J Glob Optim (2013) 56:821–844
DOI 10.1007/s10898-011-9771-5

On estimating workload in interval branch-and-bound global optimization algorithms

José L. Berenguel · L. G. Casado · I. García · Eligius M. T. Hendrix

Received: 30 July 2011 / Accepted: 13 August 2011 / Published online: 6 September 2011

Abstract In general, solving Global Optimization (GO) problems by Branch-and-Bound (B&B) requires a huge computational capacity. Parallel execution is used to speed up the computing time. As in this type of algorithms, the foreseen computational workload (number of nodes in the B&B tree) changes dynamically during the execution, the load balancing and the decision on additional processors is complicated. We use the term *left-over* to represent the number of nodes that still have to be evaluated at a certain moment during execution. In this work, we study new methods to estimate the *left-over* value based on the observed amount of pruning. This provides information about the remaining running time of the algorithm and the required computational resources. We focus on their use for interval B&B GO algorithms.

Keywords Global optimization · Interval Arithmetic · Branch-and-Bound · Workload prediction · Parallel algorithms

This work has been funded by grants from the Spanish Ministry of Science and Innovation (TIN2008-01117), Junta de Andalucía (P08-TIC-3518), in part financed by the European Regional Development Fund (ERDF). Eligius M.T. Hendrix is a fellow of the Spanish "Ramon y Cajal" contract program, co-financed by the European Social Fund.

J. L. Berenguel
TIC 146: "Supercomputing–Algorithms" Research group, University of Almería, Almería, Spain
e-mail: jlberenguel@gmail.com

L. G. Casado (✉)
Computer Architecture and Electronics Department, University of Almería, Almería, Spain
e-mail: leo@ual.es

I. García · E. M. T. Hendrix
Computer Architecture Department, University of Málaga, Málaga, Spain
e-mail: igarciaf@uma.es

E. M. T. Hendrix
e-mail: eligius.hendrix@wur.nl

1 Introduction

Branch-and-Bound algorithms are frequently used to solve NP-hard problems: Global Optimization (GO), Combinatorial Optimization, Mixed Integer Programming (MIP), etc. This type of problems can not be solved in polynomial time. Little, Murty, Sweeny and Karel gave the name of Branch-and-Bound [21]. The method was first proposed by A. H. Land and A. G. Doig in 1960 for Integer Linear Programming [19]. The B&B algorithm builds a tree of nodes which are selected, divided, evaluated and stored or rejected according to certain rules. The number of nodes to be evaluated depends on the instance to be solved and is not known in advance.

The computing time required to solve problems can be reduced using parallel computing. However, in order to obtain good performance, appropriate management of workload among processors is necessary. This requires estimation of the current and future computational burden to appropriately determine the computational resources applied to the problem at any time. We review methods for estimating the workload in B&B algorithms from literature and then develop new methods based on measured pruning. For a wide survey of parallel B&B algorithms see [11,32]. In multiple pool algorithms most authors discuss the prevention of parallel search anomalies; i.e. performing searches in a similar way to the sequential one, and moving promising nodes among processors in terms of lower bounds in minimization problems. Some examples can be found in [9,10,20,29]. As mentioned before, parallel B&B has been used widely to solve GO problems [8,27,28]. We are interested in interval parallel GO algorithms. Examples of general interval parallel GO algorithms can be found in [1,7,13–16,23,33]. These references show the relevance of knowing the workload in B&B but none of them attempts to estimate the pending work.

Knuth [18] was the first to propose a prediction method for backtracking search trees. He argued that his off-line method based on random probing can not be used for procedures like B&B. Cornuéjols et al. [4] give some intuitive reasons why Knuth's estimator may fail and describe a method to predict the size of a MIP B&B tree using three parameters: the maximum depth, the widest level and the first level at which the tree is no longer complete. Kilby et al. [17] give an extensive review on the topic and present two on-line methods for estimating the size of a backtracking search. The first method is based on a weighted sample of the branches visited by chronological backtracking. The second method uses a depth-first search from left to right. It assumes that the unexplored right sub-tree will be similar to the explored left sub-tree. Recently, Özaltın et al. [26] study a method based on monitoring the development of the MIP gap to predict when the gap becomes zero and the search ends. Authors show experimentally that applying the sum of subtree gaps leads to better predictions than using the global gap in MIP problems. The methods we develop do not assume any shape nor regularity of the search tree and use the so-called rejection ratio obtained by the algorithm.

We use the term *left-over* to represent the number of nodes to be evaluated by the B&B algorithm until it is finished. In this work, we study new methods for estimating the *left-over* value based on the observed amount of pruning. This provides information about the remaining computational burden of the algorithm that can be used to dynamically allocate resources in a parallel computing environment. For the design of our methods we have taken into account the following two important characteristics for estimating the *left-over* value:

– To know whether or not a branch will reach a solution leaf has the same complexity as solving a GO problem. Therefore, this problem is NP-hard. This shows the difficulty to estimate the number of nodes that will be generated from a given one.

– An estimation method should be simple, in the sense that it should not be hard to compute, compared to running time of the underlying B&B algorithm.

The rest of the paper is organized as follows. Section 2 describes basic characteristics of B&B algorithms. In Sect. 3 we define several concepts related to the *left-over* which are useful to develop methods to estimate *left-over* in B&B algorithms. Section 4 describes two points of view on what we call node rejection ratios: per level of the search tree and based on information from previous iterations. Moreover, we study how to predict the maximum depth reached from a node. Three new methods to estimate the *left-over* in a B&B algorithm are presented in Sect. 5. Section 6 outlines the experiments to measure performance and studies experimental results. Finally Sect. 7 presents conclusions.

2 B&B algorithm

B&B algorithms are used to find optimal solutions in optimization problems. Their performance depends on the specification of the following ingredients:

– *Division*: Divide a node such that it is covered by the union of resulting nodes.
– *Bounding*: Compute bounds of the objective function value range for each node in order to decide on its rejection.
– *Selection*: Determine which node is the next one to be visited and possibly divided.
– *Rejection*: If a node is proved not to contain optimal solutions, it can be removed from the search tree. The bounds are used for that.
– *Termination rule*: Determines when the algorithm finishes.

We describe these rules in a B&B algorithm for solving GO problems. GO problems consist of finding the global minimum (or maximum) points of a function, that is:

Let $f : S \subset \mathbb{R}^n \to \mathbb{R}$ be a continuous function. We want to find the set of points x^* for which:

$$f(x^*) = \min_{x \in S} f(x) \tag{1}$$

In a point x^* the global minimum is attained. S is the feasible set or search region. Usually S is assumed to be compact, specifically we focus on a box constrained feasible area.

Algorithm 1 describes an interval B&B algorithm to solve Problem (1). F is the interval extension of f [12], Λ is the work pool, Ω the list of nodes that reach the termination criterion, and ϵ is the accuracy of the solution nodes based on their width. Algorithm 1 runs until the work pool, Λ, is empty (line 4). A node is selected from the work pool (line 5). As selection rule we use Best-first search, selecting the node with the best lower bound. This rule is also known as "best bound first". Using this rule, the number of generated nodes is smaller compared to other strategies when the algorithm searches all solutions [1]. The upper bound $\overline{f^*}$ of the optimum is updated by evaluating the middle point $m(X)$ of the selected node (line 6). In this work we use bisection as division rule, where the widest co-ordinate of X is bisected. Interval Arithmetic is used as bounding rule to compute the lower ($\underline{F}(X)$) and upper bound ($\overline{F}(X)$) of the nodes [12,22]. The rejection rule discards node X if $\overline{f^*} < \underline{F}(X)$ because X can not contain a global optimum solution (line 12). If $\overline{f^*}$ is updated, the algorithm examines the work and final pool to determine which nodes can be rejected (CutOffTest, lines 8 and 9, respectively). The width $w(X)$ of X is defined by the size of the widest co-ordinate of X. If $w(X) \leq \epsilon$, the node is saved in the final pool Ω and not divided anymore. If it remains in Ω at the end of the execution, it is considered a candidate to contain an optimal

Algorithm 1 B&B Global Optimization Algorithm

Funct BBGO_Algorithm(S, F, ϵ)

1. $\overline{f^*} = \overline{F}(S)$ — *Upper bound of the minimum* f^*
2. $\Lambda := \{S\}$ — *Work pool*
3. $\Omega := \emptyset$ — *Final pool*
4. **while** ($\Lambda \neq \emptyset$)
5. X:=Select a node from Λ — *Selection rule*
6. $\overline{f^*} := min\{\overline{f^*}, \overline{F}(m(X))\}$ — *Update* $\overline{f^*}$
7. **if** ($\overline{f^*} = \overline{F}(m(X))$) — *If* $\overline{f^*}$ *is updated*
8. $\Lambda := CutOffTest(\Lambda, \overline{f^*})$
9. $\Omega := CutOffTest(\Omega, \overline{f^*})$
10. $Divide(X, X^1, X^2)$ — *Division rule*
11. **for** $i := 1$ **to** 2
12. **if** ($\overline{f^*} < \underline{F}(X^i)$) **reject** X^i — *Rejection rule*
13. **elsif** ($w(X^i) \leq \epsilon$) $\Omega := \Omega \cup \{X^i\}$ — *Termination rule*
14. **else** $\Lambda := \Lambda \cup \{X^i\}$
15. **return** Ω

solution (line 13). Nodes that do not satisfy rejection or termination rules are inserted in the work pool for further processing (line 14).

Other B&B algorithms are similar to Algorithm 1. Therefore, the research presented in the sequel can be adapted to other B&B algorithms.

3 Definitions around the *left-over* concept

This section provides definitions of basic parameters and concepts that help to understand the problem of predicting the remaining workload in terms of nodes to be evaluated. Each node can be considered the root of a sub-tree. We use the term *offspring of a node* to refer to the total number of nodes in the sub-tree starting at this node.

Definition 1 $Offspr(X)$ is the number of nodes generated from X by a B&B algorithm; i.e. the number of nodes in the sub-tree having X as root.

We are interested in the total offspring of nodes in pool Λ during algorithm execution. We define this as:

Definition 2 $LeftOver(\Lambda)$ is the total offspring of nodes in Λ:

$$LeftOver(\Lambda) = \sum_{X \in \Lambda} Offspr(X).$$

The issue is that the exact number of nodes $Offspr(X)$ generated from X is known only after algorithm termination. However, for each node X in the search tree, one can determine an upper bound on the number of nodes in the sub-tree generated from it. We call this upper bound:

Definition 3 $CTree(X)$ is the number of nodes of the complete tree generated from X through successive divisions until reaching accuracy ϵ.

The value of $CTree(X)$ depends on the width of X, dimension of the problem (n), used accuracy (ϵ) and division rule (bisection in our case). We denote by L the maximum depth

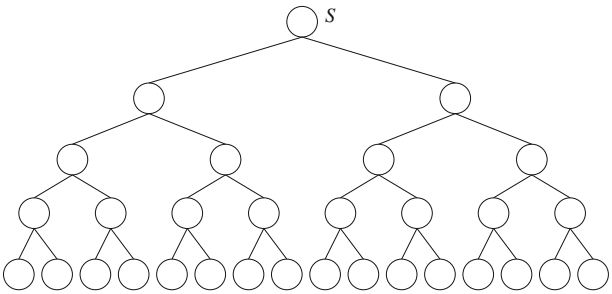

Fig. 1 Complete binary tree ($L = 4$). $LeftOver^U(\Lambda) = CTree(S) = 2^1 + 2^2 + 2^3 + 2^4 = 30$

of the search tree to reach solution nodes. Every node requires n subdivisions to halve their width. The number of levels L of the binary tree can be determined from the number of times m to halve the width such as to obtain final solutions ($w(X) \leq \epsilon$) from the starting node S. If the search region has equal sized sides, the number m is determined by

$$\frac{w(S)}{2^m} \leq \epsilon.$$

Therefore,

$$m = \left\lceil \log_2 \left(\frac{w(S)}{\epsilon} \right) \right\rceil.$$

If all width w_i of the interval on variable x_i are equal, the depth of the B&B tree is $L = m \cdot n$. Otherwise one should specify $L = \lceil \sum_{i=1}^{n} \log_2 w_i(S) - \log_2 \epsilon \rceil$.

Let $l(X)$ represent the level of a node X, where $l(S) = 0$. In the algorithm, $CTree(X)$ is the number of nodes of the complete binary tree of depth $L - l(X)$:

$$CTree(X) = 2^1 + 2^2 + 2^3 + \cdots + 2^{L-l(X)}. \tag{2}$$

This can be written as:

$$CTree(X) = 2^{(L-l(X)+1)} - 2 = 2(2^{(L-l(X))} - 1).$$

Given a pool Λ in a B&B algorithm, one can dynamically determine an upper bound on $LeftOver(\Lambda)$, denoted by $LeftOver^U(\Lambda)$ as:

$$LeftOver^U(\Lambda) = \sum_{X \in \Lambda} CTree(X).$$

Figure 1 shows a complete binary tree of four levels and the $LeftOver^U(\Lambda)$ with $\Lambda = \{S\}$.

The value of $LeftOver^U(\Lambda)$ is a large overestimate of $LeftOver(\Lambda)$. The actual value of the $LeftOver(\Lambda)$ depends on how effective the rejection is done. Estimating the rejection behavior can help us to estimate the offspring of nodes. Figures 2, 3, 4 and 5 show an example of a B&B tree. Black nodes represent the current pool Λ. Shaded nodes are those rejected by the algorithm. Figure 2 shows an example of the execution of the B&B algorithm. Nodes above the dotted line are known at a certain step during B&B algorithm execution. The nodes below the dotted line depict future iterations and obviously, they are not known. For pool Λ in this example, the difference between $LeftOver^U(\Lambda)$ and $LeftOver(\Lambda)$ is 12.

At each iteration of the B&B algorithm, the upper bound of *left-over* gets closer to its value. Figure 3 shows how the difference has decreased to 6 due to the elimination of nodes with over-estimated offspring.

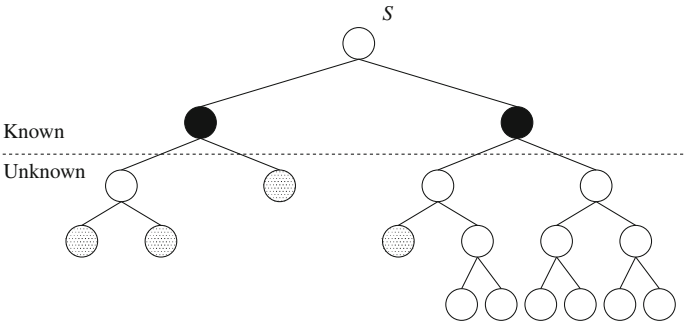

Fig. 2 Example 1 ($L = 4$). $LeftOver(\Lambda) = 16$, $LeftOver^U(\Lambda) = 28$

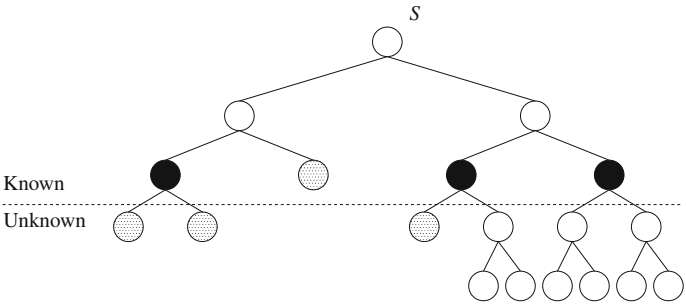

Fig. 3 Example 2 ($L = 4$). $LeftOver(\Lambda) = 12$, $LeftOver^U(\Lambda) = 18$

Let the estimated offspring of a node X be denoted by $Offspr^E(X)$. Our approach to estimate the *left-over* is to consider the sum of estimators of offspring of all the nodes in the current pool.

Definition 4 The estimated *left-over* value, denoted by $LeftOver^E(\Lambda)$, in a B&B Algorithm is the sum of the estimated offspring of all nodes in pool Λ:

$$LeftOver^E(\Lambda) = \sum_{X \in \Lambda} Offspr^E(X) \tag{3}$$

The goal is to construct good offspring estimators $Offspr^E(X)$ which are easy to calculate. One approach is to observe the amount of pruning done by the algorithm and we focus on that in the following section.

4 Rejection ratios and sub-tree depth

We study approaches to measure the amount of pruning done by the algorithm. These are used to develop offspring estimators. One way to consider pruning is to focus on the number of rejected nodes in each level of the B&B tree. Another way is to consider the rejection ratio achieved in previous iterations of the algorithm. Moreover, one can focus on the depth of a sub-tree rooted in each node. In this study, predictions are made every k iterations of the algorithm. We use j as prediction index, such that at $j \times k$ iterations a prediction is given.

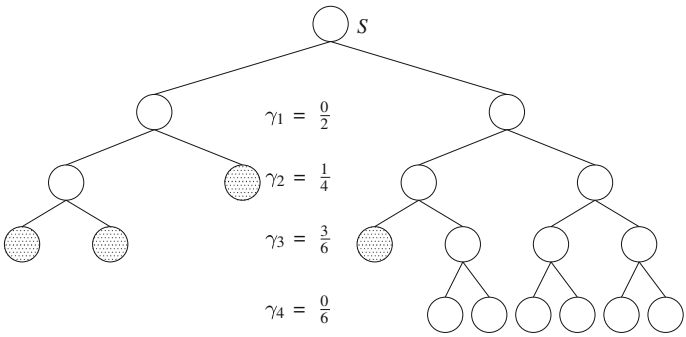

Fig. 4 Rejection ratios per level ($L = 4$), $Offspr(S) = 18$

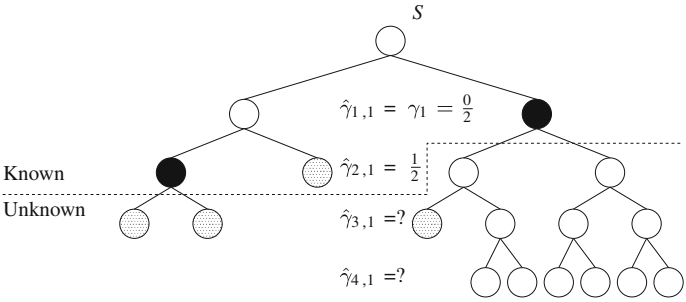

Fig. 5 Rejection ratios per level ($L = 4$, $q_1 = 3$)

4.1 Rejection ratio per level

One way to look at rejections within the search tree is to consider the rejection per level. We use index i to denote a level of the tree.

Definition 5 The rejection ratio per level γ_i, $i = 1, \ldots, L$ is the number of nodes rejected at level i divided by the number of nodes evaluated at level i. The series of numbers is called the γ-sequence.

Figure 4 illustrates the concept of the γ-sequence. The total number of nodes in the tree can be deduced from that:

$$Offspr(S) = 2^1 + 2^2(1-\gamma_1) + \cdots + 2^4(1-\gamma_1)(1-\gamma_2)(1-\gamma_3) = 18. \tag{4}$$

We will use the idea of Eq. (4) to estimate total number of nodes in the tree. For that, it is necessary to have good estimates of the γ-sequence. To estimate the rejection ratio during algorithm execution, the following concepts are introduced:

$E_{i,j}$: number of nodes evaluated at level i after $j \times k$ iterations.
$R_{i,j}$: number of nodes rejected at level i after $j \times k$ iterations.

Index q_j denotes the lowest level such that $E_{i,j} = 0$, $\forall i \geq q_j$. When the algorithm finishes $q_j = L + 1$. The observed rejection ratio at level i after $j \times k$ iterations is:

$$\hat{\gamma}_{i,j} = \frac{R_{i,j}}{E_{i,j}}, i = 1, \ldots, q_j - 1.$$

During algorithm execution, for these levels i where all nodes have been evaluated, $\gamma_i = \hat{\gamma}_{i,j}$. So after algorithm execution, the series $\hat{\gamma}_{i,j}$ gives the exact γ-sequence. Figure 5 shows an example where $q_1 = 3$. In this example $E_{2,1} = 2$ and $R_{2,1} = 1$. The challenge is to determine good estimates for the unknown (γ_3 and γ_4 in Fig. 5), or partially unknown (γ_2 in Fig. 5) values of γ_i.

4.2 Rejection ratio per iteration

Another way to look at rejections within the search tree is to measure rejections during the iterations. In each iteration, the B&B algorithm selects a node and divides it, rejecting or saving new nodes in final pool Ω or inserting them for further processing in pool Λ. We focus on the rejection behavior during the last k iterations. Using a similar notation as in Sect. 4.1, we define:

EI_j : number of nodes evaluated during the last k iterations. In this work EI_j is always $2k$, because two new nodes are generated and evaluated at each iteration using bisection.
RI_j : number of nodes rejected during the last k iterations.
FI_j : number of final nodes stored in Ω (see Algorithm 1) during the last k iterations.

The rejection ratio per iteration, denoted by φ_j, is the ratio between the number of rejected nodes plus final nodes and the number of evaluated nodes, during the last k of in total $j \times k$ iterations:

$$\varphi_j = \frac{RI_j + FI_j}{EI_j} \in [0, 1].$$

A value of $\varphi_j > 0.5$ means that new nodes are mostly rejected, so the size of the pool Λ decreases. A value of $\varphi_j < 0.5$ means that new nodes are mostly stored, so the size of the pool increases. A value of $\varphi_j = 0.5$ means that one node is rejected and one is stored, so the size of the pool stays the same. A similar reasoning applies for $\hat{\gamma}_{i,j}$.

4.3 Sub-tree depth prediction

In general, not all branches of the B&B algorithm reach the final level. Therefore, it is interesting to predict the maximum depth reached from a node to obtain a better offspring estimation.

The lower bound $\underline{F}$ on the objective function value is nondecreasing with the level [24]. Let X^F be the father node of X and X^G the grandfather node of X, such that $\underline{F}(X^G) \leq \underline{F}(X^F) \leq \underline{F}(X)$. Figure 6 outlines the idea of increasing lower bound with increasing level. For node X, a prediction of the depth of the subtree starting in X could be based on the increase measured of the lower bound from grandfather node to father node to X itself. The moment the observed slope cuts bound $\overline{f^*}$ can be used to estimate the depth. The idea is to estimate when the lower bound of the offspring of X will be greater than $\overline{f^*}$ and therefore the subtree finishes. Both observed slopes, i.e. the increase from grandfather to father and from father to X, can be used.

Using the observed slopes, the prediction of the subtree ending and the offspring dying out is given by Eqs. (5) and (6):

$$CutLevel_1(X) = \left\lceil \frac{\overline{f^*} - \underline{F}(X)}{\underline{F}(X) - \underline{F}(X^F)} \right\rceil + l(X), \tag{5}$$

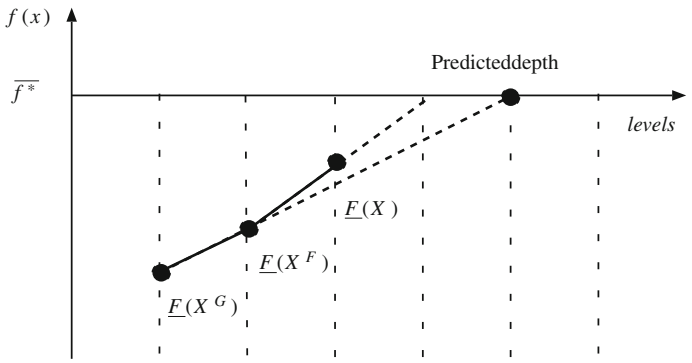

Fig. 6 Sub-tree depth prediction

$$CutLevel_2(X) = \left\lceil \frac{\overline{f^*} - \underline{F}(X^F)}{\underline{F}(X^F) - \underline{F}(X^G)} \right\rceil + l(X) - 1. \tag{6}$$

Using the largest one as long as it does not pass the depth L of the total tree gives estimate:

$$CutLevel(X) = \min\{\max(CutLevel_1(X), CutLevel_2(X)), L\}.$$

This means that if any of the slopes is 0 or if $CutLevel_1(X)$ and/or $CutLevel_2(X)$ are greater than L, then $CutLevel(X) = L$. A prediction of the depth of the tree rooted in X is

$$dp(X) = CutLevel(X) - l(X). \tag{7}$$

Notice that $dp(X)$ predicts the sub-tree depth generated from a node, but it does not give information about how many offspring will be generated and how offspring is distributed in the sub-tree.

5 *Left-over* estimators

The measured parameters $\hat{\gamma}_{i,j}, \varphi_j$ and $dp(X)$ can be used to develop offspring and *left-over* estimators. We describe and evaluate several.

5.1 PL-LEM (Per Levels *Left-Over* Estimation Method)

Let Λ_j be the state of the working pool at prediction moment j, that is after $k \times j$ iterations. Then $LeftOver(\Lambda_j)$ is the difference between $Offspr(S)$ and the number of nodes already evaluated at prediction j:

$$LeftOver(\Lambda_j) = Offspr(S) - \sum_{i=1}^{L} E_{i,j}. \tag{8}$$

Notice that using bisection, $\sum_{i=1}^{L} E_{i,j} = 2 \times k \times j$. Similar to Eq. (4) we can estimate the total tree size by

$$Offspr_j^E(S) = 2^1 + 2^2(1 - \hat{\gamma}_{1,j}) + \ldots + 2^L \prod_{i=1}^{L-1}(1 - \hat{\gamma}_{i,j}).$$

Notice that $\hat{\gamma}_{i,j}$ is only measured for $i = 1, \ldots, q_j - 1$. As the further development of the tree is unknown, we use $\hat{\gamma}_{i,j} = 0.5$ for $i = q_j, \ldots, L$. This means that the number of branches of the search tree is considered constant from level q_j until the last level. The corresponding estimate of the *left-over* is

$$LeftOver^E(\Lambda_j) = Offspr_j^E(S) - \sum_{i=1}^{L} E_{i,j}. \tag{9}$$

5.2 IG-LEM (Per k Iterations Global *Left-Over* Estimation Method)

IG-LEM is based on the rejection ratio φ_j per iteration discussed in Sect. 4.2. An exponential smoothing average is used for the rejection ratio φ_j measured during the last k iterations. Rejection estimator θ_j is the exponential smoothing average of sequence φ_j, i.e:

$$\theta_j = \alpha\theta_{j-1} + (1-\alpha)\varphi_j \in [0, 1]. \tag{10}$$

At the beginning, $\theta_0 = \varphi_0$. The value of α is between [0, 1]. We choose a value of 0.4 because it shows a good balance between smoothing and changes. The last evaluated θ_j is used to estimate $Offspr_j^E(X)$ in the following way. We apply θ_j to $CTree(X)$ (Eq. (2)) to obtain an offspring estimator for node X:

$$Offspr_j^E(X) = 2^1 + 2^2(1-\theta_j)^1 + 2^3(1-\theta_j)^2 + \cdots + 2^{L-l(X)}(1-\theta_j)^{L-l(X)-1}$$

which can be simplified to:

$$Offspr_j^E(X) = 2\frac{2^{L-l(X)}(1-\theta_j)^{L-l(X)} - 1}{2(1-\theta_j) - 1}. \tag{11}$$

Combined with (3) this gives estimator

$$LeftOver^E(\Lambda_j) = \sum_{X\in\Lambda_j} Offspr_j^E(X) = 2\sum_{X\in\Lambda_j} \frac{2^{L-l(X)}(1-\theta_j)^{L-l(X)} - 1}{2(1-\theta_j) - 1}. \tag{12}$$

Notice that (11) reflects the idea that the subtree of each node X has offspring in last level L.

5.3 IL-LEM (Per k Iterations Local *Left-Over* Estimation Method)

The IL-LEM method is similar to IG-LEM. The difference is that the predicted depth (7) of the sub-tree from node X is taken into account:

$$Offspr_j^E(X) = 2\frac{2^{dp(X)}(1-\theta_j)^{dp(X)} - 1}{2(1-\theta_j) - 1}. \tag{13}$$

Combining again (3) with subtree estimator (13), gives the *left-over* estimation

$$LeftOver^E(\Lambda_j) = \sum_{X\in\Lambda_j} Offspr_j^E(X) = 2\sum_{X\in\Lambda_j} \frac{2^{dp(X)}(1-\theta_j)^{dp(X)} - 1}{2(1-\theta_j) - 1}. \tag{14}$$

This method requires more computation than IG-LEM because it uses the determination of $dp(X)$ for each node X in the tree. Additionally, it requires the information on the lower bounds of father and grandfather node which has to be stored with the node. Nodes with the same $dp()$ value have the same $Offspr^E()$ value, which does not depend on their level in the search tree.

Table 1 Test functions

Problem	Ref.	n	gm	ϵ	L	$Offspr(S)$
Goldstein–Price	[30]	2	1	10^{-3}	24	101,668
Levy3	[30]	2	9	10^{-4}	36	337,786
Levy5	[30]	2	1	10^{-5}	42	299,656
Griewank 2	[31]	2	1	10^{-9}	82	109,390
Griewank 10	[31]	10	1	10^{-6}	310	616,446
EX1	[5]	2	1	10^{-6}	42	383,058
Branin	[30]	2	3	10^{-9}	68	146,358
Henriksen–Madsen 3	[14]	2	9	10^{-4}	36	335,896
Henriksen–Madsen 4	[14]	3	1	10^{-3}	42	216,292
Chichinadze	[6]	2	1	10^{-5}	46	697,304
Shekel 10	[30]	4	1	10^{-5}	80	8,487,156
Shekel 7	[30]	4	1	10^{-5}	80	6,939,346
Shekel 5	[30]	4	1	10^{-5}	80	313,096
Hartman 6	[30]	6	1	10^{-2}	42	877,002
Hartman 3	[30]	3	1	10^{-3}	30	454,568
Ratz-5	[30]	3	1	10^{-4}	54	2,359,306
Rosenbrock 10	[6]	10	1	10^{-5}	190	581,136
Colville	[3]	4	1	10^{-5}	84	1,211,542
Kowalik	[30]	4	1	10^{-3}	36	3,090,698
Neumaier 2	[25]	4	12	10^{-3}	48	21,399,102
Neumaier 3–10	[25]	10	1	10^{-2}	150	12,958,026
Ratz-8	[30]	9	1	10^{-4}	162	4,718,574

6 Numerical experiments

First we discuss the design of the experiments and then give and discuss the numerical results.

An Interval GO Algorithm is used to evaluate the described *left-over* estimation methods. We evaluate per level *left-over* estimator PL-LEM (based on (9)) and *left-over* estimators IG-LEM (based on (12)) and IL-LEM (based on (14)). Version 2.2.4 of C-XSC is used for Interval Arithmetic (http://www.math.uni-wuppertal.de/~xsc/).

To measure the performance of the methods, we designed three testbeds of experiments that are summarized in Table 1. For the first set of functions, f^* was given as the global upper bound $\overline{f^*}$, no accelerating devices were used [12,16] and predictions were made every $k = 1{,}000$ iterations. For the second and third sets of test functions, the global minimum value f^* was not provided. Furthermore, for the third set of functions the monotonicity test was used as accelerating device and predictions were made every $k = 100{,}000$ iterations. The following notation is used for column headers:

Problem	Problem name
Ref.	Reference with problem description
n	Problem dimension
gm	Number of global minimum points

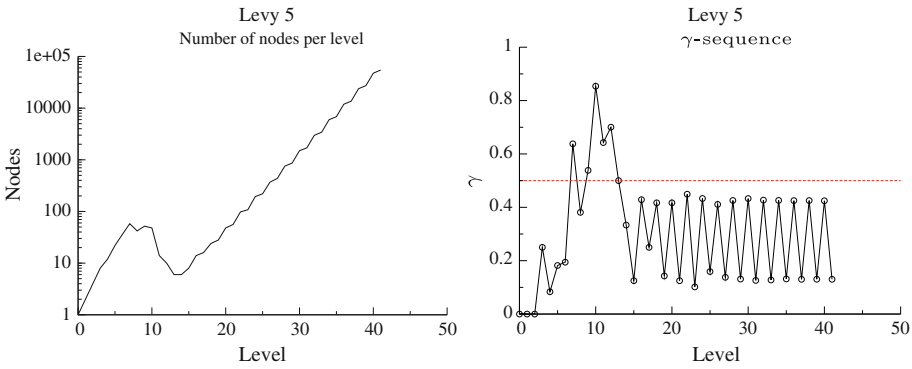

Fig. 7 Number of nodes at each level and γ-sequence for the Levy 5 problem, $\epsilon = 10^{-5}$

ϵ	The stopping criterion accuracy ($w(X) \leq \epsilon$)
L	Maximum level
$Offspr(S)$	Number of generated nodes.

The shape of the generated search tree depends on the instance and on the algorithm. To illustrate the introduced concepts, Fig. 7 gives graphs with the number of nodes per level and the γ-sequence for the Levy-5 test function. Graphs for other instances presented in Table 1 are given in Appendix A. Notice that the number of nodes per level and γ-sequence give similar information. The γ-sequence graphs have a line at $\gamma = 0.5$ to highlight that $\gamma_i < 0.5$ means an increase in the number of nodes from level i to level $i + 1$, $\gamma_i > 0.5$ means a decrease in the number of nodes and $\gamma_i = 0.5$ means that the number of nodes in level i and level $i + 1$ is the same. This can be seen in the graph showing the number of nodes per level, on the left hand side. It is interesting to observe the zigzag behavior of the γ-sequence. This is a consequence of using bisection; after each n splits the width of the box is halved giving another rejection behavior than in the intermediate levels.

To depict the *left-over* prediction made by the methods described in this paper, a graph is drawn for each instance of the first set where the case of Levy 5 is given in Fig. 8. On the axes one can find the iterations versus the *left-over* and its predictions. Figures 18–26 in Appendix B show the prediction for the other instances. Only values near the *left-over* value defined by Eq. (8) are shown. The line representing the *left-over* starts at $Offspr(S)$ and decreases with a slope of 2 as two nodes are evaluated in each iteration. The value of $Offspr(S)$ is given for all evaluated instances in Table 1.

The difference in search behavior for the different instances is shown by the figures in Appendix B. Families of test problems like Levy 3-Levy 5, Griewank 2-Griewank 10 and Henriksen–Madsen 3- Henriksen–Madsen 4 show a similar behavior in the search, even when the problems in the same family have a different number of global minimum points and/or dimension. Moreover, most of the test problems show an increasing number of nodes at final levels. This is due to the fact that the algorithm searches for all solutions.

The instances Griewank 2, Griewank 10, Branin and Chichinadze provide a specific behaviour; their lower bounds are equal to f^* for each node. If the second criterion is to select the node with the smallest upper bound, then a similar behavior of a Depth-First search is produced, causing IL-LEM and IG-LEM estimators to predict badly. On the other hand, the PL-LEM estimator performs almost perfect. If the second criterion is to select the oldest node first (LIFO), then IL-LEM and IG-LEM work as good as the PL-LEM estimator. We used LIFO as second selection rule in our experiments.

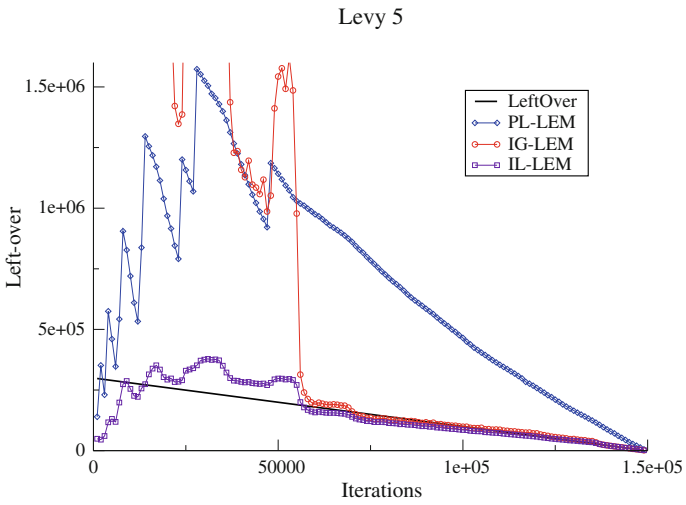

Fig. 8 Levy 5. $\epsilon = 10^{-5}$

To measure the performance of the *left-over* estimators, we define as performance indicator the Average Relative Prediction Error ($ARPE$) as:

$$ARPE = \frac{1}{N}\sum_{j=1}^{N} \frac{|LeftOver^{E}(\Lambda_j) - LeftOver(\Lambda_j)|}{LeftOver(\Lambda_j)},$$

where N is the total number of predictions made during an experiment. This number differs for each considered instance and experimental set up. Specifically, in the experiment we measure the value of $ARPE$ during different stages of the algorithm execution, which are defined by $0-20\%$, $20-40\%$, $40-60\%$, $60-80\%$ and $80-100\%$. The values of the best performing method are marked in bold. For each method, the prediction error $ARPE$ is shown in a separate table. In order to facilitate comparison of the three estimation methods, the instances are reordered within the testbeds.

The general rule looking at predictions is that if the measured past gives good information about the future, then prediction is easy. For our investigation this means that measured rejection ratios give adequate information. To highlight an extreme case, consider the Rosenbrock instance with updates of the upper bound. The PL-LEM prediction is doing extremely well, where the other predictors fail completely. Going into depth, the γ-sequence has values varying from 0 to 0.7 over the levels and are measured well by $\hat{\gamma}$. However, the iteration rejection rates are measured over all the tree and sometimes underestimate the average rejection highly and therefore overestimate the workload.

If we focus more in general on PL-LEM in Table 2 and its *ARPE* values, one observes that seven of the instances exhibit a prediction error less than 0.5 in most of the stages. For these test functions the $\hat{\gamma}$-sequence is relatively constant during the execution. This means that boxes at the same level have a similar rejection ratio value. Notice that if $\hat{\gamma}$ predicts the γ-sequence exactly, no prediction error is made, as occurs for the Griewank 2 and Griewank 10 functions. In contrast, a non-constant behavior of the $\hat{\gamma}$-sequence leads to big errors even in the last stage of execution.

Table 2 ARPE for PL-LEM prediction method

Problem	0–20%	20–40%	40–60%	60–80%	80–100%
Henriksen–Madsen 4	**26.50**	**32.80**	21.30	13.01	6.75
Goldstein–Price	12.35	4.25	0.93	**0.34**	1.31
Henriksen–Madsen 3	**2.89**	5.90	5.32	4.53	2.97
Levy 5	2.48	4.52	4.22	3.58	3.05
Levy 3	**2.43**	4.49	4.25	3.49	3.27
EX 1	**1.64**	2.75	2.54	2.23	2.42
Branin	**1.29**	2.82	3.39	2.64	0.79
Chichinadze	**1.19**	2.24	2.25	1.02	0.76
Griewank 2	0.23	0.09	**0**	**0**	**0**
Griewank 10	**0**	**0**	**0**	**0**	**0**
Hartman 6	**874.38**	221.11	83.80	27.18	10.71
Shekel 7	**130.06**	82.96	43.23	20.16	8.04
Shekel 5	126.49	65.45	29.68	16.90	7.95
Shekel 10	**125.23**	77.04	41.66	19.63	7.97
Colville	**19.92**	**15.20**	**14.59**	12.05	6.39
Hartman 3	**4.94**	5.74	4.55	3.89	3.50
Ratz-5	**0.57**	0.42	0.23	0.37	0.06
Rosenbrock 10	**0.12**	**0.05**	**0.05**	**0.05**	**0.06**
Kowalik	**7.82**	6.87	5.98	6.32	6.82
Ratz-8	0.68	**0.37**	**0.18**	**0**	**0**
Neumaier 3–10	**0.66**	**0.49**	**0.28**	**0.24**	0.40
Neumaier 2	**0.58**	**0.24**	**0.07**	**0.14**	**0.02**

Table 3 shows the ARPE value for IG-LEM. The IG-LEM method shows good predictions from (40–60%) of the execution. For many instances good estimation results occur, which are better at the final stage. One can observe that the prediction error in the last stage grows occasionally due to nodes reaching the final level of the tree.

Table 4 shows the ARPE value for IL-LEM. In general, the IL-LEM method outperforms the others. It obtains the best predictions in the last stage of execution (80–100%). Moreover, good predictions are made from (20–40%) of algorithm execution. Those values not marked in bold are quite similar to the best predictions apart from a small number of cases. The figures in Appendix B illustrate this behavior. IG-LEM and IL-LEM provide similar predictions in the last stages for most of the instances. In first stages, the prediction may differ considerably.

We systematically varied the algorithm to observe the quality of the prediction. First of all, the updates of $\overline{f^*}$ give an unpredictable aspect to the estimation, as one does not know in advance when the upper bound is updated and how this is influenced by the selection rule chosen. In several scenarios, we fixed $\overline{f^*} = f^*$ beforehand. The difference in predictability did not seem very big. A second aspect is to add a monotonicity test as rejection rule. The hypothesis is that this will increase predictability if during the search this rule provides a stationary behavior. What we observed is that the predictability did not increase, nor decrease systematically. Apparently, the prediction methods that are based on observed behaviour during the last iterations are in a sense robust to algorithm configuration change. This seems promising for the general applicability of such methods.

Table 3 ARPE for IG-LEM prediction method

Problem	0–20%	20–40%	40–60%	60–80%	80–100%
Henriksen–Madsen 4	2.10E+6	8.55E+4	1.19	0.48	0.01
Goldstein–Price	40.95	**0.18**	**0.33**	1.29	0.44
Henriksen–Madsen 3	124.30	7.85	0.54	0.20	0.01
Levy 5	143.89	6.78	**0.08**	**0.05**	0.11
Levy 3	62.08	5.93	**0.07**	**0.05**	0.15
EX 1	6.49	2.43	**0.30**	**0.26**	**0.13**
Branin	3.07	**2.17**	**0.76**	**0.54**	**0.31**
Chichinadze	9.88	3.22	1.66	**0.18**	**0.25**
Griewank 2	0.09	**0**	**0**	**0**	**0**
Griewank 10	1.10E+86	0.02	0.02	0.01	**0**
Hartman 6	7.02E+4	39.45	1.44	0.19	**0.04**
Shekel 7	1.19E+5	7.08	1.41	0.57	**0**
Shekel 5	8.98E+4	11.89	0.90	0.49	0.06
Shekel 10	4.83E+4	3.89	1.31	0.60	**0**
Colville	7.55E+9	153.75	32.03	7.71	3.21
Hartman 3	21.97	**0.27**	**0.25**	**0.12**	**0.09**
Ratz-5	181.73	**0.27**	**0.22**	**0.31**	**0**
Rosenbrock 10	1.23E+27	1.99E+5	4.30E+3	28.29	0.59
Kowalik	15.57	2.35	0.35	0.27	0.39
Ratz-8	**0.66**	1.18	1.21	0.03	**0**
Neumaier 3–10	4.43E+4	4.21	0.36	0.79	0.61
Neumaier 2	2.53	0.95	0.59	0.25	0.03

7 Conclusions and future work

Workload estimation during algorithm execution of a B&B method is a big challenge. This work presents new estimators of the *left-over* value which can be used to predict the remaining execution time. All these prediction methods have a low computational cost requiring just a few operations. They are based on the measurement of the rejection ratio per level of the search tree (the so-called γ-sequence) or per iterations during the execution. The prediction is based on the assumption that the rejection ratio measured during algorithm execution will be similar in the future. Specifically, we developed the PL-LEM predictor based on rejection per level and IG-LEM and IL-LEM based on measured rejection during the iterations. Experimental results show that in general, better predictions are obtained for methods based on the rejection ratio per iterations. They can do a reasonable good prediction after 40% of algorithm execution.

The inclusion of additional information in the methods improves the predictions but increases the computational cost. This is the case of the IL-LEM method, which uses a sub-tree depth predictor and obtains good *left-over* estimations. Additionally, the sub-tree depth predictor can be used to perform dynamic load balancing in parallel computing. The study of performance of this load balancing method will be done in future work. On the other hand, the PL-LEM method deserves a future study of inclusion of additional information to improve the predicted γ-sequence. The study of better indicators [2] of the quality of a box will be done in future work.

Table 4 ARPE for IL-LEM prediction method

Problem	0–20%	20–40%	40–60%	60–80%	80–100%
Henriksen–Madsen 4	9.45E+5	4.64E+3	**0.27**	**0.12**	**0**
Goldstein–Price	**0.84**	0.78	0.78	0.54	**0.09**
Henriksen–Madsen 3	76.43	**0.28**	**0.29**	**0.16**	**0**
Levy 5	**0.29**	**0.37**	0.15	0.12	**0.03**
Levy 3	24.20	**0.96**	0.15	0.12	**0.05**
EX 1	4.23	**1.68**	0.31	0.27	0.14
Branin	3.07	**2.17**	**0.76**	**0.54**	**0.31**
Chichinadze	7.21	2.47	**1.48**	**0.18**	**0.25**
Griewank 2	**0.04**	**0**	**0**	**0**	**0**
Griewank 10	1.10E+86	0.02	0.02	0.01	**0**
Hartman 6	2.73E+4	**7.18**	**0.35**	**0.10**	0.14
Shekel 7	489.29	**0.28**	**0.19**	**0.04**	**0**
Shekel 5	**1.04**	**0.10**	**0.08**	**0.02**	**0.01**
Shekel 10	156.18	**0.32**	**0.24**	**0.09**	**0**
Colville	5.80E+9	66.73	15.91	**3.89**	**1.07**
Hartman 3	10.55	0.31	0.30	0.17	0.11
Ratz-5	181.73	**0.27**	**0.22**	**0.31**	**0**
Rosenbrock 10	1.23E+27	1.99E+5	4.30E+3	28.29	0.59
Kowalik	7.87	**0.69**	**0.09**	**0.07**	**0.06**
Ratz-8	**0.66**	1.18	1.21	0.03	**0**
Neumaier 3–10	1.62E+3	0.57	0.67	0.60	**0.22**
Neumaier 2	2.53	0.95	0.59	0.25	0.03

Appendix A: Nodes per level and γ-sequence of test problems

See Figs. 9, 10, 11, 12, 13, 14, 15, 16, 17.

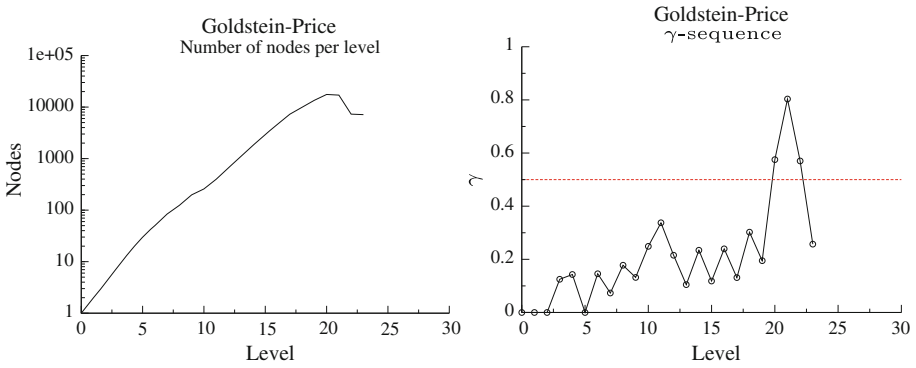

Fig. 9 Goldstein–Price problem. $\epsilon = 10^{-3}$

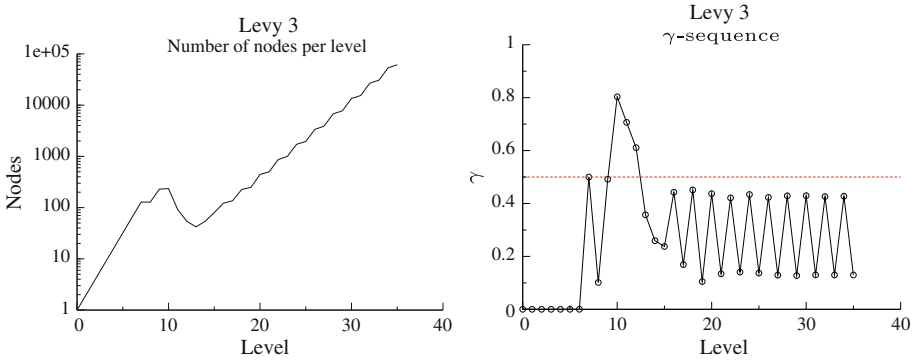

Fig. 10 Levy 3 problem. $\epsilon = 10^{-4}$

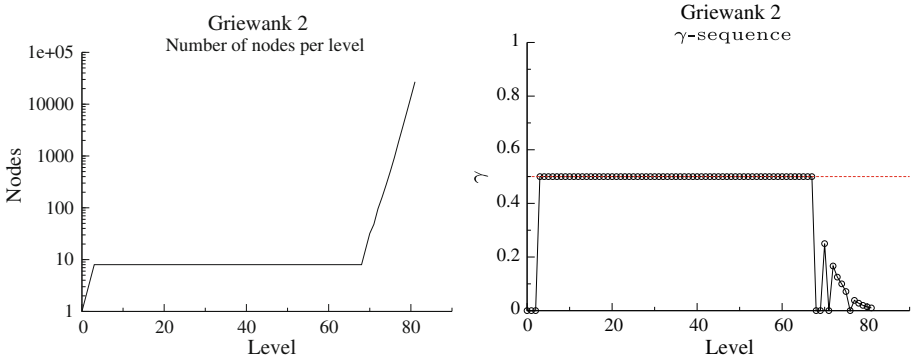

Fig. 11 Griewank 2 problem. $\epsilon = 10^{-9}$

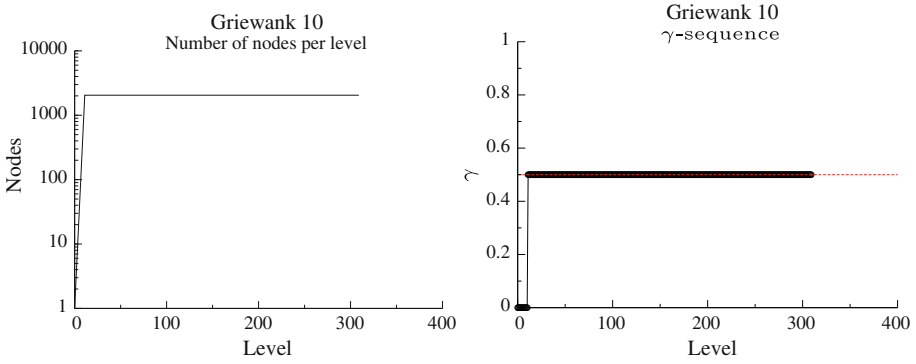

Fig. 12 Griewank 10 problem. $\epsilon = 10^{-6}$

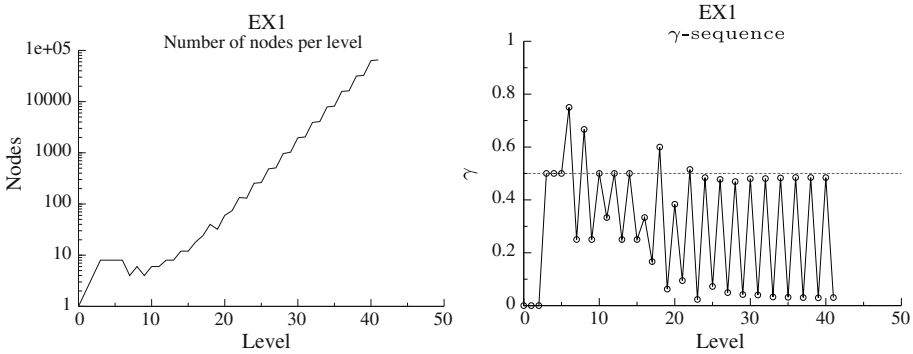

Fig. 13 EX1 problem. $\epsilon = 10^{-6}$

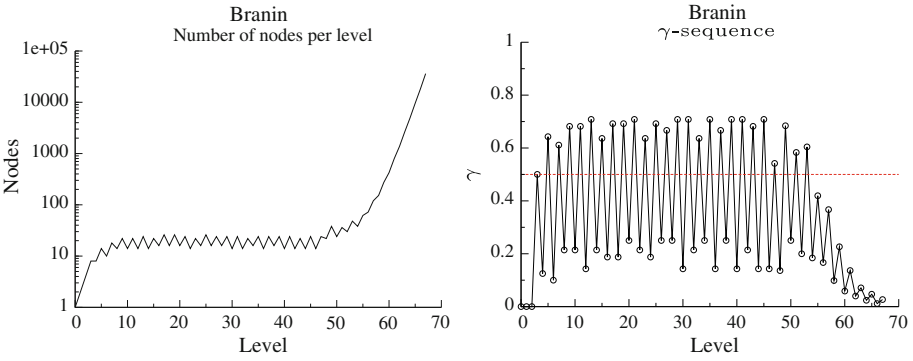

Fig. 14 Branin problem. $\epsilon = 10^{-9}$

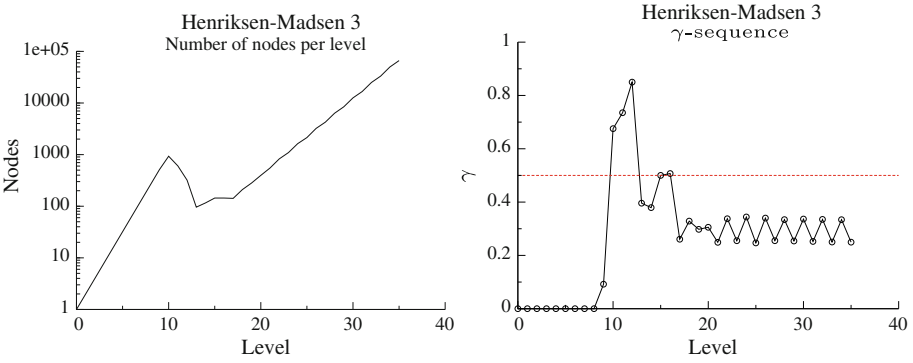

Fig. 15 Henriksen–Madsen 3 problem. $\epsilon = 10^{-4}$

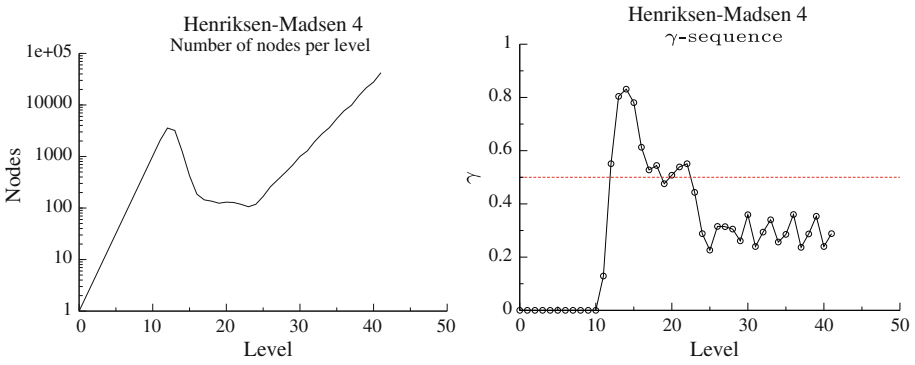

Fig. 16 Henriksen–Madsen 4 problem. $\epsilon = 10^{-3}$

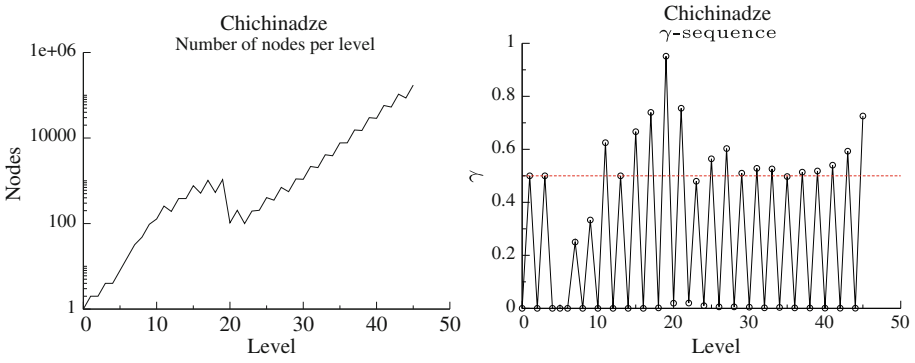

Fig. 17 Chichinadze problem. $\epsilon = 10^{-5}$

Appendix B: *Left-over* prediction figures

See Figs. 18, 19, 20, 21, 22, 23, 24, 25, 26.

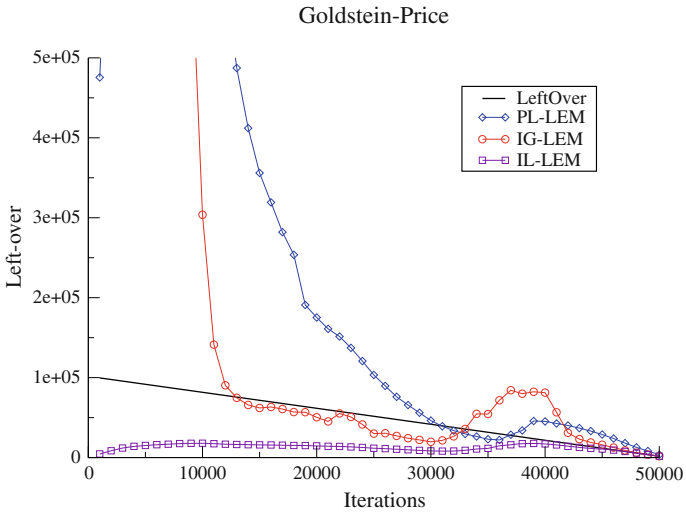

Fig. 18 Goldstein–Price. $\epsilon = 10^{-3}$

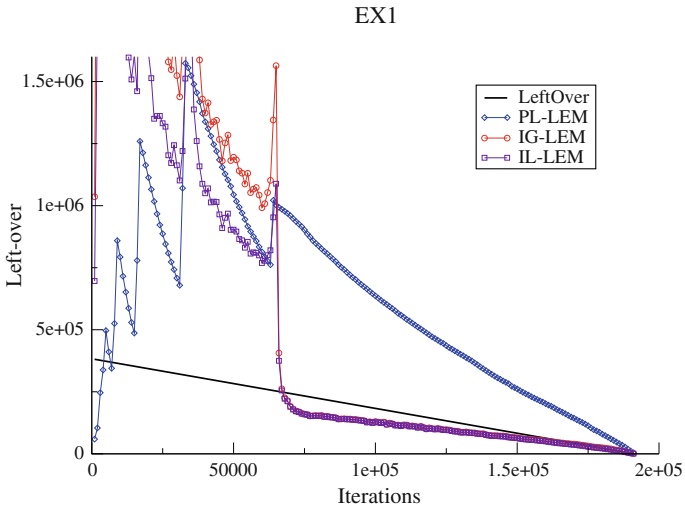

Fig. 19 EX 1. $\epsilon = 10^{-6}$

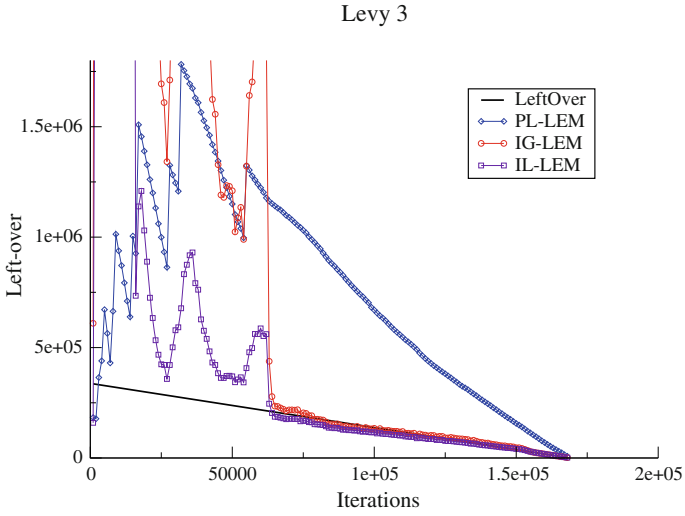

Fig. 20 Levy 3. $\epsilon = 10^{-4}$

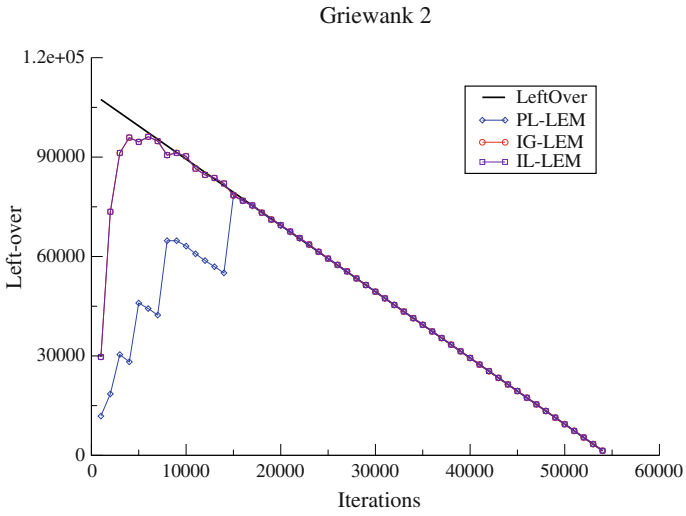

Fig. 21 Griewank 2. $\epsilon = 10^{-9}$

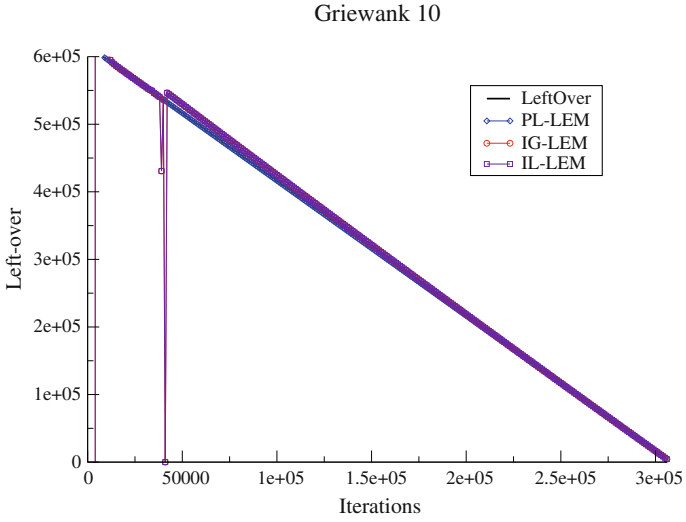

Fig. 22 Griewank 10. $\epsilon = 10^{-6}$

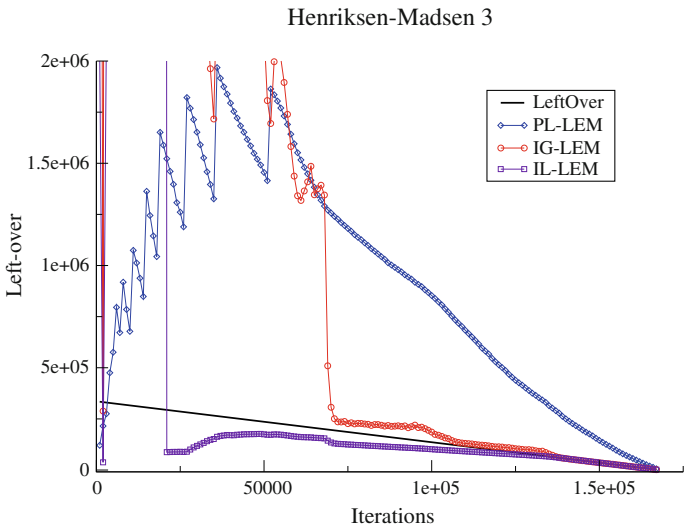

Fig. 23 Henriksen–Madsen 3. $\epsilon = 10^{-4}$

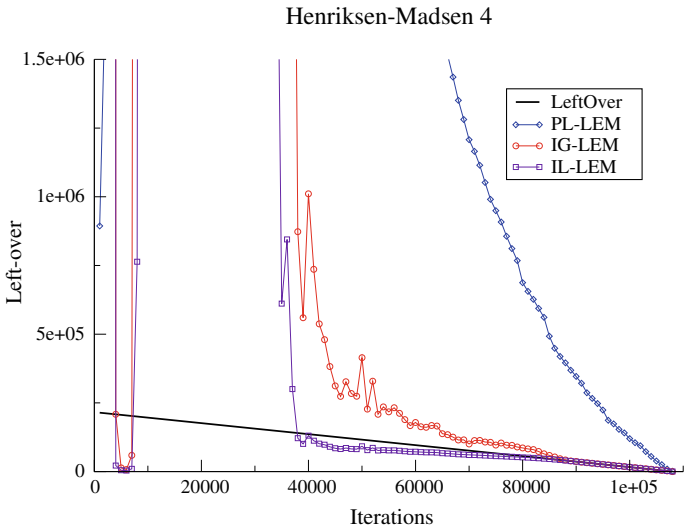

Fig. 24 Henriksen–Madsen 4. $\epsilon = 10^{-3}$

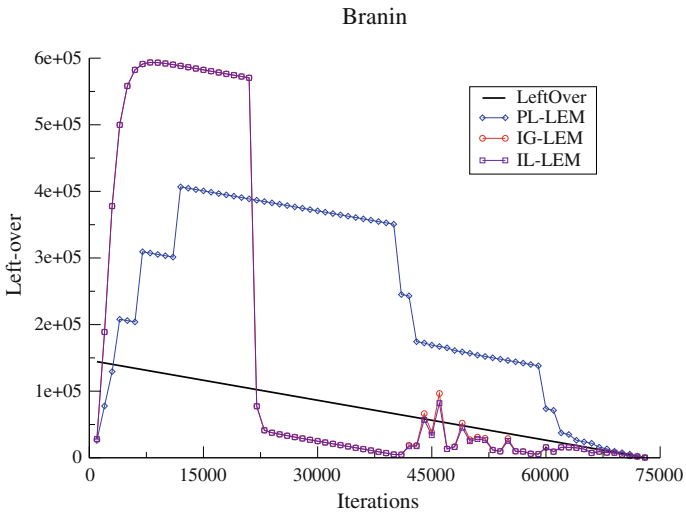

Fig. 25 Branin. $\epsilon = 10^{-9}$

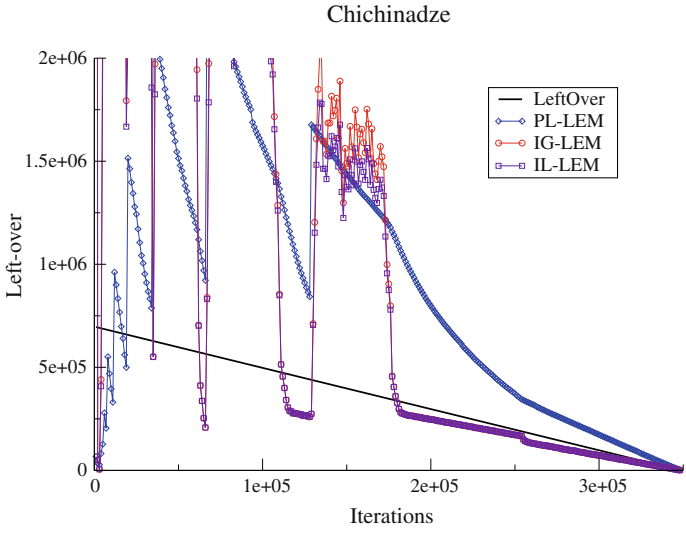

Fig. 26 Chichinadze. $\epsilon = 10^{-5}$

References

1. Berner, S.: Parallel methods for verified global optimization practice and theory. J. Glob. Optim. **9**, 1–22 (1996). doi:10.1007/BF00121748
2. Casado, L., García, I., Csendes, T., Ruíz, V.: Heuristic rejection in interval global optimization. J. Optim. Theory Appl. (JOTA) **118**(1), 27–43 (2003). doi:10.1023/A:1024731306785
3. Colville, A.R.: A comparative study of nonlinear programming codes. Technical report 320-2949. IBM Scientific Center, New York (1968)
4. Cornuéjols, G., Karamanov, M., Li, Y.: Early estimates of the size of branch-and-bound trees. INFORMS J. Comput. **18**(1), 86–96 (2006). doi:10.1287/ijoc.1040.0107

5. Csendes, T., Ratz, D.: Subdivision direction selection in interval methods for global optimization. SIAM J. Numer. Anal. **34**, 922–938 (1997)
6. Dixon, L., Szego, G. (ed.): Towards Global Optimization 2. North-Holland Publishing Company, Amsterdam (1978)
7. Eriksson, J., Lindström, P.: A parallel interval method implementation for global optimization using dynamic load balancing. Reliab. Comput. **1**(1), 77–91 (1995). doi:10.1007/BF02390523
8. Ferreira A., Pardalos P. (eds.): Solving combinatorial optimization problems in parallel. In: Lecture Notes in Computer Science, vol. 1054. Springer, Berlin (1996). doi:10.1007/BFb0027115
9. Finkel, R., Manber, U.: A distributed implementation of backtracking. ACM Trans. Program. Lang. Syst. **9**(2), 235–256 (1987)
10. Gendron, B., Crainic, T.: A parallel branch-and-bound algorithm for multicommodity location with balancing requirements. Comput. Oper. Res. **24**(9), 829–847 (1997). doi:10.1016/S0305-0548(96)00094-9
11. Gendron, B., Crainic, T.G.: Parallel branch-and-bound algorithms: survey and synthesis. Oper. Res. **42**(6), 1042–1066 (1994)
12. Hansen, E.R., Walster, G.W.: Global Optimization Using Interval Analysis. 2nd edn. Marcel Dekker, New York (2004)
13. Henriksen, T., Madsen, K.: Parallel algorithms for global optimization. Interval Comput. **3**(5), 88–95 (1992)
14. Henriksen, T., Madsen, K.: Use of a depth-first strategy in parallel global optimization. Technical report EUR-CS-92-10, Institute for Numerical Analysis, Technical University of Denmark (1992)
15. Ibraev, S.: A new parallel method for verified global optimization. Ph.D. thesis, University of Wuppertal, Wuppertal (2001)
16. Kahou, J.: Some new acceleration mechanisms in verified global optimization. Ph.D. thesis, Bergischen Unviersität Wuppertal (2005)
17. Kilby, P., Slaney, J., Thiébaux, S., Walsh, T.: Estimating search tree size. In: AAAI'06: Proceedings of the 21st National Conference on Artificial Intelligence, pp.1014–1019. AAAI Press, HJ (2006)
18. Knuth, D.: Estimating the efficiency of backtrack programs. Math. Comput. **29**(129), 121–136 (1975)
19. Land, A., Doig, A.: An automatic method of solving discrete programming problems. Econometrica **28**(3), 497–520 (1960)
20. Laursen, P.: Simple approaches to parallel branch and bound. Parallel Comput. **19**, 143–152 (1993)
21. Little, J., Murty, K., Sweeney, D., Karel, C.: An algorithm for the traveling salesman problem. Oper. Res. **11**, 972–989 (1963)
22. Moore, R.: Interval Analysis. Prentice-Hall, New Jersey (1966)
23. Moore, R., Hansen, E., Leclerc, A.: Rigorous methods for global optimization. In: Recent Advances in Global Optimization. Princeton University Press, Princeton, NJ, USA, (1992)
24. Moore, R., Kearfott, R., Cloud, M.: Introduction to Interval Analysis. SIAM, Philadelphia (2009)
25. Neumaier, A.: Interval Methods for Systems of Equations. Cambridge University Press, Cambridge (1990)
26. Özaltın, O., Hunsaker, B., Schaefer, A.: Predicting the solution time of branch-and-bound algorithms for mixed-integer programs. INFORMS J. Comput. **23**(3), 392–403 (2011). doi:10.1287/ijoc.1100.0405
27. Pardalos, P. (ed.): Parallel processing of discrete problems. In: The IMA Volumes in Mathematics and its Applications, vol. 106. Springer, Berlin (1999)
28. Pardalos, P., Resende, M., Ramakrishnan, K. (eds.): Parallel processing of discrete optimization problems: DIMACS workshop, 28–29 April 1994. In: DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 22. American Mathematical Society (1995)
29. Rao, V., Kumar, V.: Parallel depth first search. part I. Implementation. Int. J. Parallel Program. **16**(6), 479–499 (1987). doi:10.1007/BF01389000
30. Ratz, D., Csendes, T.: On the selection of subdivision directions in interval branch and bound methods for global optimization. J. Glob. Optim. **7**, 183–207 (1995)
31. Törn, A., Žilinskas, A.: Global Optimization, vol. 350. Springer, Berlin (1989)
32. Trienekens, H., De Bruin, A.: Towards a taxonomy of parallel branch and bound algorithms. Technical report EUR-CS-92-01. Erasmus University Rotterdam (1992)
33. Wiethoff, A.: Verifizierte Globale Optimierung auf Parallelrechnern. Ph.D. thesis, Universität Karlsruhe, Karlsruhe (1998)