{\max} are elements of the problem size. Debatable, but let's continue the reasoning.

As we are in the context of asymptotic analysis we can ignore the constant C and if the population size is also constant we can write that the time complexity is $O(D \times \text{iter}_{\max})$. If iter_{\max} is constant, no problem, we finally have $O(n)$ with $n = D$, assuming the problem is scalable.

But if iter_{\max} is seen as element of the input size, we can increase it to infinity in order to define the asymptotic behavior with $n = D \times \text{iter}_{\max}$.

Suppose that the user sets it to a higher value $\text{iter}'_{\max} = (\text{iter}_{\max})^2$. The time complexity is now $O(D \times \text{iter}_{\max}^2)$ that is not the same equivalence class. Hence the inconsistency of the claim, at least on the general case.

The reasoning regarding the population size is similar. If we substitute N with N^2 , the new time complexity is no longer in the same equivalence class. Additionally, the algorithm is now different as well. Therefore, it is not accurate to discuss its time complexity as we essentially have two distinct algorithms.

A workaround is to assume that t_{\max} (or N) is determined by the algorithm itself as a function of D . However, this assertion does not hold for all algorithms and cannot be generalized. Furthermore, similar to the population size, if we alter the function (for example, using $10D^2$ instead of $1000D$), we once again introduce two distinct algorithms, each with its own time complexity.

So, when we encounter a statement in a paper such as “time complexity of this algorithm is $O(n)$ with $n = C \times D \times N \times \text{iter}_{\max}$,” we should mentally substitute this assertion with “execution time of this algorithm is $C \times D \times N \times \text{iter}_{\max}$,” unless N and iter_{\max} are determined by the algorithm itself as functions of the problem size.

Claim 2. When an algorithm is $O(r(n))$ the r function is increasing

Optimization algorithms are seldom $O(1)$, leading one to easily assume that in $O(r(n))$, the r function can not decrease. However, there is no theoretical reason for that, and it is easy to find counterexamples. A rudimentary one is: on $\{1, 2, \dots, n\}$ find by random search a number at most equal to a given constant C .

Although artificial, this example illustrates two points:

1. the asymptotic behavior of an algorithm may be the one of a decreasing function;
2. it may depend on the problem to be solved.

Notice how a small difference between two problems can lead to completely different behaviors. If the problem for random search is “find x such that $x = C$,” the time complexity is $O(n)$. However, if the problem is “find x such that $x \leq C$ ” the complexity is $O(1/n)$.

Even in practical situations, it may be necessary to employ a decreasing r function. Certain algorithms automatically adapt their population size by utilizing an increasing function of the search space dimension D , such as $N = 10D$. And in such a case the time complexity is indirectly depending on N . If the evaluation cost of a position rises rapidly with D , as seen in

cases like $f(x_1, x_2, \dots) = x_1^{x_2}$, it becomes unmanageable. Consequently, the user may need to be less demanding and defines a decreasing function of D for the population size, potentially leading to a decreasing r function. Naturally, this adjustment may result in reduced effectiveness of the algorithm.

Claim 3. The time complexity of an algorithm is problem independent

Not always. See the previous claim.

More generally, an algorithm can employ various strategies during the search process, depending on the gathered information. This is often the case, among other things, when it calls upon replacement functions (surrogate functions). It would then be absurd to assign it a time complexity and claiming it is problem independent.

So we should never say “this algorithm is $O(r(n))$ ” but “this algorithm is $O(r(n))$ for this (kind of) problem”.

Claim 4. A lower time complexity implies a better efficiency. Variant: same time complexity means same efficiency.

Let’s consider two algorithms A and B that are applied to the same problem. A is $O(n)$, B is $O(n^2)$ so one may think that A is more efficient. But it doesn’t make sense if A is unable to find a solution or, more generally, if it finds one with a significantly lower probability.

That is why adding a probability threshold is useful. If A is $O_{p_1}(n)$ and B is $O_{p_2}(n^2)$ with $p_1 \geq p_2$ then A indeed is more efficient. But if $p_1 < p_2$ we can not draw any conclusion. We have a bi-objective problem here, involving time complexity and probability. Only the user can decide whether it is worth employing a lower time complexity algorithm with a reduced probability of success, and to what extent.

Claim 5. The higher the search effort the better the solution, or equal in the worst case

Let’s suppose you run the algorithm with an user-defined maximum number of time steps t_{max} and that the search effort, usually the number of evaluations, is an increasing function of the current time step t . For instance for many algorithms the search effort is Nt , where N is a population size.

If it makes use of a variable parameter depending on $t_{max} - t$ or t/t_{max} or similar formulae, its efficiency can perfectly be decreasing for an increasing t_{max} . For example, if a local search is applied with a radius that is a decreasing function of t/t_{max} , the higher the value of t_{max} , the slower the decrease in radius. This may lead to the algorithm becoming less efficient, especially for unimodal landscapes.

Claim 6. For a scalable problem the higher the dimension the higher the difficulty

A well-known counterexample is the Griewank function (see [5]). This is because as the dimension increases, the number of local minima also increases, but the domain of each of them quickly becomes very small. Therefore, any reasonable algorithm can more easily escape them and find the global minimum.

More generally finding the minimum of a function in several variables might have more apparent trends and less chance of getting stuck in local minima compared to lower-dimensional problems. For example the binary Rastrigin function defined on $(0, 1, \dots, 2^6 - 1)$ has seven local minima and one global one. Now if we define it on $(0, \dots, 2^3 - 1) \times (0, \dots, 2^3 - 1)$ the landscape becomes unimodal (figure 1).

4 Complexity vs Efficiency - Example 1

The input is the search space $S = 0, 1, \dots, m - 1^D$. Its size, i.e. the number of possible positions is $n = m^D$. The value space is $V = 0, 1, \dots, v - 1$. There are v^n possible functions from S to V that define the benchmark \mathbb{B} . For a given function f of \mathbb{B} we are looking for the position(s) x^* so that $f(x^*) = 0$.

According to the above definition of quality comparison we define the efficiency of the algorithm A for the problem f for a computing time equal to t as the probability to find a solution (here 0) after t evaluations of positions. Let’s call it $E_A(f, t)$. Of course for a deterministic algorithm it can be only 0 or 1. The average efficiency on \mathbb{B} is noted $E_A(\mathbb{B}, t)$.

We now compare two simple algorithms, exhaustive search and Random search on the set of problems defined by \mathbb{B} . What are the time complexity and the average efficiency of each of them?

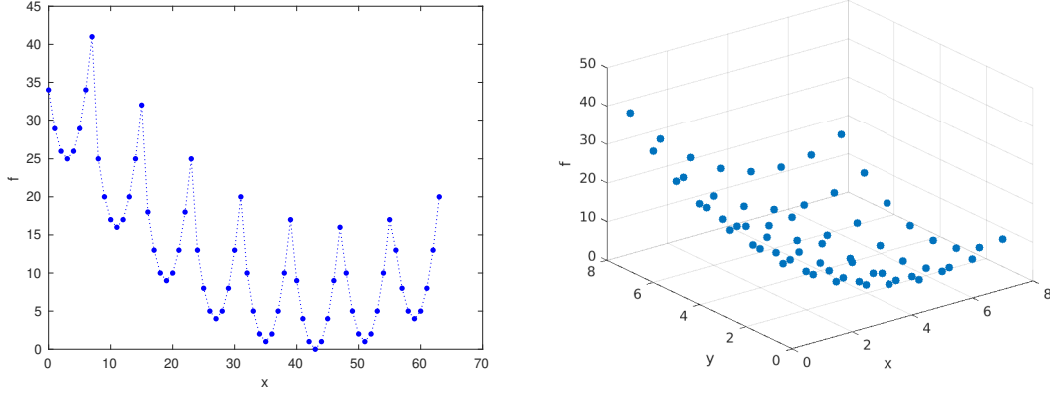


Figure 1: Binary Rastrigin is easier in 2D than in 1D

4.1 Exhaustive Search XS

The algorithm evaluates each position of the search space S in any order but for simplicity we assume the positions are ordered from 1 to n .

For a given function the worst-case probabilistic time complexity is $O_1(n)$ if there is a solution or $O_1(\infty)$ if there is no solution at all. Let's compute the average asymptotic efficiency on \mathbb{B} .

- Worst case if $f(x) \neq 0, \forall x \in S$. There are $(v-1)^n$ such functions for which $E_{XS}(f, t) = 0, \forall t$.
- General case: $f(x) = 0$ on position k and $f(x) \neq 0$ for positions of rank inferior to k . There are $(v-1)^{k-1} v^{n-k}$ such functions, for which $E_{XS}(f, k) = 1$.
- Average on \mathbb{B} :

$$E_{XS}(\mathbb{B}, t) = \frac{1}{v^n} \left(\sum_{k=1}^t (v-1)^{k-1} v^{n-k} \right) \quad (1)$$

For easier comparison we can set the search effort t to n . For this value if there is a solution (general case) it is found for sure and the mean value is simply

$$E_{XS}(\mathbb{B}, n) = 1 - \left(1 - \frac{1}{v} \right)^n \quad (2)$$

This function of n and v can be plot, as on the figure 2. When both n and v tend to infinity (i.e. for quasi-continuous problems) the efficiency tends to $1 - \frac{1}{e} \simeq 0.6321$ (see 7.3.1).

4.2 Random Search RS

At each iteration the execution time is the same, not depending on the search space size (see 7.1). So the time complexity is $O(1)$... if we don't consider the probability of success. For a given function of \mathbb{B} and probability 1 the worst-case probabilistic time complexity is $O_1(\infty)$ for RS may virtually never find a solution even if there is one.

On \mathbb{B} the asymptotic efficiencies are:

- Worst case if $f(x) \neq 0, \forall x \in S$. There are $(v-1)^n$ such functions for which $E_{RS}(f, t) = 0, \forall t$.
- General case if there are k positions x so that $f(x) = 0$. There are $\binom{n}{k} (v-1)^{n-k}$ such functions for which $E_{RS}(f, t) = 1 - \left(1 - \frac{k}{n} \right)^t$.
- Average on \mathbb{B} :

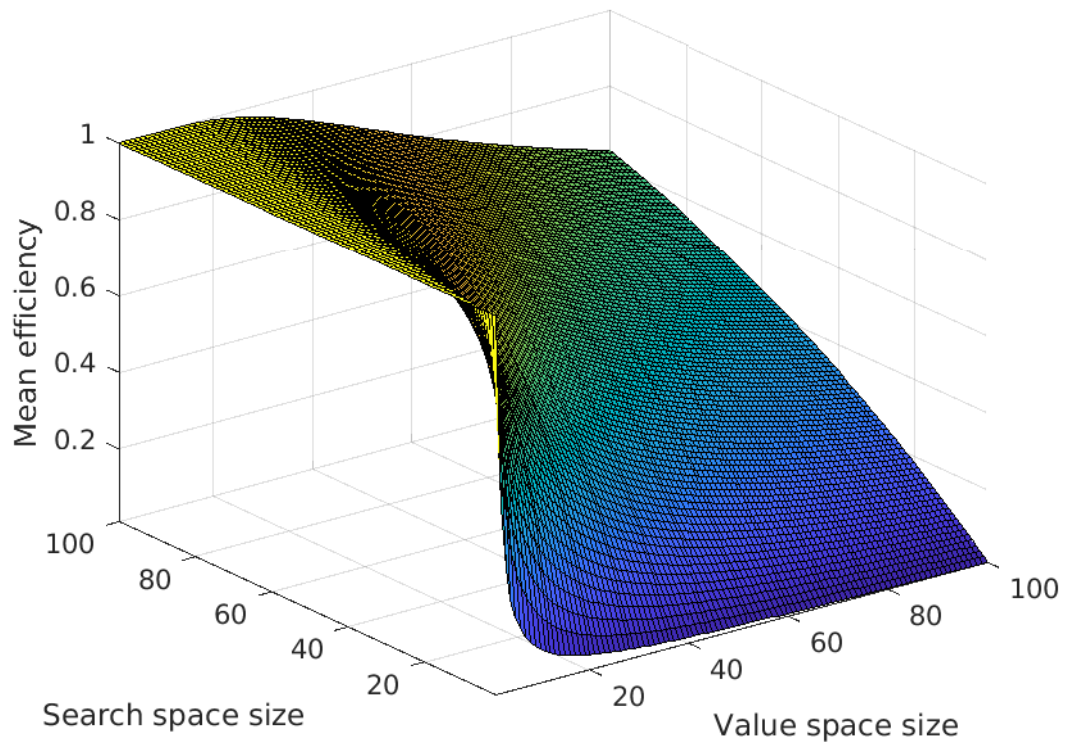
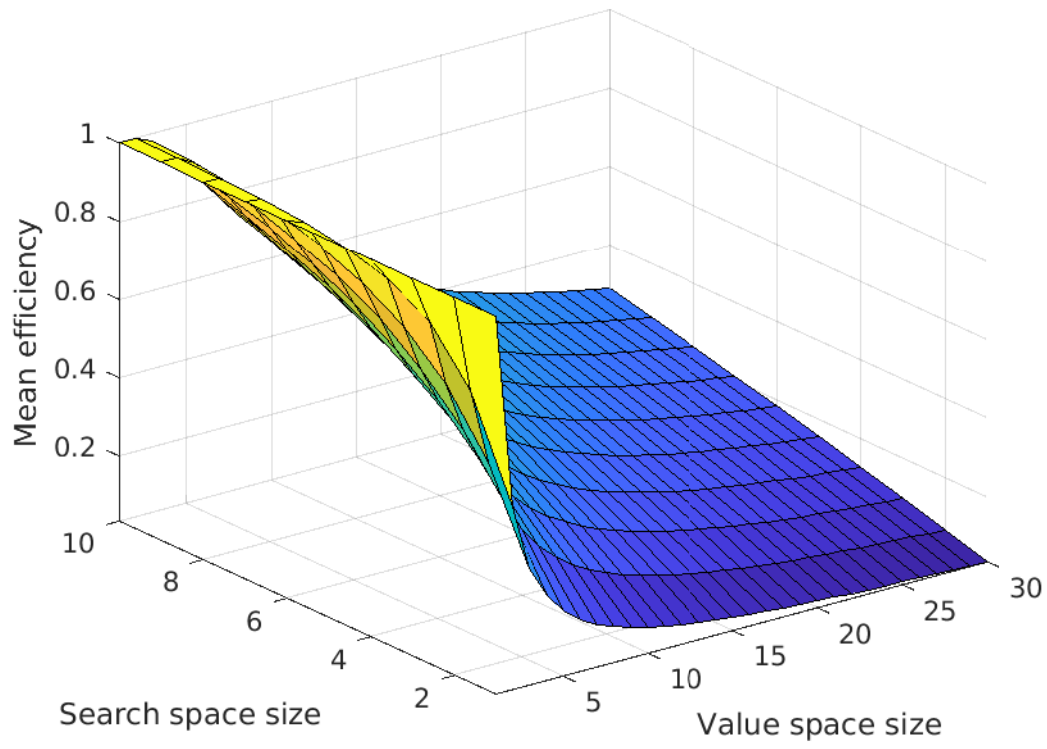


Figure 2: exhaustive search mean efficiency. Search effort = search space size.

Table 1: Time complexity and Asymptotic efficiency in average on \mathbb{B} . The input size is n .

	Time complexity	Efficiency	Probabilistic time complexity
Exhaustive search	$O(n)$	$\simeq 0.6321$	$O_1(n)$
Random search	$O(1)$	$\simeq 0.4685$	$O_1(\infty)$

Table 2: A small benchmark

		Formula	Search domain
F1	Rastrigin	$f(x) = 10D + \sum_{d=1}^D (x_d^2 - 10 \cos(2\pi x_d))$	$-5.12 \leq x_d \leq 5.12$
F2	Ackley	$f(x) = -20e^{-0.2\sqrt{\frac{\sum_{d=1}^D x_d^2}{D}}} - e^{-0.2\sqrt{\frac{\sum_{d=1}^D \cos(2\pi x_d)}{D}}} + 20 + e$	$-5 \leq x_d \leq 5$
F3	Griewank	$f(x) = 1 + \frac{1}{4000} \sum_{d=1}^D x_d^2 - \prod_{d=1}^D \cos\left(\frac{x_d}{\sqrt{d}}\right)$	$-10 \leq x_d \leq 10$
F4	Sphere	$f(x) = \sum_{d=1}^D x_d^2$	$-10 \leq x_d \leq 10$

$$E_{RS}(\mathbb{B}, t) = \frac{1}{v^n} \left(\sum_{k=1}^n \binom{n}{k} (v-1)^{n-k} \left(1 - \left(1 - \frac{k}{n} \right)^t \right) \right) \quad (3)$$

To compare with the exhaustive search we can calculate it for a search effort t equal to n . And when n and v tend to infinity the limit is then $1 - e^{\frac{1}{e}-1} \simeq 0.4685$ (see 7.3.2).

4.3 Discussion

Not surprisingly Random search is less efficient than exhaustive search but not that much: 47 % vs 63 %.

The benchmark \mathbb{B} is quite special. It is closed under permutations (c.u.p.), which means that the NFLT (No Free Lunch Theorem) is applicable[11]. Therefore all algorithms are anyway equivalent, not matter what efficiency measure is used.

But this is true only for algorithms that sample without any repetition, like Exhaustive search so not for our Random search. Actually a random search without repetition would be just a exhaustive search.

On the table 1 we can see that exhaustive Search has a higher complexity *and* a higher efficiency.

Regardless of the type of time complexity we consider, whether probabilistic or not, the most complex algorithm is also the most efficient.

$$\left\{ \begin{array}{l} A \succ B \\ A \succ B \end{array} \right.$$

5 Complexity vs Efficiency - Example 2

In practice, a benchmark is never c.u.p. (incidentally it implies that a best algorithm may exist[3]) but there is a corollary of the NFLT:

Corollary 1. When comparing two algorithms A and B , for any quality measure there exists a benchmark on which $A \prec B$ and another one on which $B \prec A$.

And we can directly derive the following other corollary:

Corollary 2. For any quality measure there exists a benchmark on which $A \succ B$ even if its time complexity is greater than the one of B

We don't even need a c.u.p. test to illustrate this corollary. Let's consider the classic small benchmark of scalable functions of the Table 2.

Now I give you two black boxes. Box 1 contains the algorithm A and box 2 the algorithm B .

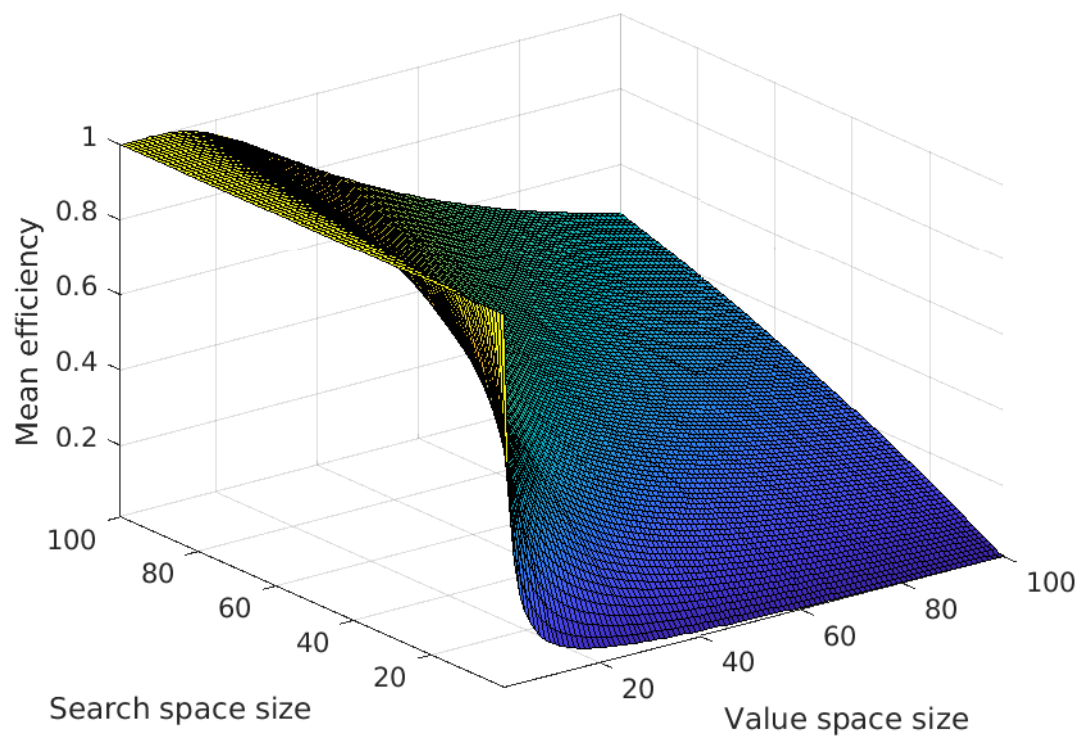
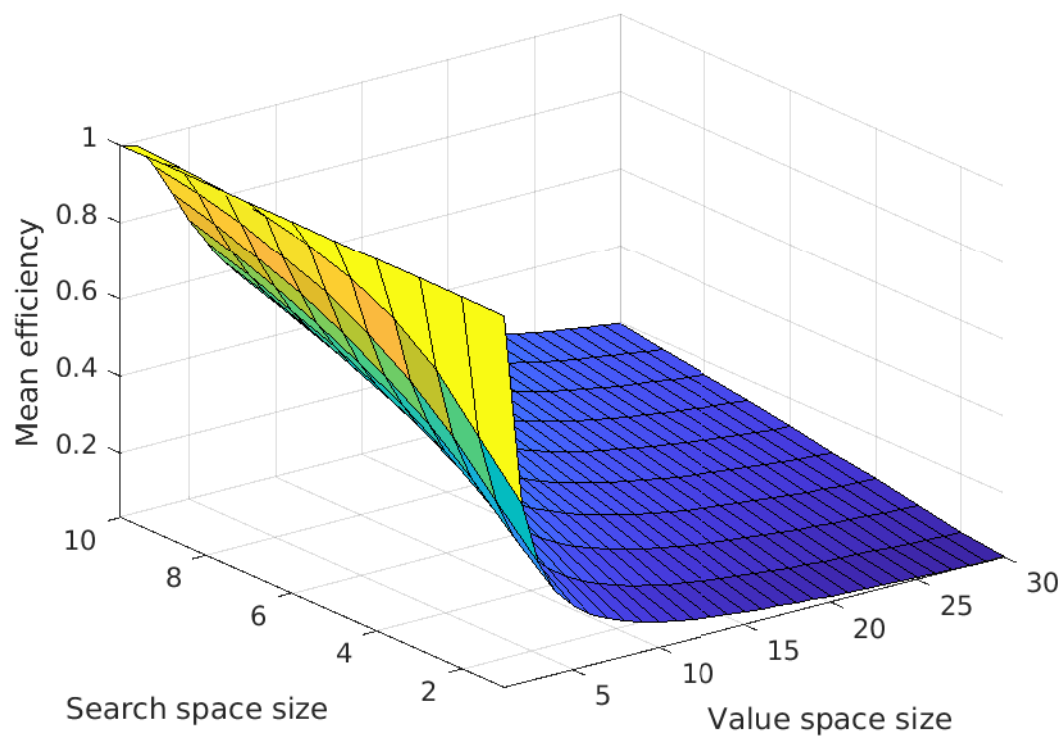


Figure 3: Random search mean efficiency. Search effort = search space size.

The stop criterion is a high number of fitness evaluation, t_{max} .

You run several times the algorithms (you don't know but they may be stochastic) and for increasing values of the dimension D . According to the results you find that B is better than A for all functions, for it always finds a smaller final value and, moreover, very quickly. So we write that B is more efficient than A : $B \succ A$.

When you open the boxes you discover that A is the uniform random search RS, whose time complexity is $O(1)$ and B the Binary search BS, whose time complexity is $O(\ln(n))$. The algorithm with the highest complexity has the best efficiency: although we have $B \succ A$ we don't have $B \prec_O A$, not even $B =_O A$.

This is, of course, because the test functions are center biased, and the three positions sampled by BS include two extremes on the frontier and one in the center. For a second experiment you slightly shift them, just replacing x_d by $x_d - \varepsilon$, with a small ε , defining a new set $(F1_\varepsilon, F2_\varepsilon, F3_\varepsilon, F4_\varepsilon)$. On $F4_\delta$ BS is still better than RS if ε is small enough but not on the three other (it never finds the solution as RS finds it, sooner or later). So now we have $A \succ B$ (three times over four) and also have $A \prec_O B$.

Two different benchmarks and two opposite conclusions².

6 In short

Assessing the time complexity of an algorithm through the analysis of how it works can be useful at times. In particular, it can highlight simplifications that do not compromise the quality of the results.

However such analyses do not allow us to assert that, inherently, algorithm A is better than algorithm B . Indeed, the observed result comparisons are only valid for specific problems. There are inevitably other problems for which, conversely, B is better than A .

That is why in an interesting long survey ([1]), and along several references about the relationships between benchmarking and the NFLT, the authors write (p. 12):

The problem instances should therefore include the difficulties that are typical of real world instances of the problem class under investigation.

Only under this condition you can possibly claim that A is better than B . And if, furthermore, its time complexity is lower, it becomes even more interesting. However, the challenge lies precisely in satisfying it. We then enter the vast domain of representative test sets, and various works on the subject clearly show that there is no consensus on what "representative" means, especially when attempting to compare metaheuristics ([1, 9]).

7 Appendix

7.1 From running time to time complexity

Let's consider a search space S of N elements and the following algorithm that search the minimum of a function f :

```

 $f_{min} = \text{Inf}$ 
for  $t=1:iter_{max}$ 
    select  $x$  at random in  $S$ 
    if  $f(x) < f_{min}$ 
         $x_{min} = x$ 
         $f_{min} = f(x)$ 

```

We execute it several times and compute the mean of the execution times. It does not depend on the problem size N but is proportional to $iter_{max}$, that is $T_{mean}(iter_{max}) = C \times iter_{max}$. Dividing by $iter_{max}$, that is not an element of the algorithm, we find the time complexity $O(C) = O(1)$.

²Actually, it's not even an illustration of the corollary because the conditions of the theorem are not satisfied, since RS allows for repetitions.

7.2 Time complexity of a few algorithms

For a single-strategy algorithm (not even speaking of hybrid methods) and for certain types of scalable problems, it is possible to define a time complexity. However, it is important to keep in mind that in many cases, this makes no sense (see 3).

- CMA-ES (Covariance matrix adaptation evolution strategy) - The time complexity of the basic version is $O(D^2)$. Some variants specifically designed for large-scale problems are sub-quadratic ([10]).
- DE (Differential evolution) - In [8] the running time is $C \times D \times \text{population} \times \text{iter}_{max}$. Therefore, for a given population, the time complexity as function of the problem size is $O(D)$.
- PSO (Particle swarm optimization) - For the basic version (constant swarm size S , global best topology) the running time is mainly function of $D \times S \times \text{iter}_{max}$. Detailed evaluations for some variants can be found in [6]. Therefore the time complexity as function of the problem size is $O(D)$.

It is interesting to note (see 4) that although CMA-ES has a higher complexity than other methods, it is often more efficient in practice.

7.3 Asymptotic mean efficiency

For the reader who is not familiar with limit calculations, here are brief demonstrations of the ones used.

The set of functions is \mathbb{B} . Both the search space size and the value space size tend to infinity (quasi-continuous problem). So we can assign the same increasing value n . We apply several times the classical limit

$$\lim_{n \rightarrow \infty} \left(1 - \frac{k}{n}\right)^n = \lim_{n \rightarrow \infty} e^{n \ln(1 - \frac{k}{n})} = e^{-k}$$

and also, for the Random search, the binomial expansion:

$$\begin{aligned} (x + y)^n &= \sum_{k=0}^n \binom{n}{k} x^{n-k} y^k \\ &= x^n + \sum_{k=1}^n \binom{n}{k} x^{n-k} y^k \end{aligned}$$

7.3.1 Exhaustive search

$$\begin{aligned} &\frac{n-1}{n^n} \left(\sum_{k=1}^n (n-1)^{k-1} n^{n-k} \right) \\ &= \frac{n^{n-1}}{n^n} \left(\sum_{k=1}^n \left(\frac{n-1}{n} \right)^{k-1} \right) \\ &= \frac{1}{n} \frac{1 - \left(\frac{n-1}{n} \right)^n}{1 - \frac{n-1}{n}} \\ &= 1 - \left(\frac{n-1}{n} \right)^n \\ &\simeq 1 - e^{-1} \end{aligned}$$

7.3.2 Random search

$$\begin{aligned}
& \frac{1}{n^n} \left(\sum_{k=1}^n \binom{n}{k} (n-1)^{n-k} \left(1 - \left(1 - \frac{k}{n} \right)^n \right) \right) \\
& \simeq \frac{1}{n^n} \left(\sum_{k=1}^n \binom{n}{k} (n-1)^{n-k} (1 - e^{-k}) \right) \\
& = \frac{1}{n^n} \left(-(n-1)^n + n^n - \sum_{k=1}^n \binom{n}{k} (n-1)^{n-k} e^{-k} \right) \\
& = 1 - \left(1 - \frac{1}{n} \right)^n - \frac{1}{n^n} \left(-(n-1)^n + \sum_{k=0}^n \binom{n}{k} (n-1)^{n-k} e^{-k} \right) \\
& \quad \simeq 1 - \frac{1}{n^n} \left(n - 1 + \frac{1}{e} \right)^n \\
& \quad = 1 - \left(1 - \frac{1}{n} \left(\frac{1}{e} - 1 \right) \right)^n \\
& \quad \simeq 1 - e^{\frac{1}{e}-1}
\end{aligned}$$

7.4 What ChatGPT says

It is well known that an AI like ChatGPT³ may be completely wrong([7]). However, the following answers seem reasonable.

Question: In optimization, is there a possible confusion between running time and time complexity?

Answer:

The confusion between running time and time complexity in optimization arises because both concepts are related to the efficiency of algorithms but refer to different aspects.

1. Running Time - Running time refers to the actual time it takes for a specific instance of an algorithm to execute on a particular input. - It is influenced by various factors, including the hardware, compiler optimizations, and specific characteristics of the input data. - Running time can vary from one execution to another, even for the same input, due to external factors.
2. Time Complexity - Time complexity, on the other hand, is a theoretical measure that expresses the amount of time an algorithm takes to complete as a function of the input size. - It focuses on the growth rate of the algorithm's running time as the input size increases. - It provides an abstract representation of the algorithm's efficiency and is used for analyzing and comparing algorithms independently of the specific machine or input.

The confusion often arises because people might use the terms interchangeably or not make a clear distinction between them. However, it's important to recognize that running time is specific to an instance of an algorithm on a particular machine with a specific input, while time complexity is a broader and more abstract measure that characterizes the algorithm's efficiency in general.

In optimization, the goal is to improve the efficiency of algorithms, and discussions may involve considerations of both running time and time complexity. Optimizing an algorithm's running time may involve improving the algorithm itself or making use of hardware-specific optimizations. Optimizing time complexity usually involves finding algorithmic improvements that reduce the growth rate of running time as the input size increases.

It's crucial to be clear about the context in which these terms are used to avoid confusion and accurately communicate the efficiency improvements achieved through optimization efforts.

Question: In optimization what is the time complexity of pure random search as function of the search space size?

Answer

³<http://chat.openai.com>

Pure random search, where solutions are generated randomly without any specific strategy or guidance, has a constant time complexity. This is because each iteration of the search involves generating a random solution, and the time it takes to generate a random solution is not dependent on the size of the search space.

In Big O notation, the time complexity of pure random search is $O(1)$, indicating constant time complexity. Regardless of the size of the search space, the time required for each iteration remains roughly the same.

Question: Is it possible for two optimization algorithms, A and B, to exist where A is more efficient than B despite having a higher time complexity?

Answer

Yes, it is possible for two optimization algorithms, A and B, to exist where A is more efficient than B despite having a higher time complexity. Time complexity is just one factor to consider when comparing the efficiency of algorithms, and it doesn't always provide a complete picture of their performance in real-world scenarios.

There are several reasons why Algorithm A might be more efficient than Algorithm B despite having a higher time complexity:

1. **Hidden Constants:** The big-O notation used to express time complexity doesn't capture the constants that may be hidden in the notation. Even if the time complexity of A is higher, it may have a smaller hidden constant, making it faster in practice for certain input sizes.
2. **Algorithm Design:** The overall design and implementation of an algorithm can significantly impact its performance. A well-designed algorithm with a higher time complexity might still outperform a poorly designed one with a lower time complexity.
3. **Input Distribution:** The behavior of algorithms can vary based on the characteristics of the input data. An algorithm may have a higher time complexity on average but perform exceptionally well on the specific types of inputs it is designed to handle.
4. **Parallelization:** Some algorithms can take advantage of parallel processing or multi-core architectures, which may not be reflected in their time complexity. In certain situations, Algorithm A might benefit more from parallelization, making it faster in practice.
5. **Memory Usage:** Algorithms with lower time complexity may require more memory, and in certain situations, the trade-off between time complexity and space complexity can influence overall efficiency.

When comparing algorithms, it's essential to consider various factors, including time complexity, space complexity, constants hidden in big-O notation, input characteristics, and practical implementation details. Benchmarking and empirical testing on specific use cases or datasets are often necessary to determine the actual efficiency of an algorithm in a given context.

References

- [1] Thomas Bartz-Beielstein, Carola Doerr, Daan van den Berg, Jakob Bossek, Sowmya Chandrasekaran, Tome Eftimov, Andreas Fischbach, Pascal Kerschke, William La Cava, Manuel Lopez-Ibanez, Katherine M. Malan, Jason H. Moore, Boris Naujoks, Patryk Orzechowski, Vanessa Volz, Markus Wagner, and Thomas Weise. Benchmarking in Optimization: Best Practice and Open Issues, December 2020. arXiv:2007.03488 [cs, math, stat].
- [2] N. G. Bruijn. *Asymptotic Methods in Analysis*. 2010.
- [3] Maurice Clerc. Iterative optimisation - When a best algorithm does exist and other frustrating theorems. May 2021.

- [4] Maurice Clerc and Dattasharma, Abhi. Comparisons in optimisation: the Eff-Res approach. *International Journal of Swarm Intelligence*, 1(3):266–298, 2014.
- [5] Yan Huang, Jian-ping Li, and Peng Wang. Unusual phenomenon of optimizing the Griewank function with the increase of dimension. *Frontiers of Information Technology & Electronic Engineering*, 20(10):1344–1360, October 2019. Number: 10 Publisher: Higher Education Press.
- [6] Naveed Iqbal, Azzedine Zerguine, and N. Al-Dhahir. Decision Feedback Equalization using Particle Swarm Optimization. *Signal Processing*, 108:1–12, March 2015.
- [7] Ryan Morisson. ChatGPT wrong over half the time on software questions - Tech Monitor, August 2023. Section: AI and automation.
- [8] Karol R. Opara and Jarosław Arabas. Differential Evolution: A survey of theoretical analyses. *Swarm and Evolutionary Computation*, 44:546–558, February 2019.
- [9] Adam P. Piotrowski, Jarosław J. Napiorkowski, and Agnieszka E. Piotrowska. Choice of benchmark optimization problems does matter. *Swarm and Evolutionary Computation*, page 101378, August 2023.
- [10] Konstantinos Varelas, Anne Auger, Dimo Brockhoff, Nikolaus Hansen, Ouassim Ait ElHara, Yann Semet, Rami Kassab, and Frédéric Barbaresco. A Comparative Study of Large-Scale Variants of CMA-ES. In Anne Auger, Carlos M. Fonseca, Nuno Lourenço, Penousal Machado, Luís Paquete, and Darrell Whitley, editors, *Parallel Problem Solving from Nature – PPSN XV*, Lecture Notes in Computer Science, pages 3–15, Cham, 2018. Springer International Publishing.
- [11] David H. Wolpert and William G. Macready. No Free Lunch Theorems for Optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.