Acta Cybernetica **14** (1999) 1-.

Improving Storage Handling of Interval Methods for Global Optimization *

Csallner A. E. †

Abstract

Global nonlinear optimization problems can be solved by interval subdivision methods with guaranteed reliability. These algorithms are based on the branch-and-bound principle and use special storage utilities for the paths not pruned from the search tree yet. In this paper the possibilities for the kinds of applied storage units are discussed.

If no ordering is kept in the storage unit then the dependence of the number of operations demanded by the storage on the iterations completed is quadratic in worst case. On the other hand, ordering the elements as it is necessary for choosing new elements from the storage unit for backtracking, the worst case for the number of storage operations done to the k-th iteration has the magnitude $k \log k$. The hybrid method defined in this paper satisfies the same complexity properties. It is also proved that the $k \log k$ magnitude is optimal.

1 Global Optimization and Interval Methods — Introduction

The global optimization problem can be defined in general as follows:

$$\min_{x \in X} f(x) \tag{1}$$

where X is a — possibly multidimensional — interval. If we denote the set of real intervals by $\mathbb{I}$ and $f : \mathbb{R}^n \to \mathbb{R}$ is the objective function of the problem, then $X \in \mathbb{I}^n$. Note, that a great class of real-life bound-constrained global optimization problems are covered by (1), e.g., problems where the parameters are given with tolerances or if the optimizers are supposed to be inside a parameter region [2].

Problem (1) can be solved with verified accuracy with the aid of interval methods (see, e.g., [1, 5, 6, 7, 8]). These methods are based on the well-known branch-and-bound principle. Thus, a search tree is built where the whole search region — the

*This work has been supported by the Grants OTKA T017241, F025743, and FKFP 0739/1997.

†Department of Computer Science, Juhász Gyula Teacher Training College, 6725 Szeged, Boldogasszony sgt. 6, Hungary, e-mail: `csallner@jgytf.u-szeged.hu`

interval X — is the root and the particular levels consist of subintervals which are partitions of their parents in the tree. Those branches that cannot be pruned have to be stored for later treatment. The kind of the storage method used can be of great importance in performance if the increase of the number of intervals to be stored is considerable.

The following outlined algorithm is a model algorithm for interval subdivision methods for global optimization.

Algorithm 1 Model algorithm for interval subdivision methods for global optimization.

Step 1 Let S be an empty storage unit, the actual box $A := X$, and the iteration counter $k := 1$.

Step 2 Subdivide A into finite number $s \geq 2$ of subintervals A_i satisfying $A = \cup A_i$ so that $\mathrm{int(A_i)} \cap \mathrm{int(A_j)} = \emptyset$ for all $i \neq j$ where 'int' denotes the interior of a set.

Step 3 Let $S := S \cup \{A_i\}$.

Step 4 Discard certain elements from S.

Step 5 Choose a new $A \in S$ and delete it from S, $S := S \setminus \{A\}$.

Step 6 While termination criteria do not hold let $k := k + 1$ and go to Step 2.

Steps 3, 4, and 5 are using S, and their running time depends obviously on the storage handling of the algorithm. In the following we shall concentrate on these parts of the algorithm.

2 Worst Cases in Storage Handling of Interval Subdivision Methods

Because the efficiency of a branch-and-bound method depends highly on which branch, i.e., stored element is chosen as next to be treated, two basically different principles can be applied to handle the list. The first is to keep the list ordered and always pick up the first (or last) element in that ordering, while the second is to let the list be unordered and search for the next element in each step a new one is needed. The former saves computational time at picking up the elements, the latter at storing them. The time necessary for storage handling can be calculated in both cases.

We shall assume that Step 4 is not involved in the considered algorithm. In practice, this is the case in most implementations because cleaning up the stored elements is usually done when choosing a new A or before storing the new A_i intervals.

If the stored elements are unordered then Step 3 consumes some $c_1 s$ operations, and Step 5 some $c_2|S|$. If there is an ordering present which provides a constant time operation for finding the new element A in Step 5 then with a most efficient method the newly arisen s subintervals are stored in some $c_3 s \log|S|$ time and the new actual interval A is delivered in constant c_4 time. We summarize these results in the following lemma.

Lemma 1 *The worst case time complexity of storage handling of Algorithm 1 is*

$$c_1 s + c_2|S| \tag{2}$$

if S is unordered, and

$$c_3 s \log|S| + c_4 \tag{3}$$

if S is ordered. The results apply to a single iteration.

The algorithm keeps running for some k_0 iteration steps, and our goal is now to determine the total time consumption of the storage handling for the whole running time of the algorithm.

2.1 Unordered Storage Handling

If the elements are unordered then we can simply consider the storage unit as a sequential list. The next theorem says that in this case the number of storage operations depends quadratically on the iterations completed.

Theorem 2 *If S of Algorithm 1 is unordered and the process of finding a new actual interval A depends on a certain property of the list elements then the time complexity of storage operations up to the k_0-th iteration is*

$$T(s, k_0) = \mathcal{O}(sk_0^2). \tag{4}$$

Proof. From (2) of Lemma 1 it follows that a single iteration step uses $c_1 s + c_2|S|$ operations for appending the newly arisen intervals to the list and to find a new actual interval, where c_1 and c_2 are independent constants. The total number of storage operations done till the k_0-th iteration is hence

$$T(s, k_0) = \sum_{k=1}^{k_0} (c_1 s + c_2|S|). \tag{5}$$

Let us check how Algorithm 1 works. After the first iteration at most s elements are put onto the empty list, and one of them is picked up as the new actual interval in the same iteration. Thus, the number of list elements becomes $s - 1$. The list grows in every iteration by at most $s - 1$. Hence, at the end of some k-th iteration in worst case the number of list elements equals

$$|S| = k(s - 1). \tag{6}$$

Now putting (5) and (6) together we have

$$T(s,k_0) = \sum_{k=1}^{k_0}(c_1 s + c_2 k(s-1)) = c_1 s k_0 + c_2(s-1)\frac{k_0^2+k_0}{2}, \tag{7}$$

delivering (4) of the assertion. □

Note, that the proof does not presume the way of either subdividing or choosing a new actual box. Hence, all methods covered by the model Algorithm 1 obey Theorem 2.

2.2 Ordered Storage Handling

The following result is only sharp if in worst case more than a number of operations linear in the storage unit's cardinality is needed to keep the elements in the storage unit ordered.

Theorem 3 *If S of Algorithm 1 is kept ordered and the selection of the new actual intervals presume considering the elements' ordering then the time complexity of storage handling up to iteration k_0 is*

$$T(s,k_0) = \mathcal{O}(sk_0(\log s + \log k_0)). \tag{8}$$

Proof. (3) of Lemma 1 says that storage handling costs $c_3 s \log|S| + c_4$ in each iteration, where c_3 and c_4 are independent constants. Hence, summing this to the k_0-th iteration we get

$$T(s,k_0) = \sum_{k=1}^{k_0}(c_3 s \log|S| + c_4). \tag{9}$$

On the other hand, from (6) of the proof of Theorem 2 (9) can be extended to

$$\begin{aligned} T(s,k_0) &= \sum_{k=1}^{k_0}(c_3 s \log(k(s-1)) + c_4) = & (10)\\ &= c_3 s \sum_{k=1}^{k_0}\log k + c_3 s k_0 \log(s-1) + c_4 k_0. & (11)\end{aligned}$$

The first term of (11) can be bounded from above using the following inequality:

$$\begin{aligned} \sum_{k=1}^{k_0}\log k &\le \int_1^{k_0+1}\log x \; dx = \frac{1}{\ln a}[x\ln x - x]_1^{k_0+1} = & (12)\\ &= \frac{1}{\ln a}(k_0+1)\ln(k_0+1) - \frac{k_0}{\ln a}, & (13)\end{aligned}$$

where a denotes the base of the logarithm function in question. Note, that its value cannot influence the magnitude of the formula.

Substituting (13) into (11) provides the magnitudes that had to be proved. □

If s is considered as a constant then till finishing the k_0-th iteration the magnitude of storage handling's time consumption is $k_0 \log k_0$ in worst case, provided that any element can be inserted to the ordered storage unit in logarithmic time.

However, the time consumption's dependence on s is higher by a $\log s$ factor than in the unordered case (see Theorem 2), though this might not mean heavy differences by the usually small values of s.

2.3 Hybrid Storage Handling

A further storage handling method can be used which has not been mentioned here yet. In reality, this is not a new one but a mixture of those from Subsection 2.1 and 2.2. The idea is [4] to keep some p_0 elements from the storage unit ordered. These elements have to be the first ones regarding to the ordering of the whole. Hence, the new actual interval can always be chosen as the first of the ordered part. When inserting newly arisen intervals, one of them can supply for the ordered part. Otherwise a new element for the ordered part can be searched for in the unordered part. In worst case it can occur that for a substantial number of iterations the ordered part can only be refilled from the unordered part consuming $\mathcal{O}(|S| - p_0)$ time in each iteration.

Therefore, let us consider the following modification; we shall fix the value of p_0 and store only the first p ($1 \le p \le p_0$) elements ordered. The remaining $|S| - p$ elements are stored in a simple list.

Thus, every time a new interval is needed, it can be reached in constant time, since the first element of the ordered part does it. The other direction, namely, storing new elements is a little bit more difficult. For each new element it is checked whether it can be inserted into the ordered part of S. If it is the case, the element is inserted. If $p = p_0$ held before the insertion, the last element is moved to the unordered part. If the new element is greater at the present ordering than any from the ordered part — assuming the ordering is increasing — then it is pushed simply onto the unordered part. This procedure can be done in some $c_5 \log p_0$ time in worst case. Since there are at most s new elements to be inserted, the total amount of time needed for the storage operations is

$$c_5 s \log p_0 + c_6 \tag{14}$$

together with the constant number of operations for picking up the new actual interval. If p_0 was considered as an independent constant, (14) could lead to the conclusion that for an arbitrary number k_0 of iterations the time complexity of storage operations is linear in the variable k_0. In worst case, however, the ordered part can become empty forcing the new element to be chosen from the unordered part. Theoretically this can occur arbitrary many times. Thus, we should enhance the performance, namely, by doing the following two things:

1. We compensate the missing elements of the ordered part only if the ordered part runs empty, and then it is filled up with p_0 elements from the unordered part.

2. We do not fix p_0 directly as a constant, but say that it is always a κ proportion of $|S|$, where κ is a constant.

We shall call this algorithm the *hybrid* method. The time complexity of this method is described in Theorem 4.

Theorem 4 *Let us implement Algorithm 1 with the storage handling described above as the hybrid method. Then the time complexity of storage handling operations is*

$$T(s, k_0) = \mathcal{O}(k_0 \log(sk_0)) \tag{15}$$

in worst case.

Proof. Upon the assumptions the elements are stored in two disjunct units, one is ordered, the other is not. If the number of all the elements is $|S|$ then the ordered part consists of at most $p_0 = \kappa|S|$ elements, where $0 < \kappa < 1$ hold.

The procedure of storage handling involves two stages. In worst case, the first stage is done through p_0 iterations, i.e., until the ordered part becomes empty. The next stage consists of a single iteration, where the ordered part is rebuilt.

In the second stage filling up the ordered part is done as follows. First of all, the first p_0 elements are ordered which needs

$$t_1 = c_5 p_0 \log p_0 \tag{16}$$

time. After this, the remaining $|S| - p_0$ elements have to be inserted into the ordered part if possible, each with a $\log p_0$ time complexity. This can be done with the following time consumption:

$$t_2 = c_6(|S| - p_0) \log p_0. \tag{17}$$

¿From (6) of Theorem 2 we know that in worst case the number of list elements after the k_0-th iteration is $|S| = k_0(s-1)$. Let us apply this to (17) and add it to (16):

$$t = c_5 p_0 \log p_0 + c_6(k_0(s-1) - p_0) \log p_0. \tag{18}$$

It is known that at the iteration in question p_0 equals $\kappa|S|$ with a given κ. But citing (6) of the proof of Theorem 2 again we have

$$p_0 = \kappa|S| = \kappa k_0(s-1) \tag{19}$$

Let us use this to (18):

$$t = (c_5\kappa + c_6(1-\kappa))k_0(s-1)\log(\kappa k_0(s-1)) \tag{20}$$

These t operations only have to be done periodically after each p_0 iterations, where p_0 grows monotonously. Applying amortized analysis t can be apportioned among the next p_0 iterations to get the time complexity of storage handling for a single iteration in average. If for this the actual $|S|$ is used, we get an underestimation due to the increasing value of $|S|$ and hence p_0. Thus, for the time complexity the following holds:

$$k_0^{-1}T(s,k_0) \leq \frac{t}{p_0} + c_7 = \frac{t}{\kappa k_0(s-1)} + c_7, \tag{21}$$

where c_7 is the time consumption of the first stage. Applying this to the form for t calculated in (20) we have

$$k_0^{-1}T(s,k_0) \leq \frac{c_5\kappa + c_6(1-\kappa)}{\kappa} \log(\kappa k_0(s-1)) + c_7, \tag{22}$$

delivering the magnitude of (15) after k_0 iterations. □

The result of Theorem 4 is a nice enhancement in storage handling. To reach it we had to make two further assumptions previous to the theorem. None of them can be left away unless damaging the bound of (15). Namely, if the ordered part is supplied continually then in worst case the time complexity becomes the same as in the unordered case (see Theorem 2). On the other hand, if p_0 is a value independent from $|S|$ then the amortized analysis leaves a term containing k_0 on first degree in the formula also resulting in a quadratic time complexity in k_0 similar to (4).

¿From the three investigated methods the dependence on s is far the best with its $\log s$ complexity. This magnitude means that in practice — where $s > 8$ hardly ever occurs — the influence of s is unimportant.

Moreover, for the dependence on the number k_0 of iterations the following theorem holds.

Theorem 5 *For the time complexity dependence on the number k_0 of iterations the magnitude $k_0 \log k_0$ is optimal in worst case.*

Proof. It will be proved that if it was not optimal then an algorithm could be given to order n elements in less than $\mathcal{O}(n \log n)$ time in worst case.

Let the n data to be ordered be denoted by $a_1, a_2, \ldots, a_n$. We can determine the $max := \max_{i=1,\ldots,n} a_i$ value in linear time, and let a be greater regarding to the ordering than max. Then the algorithm is the following. We take a list handling method for interval subdivision methods which has a smaller time complexity than $k_0 \log k_0$ provided the algorithm is stopped after k_0 iterations. Now we place the data into the storage unit and begin to select elements from it as the ordering desires it. Instead of each element removed at least two further instances of the value a is put into the storage unit. After n iterations the original data are taken from the storage unit in the right order in less than $n \log n$ time which is a contradiction. □

Corollary 6 *The ordered and hybrid list handling methods for interval branch-and-bound algorithms are both optimal.*

2.4 List Handling of Hansen Methods

In the previous three subsections it has been assumed that the sequence of storing and recalling elements of the storage unit have not necessarily the same order. However, E.R. Hansen's subdivision method [5, 8] selects the oldest element waiting in the storage unit — which is a simple FIFO list in this case — in Step 5 of Algorithm 1, thus, the time complexity for a single iteration is constant. The time complexity after k_0 iterations is linear in the variable k_0. The reason for using the algorithm of Hansen yet quite rarely is that its convergence speed is in best case the same (see [3]) as of all efficiently converging interval subdivision methods in worst case.

3 Best Cases and Concluding Remarks

It is obvious that no storage handling can perform better than linearly depending on the number of iterations since there must be a few storage handling acts in every iteration. A straight consequence of this fact and of the properties discussed in Subsection 2.4 is that the Hansen methods achieve this linear dependence.

Upon the worst cases the unordered and mixed storage handling methods are the worst with their quadratic dependence. For the unordered handling this is the best case, as well, because looking up all list elements for choosing the new actual box cannot be avoided, at all. For the mixed handling it can occur that a newly arisen subinterval can always immediately be inserted into the ordered part consuming constant time in k_0. Thus, the best case behavior shows linear dependence.

The same thoughts lead to the statements about the ordered and hybrid handling methods. These methods both have linear time complexity in best case concerning the number of iterations made.

The optimal storage handling time complexity is $k_0 \log k_0$ — where k_0 denotes the iteration of termination — for interval branch-and-bound methods if the order of the sequence of resulting elements is not necessarily the same as that of the recalled elements.

This optimal complexity is obtained for the ordered storage types and the hybrid method.

If the algorithm itself provides the ordering of new elements then the worst case is the same as the best case, i.e., linear regarding to the iterations completed (Hansen algorithm).

The best and worst cases for the rest of the methods are summarized below:

Method	**Best Case**	**Worst Case**
Unordered	k_0^2	k_0^2
Ordered	k_0	$k_0 \log k_0$
Mixed	k_0	k_0^2
Hybrid	k_0	$k_0 \log k_0$

Note, that the worst case of the ordered storage handling is only true if a best implementation, i.e., a balanced search tree is used. Otherwise the worst case can also grow to k_0^2 as for the unordered case.

Average cases cannot be treated simply deriving them from the best and worst cases, respectively, since the interval branch-and-bound methods can be applied to every programmable function and thus their influence to the storage's behavior cannot be predicted in general. Numerical tests are in progress but they are not at hand at the present.

References

[1] Alefeld G. and Herzberger J. (1983), *Introduction to Interval Computations*, Academic Press, New York.

[2] Csallner A.E. (1993), *Global Optimization in Separation Network Synthesis*, Hungarian Journal of Industrial Chemistry, 21, pp. 303–308.

[3] Csallner A.E. and Csendes T. (1995), *Convergence Speed of Interval Methods for Global Optimization and the Joint Effects of Algorithmic Modifications*, IMACS/ GAMM SCAN-95 Conference, p. 33, Wuppertal, Germany, September 26-29, 1995.

[4] Csendes T. and Pintér J. (1993), *The Impact of Accelerating Tools on the Interval Subdivision Algorithm for Global Optimization*, European Journal on Operational Research, 65, pp. 314–320.

[5] Hansen E.R. (1992), *Global Optimization Using Interval Analysis*, Marcel Dekker, New York.

[6] Kearfott R.B. (1996), *Rigorous Global Search: Continuous Problems*, Kluwer, Dordrecht.

[7] Moore, R.E. (1966), *Interval Analysis*, Prentice Hall, Englewood Cliffs NJ.

[8] Ratschek H. and Rokne J. (1993), *Interval Methods*, In: Handbook of Global Optimization, Horst R. and Pardalos P.M. (eds.), Kluwer, Dordrecht, pp. 751–828.