Finding and Proving the Optimum: Cooperative Stochastic and Deterministic Search

Jean-Marc Alliot[1] and **Nicolas Durand**[1,2] and **David Gianazza**[1,2] and **Jean-Baptiste Gotteland**[1,2]

Abstract. In this article, we introduce a global cooperative approach between an Interval Branch and Bound Algorithm and an Evolutionary Algorithm, that takes advantage of both methods to optimize a function for which an inclusion function can be expressed. The Branch and Bound algorithm deletes whole blocks of the search space whereas the Evolutionary Algorithm looks for the optimum in the remaining space and sends to the IBBA the best evaluation found in order to improve its Bound. The two algorithms run independently and update common information through shared memory.

The cooperative algorithm prevents premature and local convergence of the evolutionary algorithm, while speeding up the convergence of the branch and bound algorithm. Moreover, the result found is the **proved** global optimum.

In part 1, a short background is introduced. Part 2.1 describes the basic Interval Branch and Bound Algorithm and part 2.2 the Evolutionary Algorithm. Part 3 introduces the cooperative algorithm and part 4 gives the results of the algorithms on benchmark functions. The last part concludes and gives suggestions of avenues of further research.

1 Background

Evolutionary Algorithms (EAs) appeared in the 60s with Holland [7] and became popular in the late 80s with Goldberg [4]. They can be very efficient to solve large dimension problems but are difficult to handle (many parameters need to be chosen and are very often problem dependant). They often get trapped in local optima (premature convergence).

Interval Branch and Bound Algorithms (IBBAs) were first introduced by Hansen [5] in the 90s and combined interval analysis with a Branch and Bound algorithm to reduce the size of the domain containing the optimum. They are able to prove the optimality of the solution but can rarely handle large dimension problems.

According to Alander [1] who studied the bibliography on genetic algorithms from the 50s to 93, very few articles were related to intervals and none of them dealt with IBBA-EA cooperation. In [9], Jourdan, Basseur and Talbi proposed in 2009 a taxonomy of exact methods and metaheuristics hybridizations. It appears that most of the hybridization between metaheuristics and exact methods concern discrete or combinatorial optimization.

IBBA and EA hybridizations were introduced by Sotiropoulos [21] in 1997 and used by Zhang [25] in 2007. Both approaches are integrative combinations, as described by Puchinger and Raidl [19]. In Sotropoulos' article, the first step of the algorithm uses a branch and bound to reduce the size of domain to a list of boxes (with a size smaller than ϵ). Then a genetic algorithm initializes its population in every box and updates the upper bound of the minimum searched. A shrinking box is used to improve the lower bound of the minimum searched. A new population is generated after updating the bounds and the corresponding box list. Zhang incorporates a genetic algorithm in the Interval Branch and Bound algorithm to improve the bounds and the remaining intervals list order.

Our approach is different as the IBBA and the EA cooperate but run independently. They share and update common information that helps both of them to accelerate their convergence.

[1] Institut de Recherche en Informatique de Toulouse, name.surname@irit.fr
[2] Laboratoire "Mathématiques Appliqués et Informatique" de l'ENAC

2 Standard algorithms

2.1 Interval branch and bound

The Interval Branch and Bound Algorithm (IBBA) is basically a Branch and Bound algorithm operating in a search space of intervals. It requires to re-code the function using interval arithmetic [16].

Let us consider $I = \{[a,b] | a \leq b, (a,b) \in \mathbb{R}^2\}$ the set of compact intervals in $\mathbb{R}$, and $\boldsymbol{I}(\mathbb{R})^n$ the set of n-dimensional interval vectors (or *boxes*). The basic operations of interval arithmetic are defined as follows:

$$[a,b] + [c,d] = [a+c, b+d] \tag{1a}$$
$$[a,b] - [c,d] = [a-d, b-c] \tag{1b}$$
$$[a,b] * [c,d] = [min\{ac,ad,bc,bd\}, max\{ac,ad,bc,bd\}] \tag{1c}$$
$$[a,b]/[c,d] = [a,b].[1/d, 1/c] \text{ if } 0 \notin [c,d] \tag{1d}$$

The usual real-valued functions ($\cos$, $\sin$, $\log$, and so on) can also be extended to interval arithmetic. There are now a large number of interval arithmetic implementations with various bindings to different languages, such as MPFI [20] for C, C++, or SUN interval arithmetic implementation for Fortran 95 or C++ [13, 14].

In the rest of this document, we shall denote $\boldsymbol{x} = (x_1, \ldots, x_n)$ the real vectors, and $\boldsymbol{X} = (X_1, \ldots, X_n)$ the interval vectors (boxes). An interval function $F : \boldsymbol{I}^n \to \boldsymbol{I}$ is said to be an *interval extension* of the real-valued function $f : \mathbb{R}^n \to \mathbb{R}$ if $f(\boldsymbol{x}) \in F(\boldsymbol{X})$ whenever $\boldsymbol{x} \in \boldsymbol{X}$. An interval function F is said to be *inclusion monotonic* if $\boldsymbol{X} \subset \boldsymbol{Y}$ implies $F(\boldsymbol{X}) \subset F(\boldsymbol{Y})$.

A result due to Moore ([16], [17]) states that if F is an *inclusion monotonic interval extension* of f (or more shortly, an *inclusion function*), then $F(\boldsymbol{X})$ contains the range of $f(\boldsymbol{x})$, for all $\boldsymbol{x} \in \boldsymbol{X}$.

Interval methods for solving optimization problems rely on the above result and the use of deterministic *branch and bound* techniques to find the optima of f. An initial domain $\boldsymbol{X_0}$ is split into smaller boxes (branching) evaluated using the inclusion function F (bounding). The subregions that do not contain a global minimizer of f are discarded. The basic *cut-off* test allowing the algorithm to

Algorithm 1 Interval branch and bound (maximization)

1: $Q \leftarrow \boldsymbol{X_0}$
2: **while** Q not empty **do**
3: Extract $\boldsymbol{X}$ with highest priority p_X from Q
4: **if** $upperbound(F(\boldsymbol{X})) < f_{best}$ **or** $width(\boldsymbol{X}) \leq \epsilon_x$ **or** $width(F(\boldsymbol{X})) \leq \epsilon_f$ **then**
5: Go to step 3
6: **else**
7: Split X in two sub-boxes $\boldsymbol{X_1}$ and $\boldsymbol{X_2}$
8: **for** $i \in \{1, 2\}$ **do**
9: $e_i \leftarrow f(midpoint(\boldsymbol{X_i}))$
10: **if** $e_i > f_{best}$ **then**
11: $f_{best} \leftarrow e_i$
12: $\boldsymbol{X_{best}} \leftarrow \boldsymbol{X_i}$
13: **end if**
14: Insert $\boldsymbol{X_i}$ into Q, with priority e_i
15: **end for**
16: **end if**
17: **end while**
18: return $(\boldsymbol{X_{best}}, f_{best})$

discard a subregion $\boldsymbol{X}$ consists in comparing the bounds of $F(\boldsymbol{X})$ to the best estimator of the optimum found so far. Boxes that are not discarded are divided again in smaller boxes until the desired precision for $F(\boldsymbol{X})$ is reached (or when $\boldsymbol{X}$ becomes too small). Valid boxes are inserted in a priority queue, which is sorted according to an estimator of the quality of the box. In this simple Branch-and-Bound algorithm, the estimator is just the image of the midpoint of the box. The search stops when the whole domain has been explored (the priority queue is empty).

Several refinements of this basic algorithm exist: monotonicity test when the derivatives of f are available[3], concavity test, local search procedures for enhancing the best estimator, etc. These procedures may (or may not) be efficient, depending on the problem. In order to keep this article as simple and clear as possible, we opted to use the basic algorithm described above, discarding all these refinements. The interval branch and bound algorithm (IBBA) is described in algorithm 1

2.2 Evolutionary algorithm

Evolutionary algorithms, popularized by David Goldberg ([4]) and Michalewicz [12], are inspired by Darwin's theory of evolution. A population of individuals (points of the search space) is selected according to its fitness, and recombined using crossover and mutation operators. The process is repeated until a termination criterion is met, as described in algorithm 2.

Several refinements have been introduced in this evolution scheme, (among others *elitism*, *scaling*, *sharing*). The reader may refer to [3] for a description of genetic algorithms and other evolutionary algorithms also inspired from evolutionary theory.

In this article, we have used a real-coded genetic algorithm, where the population is made of N real values vectors. The population is randomly initialized, with uniform probability, within the bounds of the search space. Before selecting the pool of parents, a *sigma truncation* [4] scaling is applied to the fitness values, followed by a *clusterized sharing* (step 4). The selection/reproduction itself is made using the *stochastic remainder without replacement* [4] principle (step 5).

[3] This can be done by hand for simple functions. or using automatic differentiation [2] for complex programs.

Algorithm 2 Evolutionary algorithm (EA)

1: Initialize population
2: **while** termination criterion is not met **do**
3: Evaluate *raw fitness* of population elements
4: Apply *scaling* and *sharing* operations on *raw fitness*
5: Create new population according to *new fitness* criterion
6: Replace some elements by *mutation* and *crossover*
7: **end while**
8: Return best elements of population

The crossover and mutation operators are then applied with respective probabilities P_c and P_m ($P_c + P_m < 1$) to the pool of parents, as follows:

- crossover: two different elements p_1 and p_2 are randomly drawn from the parents' pool and recombined into two children using an arithmetic crossover. Each child is defined by $\alpha p_1 + (1 - \alpha)p_2$ where α is a real value randomly chosen in a given interval. The process is repeated $\lfloor \frac{N.P_c}{2} \rfloor$ times to create $\lfloor N.P_c \rfloor$ children.
- mutation: $\lfloor N.P_m \rfloor$ elements are drawn from the pool of parents. For each drawn vector, a number k of values is randomly selected, and a Gaussian noise is added to the selected values, thus providing the mutated vector. Assuming the vectors are of dimension n, k is randomly chosen so that $k \leq n$. This creates $\lfloor N.P_m \rfloor$ children.

At the end of the crossover/mutation process, the parents are replaced by their respective children and the new generation of N population elements replaces the previous one. The process is repeated until a termination criterion – maximum time here – is met.

We could have chosen other evolutionary algorithms such as Particule Swarm Optimization [10], Differential Evolution [23] or CMA-ES [6]. These algorithms might (or might not) have been more efficient than a real-coded EA. However, the goal of this article is not to find the fastest or most efficient algorithm, but to show how the two approaches (stochastic and deterministic) cooperate. We therefore chose the algorithm we were the most comfortable with.

3 Parallel cooperative algorithm

When hybridizing the genetic and interval branch and bound algorithms, we adopted the following cooperation scheme. The two algorithms run in parallel. Shared memory is used to exchange information between the two programs. A third thread is used to perform some common operations on elements of both threads.

3.1 IBBA thread

The Branch and bound thread is very similar to the Branch and bound algorithm described in section 2.1. The thread is described in algorithm 3.

The main differences between the IBBA algorithm and IBBA thread of the cooperative algorithm are outlined below:

- Shared memory is used to retrieve the best evaluation found by the evolutionary algorithm (step 4). This best evaluation is used to update the bounding value of the IBBA thread, thus speeding up the process of cutting intervals.

Algorithm 3 Cooperative algorithm, IBBA thread

1: $Q \leftarrow \boldsymbol{X_0}$
2: **while** Q not empty **do**
3: Synchronization point for UPDATE thread
4: $f_{bestag} \leftarrow GetFromSharedMem(f_{bestag})$
5: $f_{best} \leftarrow max(f_{best}, f_{bestag})$
6: Extract $\boldsymbol{X}$ with best priority p_X from Q
7: **if** $upperbound(F(\boldsymbol{X})) < f_{best}$ **or** $width(\boldsymbol{X}) \leq \epsilon_x$ **or** $width(F(\boldsymbol{X})) \leq \epsilon_f$ **then**
8: Go to step 6
9: **else**
10: Split X in two sub-boxes $\boldsymbol{X_1}$ and $\boldsymbol{X_2}$
11: **for** $i \in \{1, 2\}$ **do**
12: $e_i \leftarrow f(midpoint(\boldsymbol{X_i}))$
13: **if** $e_i > f_{best}$ **then**
14: $f_{best} \leftarrow e_i$
15: $bestbb \leftarrow midpoint(\boldsymbol{X_i})$
16: $\boldsymbol{X_{bestbb}} \leftarrow \boldsymbol{X_i}$
17: $PutToSharedMem(bestbb)$
18: **end if**
19: Insert $\boldsymbol{X_i}$ into Q, with priority e_i
20: **end for**
21: **end if**
22: **end while**
23: Signal EA thread and stop

- When the IBBA thread finds a better overall element, it updates the shared memory, and makes this element available for the EA thread (step 17).
- When the IBBA thread ends, we are sure that we have found a global optimum and the IBBA thread sends a signal to the EA thread and then terminates (step 23).

Other operations are performed on the priority queue of the IBBA thread by the UPDATE thread at the synchronization point. They are described in section 3.3.

3.2 EA thread

The evolutionary algorithm thread is also very similar to the evolutionary algorithm described in section 2.2. This thread is described in algorithm 4.

Algorithm 4 Cooperative algorithm, EA thread

1: Initialize population
2: **while** (termination criterion not met) **or** (no signal from IBBA thread) **do**
3: Synchronization point for UPDATE thread
4: Evaluate *raw fitness* of population elements
5: $PutToSharedMem(f_{bestag})$
6: $bestbb \leftarrow GetFromSharedMem(bestbb)$
7: Replace worst population element by $bestbb$
8: Evaluate $bestbb$ raw fitness
9: Apply *scaling* and *sharing* operations on *raw fitness*
10: Create new population according to *new fitness* criterion
11: Replace some elements by *mutation* and *crossover*
12: **end while**
13: Return best element of population

The main differences are outlined below:

- The EA thread puts in shared memory the best evaluation found so far (step 5), which will be retrieved by the IBBA thread.
- The EA thread gets from the shared memory the best element found so far by the IBBA thread (step 6) and then replaces its worst population element by this element.

Other operations are performed by the UPDATE thread on the EA population at the synchronization point (step 3). These operations are described in section 3.3.

3.3 UPDATE thread

The UPDATE thread is triggered every t seconds. It is described in algorithm 5.

Algorithm 5 Cooperative algorithm, UPDATE thread

1: **loop**
2: Sleep for duration t
3: Wait for and then Suspend EA thread and IBBA thread
4: **for** $i = 1$ **to** N **do**
5: $d_{min} \leftarrow +\infty$
6: $NQ \leftarrow Q$
7: **while** NQ not empty **and** $d_{min} \neq 0$ **do**
8: Extract $(\boldsymbol{X}, p_X)$ from NQ
9: **if** $upperbound(F(\boldsymbol{X})) < f_{best}$ **then**
10: Suppress $\boldsymbol{X}$ from Q
11: **else**
12: **if** $elt(i) \in \boldsymbol{X}$ **then**
13: $d_{min} \leftarrow 0$
14: **else**
15: **if** $distance(elt(i), \boldsymbol{X}) < d_{min}$ **then**
16: $d_{min} \leftarrow distance(elt(i), \boldsymbol{X})$
17: $\boldsymbol{X_c} \leftarrow \boldsymbol{X}$
18: **end if**
19: **end if**
20: **end if**
21: **end while**
22: **if** $d_{min} = 0$ **then**
23: **if** $p_X < f(elt(i))$ **then**
24: Reinsert $\boldsymbol{X}$ with new priority $f(elt(i))$ in Q
25: **end if**
26: **else**
27: $elt(i) \leftarrow Project(elt(i), X_c)$
28: **end if**
29: **end for**
30: Resume EA thread and IBBA thread
31: **end loop**

The thread first waits for the IBBA and the EA thread to reach their synchronization point, and suspends them before performing any operation.

The thread then examines in turn the N elements of the population of the EA thread. For each element $elt(i)$, it performs a lookup in the priority queue Q of the IBBA thread. This queue contains all the interval vectors (boxes) of search space that are still valid. For each element $elt(i)$, the thread finds the minimal distance d_{min} of this element to the closest box $\boldsymbol{X_c}$ in queue Q (in the process the thread also suppresses from Q boxes whose upper-bounds are lower than the current best evaluation in step 10). Then:

- if d_{min} is equal to zero, then we have found a box $\boldsymbol{X}$ that contains $elt(i)$ and $elt(i)$ is in an admissible zone of search space. Thus

$elt(i)$ is kept inside the EA population. If $f(elt(i))$ is better than the current priority $p_{\boldsymbol{X}}$ of box $\boldsymbol{X}$ that contains $elt(i)$ then we have found a better estimator for the maximum in box $\boldsymbol{X}$, and the priority of box $\boldsymbol{X}$ in queue Q is updated to $f(elt(i))$.

- if d_{min} is not zero then $elt(i)$ is outside the admissible search space. Then we project $elt(i)$ on the closest box $\boldsymbol{X_c}$ and replace in the EA population $elt(i)$ by this projection.

The projection algorithm is simple and described in algorithm 6.

Algorithm 6 Projection algorithm (step 27 of algorithm 5))

```
1: for j = 1 to n do
2:   if elt(i)(j) ∉ X_c(j) then
3:     if upperbound(X_c(j)) < elt(i)(j) then
4:       elt(i)(j) ← upperbound(X_c(j))
5:     else
6:       elt(i)(j) ← lowerbound(X_c(j))
7:     end if
8:   end if
9: end for
```

$elt(i)$ is a real vector in $\mathbb{R}^n$, while $\boldsymbol{X_c}$ is an interval real vector in $I(\mathbb{R})^n$. For each dimension j we check if $elt(i)(j)$ is inside interval $\boldsymbol{X_c}(j)$. If $elt(i)(j)$ is not inside the interval then we replace $elt(i)(j)$ by the closest element of interval $\boldsymbol{X_c}(j)$, which is either the upper bound or the lower bound of $\boldsymbol{X_c}(j)$.

The UPDATE thread has two main goals:

1. Put all the population elements of the EA thread back into the admissible search space. This will increase the speed of convergence of the EA, and will also take the EA out of local minima as soon as these minima have been ruled out by the IBBA thread. In fact, on some examples developed in section 4 we will see that even the best element of the EA thread can be suppressed and projected elsewhere by the UPDATE thread when this element is inside a local optimum.
2. Re-sort the IBBA priority queue, thus focusing the search in the IBBA thread on the "interesting" part of the search space, and increasing the IBBA convergence speed.

The UPDATE thread is a costly one, especially when there are many boxes in the priority queue Q. Thus, it should not be triggered too often, but often enough to fulfil its two goals. For simplicity's sake, we have only presented here a simple strategy (timer interval) for triggering this thread. But other, more efficient strategies can be used to trigger it, based on the size of the priority queue, the evolution of the population in the EA thread. Moreover, some implementation tricks can be used to accelerate it. However, again for simplicity's sake, we present in the following results the simple, basic algorithm.

3.4 Understanding the algorithm

In this section we are going to graphically present a few examples in order to understand how the cooperative algorithm works. Statistical tests and results will be presented in section 4.

We will first consider the Griewank function in dimension 6. Griewank is a classical example [18], even if not a very good one regarding global optimization, as Locatelli has shown in [11] that the function becomes easier to optimize for large dimensions with stochastic algorithms. Moreover the Griewank function is partially separable, which makes convergence of both EA and IBBA algorithms extremely fast.

It is now customary to use a variant of the Griewank function, the rotated Griewank function [22].

$$f(x) = \sum_{i=1}^{D} \frac{z_i}{4000} - \prod_{i=1}^{D} \cos(\frac{z_i}{\sqrt{i}}) + 1 \text{ with } z = M(x - o)$$

where M is a random rotation matrix and o a random vector. To have results easy to read we maximize here the function $g(x) = \frac{1}{1+f(x)}$

The rotated Griewank function is not separable . The non-separability of the variables turns the inclusion function of the IBBA, which is very efficient for the regular Griewank function, into a very inefficient one. It is currently impossible to find the optimum of the R-Griewank function with a simple IBBA algorithm as soon as D is larger than 7. Thus 6 is a good value to see how the cooperative algorithm works, and to compare the convergence of all three algorithms.

On Figure 1, we first compare the cooperative algorithm with the standard Evolutionary Algorithm and with the Branch and Bound algorithm[4]. These results are only an indication of the general behaviour of the cooperative algorithm, and statistical results will be presented in the next section. It is already clear however that the cooperative algorithm is much faster than both the EA and the IBBA algorithms, while **proving** the result, as the IBBA does.

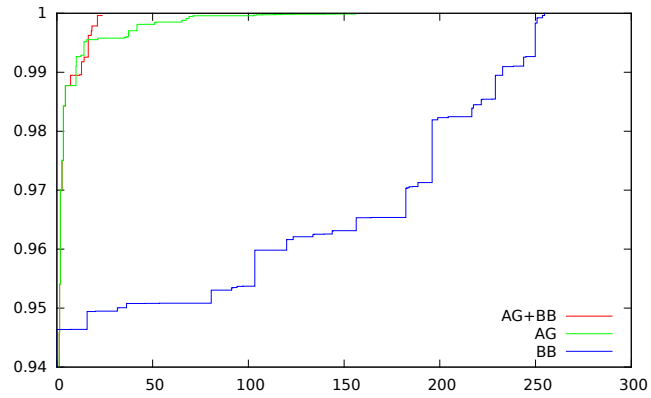

Figure 1. Comparison Cooperative/AG/BB (6 variable Griewank)

On Figure 2, we see how the cooperative algorithm finds and proves the optimum in 25s. The red line is the value of the internal

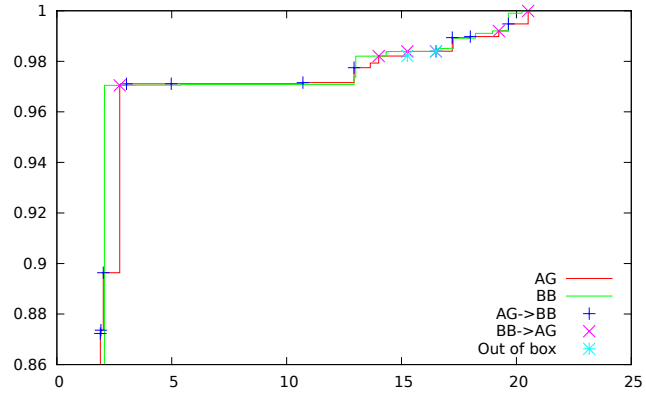

Figure 2. Understanding cooperative (6 variable Griewank)

[4] In all figures, AG+BB=cooperative algorithm, AG=evolutionary algorithm, BB=branch and bound. The x-axis is in seconds, the y-axis is the fitness.

best evaluation found by the EA thread. The green line is the internal value of the best evaluation found by the IBBA thread. Deep blue crosses are the times when the EA thread sends to the IBBA thread a better evaluation than the one that the IBBA has. The pink crosses are the times when the IBBA thread sends to the EA thread a better element than the one the EA thread has. The light blue crosses are the times when the UPDATE thread destroys the best element of the EA thread because it is outside the searchable domain (the EA thread is stuck in a local optimum). We can see on this figure that the algorithms collaborate in an extremely efficient way. All mechanisms are used during the run.

With 8 variable, the IBBA algorithm can never find a solution in a reasonable amount of time, while the cooperative algorithm can. The EA algorithm performance on the 8 variable function depends on luck as it can sometimes get stuck in a local optimum (the cooperative algorithm never does).

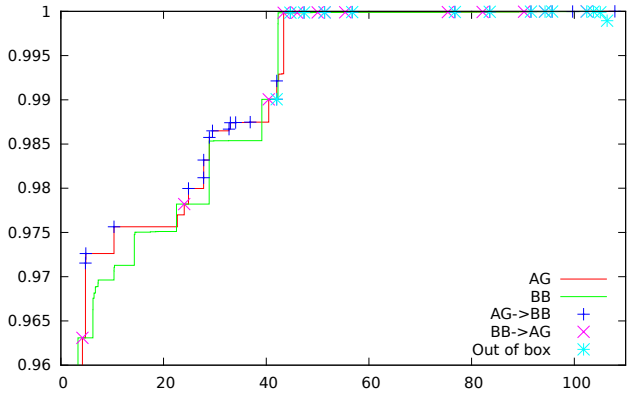

Figure 3. Understanding cooperative (8 variable Griewank)

On Figure 3, we have a detail of the convergence of the cooperative algorithm on the 8 variable rotated Griewank. We choose one of the example with the longest running time, in order to clearly see what happens here. The EA algorithm is usually dragging up the IBBA (deep blue crosses), at least at the beginning of the search. However, from 40s and up to 45s, it is the IBBA which is taking the EA algorithm out of local minima. After 45s, the algorithm is already in the vicinity of the optimum. The IBBA is more efficient than the EA in performing a local optimization (all the pink crosses at the top of the graphic). The light blue crosses at the top are simply the result of the IBBA thread "killing" search space at a very fast pace. The global optimum is found with the required precision at the last pink cross (85s). Thereafter, the algorithm is just completing the proof by searching and cutting the remaining search space (the last clear blue crosses).

As a last example we will discuss the Michalewicz function [18]:

$$f(x) = \sum_{i=1}^{D} \sin(x_i) \left(\sin(\frac{ix_i^2}{\pi}) \right)^{20}$$

This function is difficult to optimize because of the steepness of the curve (the 20-th power), and is interesting because there are very few results available for large D. In [18], the highest D for which the optimum is presented is $D = 10$, and the same goes for [15] and [8]. Of course, the optimum is never proved as it is found by stochastic algorithms. It was thus a challenge to find and prove the optimum of the Michalewicz function for $D = 12$ variables. The function optimized is $g(x) = f(x) + D$, in order to keep g positive.

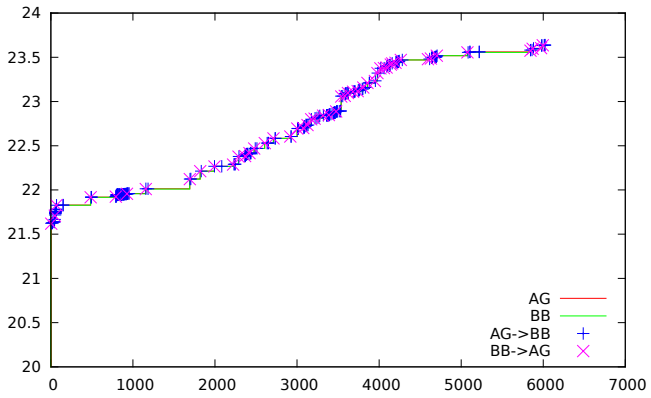

Figure 4. Understanding cooperative (12 variable Michalewicz)

On Figure 4 we see how the cooperative algorithm behaves. The cooperation is present all the way up to the optimum. On Figure 5, we have displayed the times when the IBBA thread had to kill the best element of the EA thread which was stuck in a local optimum. This happens often because the function has a lot of local optima and because the vicinity of the optimum is extremely small due to the steepness of the function.

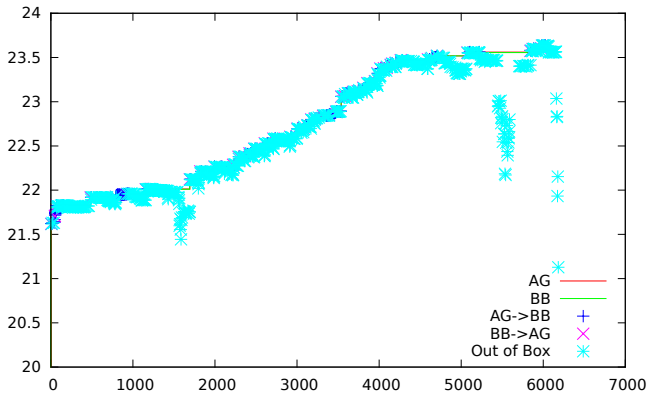

Figure 5. Killing local minima (12 variable Michalewicz)

The optimum is
$x = [2.202881, 1.570808, 1.284998, 1.923050, 1.720462, 1.570800, 1.454402, 1.756096, 1.655724, 1.570792, 1.497731, 1.696620]$
$g(x) = 23.64957$, $f(x) = 11.64957$ with $\epsilon_x = 10^{-3}$ and $\epsilon_f = 10^{-4}$.

4 Statistical tests and results

In Table 1, statistical results on the rotated Griewank function are presented on 100 runs using the EA alone, the IBBA alone, and the cooperative algorithm. The search space is $[-400, 600]^n$ with $n \in \{6, 7, 8, 9, 10\}$. The time was limited to 1800 seconds. We used a 3.40GHz Intel Xeon E3-1270.

The Evolutionnary Algorithm (described in part 2.2) parameters are: 1000 chromosomes, $P_c = 0.5$ and $P_m = 0.3$. An arithmetic crossover is used with $\alpha \in [-0.5, 1.5]$. The mutation operator adds a random noise in the $[-0.5, 0.5]$ interval to each variable of the function. We set $\sigma = 2$ for sigma truncation scaling and used the clusterized sharing described by Yin and Germay [24]. The algorithm

stops when the distance between the current best element and the optimum (1 in this case) is less than 10^{-4}, or when the alloted time (1800s) is over.

For the Interval Branch and Bound Algorithm, ϵ_x and ϵ_f (see algorithm 1) were set to 10^{-2} and 10^{-4}. The algorithm stops when the Q list is empty, or when the alloted time is over.

The same parameters and stopping criteria are used for the cooperative algorithm.

	size	6	7	8	9	10
EA	Found	100	94	92	83	15
	Mean	204	864	972	1340	1678
	Sigma	92	356	389	430	34
IBBA	Found	71	0	0	0	0
	Mean	284				
	Sigma	192				
Cooperative	Found	100	100	100	100	100
	Mean	50	62	156	215	267
	Sigma	18	47	85	317	105

Table 1. Rotated Griewank function, statistical results on 100 runs

For each algorithm, Table 1 gives the number of runs that found the optimum in less than 1800 seconds, the mean time duration, and the corresponding standard deviation in seconds.

Results show that the IBBA can only deal with small dimensions (≤ 6) in a reasonable time. The EA approach is sensitive to dimension as well. The EA would certainly give much better results if its parameters and operators were optimized for the Griewank function but we did not concentrate on this issue. The Cooperative Algorithm always gives much better results than the IBBA and EA.

5 Conclusion

In this article, we have presented a cooperative algorithm that combines the advantages of the global stochastic optimization techniques and the global deterministic techniques, and we have shown that this algorithm is able to speed up the convergence of the stochastic algorithm. But the most important result is that this algorithm is able to **prove** the optimality of the result for very difficult functions such as R-Griewank or Michalewicz, up to 12 variables, while the best available result was, as far as we know, currently limited to 6 to 8 variables.

We have also focused on presenting the algorithm as clearly as possible, using only a standard evolutionary algorithm and a standard interval branch and bound algorithm, leaving out all the acceleration, modifications and implementation tricks. The results presented are thus easily reproducible with off-the-shelf algorithms. For simplicity's sake and lack of space, we have also limited our presentation to two functions but we have run similar tests on many more functions (Rastrigin, Schwefel, etc…) with similar excellent results.

We think that this cooperative algorithm is currently the best algorithm available for proving the optimality of the result for complex and deceptive functions up to a number of variables which had, to our knowledge, never been reached.

Our next paper will present the modifications of the EA and the IBBA algorithms, along with the implementation optimizations that we have developed. These improvements tremendously speed up the cooperative algorithm and enable, for example, to find and prove the optimum of the Michalewicz function with 20 variables in less than 30 seconds on a "standard" dual core processor.

References

[1] Jarmo T. Alander, 'An indexed bibliography of genetic algorithms: Years 1957-1993', Technical report, Department of Information Technology and Production Economics, (1994).

[2] H.M. Bücker, G. Corliss, P. Hovland, U. Naumann, and B. Norris, *Automatic Differentiation: Applications, Theory, and Implementations*, Springer-Verlag, 2006. ISBN: 978-3-540-28403-1.

[3] A.E. Eiben and J.E. Smith, *Introduction to Evolutionary Computing*, Springer, 2003. ISBN : 3-540-40184-9.

[4] D. Goldberg, *Genetic Algorithms*, Addison Wesley, 1989. ISBN: 0-201-15767-5.

[5] E. Hansen, *Global optimization using interval analysis*, Dekker, New-York, 1992.

[6] N. Hansen and S. Kern, 'Evaluating the cma evolution strategy on multimodal test functions', in *Proceedings of the 8th International Conference on Parallel Problem Solving from Nature*, pp. 282–291, (2004).

[7] J.H Holland, *Adaptation in Natural and Artificial Systems*, University of Michigan press, 1975.

[8] Lhassane Idoumghar, Mohamed Melkemi, and René Schott, 'A novel hybrid evolutionary algorithm for multi-modal function optimization and engineering applications', in *Proceedings of the 13th IASTED International Conference on Artificial Intelligence and Soft Computing*, (2009).

[9] L. Jourdan, M. Basseur, and E-G Talbi, 'Hybridizing exact methods and metaheuristics: A taxonomy', *European Journal of Operational Research*, (2009).

[10] J. Kennedy and R. Eberhart, 'Particle swarm optimization', in *Proceedings of the IEEE International Conference on Neural Networks*, (1995).

[11] M. Locatelli, 'A note on the Griewank test function', *Journal of global optimization*, **25**, 169–174, (2003).

[12] Z. Michalewicz, *Genetic algorithms+data structures=evolution programs*, Springer-Verlag, 1992. ISBN: 0-387-55387-.

[13] SUN Microsystems, *C++ Interval Arithmetic programming manual*, SUN, Palo Alto, California, 2001.

[14] SUN Microsystems, *Fortran95 Interval Arithmetic programming manual*, SUN, Palo Alto, California, 2001.

[15] M. Molga and C. Smutnicki, 'Test functions for optimization needs', Technical report. http://www.zsd.ict.pwr.wroc.pl/files/docs/functions.pdf.

[16] R.E. Moore, *Interval Analysis*, Prentice Hall, NJ, 1966.

[17] R.E. Moore and Fritz Bierbaum, *Methods and applications of interval analysis*, SIAM, 1979.

[18] Hartmut Pohlheim, *Example of objective functions, documentation of the Matlab Genetic and evolutionary algorithm toolbox*, MATLAB, 2005.

[19] Jakob Puchinger and Günther R. Raidl, 'Combining metaheuristics and exact algorithms in combinatorial: A survey and classification', in *Proceedings of the International Work-conference on the Interplay between Natural and Artificial Computation*. IWINAC, (2005).

[20] N. Revol and F. Rouillier, 'Motivations for an arbitrary precision interval arithmetic and the MPFI library', *Reliable computing*, **11**(4), 275–290, (2005).

[21] D.G. Sotiropoulos, E.C. Stravopoulos, and M.N. Vrahatis, 'A new hybrid genetic algorithm for global optimization', in *Proceedings of the 2nd World Congress of Nonlinear Analysis*, (1997).

[22] P. N. Suganthan, N. Hansen, J. J. Liang, K. Deb, Y. P. Chen, A. Auger, and S. Tiwari, 'Problem definitions and evaluation criteria for the CEC 2005 special session on real-parameter optimization', KANGAL Report 2005005, (May 2005).

[23] Differential Evolution A Practical Approach to Global Optimization, *Ken Price and Rainer Storn and Jouni Lampinen*, Springer-Verlag, 2005. ISBN: 3-540-20950-6.

[24] X. Yin and N. Germay, 'A fast genetic algorithm with sharing scheme using cluster analysis methods in multimodal function optimization', in *Proceedings of the Artificial Neural Nets and Genetic Algorithm International Conference, Insbruck Austria*. Springer-Verlag, (1993).

[25] Xiaowei Zhang and Sanyang Liu, 'A new interval-genetic algorithm', in *Proceedings of the Third International Conference on Natural Computation*. ICNC, (2007).