Zuse Institute Berlin

Takustr. 7
14195 Berlin
Germany

Gregor Hendel[1], Daniel Anderson[2],
Pierre Le Bodic[3], Marc E. Pfetsch[4]

Estimating the Size of Branch-and-Bound Trees

[1] 0000-0001-7132-5142
[2] 0000-0002-5853-0472
[3] 0000-0003-0842-9533
[4] 0000-0002-0947-7193

ZIB Report 20-02 (April 2020)

Zuse Institute Berlin
Takustr. 7
14195 Berlin
Germany

Telephone: +49 30-84185-0
Telefax: +49 30-84185-125

E-mail: bibliothek@zib.de
URL: http://www.zib.de

ZIB-Report (Print) ISSN 1438-0064
ZIB-Report (Internet) ISSN 2192-7782

Estimating the Size of Branch-and-Bound Trees

Gregor Hendel* Daniel Anderson† Pierre Le Bodic‡
Marc E. Pfetsch§

April 4, 2020

Abstract

This paper investigates the estimation of the size of Branch-and-Bound (B&B) trees for solving mixed-integer programs. We first prove that the size of the B&B tree cannot be approximated within a factor of 2 for general binary programs, unless $\mathcal{P} = \mathcal{NP}$. Second, we review measures of the progress of the B&B search, such as the gap, and propose a new measure, which we call *leaf frequency*. We study two simple ways to transform these progress measures into B&B tree size estimates, either as a direct projection, or via double-exponential smoothing, a standard time-series forecasting technique. We then combine different progress measures and their trends into nontrivial estimates using Machine Learning techniques, which yields more precise estimates than any individual measure. The best method we have identified uses all individual measures as features of a random forest model. In a large computational study, we train and validate all methods on the publicly available MIPLIB and Coral general purpose benchmark sets. On average, the best method estimates B&B tree sizes within a factor of 3 on the set of unseen test instances even during the early stage of the search, and improves in accuracy as the search progresses. It also achieves a factor 2 over the entire search on each out of six additional sets of homogeneous instances we have tested. All techniques are available in version 7 of the branch-and-cut framework SCIP.

1 Introduction

Simultaneous advancement of theory, algorithms, implementation and computer hardware has, in the span of a few decades, allowed for stunning progress in the ability to solve real-world problems encoded as Mixed-Integer Programs (MIPs), see for example [36, 2]. However, in spite of these tremendous performance improvements, users of MIP solvers still face the seemingly irremediable curse that comes with solving $\mathcal{NP}$-hard combinatorial problems: it is very hard to

*Zuse Institute Berlin, Berlin, Germany; hendel@zib,de
†Computer Science Department, Carnegie Mellon University, Pittsburgh PA, USA; dlanders@cs.cmu.edu
‡Faculty of Information Technology, Monash University, Melbourne, Australia; pierre.lebodic@monash.edu
§Department of Mathematics, TU Darmstadt, Germany; pfetsch@mathematik.tu-darmstadt.de

predict how long a solver will take to solve an instance by any other means than to just wait for termination. Since the core algorithm of all state-of-the-art MIP solvers is Branch-and-Bound (B&B) [32], the main problem is that of estimating the size of the B&B tree. In this article, we focus on online tree size estimation, i.e., during the execution of the algorithm.

The first method to estimate search tree size was proposed by Knuth [30]. It works by repeatedly sampling paths of the search tree in a Monte Carlo fashion. Several extensions have been proposed based on partial backtracking [43] and on stratified sampling [12, 35]. All of these methods are carried out offline, i.e., before the actual search begins. In contrast, recent work, including the present paper, has focused on online estimation methods. These include the Weighted Backtrack Estimator [29], the Profile Estimation [14] and the Sum Of Subtree Gaps [40], which we use as reference methods for our computational study. They are explained in Sections 3 and 4.

Several Machine Learning methods have recently been proposed in the context of solving mixed integer programs [28, 5, 4, 27, 7, 22], for example, to compute approximations of strong branching decisions or to control primal heuristics. Fischetti et al. [16] proposed to train classifiers to predict at specific points in time whether a solution process will terminate before the time limit.

In our present work, we treat the more general problem of online tree size estimation over the entire duration of the search process. The main contributions are:

- a proof that approximating the size of the B&B tree within a factor of 2 is impossible, unless $\mathcal{P} = \mathcal{NP}$ (Theorem 4);
- a review of state-of-the-art methods which estimate search progress and B&B tree size online;
- a new method, based on the frequency of leaf nodes in the B&B tree, which approximates the search progress;
- new tree-size estimation methods applying Machine Learning techniques to combine individual methods;
- an experimental comparison of all methods.

The experimental results show that at the start of the search, the best ML method improves upon the best individual method in estimation accuracy by a considerable margin and by a factor of 10 compared to a method based on the gap. An efficient implementation and integration of these algorithms is available in SCIP, an academic state-of-the-art branch-and-cut framework. In its newest release, SCIP 7 displays an estimate of the search progress as a percentage in a new column of the output.

This article is structured as follows. The main theoretical result of this paper, namely on the hardness of tree size estimation, is proven in Section 2. Section 3 introduces the necessary notation for the presentation of search tree estimates. Search tree estimates which are based on approximations of the fraction of already solved nodes and/or time series forecasting are explained in Sections 4 and 5, respectively. We present an offline simulation to calibrate time series forecasting in Section 6. In Section 7, we show how to combine the tools from the previous sections via Machine Learning techniques. Afterwards, we perform computational experiments to compare all discussed methods on

Algorithm 1: Specific Branch-and-Bound algorithm B_{VP} for solving VP

```
   Input: undirected simple graph $G = (V, E)$
   Output: optimal value of the vertex packing problem for $G$
1  $\mathcal{T} = (\mathrm{VLP})$, $L = -\infty$;
2  while $\mathcal{T} \neq \varnothing$ do
3      remove $T$ from front of $\mathcal{T}$;
4      compute vertex solution $x^* \in \{0, 1, \frac{1}{2}\}$ to VLP at $T$ which
         maximizes the number of integer components;
5      if $x^*$ exists and $\mathbb{1}^\top x^* > L$ then
6          if $x^* \in \{0,1\}^V$ then
7              $L = \mathbb{1}^\top x^*$;
8          else
9              choose any $\{u, v\} \in E$ with $x_u^* = x_v^* = \frac{1}{2}$;
10             create two new nodes $T^u$ and $T^v$ in which $x_u$ and $x_v$ are
                 fixed to 0, respectively;
11             add $T^u$ and $T^v$ at the back of $\mathcal{T}$;
```

a general MIP instance set in Section 8, and on homogeneous instance sets in Section 9.

2 Hardness of Estimating the Size of the Branch-and-Bound tree

In this section, we consider a B&B algorithm for the Vertex Packing (VP) problem and show that if there existed a polynomial time oracle that estimates the size of B&B trees within a factor 2, then the optimal value of a VP instance could be determined exactly. Since computing the optimal value of VP is $\mathcal{NP}$-complete [18], this would in turn imply $\mathcal{P} = \mathcal{NP}$.

2.1 A Specific Branch-and-Bound for the Vertex Packing Problem

Given a simple and finite undirected graph $G = (V, E)$, consider the following binary program for (unweighted) vertex packing:

$$\max_{x} \quad \sum_{v \in V} x_v \tag{1a}$$

$$\text{s.t.} \quad x_u + x_v \leq 1 \quad \forall \{u, v\} \in E \tag{1b}$$

$$x \in \{0,1\}^V. \tag{1c}$$

This problem is also known as the Independent or Stable Set problem. The Linear Programming (LP) relaxation of (1) will be denoted by VLP.

To prove our complexity result, we need a specific LP-based B&B algorithm B_{VP} to solve VP; see Algorithm 1. In each node, the algorithm computes a particular optimal vertex solution to the LP-relaxation using the techniques of Nemhauser and Trotter [38]. They provide a strongly polynomial algorithm to compute an optimal vertex solution based on a bipartite graph. Moreover, by iteratively fixing components that assume value 1 and resolving VLP on a reduced graph, they reach a solution in which the set of components with integer values is maximal and all fractional values are $\frac{1}{2}$. Picard and Queyranne [42] proved that this actually yields a solution in which this set is maximum and unique. The main argument is that vertices of VLP are $\{0, 1, \frac{1}{2}\}$-valued [37].

Algorithm 1 chooses the branching candidates in a non-standard way: the algorithm will branch on a disjunction $x_u = 0 \vee x_v = 0$, where $\{u, v\} \in E$. These disjunctions are also satisfied by any feasible solution to VP, but are – by design – *not* exclusive. If the current optimal solution x^* to VLP is not integral, there exists a component with $x_u^* = \frac{1}{2}$. Then there exists an edge $e = \{u, v\}$ which is tight, i.e., $x_u^* + x_v^* = 1$, otherwise x^* would not be optimal. This implies $x_v^* = \frac{1}{2}$ and thus the edge in Step 9 exists.

Algorithm 1 uses a breadth-first node selection algorithm, which in the case of B_{VP} is equivalent to a best-bound node selection. The search could thus stop at the first node whose VLP solution is feasible for VP (by integrality of the objective function), but for convenience we suppose that all nodes at that depth are processed and pruned by LP bound.

2.2 Properties of Algorithm B_{VP}

Proposition 1. *Branching improves the dual bound by exactly $\frac{1}{2}$.*

Proof. As mentioned above, the solution obtained at every node is half-integral and the number of integer values is maximal. Thus, changing any fractional variable to 0 changes the dual bound. Therefore, the dual bound difference between a parent node and its children must be at least $\frac{1}{2}$. It is also at most $\frac{1}{2}$, as taking the VLP solution of the parent node and setting any fractional variable to 0 maintains feasibility. □

Proposition 2. *At termination of B_{VP}, all leaves have the same depth and all optimal solutions lie at these leaves.*

Proof. There always exists an optimal solution to VP, so at termination of B_{VP}, at least one optimal solution has been found. By design of B_{VP}, optimal solutions can only be found if a VLP solution at a node is integral. This node would then be a leaf of the final B&B tree. Let d denote its depth. By Proposition 1, all nodes at the same depth have the same dual bound. Since the node selection strategy is Breadth-First Search, all leaves of the current B&B tree have depth d or $d - 1$. All leaves at depth d can be pruned by bound and become leaves of the final B&B tree. Moreover, all leaves at depth $d - 1$ need to be branched on, and their children are pruned by bound (but provide all other optimal solutions if they exist). □

Proposition 3. *Let z_{VP} and z_{VLP} be the optimal VP and VLP values, respectively. Solve VP to optimality with B_{VP}, and let k denote the number of nodes of the B&B tree at termination of B_{VP}. Then $z_{VP} = z_{VLP} - \frac{1}{2}(\log_2(k+1) - 1)$.*

Proof. By Proposition 2, $k+1$ is a power of 2, the depth of all leaves at optimality is $\log_2(k + 1) - 1$, and at at least one of them, the VLP solution is feasible for VP. Since branching decreases the dual bound by $\frac{1}{2}$ at every depth after the root node (Proposition 1), all leaves have dual bound $z_{\text{VLP}} - \frac{1}{2}(\log_2(k+1) - 1)$, since z_{VLP} is the optimal value of VLP at the root node. □

2.3 Hardness Result

Consider a B&B algorithm B and an instance I that is solvable by B. Consider an algorithm that computes an estimate $\mathcal{O}(B,I) \in \mathbb{Z}_+$ for the total number of nodes $N(B,I)$ produced by B on I. We call this algorithm a *c-approximation algorithm* for a constant $c \geq 1$ if

- $N(B,I)/c \leq \mathcal{O}(B,I) \leq c\, N(B,I)$,
- $\mathcal{O}(B,I)$ can be computed in polynomial time in the encoding size of I.

Theorem 4. *There is no c-approximation algorithm for $c \leq 2$, unless $\mathcal{P} = \mathcal{NP}$.*

Proof. Assume that $\mathcal{O}(B,I)$ provides a c-approximation for B&B algorithm $B = B_{\text{VP}}$ described in Section 2.1 and instances I of VP. By Proposition 3, it suffices to compute z_{VLP} and the number of nodes of B_{VP} to determine the optimal value of the VP instance. Indeed, let $k = N(B_{\text{VP}}, I)$ and define $\ell = 2(z_{\text{VLP}} - z_{\text{VP}}) + 1 \in \mathbb{Z}_+$. Then

$$k = 2^{2(z_{\text{VLP}} - z_{\text{VP}})+1} - 1 = 2^\ell - 1.$$

Therefore, $k/c \leq \mathcal{O}(B,I)$ holds if and only if

$$(2^\ell - 1)/c \leq \mathcal{O}(B,I) \quad \Leftrightarrow \quad \ell \leq \log_2(c\,\mathcal{O}(B,I)+1) = \log_2(c) + \log_2(\mathcal{O}(B,I) + \tfrac{1}{c}).$$

Similarly, $c\,k \geq \mathcal{O}(B,I)$ holds if and only if

$$c(2^\ell - 1) \geq \mathcal{O}(B,I) \quad \Leftrightarrow \quad \ell \geq \log_2(\tfrac{1}{c}\mathcal{O}(B,I)+1) = -\log_2(c) + \log_2(\mathcal{O}(B,I) + c).$$

If $c = 1$, then $\mathcal{O}(B,I) = k$, which directly allows to compute z_{VP}. Otherwise, we have $\frac{1}{c} < 1 < c$, which implies:

$$-\log_2(c) + \log_2(\mathcal{O}(B,I) + 1) < \ell < \log_2(c) + \log_2(\mathcal{O}(B,I) + 1).$$

Defining $\alpha := z_{\text{VLP}} + \frac{1}{2} - \frac{1}{2}\log_2(\mathcal{O}(B,I) + 1)$, this is equivalent to

$$\begin{aligned} & -\log_2(c) < \ell + 2\,(\alpha - z_{\text{VLP}} - \tfrac{1}{2}) < \log_2(c) \\ \Leftrightarrow \quad & -\log_2(c) < 2(z_{\text{VLP}} - z_{\text{VP}}) + 1 + 2\,(\alpha - z_{\text{VLP}} - \tfrac{1}{2}) < \log_2(c) \\ \Leftrightarrow \quad & \frac{-\log_2(c)}{2} < \alpha - z_{\text{VP}} < \frac{\log_2(c)}{2}. \end{aligned}$$

Since $c \leq 2$, $\log_2(c) \leq 1$, implying that $-\frac{1}{2} < \alpha - z_{\text{VP}} < \frac{1}{2}$, i.e., $z_{\text{VP}} = \lfloor \alpha \rceil$, which shows that k can be computed exactly from $\mathcal{O}(B,I)$ and z_{VLP} if $c \leq 2$.

Therefore the existence of a c-approximation algorithm providing an estimate for all B&B algorithms B and all MIP instances I implies that z_{VP} can be computed in polynomial time in the size of I. But since computing z_{VP} is $\mathcal{NP}$-hard, this would imply $\mathcal{P} = \mathcal{NP}$. □

Note that this result uses only the $\mathcal{NP}$-hardness of VP. It is known that VP is also hard to approximate [21]. It may be possible to strengthen the inapproximability result of Theorem 4 using this fact, though we have not been able to do so with our current approach.

Finally, note that the c-approximation we study is offline, but it is allowed to solve polynomially many nodes of B. Therefore, if there existed an online 2-approximation, it would first need to visit super-polynomially many nodes of B, unless $\mathcal{P} = \mathcal{NP}$. Despite these pessimistic results, in the remaining part of the paper, we nevertheless aim to develop practical tree size estimation algorithms, which are consequently online, i.e., they improve their approximation during the runtime of the B&B algorithm.

3 B&B Search States and Search Completion

In this and subsequent sections, we consider mixed-integer programs

$$c^* := \min\{c^\top x \,:\, Ax \geq b,\ x \in \mathbb{Z}^{n_z} \times \mathbb{R}^{n-n_z}\}. \tag{P}$$

Here, $c \in \mathbb{R}^n$, $A \in \mathbb{R}^{\ell \times n}$, $b \in \mathbb{R}^\ell$, and $n_z > 0$ is the number of of *integrality restrictions* of the variables.

We assume familiarity with the B&B algorithm [32, 15] (described in Appendix A for completeness). For iteration (number of solved nodes) $k = 0, \ldots, m$, where m is the total size of the search tree at termination, we record the *search state* T_k, which is defined by the following data:

1. the set J_k of *inner nodes* of the search tree;
2. the set F_k of *final leaves* of the search tree, i.e., nodes that were infeasible, or whose LP relaxation was mixed-integer feasible, or nodes that were pruned;
3. the set L_k of *open nodes* of the search, which are still to be processed;
4. the *primal bound* Π_k, i.e., the value of the best solution until iteration k;
5. a *dual bound* (lower bound) for each node $\pi_k : J_k \cup F_k \cup L_k \to \mathbb{R} \cup \{-\infty, \infty\}$.

Throughout the paper, we assume 2-way branching decisions, which is most common in state-of-the-art MIP solvers. Therefore, the explored search tree is binary, and at each search state T_k, $k \geq 0$, the relation

$$2 \cdot (|F_k| + |L_k|) - 1 = |V_k| \tag{2}$$

holds, where we define $V_k := L_k \cup J_k \cup F_k$. Furthermore, since a solved node enters either F_k or J_k, the relations $|F_k| + |J_k| = k$ and therefore $|V_k| - |L_k| = k$ hold at each search state. Note that $|V_m| = m$.

Definition 5 (Search Completion)**.** *Let $T = T_0, \ldots, T_m$ be a search state sequence. For every state T_k, $k \in [m]$, in T, we define the* search completion *as the fraction of solved nodes compared to the size of the final tree*

$$\gamma_k = \frac{k}{m} = \frac{|F_k| + |J_k|}{|V_m|}. \tag{3}$$

In real optimization scenarios, m and thus γ_k are only known after the Branch-and-Bound algorithm terminated. In Sections 4 and 5 we review measures that can be used to estimate γ_k and m online.

In practice, the search completion γ_k also approximates the runtime of a MIP solver quite well. For the data set used in Sections 7 and 8, we show in Figure 1 the close correspondence between the relative time of the overall search and the search completion γ_k. To this end, Figure 1 depicts the *data density* of the observed differences between the relative time, i.e., the time normalized to $[0, 1]$, and the search completion in our data set. A high data density at a coordinate on the horizontal axis reflects a high number of observed records near this coordinate. The density has a peak at zero, in which case the search completion approximates the actual fraction of time exactly. Since deviations from zero are mostly positive, this means that the search completion is slightly pessimistic as it often underestimates the remaining time. This can be explained by the fact that solvers do not spend their entire time on the tree search itself, but also spend significant time on presolving and root node processing. Hence, it seems generally reasonable to use the search completion γ_k as a surrogate for the runtime.

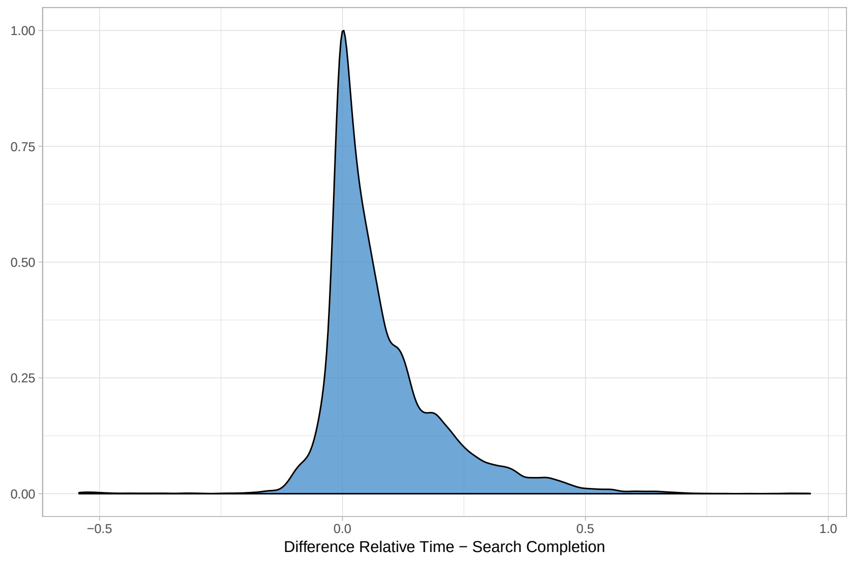


Figure 1: Density approximation of the difference between the relative time of the search and the search completion γ_k

4 Approximations of Search Completion

Let T_k for $k \in [m]$ denote a search state during the B&B algorithm. Any approximation $\hat{\gamma}_k$ of the search completion γ_k from Definition 5 provides a simple way to estimate the search tree size as

$$\hat{m} := \frac{k}{\hat{\gamma}_k}, \tag{4}$$

since k is equal to the number of solved nodes at T_k. Clearly, close approximations of γ_k can be expected to give accurate estimates of the final tree size.

In the following, we review four approximate measures of γ_k and the related weighted backtrack estimation [29]. We start with the well-known gap.

4.1 Gap

Every B&B solver reports the relative *gap* between the primal and dual bound in one form or another, which makes the gap the most common progress measure for B&B. Given a search state T_k, we extend the definition of the dual bound π_k to subsets of nodes $V' \subseteq V_k$ by setting

$$\pi_k(V') := \min_{v \in V'} \pi_k(v).$$

The dual bound $\pi_k(V_k)$ at T_k is called the *global dual bound.* We use the standard gap definition for minimization problems

$$\delta(T_k) = \delta(T_k, V_k) := \begin{cases} 1, & \text{if } \Pi_k = \infty, \\ 0, & \text{if } \Pi_k \leq \pi_k(V_k), \\ \min\left\{1, \frac{|\Pi_k - \pi_k(V_k)|}{\max\{|\Pi_k|, |\pi_k(V_k)|\}}\right\}, & \text{otherwise.} \end{cases} \tag{5}$$

The gap monotonically decreases from $1 = \delta(T_0)$ to $0 = \delta(T_m)$. Therefore, taking $1 - \delta(T_k)$ yields a monotone approximation of γ_k.

In the absence of any dedicated progress measure in the output of MIP solvers, the gap serves as the *de facto* progress measure. However, as we will see in our experiments, it provides the poorest approximation of all search progress measures defined in this section. One shortcoming of the gap is that, if the primal bound does not change, only improvements to the global dual bound are reflected in the gap. For instances for which the difficulty lies in finding an optimal solution, or towards the end of the B&B search, where the absolute gap is small, changes in the global dual bound occur very infrequently. The Sum of Subtree Gaps, presented in the next section, has been designed to take into account the change of dual bound at every node, rather than only the change of the worst dual bound.

4.2 Sum of Subtree Gaps (SSG)

The *Sum of Subtree Gaps* (SSG) has been proposed by Özaltın [40] as a better runtime estimate than the gap. Consider a family $\mathcal{V} = \{\tilde{V}_1, \dots, \tilde{V}_p\}$ of disjoint subsets of V_k, for which we define

$$f_k(\mathcal{V}) \coloneqq \sum_{\tilde{V} \in \mathcal{V}} \delta(T_k, \tilde{V})$$

as the (unscaled) sum of subtree gaps, and let

$$\mathrm{pred}^{\Pi}(k) \coloneqq \min\{k' \in [k] \,:\, \Pi_{k'} = \Pi_k\}$$

denote the last iteration of B&B up to k in which the primal bound improved. The SSG uses a special family $\mathcal{V}$, namely the $\left|L_{\mathrm{pred}^{\Pi}(k)}\right|$ subtrees rooted at the open nodes $L_{\mathrm{pred}^{\Pi}(k)}$ at the time at which the primal bound has improved last. For $v \in V_k$, let $\mathrm{subtree}(v) \subseteq V_k$ denote the set containing v and its descendants. If $v \in F_k \cup L_k$, then $\mathrm{subtree}(v) = \{v\}$. In all other cases, v is an inner node with at least two descendants. The SSG is defined as

$$\sigma(T_k) \coloneqq s_k \cdot f_k(\mathcal{V}^{\mathrm{ssg}}(k))$$

over the disjoint family

$$\mathcal{V}^{\mathrm{ssg}}(k) \coloneqq \left\{\mathrm{subtree}(v) :\, v \in L_{\mathrm{pred}^{\Pi}(k)}\right\}$$

with a scaling factor s_k defined as

$$s_k \coloneqq \begin{cases} 1, & \text{if } k = 0, \\ s_{k-1}, & \text{if } \Pi_k = \Pi_{k-1}, \\ s_{k-1} \cdot \frac{f_k(\mathcal{V}^{\mathrm{ssg}}(k-1))}{f_k(\mathcal{V}^{\mathrm{ssg}}(k))} & \text{if } \Pi_k < \Pi_{k-1}. \end{cases}$$

The scaling factor changes every time a new incumbent solution is found to ensure monotonicity.

Like the gap, the SSG $\sigma(T_k)$ is monotonically decreasing from $\sigma(T_0) = 1$ to $\sigma(T_m) = 0$. Therefore, an approximation of search completion based on SSG is given by

$$\hat{\gamma}_k^{\mathrm{ssg}} = 1 - \sigma(T_k).$$

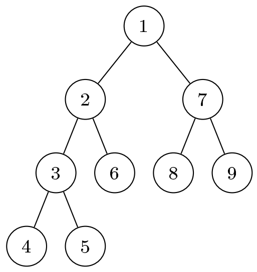


Figure 2: Example search state at termination ($m = 9$, $|F_9| = 5$, $|L_9| = 0$). We assume that the tree was traversed from left to right in depth-first order. The final leaves are $F_9 = \{4, 5, 6, 8, 9\}$ and internal nodes are $J_9 = \{1, 2, 3, 7\}$.

One limitation of both the gap and the SSG is that their value is infinite in absence of a primal bound, in which case improvements to the (global) dual bound(s) are not reflected. The progress measures presented in Sections 4.3, 4.4 and 4.5 are not based on the primal or dual bound. They can thus be used for feasibility problems and more generally for other tree search algorithms.

4.3 Tree Weight

The tree weight has been first used by Kilby et al. [29] as the denominator of the weighted backtrack estimator (studied in Section 4.4). Its use as a progress measure was proposed in [6]. This measure assigns to each node $v \in V_k$ a *weight* $2^{-d(v)}$, where $d(v)$ is the depth of v. Note that every parent weight equals the sum of the weights of its children. We call

$$\omega(T_k) := \sum_{v \in F_k} 2^{-d(v)}$$

the *tree weight* at state T_k. At the start of the search, $F_0 = \emptyset$, thus $\omega(T_0) = 0$. After every final leaf node, the tree weight strictly increases and the search ends at step m with a tree weight of 1 [6]. With those properties, the tree weight ω can be directly used as approximation of search completion,

$$\hat{\gamma}_k^{\text{tree weight}} := \omega(T_k).$$

In the running example tree from Figure 2, the leaf weights are $w_4 = w_5 = 0.125$ and $w_6 = w_8 = w_9 = 0.25$. From step 0 to 9, the tree weight therefore assumes the 6 distinct values $\omega(T_0) = \omega(T_1) = \omega(T_2) = \omega(T_3) = 0$, $\omega(T_4) = 0.125$, $\omega(T_5) = 0.25$, $\omega(T_6) = \omega(T_7) = 0.5$, $\omega(T_8) = 0.75$, $\omega(T_9) = 1$.

4.4 Weighted Backtrack Estimation (WBE)

The *weighted backtrack estimator* (WBE) of Kilby et al. [29] is a projection of the current tree weight to estimate the number of leaf nodes at completion. To this end, at search state T_k with positive tree weight $\omega(T_k) > 0$, the weighted backtrack estimate is computed as

$$\hat{m}_k^{\text{wbe}} := 2 \cdot \frac{|F_k|}{\omega(T_k)} - 1. \tag{6}$$

Table 1: Values of the leaf frequency for the running example in Figure 2

k	1	2	3	4	5	6	7	8	9
$\lvert F_k \rvert$	0	0	0	1	2	3	3	4	5
$\lambda(T_k)$	-0.5	-0.25	-0.17	0.13	0.3	0.42	0.36	0.44	0.5

Note that the tree size estimation (6) is not equivalent to the one obtained when using $\hat{\gamma}_k^{\text{tree weight}} = \omega(T_k)$ within (4), which yields

$$\hat{m}_k^{\text{tree weight}} := \frac{|F_k| + |J_k|}{\omega(T_k)}.$$

One noticeable difference is that $\hat{m}_k^{\text{wbe}}$ is only sensitive to the creation of final leaves, not open nodes.

4.5 Leaf Frequency

In this subsection we introduce a new progress measure called *Leaf Frequency*, based on the well-known observation that due to Equation (2), $|F_m|/m \approx 1/2$, i.e., the final leaf nodes comprise about one half of the overall tree at the end of the search. In order to extend this to a search progress measure at intermediate search states T_k, we define the *leaf frequency* as

$$\lambda(T_k) := \frac{1}{k}\left(|F_k| - \frac{1}{2}\right). \tag{7}$$

The subtraction of $\frac{1}{2}$ in the formula above is the necessary correction such that $\lambda(T_m) = \frac{1}{2}$ at termination. More generally, by combining (2) and the relation $|V_k| - |L_k| = k$, we obtain

$$\lambda(T_k) = \frac{1}{k}\left(|F_k| - \frac{1}{2}\right) = \frac{1}{k}\left(\frac{1}{2}(|V_k| - 2\,|L_k| + 1) - \frac{1}{2}\right) = \frac{1}{2} - \frac{|L_k|}{2k}.$$

From this equation, we derive that $\lambda(T_k) \leq \frac{1}{2}$ and that $\lambda(T_k) = \frac{1}{2}$ only for $k = m$. Moreover, $\lambda(T_k) \geq -\frac{1}{2}$ holds for all k and $\lambda(T_k) > 0$ after the first final leaf, i.e., as soon as $|F_k| \geq 1$.

Table 1 lists the leaf frequency for all $m = 9$ iterations of the running example from Figure 2. This example shows that the leaf frequency $\lambda(T_k)$ is not necessarily monotone in contrast to the other measures discussed in this section.

We propose to transform the leaf frequency into an approximation of search completion via

$$\hat{\gamma}_k^{\text{leaf-freq}} = 2 \cdot \max\{0, \lambda(T_k)\}.$$

We prefer this transformation over its alternative $\lambda(T_k) + \frac{1}{2}$, which has the disadvantage that it assumes a search completion of at least 50 % after the first final leaf node, which is often too optimistic.

5 Estimation of Tree Size via Time Series Forecasting

In the last section, we presented approximations of the search completion based on the four measures gap, SSG, tree weight, WBE, and leaf frequency, which can be translated into an estimate of the final search tree size using (4). In this section, we present a different approach which uses time series forecasting for each search progress measure. Double exponential smoothing, presented in Section 5.1, is also the approach used for runtime estimates with the SSG [40].

5.1 Double Exponential Smoothing (DES)

We consider a time series $Y = (y_t)_{t \in \mathcal{K}}$ with $\mathcal{K} \subseteq [m]$, which represents values of a suitable measure such as those from the previous section at different steps during the search. A forecast at an intermediate step $t^* \in \mathcal{K}$ uses the available data $y_1, \ldots, y_{t^*}$ to make predictions about the future evolution of Y after t^*.

We use the four measures, tree weight, leaf frequency, gap, or SSG from Section 4 (not their corresponding search completion approximations). As a fifth measure, we consider the number of open nodes $|L_k|$, which is strictly positive at all intermediate steps $0 < k < m$ and reaches 0 at $k = m$. Often, $|L_k|$ has a unimodal behavior, i.e., it increases during the first half of the search process and decreases during the second half.

Although more complex models of *Double Exponential Smoothing* (DES) exist [25], we use a simple variant of DES [23] which estimates the *level* q_t of the time series, representing a fitted value of y_t, and its *trend* s_t, which has the role of a slope. Both are computed as weighted averages of the training data, with exponentially decaying weights on older observations, as follows. Let $0 < \alpha, \beta < 1$ and denote by q_0 and s_0 initial level and trend values. For $t \in \mathcal{K}$, DES fits the level and trend component to the data recursively via

$$\begin{aligned} q_t &= \alpha\, y_t + (1-\alpha)(q_{t-1} + s_{t-1}), \\ s_t &= \beta\,(q_t - q_{t-1}) + (1-\beta)s_{t-1}. \end{aligned} \tag{8}$$

The current level and trend components q_{t^*} and s_{t^*} are used to compute a linear forecast of $h \in \mathbb{N}$ steps into the future as

$$\hat{y}_{t^*+h} := q_{t^*} + h \cdot s_{t^*}.$$

The estimation of the complete tree size m from a forecast at step t^* simply consists in predicting the remaining number of steps h^* of this time series. Depending on the target value y_m of the time series (1 for tree weight, 0.5 for leaf frequency, 0 for open nodes, SSG, and gap), the smallest number of steps h^* is computed such that $\hat{y}_{t^*+h^*}$ reaches y_m. We compute h^* as

$$h^* := \frac{y_m - q_{t^*}}{s_{t^*}}. \tag{9}$$

Recall that the leaf frequency and open nodes time series do not necessarily exhibit a clear monotonicity. Therefore, it may happen that s_{t^*} is zero or has the wrong sign to reach the target value. A trend component of 0 can also occur for monotone measures tree weight, gap, and SSG if they keep stalling. In such cases, we report twice the number of solved nodes at t^* as tree size estimation, according to the intuition that the search has not reached its midpoint if the leaf frequency has not reasonably stabilized or the number of open nodes has not started to decrease yet.

5.2 Time Series Steps and Adaptive Resolution

So far, we have left the index set $\mathcal{K}$ of the considered time series $(y_t)_{t\in\mathcal{K}}$ unspecified. One possibility lies in the full index set $\mathcal{K} = [m]$. In this case, the tree weight time series, as an example, simply consists of the tree weight values $(\omega(T_k))_{k\in[m]}$ at each search state. Another possible index set is the *leaf index set*

$$\mathcal{K}^{\text{leaf}} := \{k \in [m] : \ |F_k| = |F_{k-1}| + 1\} =: \{k_1, \dots, k_{|F_m|}\}.$$

The leaf index set considers only half as many observations as the full index set. It always contains the terminal step m. Restricted to the leaf index set, the tree weight time series becomes strictly monotone, which can be an advantage for forecasting because the trend component is always positive. Thus, we will always use the leaf index set $\mathcal{K} = \mathcal{K}^{\text{leaf}}$ from now on.

DES will assign most of the weight to the most recent individual leaf nodes. However, the information used by an estimation method at a single leaf can have a high variance. For instance, in the case of the tree weight, the relative difference between two leaf weights is exponential in the difference of their respective depth. We overcome this deficiency by essentially creating batches of 2^r leaves, starting with $r = 0$, and by adaptively increasing r over time, as more data becomes available. More precisely, for $r \in \mathbb{Z}_{\geq 0}$ we denote the *index set at resolution* $\frac{1}{2^r}$ by

$$\mathcal{K}^{\frac{1}{2^r}} := \{k_{2^r}, k_{2\cdot 2^r}, k_{3\cdot 2^r}, \dots, k_R\}, \quad \text{where } R := \left\lfloor \frac{|F_m|}{2^r} \right\rfloor.$$

For $r \geq 0$, we always batch 2^r leaves together into a single time series step. At T_k, the time series has only been recorded for all $\mathcal{K}^{\frac{1}{2^r}} \cap [k]$.

We dynamically change the resolution in order to guarantee a maximum number of $C = 1024$ time series values, see also Section 6. Using powers of 2 for the resolution provides an efficient way to update the time series data during the search. For each time series we store at most C values at resolution 1 ($r = 0$). As long as the storage size is not exhausted (i.e., $|\mathcal{K}^{\frac{1}{2^r}} \cap [k]| < C$), every new value is appended, and the DES level and trend values are incrementally updated using (8). When the storage size is reached, we divide the current resolution by 2. At each resolution update we keep only every second observation, which is an aggregation of the information of two consecutive leaves, since the data at a leaf is already cumulative. This effectively shrinks the number of stored time series values to half the capacity $\frac{C}{2}$. We then recompute the DES fit for the compressed index set from scratch.

At a time series step t^*, we first make a prediction of the remaining number of time series steps h^* according to (9). We then take into account the current resolution to rescale h^* and estimate the total tree size m at termination via

$$\hat{m} := 2 \cdot (t^* + 2^r h^*) - 1, \tag{10}$$

where the term in parentheses is an estimation of the final number of terminal nodes $|F_m|$ and therefore turned into an estimate of the total tree size via (2).

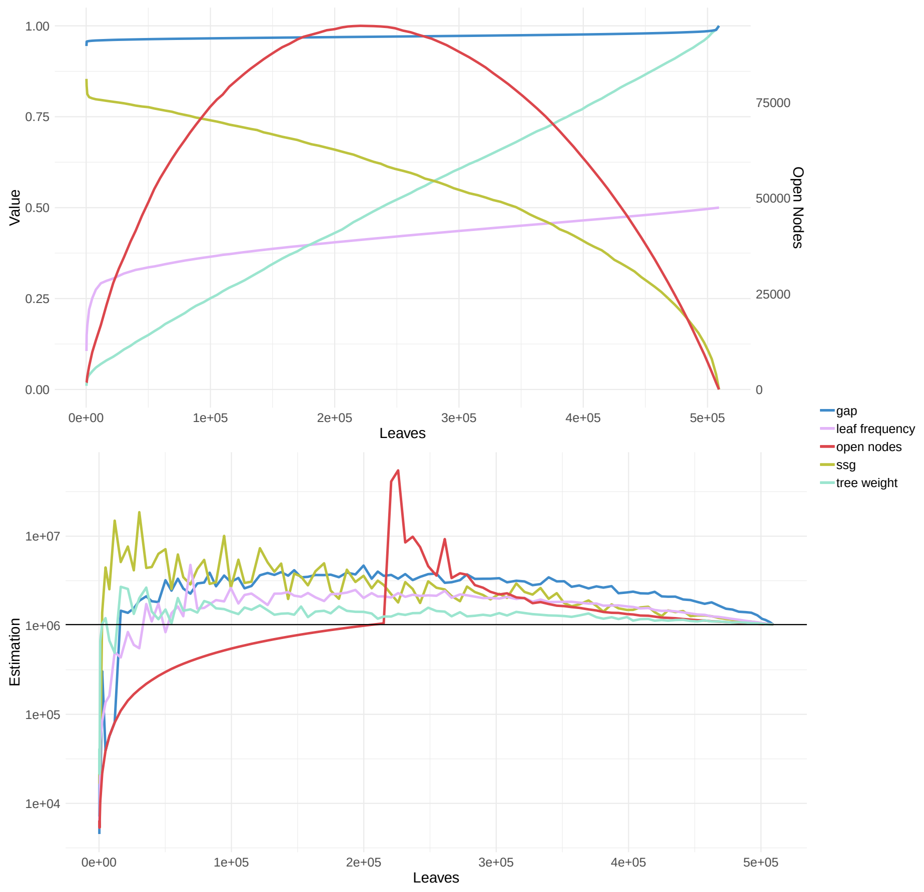


Figure 3: Examples of the five time series and their DES estimates as a function of leaves for instance `danoint`.

5.3 Example Instance `danoint`

In order to illustrate the DES based tree size estimates, we discuss their behavior for the MIPLIB instance `danoint` in detail. In Figure 3, we visualize the discussed time series and resulting tree size estimations. The data for the plots have been obtained with SCIP 6.0, which we extended by the necessary functionality to record all discussed time series and estimations, see also Appendix C. The time series estimations are all obtained by a DES forecast. As time series index, we use the index set $\mathcal{K}^{\text{leaf}}$ with an adaptive resolution as explained in Section 5.2. We allow a maximum number of $C = 1024$ observations. The parameters α and β have been calibrated for each time series individually, see Section 6. Our implementation outputs the estimations at every leaf node at which another percent of tree weight has been reached, yielding at most 100 records per instance. We compare five tree size estimations. The tree weight, leaf frequency, gap[1], SSG, and open nodes are illustrated as a function of the number of leaf nodes in the top plot of Figure 3. Note that the number of open

[1]For technical reasons, we show the complement $1 - \delta(T_k)$ of the gap.

nodes is shown with respect to the second scale at the right of the top plot. The bottom plot illustrates how the corresponding estimations evolves with an increasing number of leaf nodes.

Figure 3 depicts tree weight values that appear to increase almost linearly with the number of leaves. The corresponding tree weight estimations start steep at the beginning, yielding underestimations of the final tree size during the first 10,000 leaves. After approximately 20,000 leaf nodes, the estimations stays very close to the true final tree size of 1 million nodes, which is visualized as a black horizontal line. The tree weight estimation is consistently closer to the actual tree size than the other estimations during most of the search.

The values of the leaf frequency rise very steeply at the beginning of the search, but stabilize very soon and increase at a lower pace afterwards. The steep incline yields underestimations of the actual tree size at first. Later, it shows an overestimation of the actual tree size that is comparable to the SSG estimation. As noted before, the leaf frequency is not monotonic in general, although it appears strictly increasing in the top part of Figure 3.

The number of open nodes increases during the first half of the search and drops during the second half until it reaches zero at termination. Linear forecasts cannot cope with this reversing trend by themselves. As long as the number of open nodes increases, also the corresponding trend into the future stays positive, in which case we use twice the number of already explored nodes as estimation. When the number of open nodes starts to decrease, and hence the trend starts to become negative, the first few forecasts overestimate the final tree size, especially at the turning point. For the remainder of the search, the estimations converge to the true tree size. This example shows that the proposed time series estimations on tree weight and leaf frequency can yield quite accurate predictions early during the search.

6 Calibration of Double Exponential Smoothing

The quality of double exponential smoothing highly depends on the choice of the level and trend parameters α and β in (8). This section explains the calibration procedure of α, β, and the choice of capacity C for adaptive resolution. In our later experimental evaluations, α and β are then fixed for all instances. A direct calibration technique for α and β could consist, for example, in a grid search, where at every grid point we evaluate the accuracy of estimates on a set of MIP instances. In order to avoid repeating the same MIP runs, we solve each instance once and record the entire search, on which we can evaluate tree size estimation methods via an offline simulation[2] as in [8].

6.1 Accuracy Measures for Tree Size Estimation Methods

Similar to primal heuristics, tree size estimation methods need to be evaluated throughout the entire search process, rather than at the end of the search. An estimation may over- or underestimate the actual tree size at termination. We

[2]The simulation code is written in `R` [44] using the package `forecast` [26, 24]. It is publicly available from `https://github.com/GregorCH/treesize-estimation`.

therefore minimize the *normalized ratio*

$$E(\hat{m}, m) = \max\left\{e(\hat{m}, m), \frac{1}{e(\hat{m}, m)}\right\}, \quad \text{where } e(\hat{m}, m) := \frac{\hat{m}}{m},$$

penalizing under- and overestimations by the same factor (see, e.g., [29, 40]).

We report all ratios at different levels of tree weight $\omega(T_k)$, which is available during the search and normalized by definition. We simulate up to 95 estimations per tree and time series at those records at which the tree weight level first reaches $p \in \{0.01, 0.02, \dots, 0.95\}$. There can be fewer than 95 estimations as multiple weight levels can be first reached with a single leaf. As an example, the number of considered records for the `pigeon-10` tree is 91. At a resolution capacity $C = 1024$, the total number of considered records over the trees of all 233 instances that we use below is 10,322.

6.2 Simulation Setup

As training set, we use the combination of the following four MIP benchmark sets: MIPLIB 3 [9], MIPLIB 2003 [1], MIPLIB 2010 [31], and Coral [13]. Each of those 496 instances is solved using SCIP 6.0 [19] and the entire search tree is recorded as a VBC file [34]. Instances not solved to optimality within 2 hours or with fewer than 10 leaves are discarded, leaving 233 instances. The input data for every tree size estimation method for all leaves is then extracted from the VBC trees[3]. Our simulation code applies adaptive resolution to this offline data exactly as if it had been collected online. For example, the largest tree in our data set belongs to the instance `pigeon-10` and has almost 14 million leaf nodes. Its compressed tree contains 7507 records, at a final resolution of $\frac{1}{8192}$.

6.3 Calibration of the Adaptive Resolution Capacity C

Adjusting the resolution capacity C can lower the variance of the observations made at leaves by adaptively batching them. While we would like to optimize C, α and β together, this is computationally expensive. Instead, we first optimize C by a line search. For every value of C, we use ETS[4] [45], which considers errors, trend, and seasonality components of a time series, hence the name ETS. The key property of ETS for our simulation is that it optimizes α and β at each record to the available training data, thereby allowing for a calibration of C independent of the DES parameters.

We test capacities between $C = 2^3$ and $C = 2^{15}$, for which we compute the geometric mean normalized ratio E of the estimation obtained by an ETS forecast for each of the four measures tree weight, leaf frequency, open nodes, and SSG, see Figure 4. The rightmost entry uses an infinite capacity, which corresponds to not using adaptive resolution. At a capacity of 1024, all four measures achieve their best or second best normalized ratio over all tested capacities. Therefore, we consider the choice of 1024 as a good compromise between a coarse view on the entire search process and a fine view on the local, recent behavior. Note that in particular, the obtained normalized ratios at $C = 1024$ are consistently better than without adaptive resolution ($C = \infty$).

[3]Using a customized version of the Python code https://github.com/pierre-lebodic/bnb-mip-estimates/.

[4]We use the `forecast`-package of `R`, which features the ETS method.

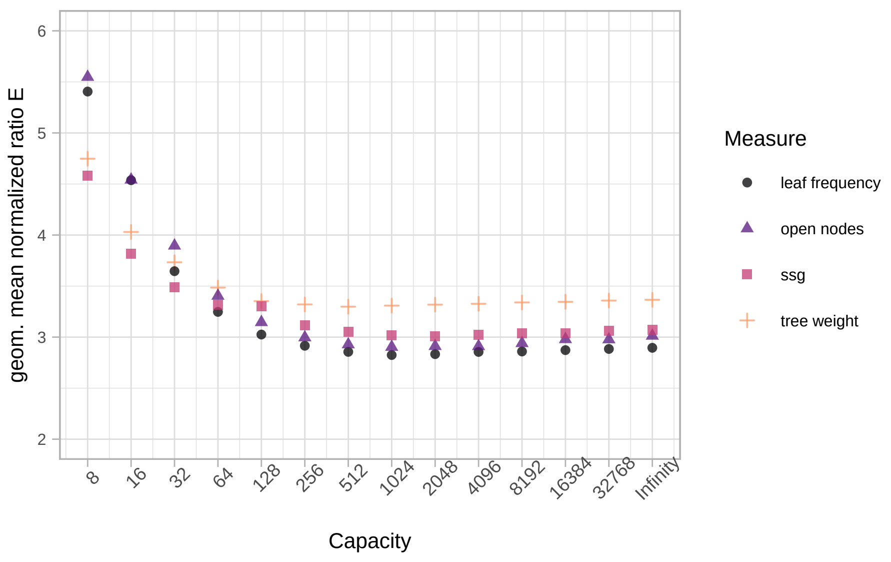


Figure 4: Normalized Ratio at different capacities C used by adaptive resolution.

Table 2: DES parameters that minimize the geometric mean normalized ratio $E(\hat{m}, m)$ for each time series on the training set.

Time Series	α	β	E	E ETS
Tree Weight	0.65	0.15	3.32	3.30
Leaf Frequency	0.30	0.33	2.71	2.82
SSG	0.60	0.15	2.71	3.01
Open Nodes	0.60	0.15	2.69	2.90

6.4 Calibration of the DES Parameters α and β

For every time series, we calibrate the values of α and β that minimize the geometric mean normalized ratio. We conduct a grid search by varying α in steps of 0.05 between 0.1 and 0.95 and β in steps of $\frac{0.05}{\alpha}$ between $\frac{0.1}{\alpha}$ and 1. Table 2 shows the obtained values of α and β per time series, their corresponding geometric mean normalized ratio E as well as the obtained normalized ratios using ETS instead (at $C = 1024$), which fits the parameters α and β adaptively to the time series at hand. Note that we did not explicitly optimize the parameters for the gap time series, for which we use the same parameters as for the SSG.

Interestingly, the calibrated parameters from Table 2 result in a lower (and hence, better) normalized ratio over the entire data set than could be obtained by ETS, despite the additional degrees of freedom of ETS. We explain this surprising result by the fact that ETS is too sensitive to the past behavior of the time series compared here, which yields a larger normalized ratio if an unpredictable event such as a new incumbent solution alters the search process significantly. The parameters from Table 2 may not fit the training data as accurately, but produce more robust estimations on average.

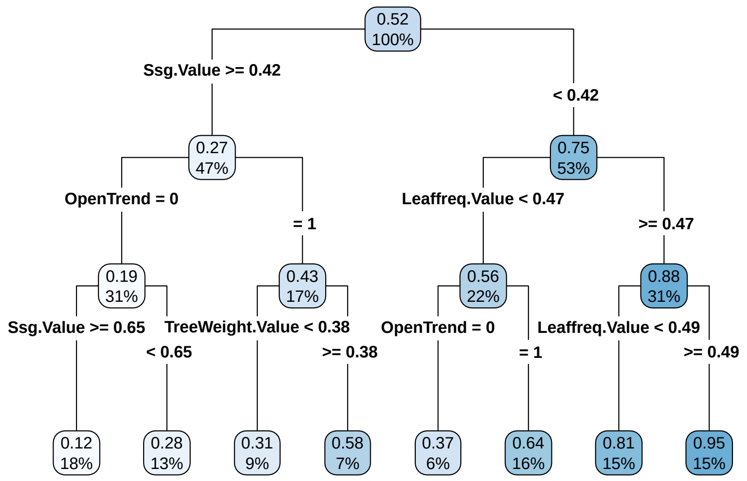


Figure 5: A visualization of the learned regression tree. In contrast to the linear models, this tree can only output one out of 8 distinct values as approximation of the search completion.

7 Learning the Search Completion

Recall that any approximation $0 < \hat{\gamma} \leq 1$ of γ_k can be trivially turned into a tree size estimate by computing $\hat{m} = k/\hat{\gamma}$, see (4). In this section, we use Machine Learning methods to train such approximations. This section builds upon both Sections 4 and 5 and the methods introduced therein.

7.1 Data Set

We use the same setup as in Section 6, but we also add MIPLIB 2017 [20], resulting in a set of 671 unique instances. We consider the 276 instances which can be solved within 2 hours and require at least 100 nodes, with up to 99 estimations per instance, resulting in a data set comprising 16k observations.

In order to validate the generalization of the learners on unseen instances, we randomly split the instances of our data set into 80 % training and 20 % test set, such that all records for one instance are fully contained in either the training or the test set.

7.2 Training

We formulate the problem to approximate the search completion based on the available measurements as a regression task. We have nine features: at each observation, we take both the measured value and the DES trend of the tree weight, SSG, leaf frequency, and gap time series. As a ninth feature, we use a boolean indicator equal to 1 if the number of open nodes has a decreasing trend component, and 0 otherwise.

We train four different regression models on all nine features, unless indicated otherwise, to learn the search completion:

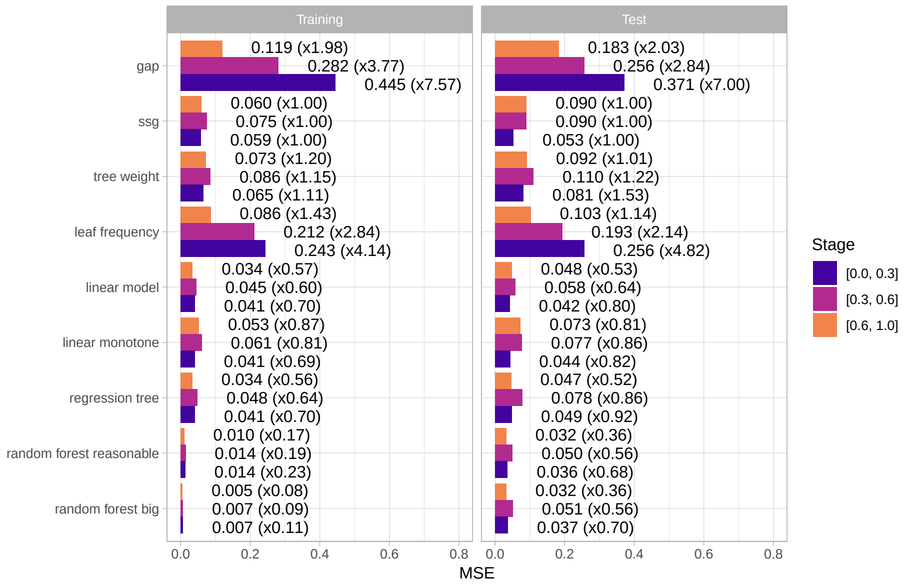


Figure 6: MSE between the search completion γ_k and its approximations.

1. `linear model:` a linear regression;
2. `linear monotone:` a linear regression on only two of the nine features: tree weight value and SSG value;
3. `regression tree:` a single regression tree;
4. `random forest:` a random forest regression.

A regression tree [11] partitions the training data recursively by selecting a feature and a value to split such that the variance after the split is minimized. The learned regression tree is depicted in Figure 5.

Random Forests [10] are ensembles of regression trees. For a random forest, a (predefined) number N of regression trees is trained on independently bootstrapped subsets of the training data samples, which contain approx. 60 % of training data. Also the number of features to consider at each split is limited. These choices prevent a random forest from overfitting. An approximation by a random forest regression is the mean approximation of its N regression trees. We conduct the learning procedure with the R package `randomForest`. This package also allows to set a limit on the number of samples at the terminal nodes of the tree, to prevent overfitting. We train a large random forest `random forest big` with $N = 200$ trees and a minimum terminal node size of 25 as well as one called `random forest reasonable` with $N = 100$ trees and a minimum node size of 75.

Figure 6 provides a comparison of the quality of the learned approximations compared to the quality of the base measures tree weight, SSG, leaf frequency, and gap. Unlike the other sections, we are interested in good approximations of the search completion $\gamma_k \in [0, 1]$. Using the final tree size instead as label for training will likely not give a meaningful generalization to unseen instances. Therefore, we report the *Mean Squared Error* (MSE) $(\hat{\gamma} - \gamma_k)^2$ between the approximation and the actual value, which is the target function that the learners attempt to minimize. We report the MSE separately for the training and test set. In parentheses, we show the ratio between the corresponding MSE and the MSE obtained by the SSG method, chosen as a baseline as it is the best performing method among the individual (untrained) measures of search completion. We further categorize the observations into an early, intermediate, and late stage based on the value of the tree weight measure $\omega(T_k)$: we call an observation early if $0 \leq \omega(T_k) \leq 0.3$, intermediate for $0.3 < \omega(T_k) \leq 0.6$, and otherwise late.

Figure 6 clearly shows that the gap measure alone offers the worst individual approximation quality, especially during the early and intermediate stages. Instead, the tree weight and SSG measures provide substantially closer approximations of search completion throughout all stages, with a slight advantage for the SSG.

Moreover, Figure 6 demonstrates that all learned methods improve upon the results of the best individual method SSG throughout all three stages on the test set. The monotone linear regression, which combines the SSG and tree weight measures, and the linear regression on all nine features improve upon the SSG by relative factors between 15–20 % and 20–50 %, respectively. Interestingly, a single regression tree achieves a comparable performance to a linear regression during the late stage, although the regression tree outputs one out of eight discrete values only.

The two trained random forest models achieve the best test results by a significant margin on the training set. On the test set, they improve upon the SSG by 30 %–64 % in the MSE.

The reason for the better performance of the regression forests compared to linear regression is that the feature values are only used indirectly by the forests, namely for deciding the "bucket" into which an observation falls, but the actual value that the regression forest assigns to an input is not computed as a weighted combination of the input as by linear regression. This allows to better integrate a logical feature such as the decreasing/increasing trend of the open nodes.

Among the tested methods, a random forest regression outperforms all other methods in all respects on both the training and the test set. Regarding the linear methods, it is surprising how well even the monotone estimation based on only tree weight and SSG works. The next section compares the estimation performance of these learned search completions to the individual methods from the previous sections.

Remarks We have also experimented with other regression techniques such as gradient boosted trees [17] and neural networks to approximate search completion. Furthermore, we also tried to combine the individual tree size estimates (including WBE and profile). In all cases, we omit the obtained results for brevity, which were comparable, but inferior to the accuracy of the presented random forests. In conclusion, it seems that attempting to approximate search completion instead of the actual tree size indeed works far better, since the wide variance of tree size estimations makes fitting a linear model, particularly regression, to the data very difficult and ineffective.

8 Comparison of Tree Size Estimation Methods

We are now ready to compare the tree size estimation quality of all discussed methods. We categorize the discussed methods into four groups based on how an estimation is derived.

1. The first group comprises the four search progress measures from Section 4. We treat (a suitable transformation of) each measure as approximation of the search completion γ_k and compute a tree size estimation using (4).
2. Two reference methods, the profile estimation [14] and the WBE [29], are treated as a separate group because they use different approaches. The profile estimation is only shortly presented in Appendix B, since it is conceptually different to the other methods and is not competitive in our experiments.
3. For the third group, all estimates are computed via DES, see Section 5. As explained in Section 5.2, the considered time series use an adaptive resolution with maximum capacity of $C = 1024$ and are indexed over the leaf index set $\mathcal{K}^{\text{leaf}}$. For each time series, the corresponding parameters α and β from Table 2 are used to update DES and produce estimations. For the gap time series, we use the same parameters as for the SSG.
4. The last group comprises five learned approximations of search completion using linear or forest regression, see Section 7.

For the two groups that approximate search completion, the computed approximation is corrected via $\tilde{\gamma} = \max\{\hat{\gamma}, 10^{-6}\}$, which is necessary because the gap and SSG approximations may still be at 0, and the linear regressions may even yield negative approximations in rare cases. As data set, we use the same 16k observations as in Section 7.1 and the same training/test split. We use the learned predictions based on the training set of Section 7 and apply them to the test set. We omit the results for the training set, on which the learned methods from Section 7 already have an advantage, and focus on the test set, which comprises a total of 3452 records.

We summarize the obtained estimation ratios on the test set records in three Tables 3–5, where we distinguish an early, intermediate, and late stage depending on the tree weight value. For every estimation method, we report the geometric mean normalized ratio E of its tree size estimation. This value is bounded from below by 1, which would be a perfect estimation. Nine of the tested methods derive their estimation from an approximation of the search completion. For those, we report the MSE between the approximation and the actual search completion. As in [40], we also present the percentage of observations that are ϵ-*accurate*, i.e., which satisfy $E(\hat{m}, m) \leq \epsilon$ for $\epsilon = 2, 3, 4$.

Analysis of the Search Completion Measures Among the search completion approximations, the methods `SSG` and `tree weight`, which already showed that they have a better approximation quality of the actual search completion than their counterparts `gap` and `leaf frequency`, are also better in terms of normalized ratio. Interestingly, the `tree weight` estimation yields considerably better estimations than `SSG` despite its worse approximation quality, as measured by the MSE.

The `tree weight` search completion achieves the lowest or second lowest ratio across all search completion methods across stages. Only during the late

Table 3: Estimate comparison during the early stage ($0 \leq \omega(T_k) \leq 0.3$) on test set

Method	n	MSE	E	2-Acc	3-Acc	4-Acc
Search Completion Approximation						
`gap`	861	0.371	36.305	20.7 %	34.7 %	40.0 %
`SSG`	861	0.053	5.702	43.0 %	58.9 %	66.3 %
`tree weight`	861	0.081	4.134	30.5 %	45.5 %	61.6 %
`leaf frequency`	861	0.256	6.590	29.0 %	44.5 %	52.5 %
Custom Estimations						
`profile`	861	–	79.930	19.5 %	30.3 %	35.7 %
`WBE`	861	–	4.686	30.4 %	46.9 %	59.1 %
Double Exponential Smoothing						
`open nodes`	861	–	5.717	40.9 %	52.1 %	58.1 %
`gap`	861	–	5.686	36.6 %	50.4 %	58.5 %
`SSG`	861	–	4.695	39.1 %	55.4 %	62.6 %
`tree weight`	861	–	4.532	35.1 %	51.9 %	60.7 %
`leaf frequency`	861	–	4.876	42.4 %	56.9 %	61.9 %
Learned Methods						
`linear model`	861	0.042	7.580	47.0 %	59.7 %	67.4 %
`linear monotone`	861	0.044	4.088	50.1 %	62.8 %	70.6 %
`regression tree`	861	0.049	3.447	49.6 %	62.0 %	69.9 %
`random forest big`	861	0.037	2.835	51.9 %	71.0 %	75.8 %
`random forest reasonable`	861	0.036	2.830	54.2 %	70.7 %	75.7 %

Table 4: Estimate comparison during the intermediate stage ($0.3 < \omega(T_k) \leq 0.6$) on test set

Method	n	MSE	E	2-Acc	3-Acc	4-Acc
Search Completion Approximation						
`gap`	1016	0.256	11.142	41.3 %	52.4 %	60.3 %
`SSG`	1016	0.090	3.891	62.7 %	73.1 %	76.2 %
`tree weight`	1016	0.110	3.034	53.4 %	75.7 %	81.2 %
`leaf frequency`	1016	0.193	3.196	54.8 %	62.8 %	69.7 %
Custom Estimations						
`profile`	1016	–	28.961	34.2 %	38.6 %	44.9 %
`WBE`	1016	–	3.242	57.1 %	70.5 %	78.7 %
Double Exponential Smoothing						
`open nodes`	1016	–	3.049	59.0 %	69.9 %	74.3 %
`gap`	1016	–	3.634	47.7 %	64.0 %	71.2 %
`SSG`	1016	–	2.924	54.8 %	70.1 %	78.0 %
`tree weight`	1016	–	3.069	56.2 %	71.2 %	76.2 %
`leaf frequency`	1016	–	2.926	59.0 %	72.2 %	78.9 %
Learned Methods						
`linear model`	1016	0.058	2.495	65.6 %	73.8 %	81.4 %
`linear monotone`	1016	0.077	2.678	62.5 %	75.3 %	78.3 %
`regression tree`	1016	0.078	2.554	60.4 %	73.5 %	78.9 %
`random forest big`	1016	0.051	2.273	65.2 %	79.3 %	84.4 %
`random forest reasonable`	1016	0.050	2.292	65.2 %	79.3 %	83.4 %

Table 5: Estimate comparison during the late stage ($0.6 < \omega(T_k)$) on test set

Method	n	MSE	E	2-Acc	3-Acc	4-Acc
Search Completion Approximation						
`gap`	1575	0.183	3.835	65.1 %	76.5 %	81.4 %
`SSG`	1575	0.090	2.118	78.9 %	84.6 %	89.4 %
`tree weight`	1575	0.092	1.688	81.4 %	87.2 %	88.5 %
`leaf frequency`	1575	0.103	1.662	77.8 %	85.3 %	88.8 %
Custom Estimations						
`profile`	1575	–	31.143	47.9 %	55.6 %	58.2 %
`WBE`	1575	–	1.714	79.6 %	86.6 %	88.4 %
Double Exponential Smoothing						
`open nodes`	1575	–	1.637	81.0 %	86.5 %	89.0 %
`gap`	1575	–	2.175	70.7 %	80.0 %	84.1 %
`SSG`	1575	–	1.689	78.0 %	84.4 %	87.7 %
`tree weight`	1575	–	1.681	81.0 %	85.6 %	87.7 %
`leaf frequency`	1575	–	1.662	81.5 %	88.4 %	90.9 %
Learned Methods						
`linear model`	1575	0.048	1.581	83.1 %	88.6 %	90.7 %
`linear monotone`	1575	0.073	1.669	80.2 %	85.3 %	89.7 %
`regression tree`	1575	0.047	1.511	85.0 %	89.2 %	91.2 %
`random forest big`	1575	0.032	1.443	86.9 %	90.9 %	92.4 %
`random forest reasonable`	1575	0.032	1.446	87.2 %	90.8 %	92.6 %

stage, the `leaf frequency` approximation of search completion yields a better normalized ratio. The two estimations `tree weight` and `WBE` are comparable in that they use the same measure, namely the tree weight measure ω, from which they compute an estimated number of nodes (`tree weight`) or leaves (`WBE`) of the final search tree. However, the tree weight estimation yields better estimations during all stages.

Analysis of Double Exponential Smoothing Forecasts Throughout all stages, the measures `SSG` and `gap` yield much better estimations within a time series approach than by projecting them as a measure of search completion. Both as search completion and as time series, a direct comparison between the `SSG` and `gap` estimations always shows a favorable behavior of the `SSG` method. This confirms the findings of [40] that a time series forecasting using `SSG` has a substantially better tree size estimation accuracy than the gap. The results also show that all three `gap`, `leaf frequency`, and `SSG` benefit from the use of DES. In contrast, the `tree weight` measure is best used as an approximation of search completion.

The custom method `profile` has by far the largest normalized ratio across all stages. A closer look at the individual records reveals that the `profile` estimation is the only method with serious overestimations. The most extreme overestimation of the `profile` method is on instance `ns1952667`, where the estimated tree size of $7 \cdot 10^{44}$ overestimates the actual, moderate tree size of 19k nodes by 40 orders of magnitude. Besides, the `profile` estimation does not necessarily converge to the actual tree size at termination, unlike all other tested methods.

Analysis of Learned Approximations As one may expect, for all time series, the estimate becomes more accurate at later stages with more data being available. As for the learned methods, the random forests outperform all other tested methods by a considerable margin in terms of ϵ-accuracy and E, as could be expected from their approximation quality measured by the MSE. The large normalized ratio during the early stage of the `linear regression` mostly arises from a few negative approximations of search completion, and despite the applied correction that maps these approximations into the allowed range. Note that the regression forest approximation is always positive, since it is the mean value of a subset of training labels, which are themselves positive search completions.

Despite their good performance, regression forests also have disadvantages. Their estimation cost grows linearly with the number of trees in the forest. In contrast, the costs to compute one linear (monotone) regression approximation in the nine-dimensional feature space is negligible[5]. Random Forests are therefore especially advantageous if the estimates do not have to be computed at every node, but infrequently during the search.

In this section, we have used observations across a variety of publicly available MIP benchmark sets, which have been compiled to cover a broad range of MIP applications. For instances from one specific type of application, the learning procedure should ideally be repeated to capture the search process of the B&B algorithm on those instances better than by our pretrained general purpose approximations.

Remarks One important influence on the behavior of a B&B-solver is the way in which improving solutions are found during the search. One can suspect that initializing the solution process with an optimal solution improves the quality of the predictions of the B&B-tree size. However, evaluating the normalized ratios as above shows that they actually increase slightly. This happens, because less records in the last phase of the search are available. If one factors this influence out, the ratios slightly decrease as expected. We do not present detailed results, because our goal was to predict the size of the B&B-tree for a general solution run, in which an optimal solution is not available.

9 Comparison on Homogeneous Instance Sets

In this section, we compare the estimation accuracy on six different homogeneous sets of MIP instances. All instance sets have been obtained from the public git repository of submissions[8] to MIPLIB 2017 [20]. While the MIPLIB 2017 collection of 1065 instances contains at most five instances from each such set, there are many more instances available from the repository. The selection of the six sets, which are shown in Table 6, has been made with respect to the following criteria:

1. sufficient number of instances,

[5]Also note that the monotonicity of the linear monotone regression can be slightly more pleasing when used as a progress bar during the search, although it is less accurate on average.

[6]23 instances $\times$ 4 different random seeds.

[7]42 instances $\times$ 4 different random seeds.

[8]`https://git.zib.de/miplib2017/revised-submissions-final.git`

Table 6: Data collected on homogeneous instance sets

Set	Example	Inst.	solv.	Records	Reference
`chromaticindex`	`chromaticindex1024-7`	121	105	1287	[33]
`generated`	`gen-ip002`	92	81	7772	–
`iis`	`iis-glass-cov`	92[6]	69	6765	[41]
`map`	`map06`	180	180	12240	[3]
`network-flow`	`g200x740`	168[7]	156	3536	[39]
`opm2`	`opm2-z12-s8`	150	92	1874	–

2. solvability with SCIP during the performance evaluation for MIPLIB 2017,
3. large enough trees.

A sufficient number of instances guarantees that we can obtain a meaningful separation into training and test set. For the sets of `iis` [41] and `network-flow` instances, we performed runs with four (default+3) random seeds to inflate the actual instance sets of 23 and 42 instances to 92 and 168 instance-seed combinations. As before, for each set, we first discard instances that could not be solved. Second we split the obtained records into 80 % instances for training and 20 % for testing. If multiple random seeds were used, the split ensures that the test set contains only unseen instances (and their respective seeds). For each set, we train independent search completion approximations on the obtained training sets.

Figure 7 summarizes the accuracy of all discussed methods in terms of the geometric mean normalized ratio E for all test records per instance set. For simplicity, we report the accuracy over the entire search process regardless of the early, intermediate, and late stages measured by tree weight. As in the previous section, we classify the tree size estimates into four groups. `Search Completion` comprises the four measures of search completion from Section 4. `DES` comprises five estimates that are derived from DES forecasts as explained in Section 5.1. All four measures of search completion also allow a DES forecast and hence appear twice in the figure. The group `Custom` comprises two further reference methods, `WBE` [29] and `profile` [14]. The last group `Learned` finally comprises five different learned search completion methods as presented in Section 7. In addition, the last group is enriched by `random forest miplib`, which corresponds to the random forest model `random forest big` from the previous sections 7 and 8 without further training on the application-specific data sets.

The learned random forests consistently achieve the smallest ratios among the methods. The random forest estimates are particularly accurate on the `map` instances, for which they achieve (almost) best possible ratios of 1.00 and 1.01, respectively. This very good result is possible because the instance set of 180 instances can be further grouped into 9 different subsets of instances. For each subset, the SCIP solution process is identical, because there is only little variation in the input data. The random forests can reliably recognize the similarities in the solution process and always assign the correct search completion even on unseen instances from the test set.

It is noteworthy that the individual estimates based on forecasting or search completion approximation vary substantially between the different sets. For instances from the set `opm2`, all individual forecasts and search completion methods yield ratios between 2.9 and 5.1, but can be effectively combined into a random forest with an acceptable ratio $E < 2$. Unsurprisingly, the random forests with application specific training always outperform the general random forest `random forest miplib`, while the latter one still achieves a superior performance to the DES forecasting estimates on all tested instance sets, and outperforms the linear regression estimates on three of the six tested instance sets.

10 Conclusion and Outlook

In this paper, we proved that it is impossible to approximate the size of the tree of a specific B&B algorithm within a factor of 2 in polynomial time, unless $\mathcal{P} = \mathcal{NP}$. We discussed and compared all state-of-the-art online methods and new methods to predict the size of the B&B tree at termination. We grouped the presented tree size estimates into approximations of the search completion on the one side and time series based estimates on the other side. We improved the time series forecasts by a careful calibration of the corresponding parameters, and by introducing adaptive resolution.

By far the best estimation quality is achieved by combining value and trend components of the individual time series into a learned regression forest that approximates search completion better than any individual method and yields superior estimation accuracy even on unseen test instances. As an additional validation, our study on six different homogeneous instance sets showed that the performance of a random forest can be significantly improved by training on a particular instance set. Nevertheless, using heterogenous general training data generalizes quite well to outperform most of the individual estimates.

Our results provide clear evidence that accurate tree size estimation requires a combination of several atomic measures such as tree weight and SSG to compensate for their individual weaknesses.

An efficient implementation of these estimates is included into SCIP as of version 7.0; more details are presented in Appendix C. It manifests during the solution process as a new display column of approximate search completion. By default, the display column shows the monotone linear regression because its approximation is easy to explain, improves upon the accuracy of the tree weight and SSG, and can be computed faster than its regression forest counterpart. For an even more accurate search completion approximation, user regression forests can be trained from SCIP log files via an external R script and input into SCIP.

The present work can be extended in various ways. Several of our methods require a binary search tree, which is the default in most state-of-the-art solvers. Measures such as the tree weight are easily generalized to nonbinary search trees at the cost of additional memory and computational effort [8]. The time series forecasting methods we used are mainly linear, investigating nonlinear functions or more advanced forecasting techniques might be beneficial. Finally, we hope that this work also inspires the use of the presented methods for algorithmic improvements. Two promising directions are the use for triggering restarts within a single solution process (see also [6]), or the use of search completion proxies in a massively parallel solver such as [46] for a better load-balancing.

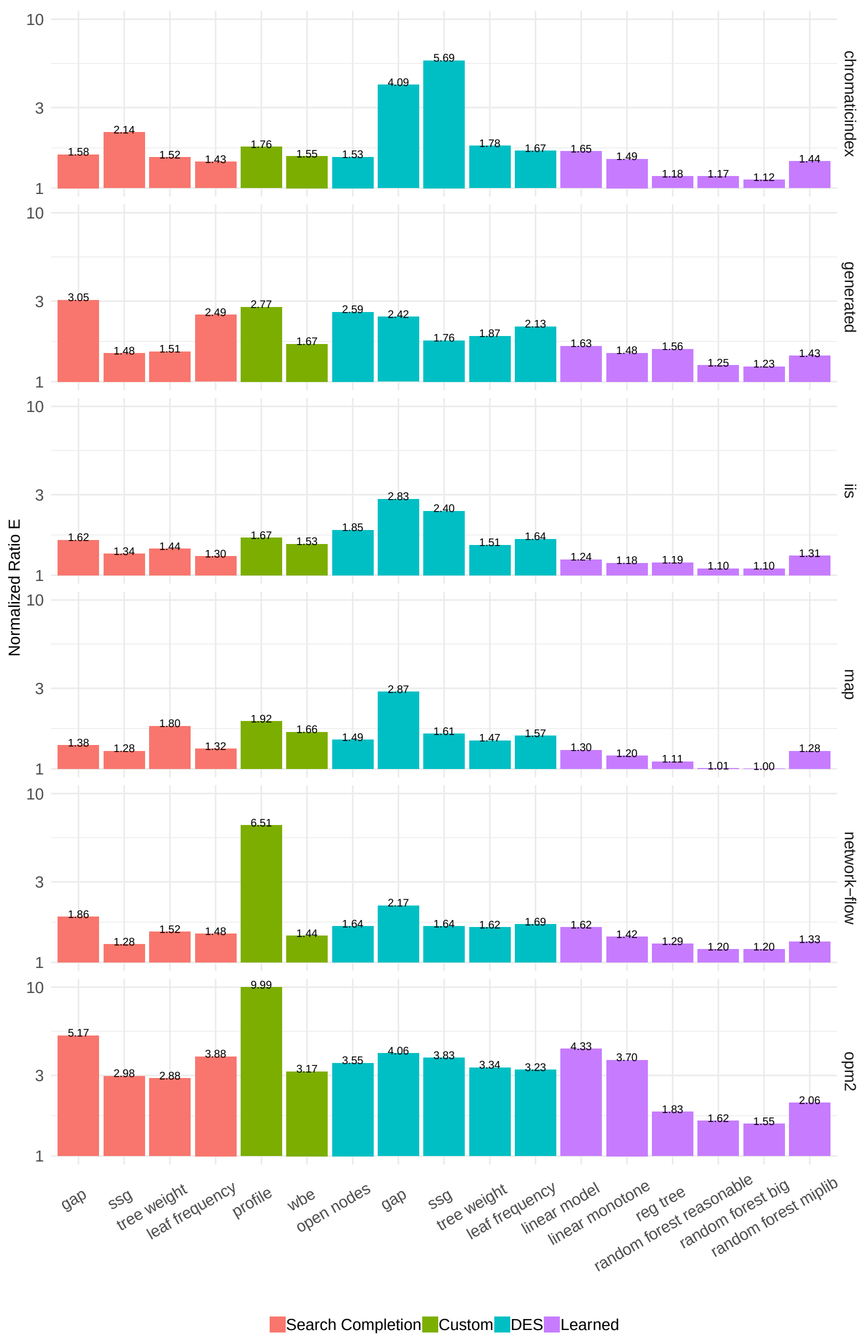


Figure 7: Geometric mean normalized ratio for six homogeneous MIP test sets.

Acknowledgments

The authors would like to thank Christoph Bergmeir for his many useful insights into time series forecasting and Lovis Anderson with his expertise on neural network training. The work for this article has been partly conducted within the Research Campus MODAL funded by the German Federal Ministry of Education and Research (BMBF grant number 05M14ZAM). The last author is supported by the DFG CRC/TRR 154, Subproject A01.

References

[1] T. Achterberg, T. Koch, and A. Martin. MIPLIB 2003. *Operations Research Letters*, 34(4):1–12, 2006.

[2] T. Achterberg and R. Wunderling. Mixed integer programming: Analyzing 12 years of progress. In M. Jünger and G. Reinelt, editors, *Facets of Combinatorial Optimization*, pages 449–481. Springer, 2013.

[3] K. Ahmadizadeh, B. Dilkina, C. P. Gomes, and A. Sabharwal. An empirical study of optimization for maximizing diffusion in networks. In *Principles and Practice of Constraint Programming*, volume 6308 of *Lecture Notes in Computer Science*, pages 514–521, 2010.

[4] A. M. Alvarez, Q. Louveaux, and L. Wehenkel. Online learning for strong branching approximation in branch-and-bound. Technical report, Université de Liège, 2016.

[5] A. M. Alvarez, Q. Louveaux, and L. Wehenkel. A machine learning-based approximation of strong branching. *INFORMS Journal on Computing*, 29(1):185–195, 2017.

[6] D. Anderson, G. Hendel, P. Le Bodic, and J. M. Viernickel. Clairvoyant restarts in branch-and-bound search using online tree-size estimation. In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence (AAAI-19)*, pages 1427–1434. AAAI Press, 2019.

[7] M.-F. Balcan, T. Dick, T. Sandholm, and E. Vitercik. Learning to branch. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018*, pages 353–362, 2018.

[8] G. Belov, S. Esler, D. Fernando, P. Le Bodic, and G. L. Nemhauser. Estimating the size of search trees by sampling with domain knowledge. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pages 473–479, 2017.

[9] R. E. Bixby, S. Ceria, C. M. McZeal, and M. W. Savelsbergh. An updated mixed integer programming library: MIPLIB 3.0. *Optima*, 58:12–15, 1998.

[10] L. Breiman. Random Forests. *Machine Learning*, 45(1):5–32, 2001.

[11] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Chapman & Hall CRC, 1983.

[12] P. C. Chen. Heuristic sampling: A method for predicting the performance of tree searching programs. *SIAM Journal on Computing*, 21(2):295–315, 1992.

[13] Coral MIP benchmark library, 2016. http://coral.ise.lehigh.edu/data-sets/mixed-integer-instances.

[14] G. Cornuéjols, M. Karamanov, and Y. Li. Early estimates of the size of branch-and-bound trees. *INFORMS Journal on Computing*, 18(1):86–96, 2006.

[15] R. J. Dakin. A tree-search algorithm for mixed integer programming problems. *The Computer Journal*, 8(3):250–255, 1965.

[16] M. Fischetti, A. Lodi, and G. Zarpellon. Learning MILP resolution outcomes before reaching time-limit. In L.-M. Rousseau and K. Stergiou, editors, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 275–291. Springer, Cham, 2019.

[17] J. H. Friedman. Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, 29(5):1189–1232, 2001.

[18] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, 1979.

[19] A. Gleixner, M. Bastubbe, L. Eifler, T. Gally, G. Gamrath, R. L. Gottwald, G. Hendel, C. Hojny, T. Koch, M. E. Lübbecke, S. J. Maher, M. Miltenberger, B. Müller, M. E. Pfetsch, C. Puchert, D. Rehfeldt, F. Schlösser, C. Schubert, F. Serrano, Y. Shinano, J. M. Viernickel, M. Walter, F. Wegscheider, J. T. Witt, and J. Witzig. The SCIP Optimization Suite 6.0. ZIB-Report 18-26, Zuse Institute Berlin, 2018.

[20] A. Gleixner, G. Hendel, G. Gamrath, T. Achterberg, M. Bastubbe, T. Berthold, P. M. Christophel, K. Jarck, T. Koch, J. Linderoth, M. Lübbecke, H. D. Mittelmann, D. Ozyurt, T. K. Ralphs, D. Salvagnin, and Y. Shinano. MIPLIB 2017: Data-driven compilation of the 6th mixed-integer programming library. Technical report, Optimization Online, 2019. http://www.optimization-online.org/DB_HTML/2019/07/7285.html.

[21] J. Håstad. Clique is hard to approximate within $n^{1-\varepsilon}$. *Acta Mathematica*, 182(1):105–142, 1999.

[22] G. Hendel. Adaptive large neighborhood search for mixed integer programming. ZIB-Report 18-60, Zuse Institute Berlin, 2018.

[23] C. C. Holt. Forecasting seasonals and trends by exponentially weighted moving averages. *International Journal of Forecasting*, 20(1):5–10, 2004.

[24] R. Hyndman, G. Athanasopoulos, C. Bergmeir, G. Caceres, L. Chhay, M. O'Hara-Wild, F. Petropoulos, S. Razbash, E. Wang, and F. Yasmeen. *forecast: Forecasting functions for time series and linear models*, 2019. R package version 8.5.

[25] R. Hyndman, A. Koehler, J. Ord, and R. Snyder. *Forecasting with exponential smoothing: the state space approach.* Springer-Verlag, 2008.

[26] R. J. Hyndman and Y. Khandakar. Automatic time series forecasting: the forecast package for R. *Journal of Statistical Software*, 26(3):1–22, 2008.

[27] E. B. Khalil, B. Dilkina, G. Nemhauser, S. Ahmed, and Y. Shao. Learning to run heuristics in tree search. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence (IJCAI)*, pages 659–666, 2017.

[28] E. B. Khalil, P. Le Bodic, L. Song, G. Nemhauser, and B. Dilkina. Learning to branch in mixed integer programming. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI-16)*, pages 724–731. AAAI Press, 2016.

[29] P. Kilby, J. K. Slaney, S. Thiébaux, and T. Walsh. Estimating search tree size. In *Proceedings of the 21st National Conference on Artificial Intelligence – Volume 2*, pages 1014–1019. AAAI Press, 2006.

[30] D. E. Knuth. Estimating the Efficiency of Backtrack Programs. *Mathematics of Computation*, 29(129):121–136, 1975.

[31] T. Koch, T. Achterberg, E. Andersen, O. Bastert, T. Berthold, R. E. Bixby, E. Danna, G. Gamrath, A. M. Gleixner, S. Heinz, A. Lodi, H. Mittelmann, T. Ralphs, D. Salvagnin, D. E. Steffy, and K. Wolter. MIPLIB 2010. *Mathematical Programming Computation*, 3(2):103–163, 2011.

[32] A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520, 1960.

[33] P. Le Bodic and G. L. Nemhauser. How important are branching decisions: Fooling MIP solvers. *Operations Research Letters*, 43(3):273–278, 2015.

[34] S. Leipert. The tree interface – version 1.0 user manual. Technical report, Zentrum für Angewandte Informatik Köln, 1996.

[35] L. H. S. Lelis, L. Otten, and R. Dechter. Predicting the size of depth-first branch and bound search trees. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, IJCAI '13, pages 594–600. AAAI Press, 2013.

[36] A. Lodi. Mixed integer programming computation. In M. Jünger, T. M. Liebling, D. Naddef, G. L. Nemhauser, W. R. Pulleyblank, G. Reinelt, G. Rinaldi, and L. A. Wolsey, editors, *50 Years of Integer Programming 1958-2008 – From the Early Years to the State-of-the-Art*, pages 619–645. Springer, Berlin, Heidelberg, 2010.

[37] G. L. Nemhauser and L. E. Trotter. Properties of vertex packing and independence system polyhedra. *Mathematical Programming*, 6(1):48–61, 1974.

[38] G. L. Nemhauser and L. E. Trotter. Vertex packings: Structural properties and algorithms. *Mathematical Programming*, 8(1):232–248, 1975.

[39] F. Ortega and L. A. Wolsey. A branch-and-cut algorithm for the single-commodity, uncapacitated, fixed-charge network flow problem. *Networks*, 41(3):143–158, 2003.

[40] O. Y. Özaltın, B. Hunsaker, and A. J. Schaefer. Predicting the solution time of branch-and-bound algorithms for mixed-integer programs. *INFORMS Journal on Computing*, 23(3):392–403, 2011.

[41] M. E. Pfetsch. Branch-and-cut for the maximum feasible subsystem problem. *SIAM Journal on Optimization*, 19:21–38, 2008.

[42] J.-C. Picard and M. Queyranne. On the integer-valued packing problem variables in the linear vertex packing problem. *Mathematical Programming*, 12:97–101, 1977.

[43] P. W. Purdom. Tree Size by Partial Backtracking. *SIAM Journal on Computing*, 7(4):481–491, 1978.

[44] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2018.

[45] G. A. R.J. Hyndman. *Forecasting: Principles and Practice*. OTexts: Melbourne, Australia, 2 edition, 2018.

[46] Y. Shinano, T. Achterberg, T. Berthold, S. Heinz, and T. Koch. ParaSCIP – a parallel extension of SCIP. In C. Bischof, H.-G. Hegering, W. E. Nagel, and G. Wittum, editors, *Competence in High Performance Computing 2010*, pages 135–148. Springer, 2012.

Appendices

A Formal description of the B&B algorithm

A general B&B algorithm as used in Section 3 is described in Algorithm 2.

Algorithm 2: Branch-and-Bound search algorithm for MIP

```
   Input: MIP (P)
   Output: Search state sequence $T = T_0, \dots, T_m$
1  $k \leftarrow 0$, $F_0 \leftarrow \emptyset$, $J_0 \leftarrow \emptyset$, $L_0 \leftarrow \{\text{root}(P)\}$;
2  $\Pi_0 \leftarrow \infty$, $\pi_0(\text{root}(P)) \leftarrow -\infty$;                // primal/dual bounds
3  while $L_k \neq \emptyset$ do
4      $k \leftarrow k + 1$;
5      select $v \in L_{k-1}$ and set $L_k \leftarrow L_{k-1} \setminus \{v\}$;
6      $\pi_k(v) \leftarrow$ objective value of relaxation of $P_v$;       // $\infty$ if infeasible
7      $\Pi_k \leftarrow \Pi_{k-1}$;
8      if relaxation solution is mixed-integer feasible and $\pi_k(v) < \Pi_k$ then
9          $\Pi_k \leftarrow \pi_k(v)$
10     if $\pi_k(v) \geq \Pi_k$ then
11         $J_k \leftarrow J_{k-1}$, $F_k \leftarrow F_{k-1} \cup \{v\}$;          // add $v$ to terminal nodes
12     else
13         $F_k \leftarrow F_{k-1}$, $J_k \leftarrow J_{k+1} \cup \{v\}$;            // add $v$ to inner nodes
14         create two children $v'$ and $v''$, $L_k \leftarrow L_k \cup \{v', v''\}$;
15 $m \leftarrow k$;
16 return $T \coloneqq (T_i = (J_i, F_i, L_i, \Pi_i, \pi_i) : i = 0, \dots, m)$;
```

The initial search state of the B&B tree is

$$T_0 = (J_0 = \emptyset,\ F_0 = \emptyset,\ L_0 = \{\text{root}(P)\},\ \Pi_0 = \infty,\ \pi_0 = -\infty)\,,$$

where root(P) denotes a (yet) isolated node that represents the original MIP problem without any tentative branching restrictions. We denote by P_v the local MIP problem associated with a tree node v. The first iteration $k = 1$ of Algorithm 2 will select root(P) as the only open node and solve the LP relaxation $P_{\text{root}(P)}$ in line 6.

The optimal objective value of the LP relaxation is computed, which determines the dual bound of the current node. The global primal bound Π_k is updated to $\pi_k(v)$, if the solution of the relaxation is integer feasible and decreases the bound, otherwise $\Pi_k = \Pi_{k-1}$. If the optimal solution to the root LP relaxation satisfies the integrality requirements for (P), it is necessarily optimal. Even in this extreme case that the solution process only requires a single node, i.e., the root node root(P), there will be two search states T_0 and T_1.

In all other cases, at least one branching operation takes place. Every time that the solution of the relaxation of P_v yields a dual bound $\pi_k(v)$ that is still smaller than the current primal bound Π_k, a branching operation in Line 12 creates two children for the currently active node v and adds them to L_k. We restrict ourselves to dichotomous (2-way) branching decisions in this paper, which are the most widely used types of branching decisions in current solvers, but the results can be easily extended to other branching schemes.

Algorithm 2 records a sequence of $m + 1$ search states $T_0, \dots, T_m$. We will refer to $T_1, \dots, T_{m-1}$ as *intermediate* states and to T_m as the *final* state. All states represent trees that have the same root and additional information regarding the primal and dual bounds. Between two successive search states T_k and T_{k+1} at some $k \in \{0, \dots, m-1\}$, the algorithm permanently removes the current node v from the set of open nodes L_k to either mark it as a final leaf or split it into subproblems by branching. Concretely, v can be marked to be a final leaf if its dual bound $\pi_k(v)$ exceeds the primal bound Π_k.

In particular, v is always added to F_k, if its relaxation is shown to be infeasible. Conversely, it may happen that the relaxation is mixed-integer feasible, which may result in a new primal bound Π_k. Also in this case, v is added to F_k in line 10 of Algorithm 2.

Every node $v \in F_k \cup J_k$ is denoted ***solved***. Note that the number of solved nodes increases by 1 with every loop iteration, such that $k = |F_k \cup J_k|$, i.e., the number of solved nodes is always equal to k. An important detail of Algorithm 2 is that pruning is performed explicitly so that each pruned node is counted. In practice, solvers such as SCIP do not report nodes that have been pruned as solved nodes. This ensures that each search state T_k represents a binary tree.

B Tree Profile Estimation

For search state T_k, let $d_k^{\max} := \max_{v \in F_k \cup J_k} d(v)$ denote the maximal depth of any solved node at T_k. A *depth profile* $\mathcal{D}_k$ is defined as

$$\mathcal{D}_k := \{|D_{k,i}| \,:\, i = 0, \ldots, d_k^{\max}\}, \text{ where } D_{k,i} := \{v \in F_k \cup J_k :\, d(v) = i\}.$$

The *maximum width depth* is $d_k^{\text{width}} := \operatorname{argmax}_i |D_{k,i}|$ and the *last full depth* $d_k^{\text{full}} := \max\{i \,:\, |D_{k,i}| = 2^i\}$. Following the intuition that between a (reasonably initialized) search state T_k and T_m these statistics do not differ too much, the *profile* estimate [14] approximates the growth factors[9]

$$\rho_{m,i} := \frac{|D_{m,i}|}{|D_{m,i-1}|}$$

as follows:

$$\rho_{k,i} := \begin{cases} 2, & \text{if } 1 \leq i \leq d_k^{\text{full}}, \\ 1 + \frac{d_k^{\text{width}} - i}{d_k^{\text{width}} - d_k^{\text{full}}} & \text{if } d_k^{\text{full}} < i \leq d_k^{\text{width}}, \\ 1 - \frac{i - d_k^{\text{width}}}{d_k^{\max} - d_k^{\text{width}}}, & \text{if } d_k^{\text{width}} < i \leq d_k^{\max}. \end{cases}$$

The growth factors are finally turned into an estimation of the final tree size as

$$\hat{m}^{\text{profile}} := 1 + \sum_{i=1}^{d_k^{\max}} \prod_{j=1}^{i} \rho_{k,i}.$$

C Implementation in SCIP

With the aim to provide a unified and understandable theoretical presentation, the B&B measures from Section 4 and the forecasting methods from Section 5 have been introduced as functions of the B&B search state T_k. The SSG as well as the time series methods require information from past search states, the former in form of a primal bound updating search state, the latter because double exponential smoothing takes into account the entire search sequence for forecasting.

[9]The growth factors are called γ-*sequence* in the original publication [14].

Clearly, it is computationally prohibitive to explicitly save the entire search state sequence. Furthermore, the measures and time series can be efficiently incremented during the search and need not be recomputed from scratch. This section provides some details about our SCIP implementation of the estimation methods. Our code is encapsulated as an event handler plugin that reacts on node events of the main search. It is invoked when a branching occurs in line 12 in Algorithm 2 or when a node becomes a final leaf in line 10. We extended the node event system of SCIP in order to capture such events even for open nodes that are pruned (e.g., because of a new primal bound). The event handler maintains the statistics $|F_k|$, $|J_k|$, $|L_k|$ for an internal model tree that counts all internal or final leaves as solved. We hence ensure that our model tree is in fact binary, so that our assumptions regarding the tree weight $\omega(T_m) = 1$ or the leaf frequency measure hold at the end of the search.

Some further remarks about specific implementations of the measures and estimations

Tree Weight, Leaf Frequency, Gap The values of all three measures tree weight $\omega(T_k)$, leaf frequency $\lambda(T_k)$ and gap $\delta(T_k)$ are updated in constant time from their values at T_{k-1}.

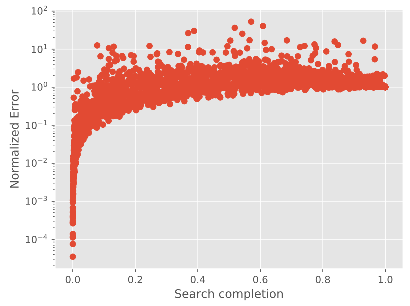
SSG The SSG [40] is more involved than the previous measures because it requires efficient updates of the individual subtree dual bounds for the different subtrees rooted at $L_{\text{pred}^{\Pi}(k)}$, i.e. the open nodes at the last primal bound improvement. For each subtree, we keep a priority queue of all open nodes $v \in L_k$ sorted by their dual bound. This allows for an efficient removal or addition of nodes with at most logarithmic effort $\mathcal{O}(\log(\max\{|subtree(v)| : v \in L_{\text{pred}^{\Pi}(k)}\}))$. The value of the SSG only changes if a dual bound defining node is removed from its respective subtree after the node has been branched or pruned. The priority queues are reinitialized at each update of the primal bound.

WBE The weighted backtrack estimation [29] can be computed in constant time from the value of the tree weight measure and the statistic $|F_k|$ of the model tree.

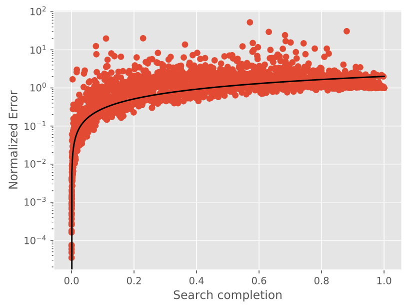
Profile Estimation For the profile estimation [14], we incrementally update the depth profile $\mathcal{D}_k$ in constant time by adding 1 to the entry $d(v)$ corresponding to the selected node v. The computation of $\hat{m}^{\text{profile}}$ can be done in $\mathcal{O}(d_k^{\max})$. In order to save time, we store the statistics $(d_{k'}^{\text{full}}, d_{k'}^{\max}, d_{k'}^{\text{width}})$ at the last step $k' < k$ when the estimation was computed, as well as the estimated tree size. If the statistics have not changed between k' and k, we report the same estimation.

D Further Improvements via Correction Factors

As observed in Section 8, the normalized ratio of each forecasting based estimation decreases as the search progresses. More interestingly, however, it can also be observed that the ratios of some methods tend to follow a particular pattern with respect to the search completion γ_k. Figure 8a depicts the ratio of the leaf frequency estimate for each record of the data set from Section 7.1. On a



(a) Errors versus search completion



(b) A curve fit to the ratio versus search completion

Figure 8: Average errors of the leaf-frequency-based tree size estimate versus the search completion.

Table 7: Geometric mean normalized ratio of the corrected leaf frequency estimates compared to the original method.

Method	Stage $0 \leq \omega(T_k) \leq 0.3$	$0.3 < \omega(T_k) \leq 0.6$	$0.6 < \omega(T_k)$
Training			
leaf-frequency	5.250	2.679	1.650
corr-leaf-freq	4.264	2.657	1.841
Testing			
leaf-frequency	5.106	2.733	1.645
corr-leaf-freq	4.397	2.721	1.876

logarithmic scale, the ratios display a logarithmic trend, which suggests a polynomial relationship between the normalized ratio and the search completion. We may exploit this trend in order to correct the estimate and obtain a more accurate method. We use a least-squares regression to fit a relationship between the logarithmic ratio and the logarithm of the search completion and use this result as a correction factor for the forecast estimate of the leaf frequency time series. Our method estimates this ratio to be $e = 2.03\,\gamma_k^{0.853}$, the reciprocal of which is hence our correction factor.

Since knowledge of the search completion is not available during the search, in an online setting, we instead employ the current tree weight as an approximation of the search completion. To evaluate the method, we randomly split the data set: 80% of the data is used for training, and the rest is left for testing. Table 7 compares the geometric mean normalized ratios of the corrected estimate, which we call `corr-leaf-freq`, to the original leaf frequency estimate. Observe that early in the search, the corrected estimate is significantly more accurate. Midway through the search, the two are indistinguishable, and towards the end, the uncorrected estimate is superior. This suggests that the correction factor is valuable especially during the early stage of the search, i.e., when the tree weight value is still small.