J Glob Optim (2010) 47:247–271
DOI 10.1007/s10898-009-9472-5

A fast memoryless interval-based algorithm for global optimization

M. Sun

Received: 17 November 2008 / Accepted: 7 September 2009 / Published online: 19 September 2009

Abstract We present a global optimization algorithm of the interval type that does not require a lot of memory and treats standard constraints. The algorithm is shown to be able to find one globally optimal solution under certain conditions. It has been tested with many examples with various degrees of complexity and a large variety of dimensions ranging from 1 to 2,000 merely in a basic personal computer. The extensive numerical experiments have indicated that the algorithm would have a good chance to successfully find a good approximation of a globally optimal solution. More importantly, it finds such a solution much more quickly and using much less memory space than a conventional interval method. The new algorithm is also compared with several noninterval global optimization methods in our numerical experiments, again showing its clear superiority in most cases.

Keywords Global optimization · Interval-based algorithm · Memoryless · Constraints

1 Introduction

Many important real world problems aim at finding the globally optimal value of an objective function f(x) and at least one global optimizer over a bounded multidimensional interval domain X in R^n, possibly subject to some equality and inequality constraints. Mathematically the problem is stated as

$$\begin{array}{ll} \text{minimize} & f(x), \\ \text{subject to} & h(x) = 0,\ g(x) \leq 0,\ x \in X. \end{array} \tag{1}$$

The global problem presents a number of more difficult challenges than local optimization problems. The most difficult issue is perhaps the lack of a single verifiable sufficient condition for a globally optimal solution unless it is a very special case. Thus, either a global

M. Sun (✉)
Department of Mathematics, The University of Alabama, Tuscaloosa, AL 35487-0350, USA
e-mail: msun@gp.as.ua.edu

behavior of f(x) (e.g., Lipschitz constant [5]) is used or the entire search domain is examined by global search algorithms.

Stochastic algorithms search the whole domain only in a probabilistic fashion so that at most they can yield a good estimate of a globally optimal solution in a probabilistic sense. Thus, when such a particular search program stops after a finite number of steps, there is no reliable way to judge the quality of the estimated solution. They are often considered as heuristic. However, stochastic search methods (such as the simulated annealing methods and genetic algorithms) have been more popular choices than interval methods because of their simplicity of implementation, relative quickness for reaching an approximate solution, less memory demands, and a wider range of applicable problems. Many stochastic search methods have been designed for solving unconstrained problems. Under the presence of constraints, their performance deteriorates further and there are even fewer theoretical justifications.

Deterministic algorithms offer attractive alternatives for solving problem (1). They are generally based on the idea of branch and bound [15]. Among them, interval methods offer both sound theoretical foundation and reliable numerical solutions [20]. Under the framework of interval branch and bound, a number of advantages are well known. (1) It guarantees convergence to all global solutions under fairly weak assumptions. (2) It offers reliable stopping criteria so that the algorithm does not have to run longer than necessary. (3) It is numerically robust and handles round-off errors conveniently and effectively. (4) It handles constraints with relative ease and without jeopardizing theoretical justifications. Despite such attractive features of the interval method, most published reports on their applications seem to be generally limited to optimization problems in low dimensions (say, much less than 100 according to our recent survey of literature). Obviously, there are three major concerns in solving large dimensional problems: large amount of memory space, slow speed of convergence, and requirement of acceptable bounds of the objective function over any interval subdomains. The last of the three is generally not specific to the dimension of problem (1). It is rather tied to the nature of the problem itself. Thus our new algorithm aims at easing the first two concerns only. Although interval algorithms can converge exponentially to the globally optimal objective function value [6], the number of subboxes to be saved and processed in an interval method could also increase exponentially with the dimension of the search domain. That raises memory requirement and slows down convergence. This could prevent any conventional interval method from becoming a practical choice for solving many large scale optimization problems. If the memory problem can be significantly alleviated, speed of convergence would be greatly improved. Consequently, interval methods would become more attractive than noninterval methods at least for the optimization problems where the function bounds are available.

Inspired by such observations, we have investigated some new strategies associated with the interval branch and bound methodology both theoretically and numerically. This paper reports one new version of the interval-based algorithm that shows improvement both in memory space usage and in overall speed of convergence. It is in fact essentially memoryless and yet still converges to a globally optimal solution in many cases. When it converges, it does so much more quickly than the standard interval method.

The rest of the paper is organized as follows. In Sect. 2, we review major features of the standard interval method. Our new algorithm is presented in Sect. 3 along with theoretical convergence results. Numerical testing results for a relatively large pool of examples are given in Sect. 4, followed by final comments and conclusions in Sect. 5. Finally 15 repeatedly used examples with variable dimensions are listed in the appendix.

2 Standard interval method

The standard branch and bound method was originally introduced in [8] and [14], and more recently presented in [15]. Its main idea is the recursive refinement of partition of the search domain and underestimation (providing guaranteed lower bound) of f(x) over the partitioned subdomains. Interval methods (e.g., [1,12,21,22,25]) are in the general framework of branch and bound along with interval arithmetic. The interval arithmetic provides an effective means of underestimation of programmable functions, and offers an additional benefit of including roundoff errors. Following the initial works in late 1950s and early 1960s, research on interval methods became a more heated topic from late 1970s to early 1990s among many researchers in several fields. During that period of time, computers were becoming increasingly more popular and more powerful. Improved computer programming languages also helped promotion of interval methods. A solid foundation had been laid by the end of 1980s. Subsequent improvements were done since 1990s (e.g., [4,5,17,24,29,30,33]). Our list of references is not meant to be complete.

Let f^* be the global minimum value of the objective function f(x), x^* a global minimizer in X, and X^* the set of all the global solutions. As in the interval analysis literature, we use boxes and intervals interchangeably. A typical interval method uses 2 major objects, a list L that holds all the subintervals of partitions that remain to be processed, and an inclusion function F(Y) that offers a lower bound and an upper bound of f(x) over any box Y to be processed. We write

$$F(Y) = [\mathrm{Lb}(F(Y)), \mathrm{Ub}\,(F(Y))]\,.$$

Later in the next section we also use

$$\mathrm{Inf}\,(f(Y)) = \mathrm{Inf}\,\{f(x) :\ x \in Y\}\,.$$

The general procedure would consist of these major steps.

Algorithm 1 (Standard interval algorithm for global optimization)

1. Initialization. Set the list $L = \phi$. Set the working box $Y = X$.
2. Subdivision of Y. The algorithm splits up Y into subboxes. Usually, bisection is used for this purpose. Add the resulting subboxes to L.
3. Deletion conditions: To increase efficiency of the method, unwanted boxes V (where no global minimizer can be located) need to be identified and then deleted. The most commonly used deletion condition is based on f-values.

$$f_{\mathrm{best}} < \mathrm{Lb}\,(F(V))\,, \tag{2}$$

   where f_{best} is the currently known best value of the objective function (or its upper bound). The deletion condition (2) is similar to the so-called midpoint test [16]. f_{best} is usually updated by sampling the midpoint of the working box [16] or any other special box [12]. When constraints are present in (1), each processing subbox can be checked against the constraint satisfaction requirements. By using an inclusion function for each of the constraint functions, one can tell if a subbox is definitely infeasible or indeterminate. Definitely infeasible subboxes can be deleted. Other deletion conditions are possible, including the ones using information on derivatives of f(x).
4. Selection of a new working box Y from L.

5. Termination criterion. Obviously, any interval algorithm stops if there are no more boxes to be processed (the list is empty). For instance, this situation occurs when the original constrained problem is actually infeasible. But practically, it may stop earlier according to some other termination criteria.

Two well known early versions of interval methods (Ichida-Fujii and Hansen) fall into this framework. Ichida-Fujii's algorithm selects a new working box based on the smallest lower bound of inclusion function, while Hansen's algorithm selects a new working box based on the oldest age or on the largest size. Actually, there was another early version of Moore-Skelboe [27]. But that version didn't use any deletion conditions. Thus it does not quite fit the standard framework stated above. By now, there are a large variety of implemented versions of the interval method (e.g., [4,24,29,33]). There are also several accelerating devices reported in the literature. Interval methods have been used for solving many different kinds of mathematical problems: optimization of functions, systems of linear and nonlinear equations, ordinary differential equations, partial differential equations, and optimal controls, just to name some. A quick survey of a large number of published reports on their applications seems to indicate that they are generally limited to problems in low dimensions (say, much less than 100 in most cases). Obviously, there are indeed two major concerns in solving large dimensional problems: large amount of memory space required to hold boxes for further processing, and slow speed of convergence due to a large number of boxes to be processed. These two disadvantages are usually considered as unavoidable since most optimization problems are NP-hard (e.g., [31]). Although interval algorithms can converge exponentially to the globally optimal objective function value [6], the number of subboxes to be saved and processed in a standard interval method could also increase exponentially with the dimension of the domain. That raises memory requirement and slows down convergence. These two disadvantages of the interval methods may also occur when the structure of the optimization problem is too complicated or the inclusion function is not tight enough so that not many boxes could be deleted.

Maintaining a memory structure is seen as a very common strategy in many global optimization methods. Several examples can be mentioned for this purpose. Genetic algorithms and its variations explicitly maintain a population of candidate solutions. The size of this memory structure is normally fixed and predetermined. But the "best" size is generally problem dependent. Tabu search [10] maintains a tabu list that represents information about recently visited solutions. The length of tabu list is usually fixed as well. A standard interval algorithm keeps track of a list of all the subboxes that might contain some global solutions. The size of such a list is typically unlimited. In case unisolated global solutions exist, this list can grow very quickly without a finite bound. In order to reach theoretically guaranteed convergence of all the optimal solutions, all those boxes must be processed enough times. Thus this could lead to a computational nightmare, making the theoretical guarantee practically fruitless. A more general branch-and-bound method works in a similar way, but without necessarily using interval arithmetic. Of course, there exist global search algorithms that do not use any memory mechanism. Instead, they rely on randomness to have a chance to get close to a global solution. Such algorithms include random search and simulated annealing. However, the random search is never considered as an effective global search method. Simulated annealing is believed to outperform the random search due to its Metropolis criterion used to overcome thermal energy barriers in order to escape from local solutions. There is even a memory-based version of simulated annealing [2]. Finally we point out that even some local search methods also employ a memory structure. One typical example is the BFGS quasi-Newton method where an approximate inverse Hessian matrix has to be memorized between

two consecutive iterations of update. The size of this memory structure is also considerable when n is large.

3 Memoryless interval-based algorithm and its convergence

The standard interval method will capture all the global solutions since its final list would include X^*. As long as every box in the list has a chance to be processed infinitely often, those global solutions can be positively identified to any desired degree of accuracy if no hard limits are prescribed. Such a nice guarantee does come with a significant overhead cost in terms of memory space and CPU time consumption. It is also worthwhile to point out that the standard Ichida-Fujii interval algorithm does not delete any global solution. But still only one global solution is guaranteed to be estimated accurately. The other global solutions cannot be pinpointed to a desired precision because they are buried in the huge memory structure and do not get processed enough during the course of algorithm. Fortunately, not many practical optimization problems really require all the global solutions. In fact, most optimization methods do not find all of global solutions. But they are still extensively used in practice anyway. For example, local optimization methods and stochastic global optimization methods have been widely used. But none of them actually finds all the global solutions. Some of them even do not get any global solution at all. It is commonly believed that any optimization method that is capable of identifying one global solution or a good estimate of one global solution within a reasonable time frame would be of a good practical value. Thus, in our memoryless interval-based algorithm, we only target one global solution in a way similar to Ichida-Fujii interval algorithm. But for the other global solutions, we no longer commit any computer memory and CPU time since they may not be extracted accurately anyway. We trade the loss of other global solutions with much improved memory requirement and much faster convergence speed. Our main focus of improvement will be on the list L used in the regular interval algorithms. We actually abandon the list completely, breaking away from the standard memory philosophy of the interval method and the branch-and-bound method in general.

Algorithm 2 (Memoryless interval-based algorithm for global optimization)
Given f(x), X, and F(.).

Step 1. Initialization:

Step 1a. Set a working interval $Y = X$.
Step 1b. Get F(Y).

Save $f_{best} = f(c)$, where $c = Mid(Y)$.

Step 1c. Set $y = Lb(F(Y))$.

Step 2. Update:

Step 2a. Take any k in $\{i : Wid(Y) = Wid(Y_i)\}$, where $Y = Y_1 \times Y_2 \cdots \times Y_n$.
Step 2b. Bisect Y normal to the coordinate direction k, obtaining intervals V_1 and V_2.
Step 2c. Get $F(V_1)$ and $F(V_2)$.
Step 2d. Set $y_1 = Lb(F(V_1))$, $y_2 = Lb(F(V_2))$.

Step 2e. Deletion. Check deletion condition(s) to see if V_1 and V_2 can be deleted. For example,

$$f_{best} < y_i \rightarrow \text{Delete } V_i, \text{ for } i = 1, 2.$$

Step 2f. Selection. The smaller of y_1 and y_2 surviving from the deletion becomes y, and the corresponding V_i becomes Y.

Step 2g. Update $f_{best} = \min\{f_{best}, f(c)\}$, where $c = \text{Mid}(Y)$.

Step 3. If one of the prescribed termination criteria holds, then stop with output:

$$f^* \cong y, f^* \in F(Y), x^* \cong \text{Mid}(Y).$$

Step 4. Go to Step 2.

This is an interval-based algorithm with the least amount of memory requirement. Thus it is likely the fastest. However, we need to justify both theoretically and numerically that under certain conditions, this drastic reduction of memory won't prevent the algorithm from finding a good estimate of one global solution. In the remainder of this section, we present some theoretical results about the properties of the algorithm. A large amount of supporting numerical evidence will be postponed until the next section.

Annihilation of the list from the framework of regular interval algorithm drastically alters the characteristics of the interval algorithm. It is obvious that the new algorithm can capture at most one global solution instead of all the global solutions for the standard version. It is our hope that one global solution will be retained for sure under appropriate conditions. In the other cases, the algorithm would reach a global solution at an acceptable success rate. But retaining one global solution would demand certain properties of the inclusion function F(.). We state a set of sufficient conditions in the following theorem.

Theorem *Let $f(x)$ have a finite number of global solutions and its inclusion function $F(.)$ satisfy*

$$Wid(F(Y)) \dashrightarrow 0 \, as \, Wid(Y) \dashrightarrow 0.$$

Furthermore, $Lb(F(U)) < Lb(F(V))$ whenever $Inf(f(U)) < Inf(f(V))$ and $U^\circ \cap V^\circ = \phi$. Let $\{(y_k, Y_k)\}$ be the sequence generated by the algorithm, and $\{f_k\}$ the f_{best} sequence. Then the following statements hold.

(a) $Wid(Y_k) \rightarrow 0$.
(b) Y_k *contains at least one global solution.*
(c) Y_k *contains at least one* x_k *with* $f(x_k) = f_k$.
(d) *At least one box should survive the deletion step. In other words, the algorithm should always exit at Step 3.*
(e) $y_k \rightarrow f^*$, $f_k \rightarrow f^*$,
(f) $x_k \rightarrow x^*$, $Mid(Y_k) \rightarrow x^*$ *for some* $x^* \in X^*$.

Proof

(a) It follows from the fact that the bisection is done perpendicular to the direction of the maximum side of Y.
(b) In view of mathematical induction arguments and the theoretical convergence analysis of a list-based interval algorithm in [29] or some other references, we just need to make sure that the selected box in Step 2f would contain at least one global solution, provided that the previously selected box does so. Note that

$$f^* = \min\{\text{Inf}(f(V_1)), \text{Inf}(f(V_2))\}.$$

The minimizing box above would contain a global solution. That minimizing box is also the selected box in Step 2f of the algorithm by applying the stated assumption to V_1 and V_2 if $\mathrm{Inf}(f(V_1)) \neq \mathrm{Inf}(f(V_2))$. When $\mathrm{Inf}(f(V_1)) = \mathrm{Inf}(f(V_2))$, the selected box would also contain a global solution.

(c) According to Step 2g, $f_k \leq f(\mathrm{Mid}(Y_k))$. From (b) above, we have

$$f^* = \inf \{f(x) : x \in Y_k\}.$$

Thus

$$\inf \{f(x) : x \in Y_k\} \leq f_k \leq f(\mathrm{Mid}(Y_k)).$$

Hence the desired result follows from the intermediate value theorem as f(x) is at least continuous.

(d) From (c), $f(x_k) = f_k$. Thus

$$y_k = \mathrm{Lb}(F(Y_k)) \leq f(x_k) = f_k,$$

which violates the deletion condition. So as one of V_1 and V_2 in the algorithm, Y_k should survive the deletion step.

(e) It follows from (a)–(c) in view of the first assumption on F(.).

(f) It follows from (a) and (e). □

Corollary *When the inclusion function is perfectly tight, the new algorithm is guaranteed to converge to a global solution.*

It is well known that the first assumption on F(.) is used to guarantee global convergence

$$\min \{\mathrm{Lb}(F(V)) : V \in L_k\} \to f^*$$

for the Moore-Skelboe algorithm and the Hansen algorithm (as well as many other interval algorithms), where L_k is the list at iteration k. Without further assumptions on the inclusion function F(.), the Moore-Skelboe algorithm can converge arbitrarily slowly [25]. However, if F(.) is assumed to be of order α, then the Hansen algorithm offers this exponential convergence

$$f^* - \mathrm{Lb}(F(Y_k) \leq C(2\mathrm{Wid}(X))^{\alpha}(k+1)^{-\alpha/n}.$$

The same exponential convergence was established in [6] for the Moore-Skelboe algorithm when F(.) is further assumed to be isotonic. Thus it is reasonable to impose additional assumptions in order to guarantee theoretical convergence. One such additional assumption is stated in the theorem that ensures convergence of the memoryless algorithm to one global solution. That is the main difference between the two versions as far as the theoretical convergence properties are concerned. However, the guaranteed convergence to a global solution under the extra condition sets the new memoryless algorithm apart from typical heuristic global search methods where no such conditional guarantee is available (e.g. [23]).

Although the theorem only states sufficient conditions for convergence, our numerical test results will show that even when the stated sufficient conditions are not completely satisfied, our algorithm can still have a good chance to find one global solution. For example, function #8 listed in the appendix clearly violates the assumptions of the theorem. But we have confirmed that the new algorithm found one global solution successfully even when $n = 2{,}000$. Thus the new interval-based algorithm offers these good convergence characteristics: quick to reach an end of execution, little memory space requirement, good success rate in the general situation (although not always 100%). It is applicable in a much wider

range of optimization problems than the standard interval method because of much improved memory requirement and convergence speed. In view of the corollary above, we also believe that the tighter the inclusion function the greater the reliability of the new algorithm. The assumptions of the theorem might not be satisfied if the inclusion function F(.) is not tight enough in some uniform fashion. In those cases, the new algorithm might be thought of as an interval-based heuristic global search method. Thus it offers a good compromise between guaranteed global search methods and heuristic ones. It possesses a good degree of reliability and a fast pace of convergence as confirmed by a large amount of numerical evidence shown in the next section.

When constraints are present in (1), the algorithm can be adjusted to handle them. Let us point out such adjustments as follows.

(a) Initialization. If an initial feasible solution is available, use its objective function value to initialize f_{best}. Otherwise, f_{best} is set to a nominal value which could be a fairly large number if nothing about a feasible f-value is known. In our numerical experiments, a penalized objective function value of an infeasible initial solution is used as the starting nominal f_{best}-value (the worst case scenario).
(b) Feasibility tolerance and additional deletion conditions.

Feasibility tolerances are defined by these two parameters.

ε_h = a small threshold of absolute value of the equality constraint functions. Any active box V with $H_i(V)$ lying outside $[-\varepsilon_h, \varepsilon_h]$ will be designated as inactive (hence deleted) due to violation of the equality constraint $h_i(x) = 0$.

ε_g = a small threshold of upper bound of the inequality constraint functions. Any active box V with $G_j(V)$ lying outside $(-\infty, \varepsilon_g]$ will be considered as inactive (thus deleted) due to violation of the inequality constraint $g_j(x) \leq 0$.

(c) Additional exit point at end of Step 2e. If there is no surviving box at end of Step 2e, we would exit the algorithm and conclude that there is no optimal solution with f-value at least as small as f_{best}. In particular, if the initial f_{best} was set to a large enough value, then the conclusion at this point would be "no feasible solution".
(d) f_{best} gets updated in Step 2g only by ε-feasible points. Feasible points are less frequently encountered by the algorithm when ε_h and ε_g are smaller.

We note that the presence of constraints also increases chances of losing more global solutions. Therefore, it likely reduces success rate of the algorithm. Actually both the standard interval algorithm and the new interval-based algorithm suffer degradation of overall efficiency due to one common scenario, that is, the algorithms do not encounter enough feasible solutions at early stages. This scenario is very likely to occur when equality constraints are present. In that situation, not enough boxes can be deleted based on the deletion conditions. This forces a significant increase in memory requirement in the standard interval algorithm. Since the new memoryless algorithm does not allow for extra memory, it may be forced to lose all the global solutions by mistakenly preferring a lower infeasible y-value over a higher best feasible y-value. We are still investigating other strategies concerning this issue.

4 Numerical results

In our implementation of algorithms 1 and 2 (as well as the other ones used for comparison), several other acceleration devices are incorporated whenever appropriate.

L_p = a primary list of boxes that represents the remaining region to be searched.

L_s = a saved list of boxes that are not deleted but do not need to be further processed (i.e. inactive) according to some prescribed tolerances (ε_{box}, ε_f) listed below.

ε_{box} = a small box size threshold. Any active box V with size Wid(V) less than ε_{box} will be moved from L_p to L_s.

ε_f = a small threshold of deviation of the objective function values over a box. Any active box with the fluctuation of the objective function value less than ε_f will be stored into L_s as well.

nf_{max} = the maximum number of function (f(.) or F(.)) calls allowed. It is checked only once for every certain number of iterations. This limit is relaxed when an algorithm continues to improve its best solution.

cpu_{max} = the maximum CPU time allowed. It is also checked once for every certain number of iterations.

An initial solution is supplied to each algorithm for every test example. It is used to initialize f_{best}. However, when constraints are present, an infeasible initial solution is intentionally selected which increases degree of difficulty and reduces success rate under the specified stopping conditions. None of the initial solutions is close to the corresponding optimal solution.

We have used the Ichida-Fujii interval algorithm (IFIA, see [16]), and four (noninterval) stochastic methods to solve the same test problems for comparison. The Ichida-Fujii interval algorithm is the closest existing interval algorithm that utilizes a list as memory structure. The four stochastic methods are a simulated annealing algorithm (SA, see [18]), a random search algorithm (RS), a genetic algorithm (GA, see [11,19]), and a differential evolution genetic algorithm (DEGA, see [28]). The first two of the four stochastic methods can be considered as memoryless as well. But the other two also require a memory structure that is normally called population in the context of the genetic algorithm. As usual, those four noninterval algorithms do not have precise stopping conditions that would ensure immediate exit once a globally optimal solution is found unless problem (1) is special enough. But within any of the six algorithms we never intentionally used any special information of objective functions or constraint functions although some of them are of fairly special structure (such as linear, quadratic functions, or separable functions). Therefore, we have to use some artificial stopping conditions such as these: (a) the number of calls of the objective function reaches a prescribed limit; (b) the total computational time exceeds a specified limit; (c) the number of generations is too big for GA or DEGA; (d) there is no improvement of f_{best} over a specified number of iterations. All such artificial stopping conditions may potentially result in a premature termination of any algorithm. They may also result in some wastes of unnecessary iterations after reaching an optimal solution. Unfortunately, these are just some of the pitiful features of many noninterval algorithms. Some of the stopping conditions are also applied to the interval algorithms for fair comparison.

The comparison results presented below are by no means comprehensive and 100% realistic. In fact, we find it difficult to do a very realistic comparison since each algorithm has a number of its own control parameters that could affect its individual performance. It is difficult and probably impossible to find "equivalent" sets of all control parameters for two different algorithms. Performances of some algorithms are more sensitive to good values of control parameters than others. But no attempt was made to find best values of control parameters for each tested algorithm. Only reasonably guessed values are adopted. However, for each example, we have used the same set of the shared parameters (such as nf_{max} and cpu_{max}) and the same initial solution for all algorithms. To further increase reliability of the test results on stochastic methods, each example is run 5–40 times using randomly selected initial solutions and seeds for the random number generator while the other parameters are kept fixed. The

mean and standard deviation information will be summarized. Thus, the results below could still provide a good idea of effectiveness of the new algorithm in comparison with the others.

The two interval-based algorithms handle constraints directly and explicitly. But all the stochastic algorithms that we have used for comparison do not have any special mechanism built in to treat the constraints. Therefore, technically speaking, they can be compared with the interval algorithms only for unconstrained problems. However, as done in [29], penalized objective functions have been used to run the stochastic methods for the constrained problems (thus #(g-calls) = #(h-calls) = #(f-calls)). For simplicity, we have used a common penalty coefficient value of 100 for all the constraints and all the constrained examples although a desirable penalty coefficient value is usually problem dependent. Additional runs might have been done to calibrate its value. But we did not do that due to fairness concern. However, constrained and unconstrained problems are separately compared so that we may get a better idea of effect of constraints on the performance of the algorithms.

We also tested Hansen's interval algorithm, a tree annealing algorithm [3], and two versions of tabu search methods (see [7,26]). Hansen's algorithm differs from Ichida-Fujii's algorithm in the way every new working box Y is selected. It is not as aggressive as Ichida-Fujii's algorithm to zoom in on one global solution. Instead, it attempts to zoom in on all global solutions. As a consequence, it usually takes a lot more CPU time as well as function calls to reach a desired termination condition. Our test results confirmed this. Under the hard stopping conditions we used, it performed poorly on a majority of our test examples. The other three search methods also did not do well on the average under our hard stopping conditions. Thus their test results are not included in this section.

As stated in Sect. 2, an inclusion function is usually required for solving each optimization problem by using an interval method. Inclusion functions are not unique. We used natural interval extensions when they are available. When they are not available, we use extended custom made inclusion functions. In any event, no derivatives are ever used.

To test performance of the new algorithm (denoted by MLIA), we have used a large number of examples with or without constraints. They are selected more or less randomly from several recent publications. Most of these examples have been widely used by other people for testing their new optimization algorithms (e.g., [9,13,19,29,32]). Among those are: Rastrigin function, Goldstein-Price function, piecewise function, Levy functions, Branin function, Shubert function, our linear complementarity problem, our discrete Halmilton-Jacobi-Bellman equation problem, De Jong function, Colville function, Griewank function, Rosenbrock function, Zakharov function, sphere function, Schwefel functions, step function, and Ackley function. Modified versions of some of those functions have been included as well. Since algorithms 1 and 2 use midpoints as sample points, some original search domains have been intentionally altered so that the exact optimal solution won't be reached accidentally as middle points are sampled at early stages. Other modifications are made to shift the best known value of the objective function to zero or to make numerical overflows less likely to occur in large dimensional cases. Among those examples, 15 of them are formulated with flexible dimensions. We vary those dimensions as 4, 10, 40, 100, 200, 400, 600, 800, 1,000, and 2,000. Different dimensions resulted in different test examples. The total number of examples we have tested is well over 100. Their dimensions vary from 1 to 2,000. Many of those examples are often regarded as difficult benchmark examples by other people. Obviously it is not a good idea to explicitly state all those examples. But the 15 examples with variable dimensions are listed in the appendix since they are repeatedly used. Our preliminary test results on all examples are presented below only in terms of their overall statistics (i.e. means and standard deviations).

We have used two sets of various ranking scores to quantitatively measure the performance of each algorithm and to compare different algorithms as a group. For individual algorithms, we used three scores. One of them is a composite ranking score to quantitatively compare various results. A composite ranking score R_q reflects the quality of the final solution in terms of the objective function value as well as the maximum amount of constraint violation. More precisely, we first calculate a ranking score r_f based on the final best objective function value (called f_{best}).

Objective function value: Ranking score $r_f(f^* = 0)$					
f_{best}	$f_{best} < 0.001$	$f_{best} < 0.01$	$f_{best} < 1.0$	$f_{best} < 10$	$f_{best} \geq 10$
r_f	1 (best)	2	3	4	5

Then we calculate a ranking score r_c based on the maximum amount of constraint violation of the final best solution $V_c = \max\{|h_i(\text{xbest})|, \max\{0, g_j(\text{xbest})\} : i = 1, \ldots, m, j = 1, \ldots, p\}$.

Constraint violation amount: Ranking score $r_c(V_c^* = 0)$					
V_c	$V_c < 0.01$	$V_c < 0.1$	$V_c < 1.0$	$V_c < 10$	$V_c \geq 10$
r_c	1 (best)	2	3	4	5

The composite ranking score for solution quality is then defined as

$$R_q = \max\{r_f, r_c\}.$$

The other two scores are the total number of objective function (f(.) or F(.)) calls (n_f or n_F) as well as the total CPU time consumption (R_{cpu}). Those two scores would reflect the effectiveness of algorithm. The total number of objective function calls is a major effectiveness indicator. But CPU time would also include various CPU time overheads required by each algorithm. For constrained optimization problems, each algorithm would require a certain number of calls of the constraint functions. Those calls have been omitted when the number of function calls is calculated mainly for this reason. Typically for our test examples constraint functions are relative simpler than the objective functions and the number of objective function calls is usually more than the number of calls of constraint functions. However, the total CPU time counts would include effects of constraint function calls. We also observe that the original objective function and its inclusion function would require significantly different computational efforts. So they are separately counted. Then additional numerical tests are performed to estimate how many f-calls (say, N_{Ff}) would be equivalent to a single F-call as far as CPU time is concerned. This factor is then used to determine a combined number of objective function calls.

$$R_{nf} = n_f + N_{Ff} \times n_F. \tag{3}$$

The total CPU times are recorded in milliseconds and presented below in seconds. They are included as a reference and the actual CPU times might be less than the reported figures since multiple applications could be executed at the same time for some test runs. Thus the numbers of function calls would be more reliable indicators of efficiency of algorithms. Those three scores are independently calculated for each algorithm without any reference to the other algorithms.

To quantitatively compare six different algorithms as a group, we introduce another set of ranking scores based on the three different types of scores described previously: the overall quality score of solution, the number of function calls, and the total CPU count. However, when two algorithms have close enough scores, their new group ranks will be considered the same to account for various sources of uncertainties or randomness such as selection of test

examples. For example, one algorithm that was ranked 1 for testing one particular example may not be ranked 1 again when another example is tested. More precisely, we define the group quality rank as

$G_q = 1$ if R_q is the lowest or within 0.5 of the lowest,
$G_q = 1$ if R_q is within 0.1 of any existing R_q that has been ranked 1.
$G_q = 2$ is defined similarly among the remaining R_q values.
This process is continued until all the algorithms are ranked.

We define the group CPU rank as

$G_{cpu} = 1$ if R_{cpu} is the lowest or within five times of the lowest,
$G_{cpu} = 1$ if R_{cpu} is within twice of any existing R_{cpu} that has been ranked 1.
$G_{cpu} = 2$ is defined similarly among the remaining R_{cpu} values.
This process is continued until all the algorithms are ranked.

We define the group #(f-calls) rank similarly as

$G_{nf} = 1$ if R_{nf} is the lowest or within five times of the lowest,
$G_{nf} = 1$ if R_{nf} is within twice of any existing R_{nf} that has been ranked 1.
$G_{nf} = 2$ is defined similarly among the remaining R_{nf} values.
This process is continued until all the algorithms are ranked.

Thus if two algorithms have different group rankings, then we know that their performances are significantly different (at least of about 1 order of magnitude in difference).

For each of the higher dimensional examples, run-time errors (such as numerical overflow) occur when the dimension exceeds a certain value. A call of the inclusion function involves calculation of a lower bound and an upper bound of the objective function over a working box. Since those bounds may be far from being tight, the chances of running into numerical overflows are greater for interval-based algorithms. Typically, summations and products appear in the analytical formula of f(.) in variable dimensions, and those values easily reach numerical overflow threshold of machine numbers in a small personal computer without special treatments of numerical infinities. Sometimes, exponential functions appear in the objective function. Then it is more likely to have the overflow problem. When the dimension is up to 2,000, we already observed a number of numerical overflow cases. However, we believe that the new algorithm is still applicable as long as numerical overflows can be avoided by reformulation of the problem, by increase of precision of variables, or by certain special treatments of numerical infinities. We are still investigating those ideas with partial successes so far. In case of run-time errors, test results for that example would not be included in the article.

Our examples are grouped according to their dimensions (1–6 for small dimensions, 9–13 for medium dimensions, 40–2,000 for higher dimensions) as well as their constraints (constrained or unconstrained). Limits on the number of function calls and CPU time consumption are set to different values for the different ranges of dimensions. Numerical results of 12 different sets of examples are presented below. A separate table is displayed for each set of examples. Each table is divided into two parts. The scores in the top half are merely based on test results of each individual algorithm. The first column shows different algorithms used for that set of examples. Each of the remaining columns contains the mean and standard deviation values due to multiple examples as well as possible multiple runs of the same examples. The second column indicates the average performance ranking score R_q for all the examples in this set. Since each noninterval algorithm is run 5–40 times, those scores will be averaged for any fixed example first. Then the average of those averaged values appears in the second

column. If only the minimum of 5–40 scores for each example is calculated, then the average of the minimum values is presented in the third column. For interval-based algorithms, the entries in second and third columns must be the same since only one run was done for each example. The forth column lists the mean and standard deviation of $n_f = \#$(f-calls), the total number of calls of the original objective function f(x) for each run. For interval-based algorithms, we have split that number into two parts, n_f and n_F (the total number of calls of its inclusion function F(X)). Usually, a call of F(X) is computationally more expensive than a call of f(x) (in fact, usually at least several times more expensive). The next column displays the mean and standard deviation of the total CPU time per run of each method that would include real CPU time plus all the auxiliary times such as I\O times. However, we did not count the CPU time if it takes less than one millisecond. But, the mean and standard deviation of CPU times displayed may be less than one millisecond since they are mathematically calculated from the individual CPU time values.

The bottom half of table shows various group rankings. The second, third, and sixth columns are based on the second, third and sixth columns of the top half, respectively. The data in the forth and fifth columns of the top half are combined into a single number displayed in the fifth column of the bottom half. Then a corresponding group ranking G_{nf} is shown in the forth column. We used numerical experiments to estimate the number of f-calls that is equivalent to a single call of F as far as CPU time is concerned. That number N_{Ff} varies from 1.15 to 23.87 for our examples, which is used in (3) to combine n_f and n_F to form R_{nf} in column five.

All of the test results have been generated by an AMD Turion 64 X2 mobile technology TL-58/1.9 GHz laptop computer with 2 GB of RAM under 32 bit Windows Vista environment. As we present test results of each set of examples, we will also make some important observations and offer brief discussions about the results.

Example set 1 (small dimensions, unconstrained) This set contains 41 examples of small dimensions ranging from 1 to 6 that do not contain any constraints other than the bound constraints. Summary of the test results is shown in Table 1. The new algorithm MLIA is about 2 orders of magnitudes faster than its standard counterpart IFIA in terms of the number of objective function calls while the accuracy is about 1 order of magnitude worse. In terms of CPU time counts, MLIA is about 3 orders of magnitudes faster than IFIA, and 2 orders of magnitudes faster than the others. This implies that for low dimensional problems, the extra time required for algorithm IFIA to manage the memory data structure is significant. Algorithm IFIA scored 1 in quality rank for 37 of the 41 examples, while algorithm MLIA scored 1 for 22 times. Among the four noninterval algorithms, DEGA performed best in terms of accuracy as well as efficiency. SA tends to exit prematurely sometimes (probably due to a fast cooling schedule that was adopted). As a result, it used a much smaller number of calls of f(x) than the other noninterval algorithms. Among all six algorithms, MLIA is clearly the best in terms of convergence speed. In terms of quality of final solutions, MLIA is ranked above the average.

Example set 2 (small dimensions, constrained) This set contains 20 examples of small dimensions between 1 and 6 with additional equality and\or inequality constraints. Fourteen of them have only inequality constraints. Now the overall ranking R_q would reflect the quality of final solution in terms of its objective function value as well as the amount of constraint violation. Summary of its test results is in Table 2. Generally speaking, the performance of each algorithm is down compared with its performance on unconstrained problems. For the interval-based algorithms, this is partially due to the fact that we used the worst case scenario in the initialization step as pointed out in Sect. 3.

Table 1 Summary of test results for example set 1 (small dim, unconstrained)

Method	avg-R_q	min-R_q	n_f: #(f-calls)	n_F: #(F-calls)	R_{cpu}: CPU time (s)
Individual ranks					
SA	2.8, 1.4	1.8, 1.4	2287, 2185	0, 0	0.102, 0.085
GA	2.9, 1. 5	2.4, 1.5	50408, 21888	0, 0	3.045, 1.565
DEGA	1.5, 1.0	1.2, 0.6	11081, 15863	0, 0	0.244, 0.314
RS	2.9, 1.6	2.9, 1.6	43731, 14579	0, 0	0.934, 0.509
IFIA	1.2, 0.9	1.2, 0.9	3175, 9526	28, 7278	12.616, 46.179
MLIA	2.3, 1.6	2.3, 1.6	92, 73	96, 62	0.014, 0.014
Method	avg-G_q	min-G_q	G_{nf}	R_{nf}	G_{cpu}
Relative group ranks					
SA	3	2	2	2287	2
GA	3	2	3	50408	3
DEGA	1	1	2	11081	2
RS	3	3	3	43731	3
IFIA	1	1	2	18236	4
MLIA	2	2	1	458	1

Table 2 Summary of test results for example set 2 (small dim, constrained)

Method	avg-R_q	min-R_q	n_f: #(f-calls)	n_F: #(F-calls)	R_{cpu}: CPU time (s)
Individual ranks					
SA	4.0, 0.8	2.2, 1.2	2128, 1503	0, 0	0.093, 0.050
GA	3.1, 1.2	2.7, 1.4	53180, 11538	0, 0	3.137, 0.892
DEGA	1.3, 0.7	1.0, 0.0	10202, 15668	0, 0	0.175, 0.206
RS	3.1, 1.3	3.1, 1.3	46985, 8840	0, 0	0.942, 0.421
IFIA	2.0, 1.6	2.0, 1.6	1859, 5432	10050, 14950	57.689, 93.472
MLIA	3.4, 1.8	3.4, 1.8	18, 31	81, 56	0.010, 0.010
Method	avg-G_q	min-G_q	G_{nf}	R_{nf}	G_{cpu}
Relative group ranks					
SA	4	2	2	2128	2
GA	3	3	3	53180	3
DEGA	1	1	2	10202	2
RS	3	3	3	46985	3
IFIA	2	2	3	34869	4
MLIA	3	4	1	286	1

DEGA continues to be the best in quality ranks. Every other algorithm performed not as good as for the unconstrained problems in example set 1. However, MLIA remains the best in efficiency ranks. Although MLIA appears to be the second worst in terms of the average

Table 3 Summary of test results for example set 3 (medium dim, unconstrained)

Method	avg-R_q	min-R_q	n_f: #(f-calls)	n_F: #(F-calls)	R_{cpu}: CPU time (s)
Individual ranks					
SA	1.9, 1.2	1.3, 0.62	39189, 29067	0, 0	0.901, 0.684
GA	4.0, 0.03	4.7, 0.49	505675, 372767	0, 0	55.9, 43.5
DEGA	2.6, 1.6	2.2, 1.6	143016, 101377	0, 0	3.96, 3.17
RS	5.0, 0.0	4.9, 0.4	250001, 0.0	0, 0	16.4, 1.58
IFIA	1.6, 1.4	1.6, 1.4	37490, 82192	28468, 57973	1087, 2250
MLIA	1.9, 1.6	1.9, 1.6	401, 194	388, 102	0.062, 0.048
Method	avg-G_q	min-G_q	G_{nf}	R_{nf}	G_{cpu}
Relative group ranks					
SA	1	1	2	39189	2
GA	3	4	3	505675	3
DEGA	2	3	2	143016	2
RS	3	4	2	250001	3
IFIA	1	1	2	233671	4
MLIA	1	2	1	2135	1

quality ranks, it still scored 1 for 6 out of 20 examples while IFIA scored 1 for 13 times. It is very clear that MLIA is the quickest to reach a stopping condition. In fact, the number of function calls is even less than the number for unconstrained problems in Table 1. This is probably due to the additional deletion condition based on the constraints. But the same effect does not show up for IFIA because constraints might have negative effects as well, as pointed out at the end of Sect. 3. Thus MLIA is capable of finding a solution quickly. Some of the solutions are still globally optimal, while more are not globally optimal. Since it takes so much less time to reach the conclusion, we would be able to spend some time to do something else with the unsuccessful runs. In fact, MLIA could have been applied around 100 times as far as CPU time is concerned. With the starting intervals of those repeated runs carefully selected, there would be a much better chance to reach a global solution. This strategy is being investigated and meaningful results would be reported elsewhere. SA shows a much worse average value of f_{best} due to many runs with premature exits. However, we noticed that if only the best of 40 runs of SA were counted for each example, the average ranking score of SA would be 2.2. To the contrary, RS would still have the average ranking score of 3.1 under the same scheme. These results perhaps indicate these important findings. (a) Globally optimal feasible solution is more likely to get lost in the memoryless algorithm because infeasible objective function value may be smaller than globally optimal feasible objective function value. (b) RS is not a good global search algorithm.

Example set 3 (medium dimensions, unconstrained) This set contains 15 unconstrained examples of medium dimension 10. Summary of the test results is in Table 3. The new algorithm is about 2 orders of magnitudes faster than its standard version in terms of the number of objective function calls, and its CPU time count is decreased to 4 or 5 orders of magnitudes, while it maintains a compatible degree of quality. For these 15 examples, MLIA's performance somehow exceeds our normal expectations. DEGA is clearly getting behind to become an average performer.

Table 4 Summary of test results for example set 4 (medium dim, constrained)

Method	avg-R_q	min-R_q	n_f: #(f-calls)	n_F: #(F-calls)	R_{cpu}: CPU time (s)
Individual ranks					
SA	3.2, 1.3	2.0, 1.4	15357, 5800	0, 0	0.351, 0.166
GA	3.6, 1.6	2.6, 1.8	354164, 212202	0, 0	39.6, 25.2
DEGA	2.4, 0.7	1.0, 0.0	106536, 59264	0, 0	1.65, 0.814
RS	4.6, 0.6	4.6, 0.6	200821, 1070	0, 0	13.5, 7.69
IFIA	4.4, 0.5	4.4, 0.5	0, 0	198721, 11551	5436, 2.74
MLIA	3.8, 1.6	3.8, 1.6	17, 39	115, 129	0.0188, 0.0129
Method	avg-G_q	min-G_q	G_{nf}	R_{nf}	G_{cpu}
Relative group ranks					
SA	2	2	2	15357	2
GA	2	3	3	354164	3
DEGA	1	1	3	106536	2
RS	4	5	3	200821	3
IFIA	4	5	3	537936	4
MLIA	3	4	1	323	1

Example set 4 (medium dimensions, constrained) This set contains five constrained problems with dimensions ranging from 9 to 13. Four of them have equality constraints and the other one has nine inequality constraints. Summary of its test results is in Table 4. Again, the constraints very much affected every algorithm's performance. Constraints made MLIA to exit more quickly. The quality of solutions identified by MLIA is ranked slightly below the average. But the speed to reach its final solution is clearly far better than the others. The tough constraints made the standard interval algorithm IFIA worse than the memoryless version in quality ranks. In fact, the data show that IFIA did not encounter any feasible solutions at all after processing so many boxes.

Example set 5 (40-dimension, unconstrained) This set contains 15 unconstrained problems with dimensions all equal to 40. Summary of its test results is in Table 5. Now MLIA becomes the best in all aspects of rankings. Three of the four noninterval algorithms (GA, DEGA, RS) performed very poorly. Thus those three noninterval algorithms will no longer be tested for the remaining sets of examples that are supposed to be more difficult than the current set. Since those examples are of higher dimensions with everything else exactly the same, we do not expect any better outcomes for those algorithms. We did not test enough constrained problems of dimensions 40 or higher. So no results on constrained problems of higher dimensions will be reported below.

Example set 6 (100-dimension, unconstrained) This set again contains 15 unconstrained problems with dimensions all equal to 100. Summary of its test results is in Table 6. Noninterval algorithms GA, DEGA, and RS were not tested due to their poor performance for the previous set of examples. Thus we marked them with "poor" in the table and corresponding rankings would be the worst. For the first time (actually the only time), CPU time count of another algorithm (SA) beats MLIA's. This is somewhat not expected. But a closer look at the original output files reveals that one of the 15 examples will run into numerical overflow

Table 5 Summary of test results for example set 5 (40-dim, unconstrained)

Method	avg-R_q	min-R_q	n_f: #(f-calls)	n_F: #(F-calls)	R_{cpu}: CPU time (s)
Individual ranks					
SA	2.7, 1.5	2.1, 1.3	5.08e6, 3.25e6	0, 0	16.0, 25.5
GA	5.0, 0.0	4.9, 0.3	1.45e7, 2.91e6	0, 0	2611, 152
DEGA	5.0, 0.0	4.9, 0.3	1.03e7, 0	0, 0	45.2, 64.9
RS	5.0, 0.0	4.9, 0.3	500001, 0	0, 0	144, 62.9
IFIA	2.0, 1.7	2.0, 1.7	39021, 66024	32761, 53983	1452, 2490
MLIA	2.1, 1.7	2.1, 1.7	1553, 856	1460, 543	5.05, 17.8
Method	avg-G_q	min-G_q	G_{nf}	R_{nf}	G_{cpu}
Relative group ranks					
SA	2	1	3	5.08e6	1
GA	3	2	3	1.45e7	3
DEGA	3	2	3	1.03e7	2
RS	3	2	2	500001	2
IFIA	1	1	2	279374	3
MLIA	1	1	1	10871	1

Table 6 Summary of test results for example set 6 (100-dim, unconstrained)

Method	avg-R_q	min-R_q	n_f: #(f-calls)	n_F: #(F-calls)	R_{cpu}: CPU time (s)
Individual ranks					
SA	2.9, 1.6	2.4, 1.7	420188, 112658	0, 0	337, 1180
GA	Poor	Poor	Poor	0, 0	Poor
DEGA	Poor	Poor	Poor	0, 0	Poor
RS	Poor	Poor	Poor	0, 0	Poor
IFIA	2.1, 1.8	2.1, 1.8	24752, 43914	24841, 43877	1696, 3016
MLIA	2.3, 1.9	2.3, 1.9	3293, 1540	3361, 1572	663, 2558
Method	avg-G_q	min-G_q	G_{nf}	R_{nf}	G_{cpu}
Relative group ranks					
SA	2	1	2	420188	1
GA	3	2	3	Poor	3
DEGA	3	2	3	Poor	3
RS	3	2	3	Poor	3
IFIA	1	1	2	143405	2
MLIA	1	1	1	24396	1

when its dimension is increased to the next value of 200. At the current dimension, that example behaved so badly that SA had to stop after reaching the preset limit on n_f, while IFIA and MLIA had to stop after reaching the preset limit on CPU time. If that example had been

Table 7 Summary of test results for example set 7 (200-dim, unconstrained)

Method	avg-R_q	min-R_q	n_f: #(f-calls)	n_F: #(F-calls)	R_{cpu}: CPU time (s)
Individual ranks					
SA	3.2, 1.5	2.8, 1.8	545247, 116822	0, 0	79.1, 86.5
GA	Poor	Poor	Poor	0, 0	Poor
DEGA	Poor	Poor	Poor	0, 0	Poor
RS	Poor	Poor	Poor	0, 0	Poor
IFIA	1.9, 1.7	1.9, 1.7	25442, 35355	25658, 35276	1202, 22
MLIA	2.1, 1.8	2.1, 1.8	7057, 2706	7230, 2682	13.5, 19.6
Method	avg-G_q	min-G_q	G_{nf}	R_{nf}	G_{cpu}
Relative group ranks					
SA	2	2	2	545247	2
GA	3	3	3	Poor	4
DEGA	3	3	3	Poor	4
RS	3	3	3	Poor	4
IFIA	1	1	1	238970	3
MLIA	1	1	1	78849	1

omitted, (CPU_{mean}, CPU_{SD}) values for SA, IFIA, and MLIA would have been (32.9, 29.4 s), (1170, 2312 s), and (2.63, 2.0 s), respectively. Again, MLIA becomes the fastest very clearly. But in any event, MLIA is still a lone top performer in terms of the number of function calls, and offered better quality solutions than SA. SA took over 56 h to get the reported numerical results. We also observed that both quality and efficiency of SA on different runs do not vary significantly. Thus, for each of the remaining sets of examples of higher dimensions, SA will be run only 5 times in order to keep the total run time under control.

Example set 7 (200-dimension, unconstrained) This set contains 14 unconstrained problems with dimensions all equal to 200. Summary of its test results is in Table 7. MLIA and IFIA are close in quality of solutions. But MLIA is clearly more efficient. Actually another example was tested and a numerical overflow occurred under one of the two methods. For fair comparison, the numerical results for that example are not included in this table. It is expected that numerical overflows become more likely to occur when the dimension of problem exceeds 100. It is even worse under interval-based algorithms because they use loose function bounds instead of the actual function values.

Example set 8 (400-dimension, unconstrained) This set contains 13 unconstrained problems with dimensions all equal to 400 after we have excluded two other functions due to encounters of numerical overflows. Summary of its test results is in Table 8. We note that the average quality of final solutions under both interval-based algorithms show a sign of improvement compared with the previous sets. That is due to the fact that both interval-based algorithms received bad scores for those omitted examples before numerical overflows occurred.

Example set 9 (600-dimension, unconstrained) This set contains ten unconstrained problems with dimensions all equal to 600. Summary of its test results is in Table 9. Again, better quality scores are observed for interval-based algorithms because more "hard" examples have

Table 8 Summary of test results for example set 8 (400-dim, unconstrained)

Method	avg-R_q	min-R_q	n_f: #(f-calls)	n_F: #(F-calls)	R_{cpu}: CPU time (s)
Individual ranks					
SA	3.8, 1.3	3.3, 1.8	597258, 65346	0, 0	202, 341
GA	Poor	Poor	Poor	0, 0	Poor
DEGA	Poor	Poor	Poor	0, 0	Poor
RS	Poor	Poor	Poor	0, 0	Poor
IFIA	1.6, 1.5	1.6, 1.5	29506, 29107	29878, 28970	1484, 2293
MLIA	1.8, 1.8	1.8, 1.8	13553, 5210	13893, 5164	75.9, 147
Method	avg-G_q	min-G_q	G_{nf}	R_{nf}	G_{cpu}
Relative group ranks					
SA	2	2	1	597258	1
GA	3	3	2	Poor	3
DEGA	3	3	2	Poor	3
RS	3	3	2	Poor	3
IFIA	1	1	1	376934	2
MLIA	1	1	1	154650	1

been dropped due to their numerical overflows. But the quality of SA continues to deteriorate despite the dropouts. This may prove that SA is more susceptible to dimension increase than the interval methods.

Example set 10 (800-dimension, unconstrained) This set contains eight unconstrained problems of dimensions all equal to 800. Summary of its test results is in Table 10.

Example set 11 (1,000-dimension, unconstrained) This set contains only six unconstrained problems with dimensions all equal to 1,000. Summary of its test results is in Table 11. It is interesting to observe that the solution quality of MLIA exceeds that of IFIA while the number of functions calls is getting closer. However, the CPU time counts still differ by at least one order of magnitude. This implies a significant overhead in CPU time when a memory structure is created and constantly updated.

Example set 12 (2,000-dimension, unconstrained) This final set also contains six unconstrained problems with dimensions all equal to 2,000. Test results are summarized in Table 12. It is observed that the increased computational demand for IFIA in this large dimensional case resulted in more frequent run-time errors. Such errors prevented IFIA from reaching a final solution for five out of the six examples. This shows an additional benefit of MLIA over IFIA. SA is also tested and all the runs resulted in poor solutions.

Now we rank the performance of six algorithms based on test results of first 11 different sets of examples. Example set 12 is not included in this ranking because Table 12 is far from complete. Table 13 shows the overall quality ranking scores G_q of the algorithms. On the average of so many examples, our new memoryless interval-based algorithm MLIA is shown to be very compatible with the standard interval algorithm IFIA as far as the quality of the final solutions is concerned. Table 14 shows an important efficiency ranking based on the total effective number of function calls G_{nf}. It is this ranking that clearly shows a significant advantage of MLIA over all the other methods.

Table 9 Summary of test results for example set 9 (600-dim, unconstrained)

Method	avg-R_q	min-R_q	n_f: #(f-calls)	n_F: #(F-calls)	R_{cpu}: CPU time (s)
Individual Ranks					
SA	4.1, 0.9	3.0, 1.9	618323, 28261	0, 0	421, 755
GA	Poor	Poor	Poor	0, 0	Poor
DEGA	Poor	Poor	Poor	0, 0	Poor
RS	Poor	Poor	Poor	0, 0	Poor
IFIA	1.0, 0.0	1.0, 0.0	29159, 26917	29881, 26920	1503, 2113
MLIA	1.3, 0.9	1.3, 0.9	18879, 7727	19541, 7764	244, 554
Method	avg-G_q	min-G_q	G_{nf}	R_{nf}	G_{cpu}
Relative group ranks					
SA	2	2	1	618323	1
GA	3	3	2	Poor	3
DEGA	3	3	2	Poor	3
RS	3	3	2	Poor	3
IFIA	1	1	1	445959	2
MLIA	1	1	1	243398	1

Table 10 Summary of test results for example set 10 (800-dim, unconstrained)

Method	avg-R_q	min-R_q	n_f: #(f-calls)	n_F: #(F-calls)	R_{cpu}: CPU time (s)
Individual ranks					
SA	4.2, 0.8	2.8, 2.0	600693, 32343	0, 0	205, 290
GA	Poor	Poor	Poor	0, 0	Poor
DEGA	Poor	Poor	Poor	0, 0	Poor
RS	Poor	Poor	Poor	0, 0	Poor
IFIA	1.3, 0.7	1.3, 0.7	29649, 27693	30851, 28016	1650, 3023
MLIA	1.4, 1.1	1.4, 1.1	22524, 9663	23626, 10054	134, 81.4
Method	avg-G_q	min-G_q	G_{nf}	R_{nf}	G_{cpu}
Relative group ranks					
SA	2	2	1	600693	1
GA	3	3	2	Poor	3
DEGA	3	3	2	Poor	3
RS	3	3	2	Poor	3
IFIA	1	1	1	447287	2
MLIA	1	1	1	278660	1

Table 11 Summary of test results for example set 11 (1000-dim, unconstrained)

Method	avg-R_q	min-R_q	n_f: #(f-calls)	n_F: #(F-calls)	R_{cpu}: CPU time (s)
Individual ranks					
SA	4.9, 0.3	4.8, 0.4	500002, 0	0, 0	226, 352
GA	Poor	Poor	Poor	0, 0	Poor
DEGA	Poor	Poor	Poor	0, 0	Poor
RS	Poor	Poor	Poor	0, 0	Poor
IFIA	1.7, 1.6	1.7, 1.6	35000, 30775	37001, 31223	2857, 4868
MLIA	1.5, 1.2	1.5, 1.2	267, 13969	28634, 14730	215, 133
Method	avg-G_q	min-G_q	G_{nf}	R_{nf}	G_{cpu}
Relative group ranks					
SA	2	2	1	500002	1
GA	3	3	2	Poor	3
DEGA	3	3	2	Poor	3
RS	3	3	2	Poor	3
IFIA	1	1	1	580876	2
MLIA	1	1	1	372734	1

Table 12 Summary of test results for example set 12 (2,000-dim, unconstrained)

Method	avg-R_q	min-R_q	n_f: #(f-calls)	n_F: #(F-calls)	R_{cpu}: CPU time (s)
Individual ranks					
SA, GA, DEGA, RS: (all poor)					
IF	4.3, 1.6	4.3, 1.6	n/a	n/a	n/a
MLIA	1.7, 1.6	1.7, 1.6	24809, 12840	47167, 23192	654, 479
Method	avg-G_q	min-G_q	G_{nf}	R_{nf}	Gn_{cpu}
Relative group ranks					
SA, GA, DEGA, RS: (all poor)					
IF	2	2	n/a	n/a	n/a
MLIA	1	1	1	381034	1

5 Final comments and conclusions

In conclusion, the major observed advantages of the new algorithm include:

(a) It is more robust than a conventional interval method as well as noninterval methods. You can set up its parameters so that the algorithm would perform relatively well for a large variety of test problems. For a conventional interval method, parameter values are more sensitive to test problems (mostly noticeably to the magnitude of problem dimension).

Table 13 Overall quality score G_q of algorithms (1 = best)

G_q score	SA	GA	DEGA	RS	IFIA	MLIA
Set 1	3	3	1	3	1	2
Set 2	4	3	1	3	2	3
Set 3	1	3	2	3	1	1
Set 4	2	2	1	4	4	3
Set 5	2	3	3	3	1	1
Set 6	2	3	3	3	1	1
Set 7	2	3	3	3	1	1
Set 8	2	3	3	3	1	1
Set 9	2	3	3	3	1	1
Set 10	2	3	3	3	1	1
Set 11	2	3	3	3	1	1
Mean, SD	2.18, 0.75	2.91, 0.30	2.36, 0.92	3.09, 0.30	1.36, 0.92	1.45, 0.82
Final rank	3	5	4	6	1	2

Table 14 Overall efficiency score G_{nf} of algorithms (1 = best)

G_{nf} score	SA	GA	DEGA	RS	IFIA	MLIA
Set 1	2	3	2	3	2	1
Set 2	2	3	2	3	3	1
Set 3	2	3	2	2	2	1
Set 4	2	3	3	3	3	1
Set 5	3	3	3	2	2	1
Set 6	2	3	3	3	2	1
Set 7	2	3	3	3	1	1
Set 8	1	2	2	2	1	1
Set 9	1	2	2	2	1	1
Set 10	1	2	2	2	1	1
Set 11	1	2	2	2	1	1
Mean, SD	1.73, 0.65	2.64, 0.50	2.36, 0.05	2.45, 0.52	1.72, 0.79	1, 0
Final rank	3	6	4	5	2	1

(b) It is a deterministic derivative free global optimization method. Typically, a deterministic global optimization method requires keeping track of partition of the search domain. But this algorithm is memoryless. This memoryless feature would allow the algorithm to solve large scale optimization problems that might be insolvable by other methods due to lack of enough computational resources (memory space and CPU time).

(c) It reaches a final solution a lot more quickly than all the other algorithms that have been tested. Furthermore, there is a good chance for the final solution to be a very good estimate of a globally optimal solution of the original problem. Since it is so quick to reach a final solution, there is a lot of time saved for potentially conducting additional searches in case the current final solution is not satisfactory.

We also realize some disadvantages of the memoryless algorithm:

(a) Clearly it cannot locate all the global solutions.
(b) Another noticeable concern about the new algorithm is that its convergence to a global solution is theoretically guaranteed under a stronger assumption on the inclusion function. When such assumption is not satisfied, all the global solutions could be lost.

Not every feature of the new algorithm has been extensively tested yet. We are constantly improving it as more analysis and more tests are done. Nevertheless, the preliminary results are encouraging and show a good promise for it to become one of practical effective global optimization methods.

Acknowledgments The author is grateful to the AMA Department of Hong Kong Polytechnic University for various support while this research was partially carried out there, and to two anonymous referees for the valuable comments and suggestions that greatly improved the presentation of the paper.

Appendix: Test examples with variable dimensions

1. Modified Rosenbrock function over $[-100, 100]^n$

$$f(x) = \sum_{i=1}^{n-1} \left[100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2\right].$$

2. Zakharov function over $[-9, 11]^n$

$$f(x) = \sum_{i=1}^{n} x_i^2 + \left(0.5 \sum_{i=1}^{n} i x_i\right)^2 + \left(0.5 \sum_{i=1}^{n} i x_i\right)^4.$$

3. Sphere function over $[-95, 105]^n$

$$f(x) = \sum_{i=1}^{n} x_i^2.$$

4. Schwefel function 2.22 over $[-10, 8]^n$

$$f(x) = \sum_{i=1}^{n} |x_i| + \prod_{i=1}^{n} |x_i|.$$

5. Schwefel function 1.2 over $[-10, 8]^n$

$$f(x) = \sum_{i=1}^{n} \left(\sum_{j=1}^{i} x_j\right)^2$$

6. Schwefel function 2.21 over $[-100, 80]^n$

$$f(x) = \max\{|x_i| : i = 1, \ldots, n\}.$$

7. Schwefel function 2.26 over $[-500, 600]^n$

$$f(x) = -\sum_{i=1}^{n} x_i \sin\left(\sqrt{|x_i|}\right) + 418.98288727n.$$

8. Step function over $[-100, 200]^n$

$$f(x) = \sum_{i=1}^{n} ([x_i + 0.5])^2.$$

9. Generalized Rastrigin function over $[-6.12, 5.12]^n$

$$f(x) = \sum_{i=1}^{n} \left[x_i^2 - 10\cos(2\pi x_i) + 10\right].$$

10. Ackley function over $[-31, 33]^n$

$$f(x) = -20\exp\left\{-0.2\left[\sum_{i=1}^{n} x_i^2/n\right]^{1/2}\right\} - \exp\left\{\sum_{i=1}^{n}\cos(2\pi x_i)/n\right\} + 22.7182818$$

11. Modified Griewank function over $[-590, 600]^n$

$$f(x) = 1 + \sum_{i=1}^{n} x_i^2/4000 - \Pi_{i=1}^{n}\cos(x_i/\sqrt{i}).$$

12. Another modified Griewank function over $[-590, 600]^n$

$$f(x) = \sum_{i=1}^{n} x_i^2/4000 - \prod_{i=1}^{n}\left[2 + \cos\left(x_i/\sqrt{i}\right)\right]/3 + 1.$$

13. Locatelli's modification #2 of Griewank function over $[-590, 600]^n$

$$f(x) = \sum_{i=1}^{n} x_i^2/4000 - \sum_{i=1}^{n}\ln\left[2 + \cos\left(x_i/\sqrt{i}\right)\right] + n\ln 3.$$

14. Locatelli's modification #3 of Griewank function over $[-590, 600]^n$

$$f(x) = \sum_{i=1}^{n} x_i^2/4000 - \sum_{i=1}^{n}\ln\left[2 + \cos\left(\sum_{j=1}^{n} A_{ij}x_j\right)\right] + n\ln 3,$$

where $A_{ij} = 1$ if $i \neq j$, and $A_{ii} = n + 1$.

15. Neumaier rational function over $[-100, 100]^n$

$$f(x) = 1 + f_k, \text{ if } x = x^{(k)} \text{ for some } k$$
$$1 + \sum_{k=1}^{n}(2f_k + r_k(x))/r_k(x)^2 / \sum_{k=1}^{n} 2/r_k(x)^2 \text{otherwise},$$

where for i, j, k = 1, …, n

$$x_i^{(k)} = 0.1\,(k-1) + (i-1)\sin(i-1),$$
$$f_k = -\cos\left(0.1(k-1)\right),$$
$$R_{ij}^{(k)} = i\,\delta_{ij},$$
$$r_k(x) = \left\|R^{(k)}\left(x - x^{(k)}\right)\right\|^2.$$

References

1. Alefeld, G., Herzberger, J.: Introduction to Interval Computations. Academic Press, New York (1983)
2. Ali, M.M., Torn, A., Viitanen, S.: A direct search variant of the simulated annealing algorithm for optimization involving continuous variables. Comput. Oper. Res. **29**, 87–102 (2002)
3. Bilbro, G.L., Snyder, W.E.: Optimization of functions with many minima. IEEE Trans. Syst. Man Cybern. **21**(4), 840–849 (1991)
4. Clausen, J., Zilinskas, A.: Subdivision, sampling, and initialization strategies for simplical branch and bound in global optimization. Comput. Math. Appl. **44**, 943–955 (2002)
5. Csallner, A.E.: Lipschitz continuity and the termination of interval methods for global optimization. Comput. Math. Appl. **42**, 1035–1042 (2001)
6. Csallner, A.E., Csendes, T.: The convergence speed of interval methods for global optimization. Comput. Math. Appl. **31**, 173–178 (1996)
7. Cvijovic, D., Klinowski, J.: Tabu search: an approach to the multiple minima problem. Science **267**, 664–666 (1995)
8. Falk, J.E., Soland, R.M.: An algorithm for separable nonconvex programming problems. Manag. Sci. **15**, 550–569 (1969)
9. Floudas, C.A., Pardalos, P.M.: A Collection of Test Problems for Constrained Global Optimization Algorithms. Springer, Berlin (1990)
10. Glover, F.: Tabu search—part I. ORSA J. Comput. **1**(3), 190–206 (1989)
11. Goldberg, D.E.: Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley, Reading (1989)
12. Hansen, E.R.: Global Optimization Using Interval Analysis. Marcel Dekker, NY (1992)
13. Hedar, A.R., Fukushima, M.: Derivative-free filter simulated annealing method for constrained continuous global optimization. J. Glob. Optim. **35**, 521–549 (2006)
14. Horst, R.: An algorithm for nonconvex programming problems. Math. Program. **10**, 312–321 (1976)
15. Horst, R., Tuy, H.: Global Optimization, Deterministic Approaches. Springer, Berlin (1990)
16. Ichida, K., Fujii, Y.: An interval arithmetic method for global optimization. Computing **23**, 85–97 (1979)
17. Kearfott, R.B.: A review of techniques in the verified solution of constrained global optimization problems. In: Kearfott, R.B., Kreinovich, V. (eds.) Applications of Interval Computations (El Paso, TX), Applied Optimization, vol. 3, pp. 23–60. Kluwer, Dordrecht (1996)
18. Kirkpatrick, S., Gelatt, C.D. Jr., Vecchi, M.P.: Optimization by simulated annealing. Science **220**, 671–680 (1983)
19. Michalewicz, Z.: Genetic Algorithms + Data Structures = Evolution Programs. 3rd edn. Springer, Berlin (1996)
20. Moore, R.E.: Methods and Applications of Interval Analysis. SIAM Publication, Philadelphia (1979)
21. Moore, R.E., Kearfott, R.B., Cloud, M.J.: Introduction to Interval Analysis. SIAM Publication, Philadelphia (2009)
22. Neumaier, A.: Interval Methods for Systems of Equations. Encyclopedia of Mathematics and its Applications 37. Cambridge University Press, Cambridge (1991)
23. Pardalos, P.M., Romeijn, E.: Handbook of Global Optimization—Volume 2: Heuristic Approaches. Kluwer, Dordrecht (2002)
24. Pedamallu, C.S., Ozdamar, L., Csendes, T., Vinko, T.: Efficient interval partitioning approach for global optimization. J. Glob. Optim. **42**, 369–384 (2008)
25. Ratscheck, H., Rokne, J.: New Computer Methods for Global Optimization. Wiley, New York (1988)
26. Siarry, P., Berthiau, G.: Fitting of Tabu Search to optimize functions of continuous variables. Intern. J. Numer. Methods Eng. **40**(13), 2449–2457 (1997)
27. Skelboe, S.: Computation of rational interval functions. BIT **14**, 87–95 (1974)
28. Storn, R., Price, K.: Differential evolution—a simple and efficient heuristic for global optimization over continuous spaces. J. Glob. Optim. **11**, 341–359 (1997)
29. Sun, M., Johnson, A.W.: Interval branch and bound with local sampling for constrained global optimization. J. Glob. Optim. **33**, 61–82 (2005)
30. Van Voorhis, T.: A global optimization algorithm using Lagrangian underestimates and the interval Newton method. J. Glob. Optim. **24**, 349–370 (2002)
31. Vavasis, S.A.: Nonlinear Optimization: Complexity Issues. Oxford University Press, New York (1991)
32. Yao, X., Liu, Y., Lin, G.: Evolutionary programming made faster. IEEE Trans. Evol. Comput. **3**, 82–102 (1999)
33. Zhang, X., Liu, S.: Interval algorithm for global numerical optimization. Eng. Optim. **40**, 849–868 (2008)