Efficient detection of repeating sites to accelerate phylogenetic likelihood calculations

Kassian Kobert[1], Alexandros Stamatakis[1,2], and Tomáš Flouri[1,2]

[1]Heidelberg Institute for Theoretical Studies, Germany
[2]Karlsruhe Institute of Technology, Institute for Theoretical Informatics, Karlsruhe

January 4, 2016

Abstract

The phylogenetic likelihood function is the major computational bottleneck in maximum likelihood and Bayesian phylogenetic inference. Given the alignment, a tree and the evolutionary model parameters, the likelihood function computes the conditional likelihood vectors for every node of the tree. Vector entries for which all input data are identical result in repeated likelihood operations which, in turn, yield identical conditional values. Such operations can be omitted for improving run-time and, by using appropriate data structures, also reduce memory usage. Here, we present a novel, fast method for identifying *and* omitting such redundant operations in phylogenetic likelihood calculations, and assess the performance improvement that is attained by our method. Using empirical and real data sets, we show that a prototype implementation of our method yields up to 10-fold speedups and uses up to 78% less memory than one of the fastest and most highly tuned implementations of the phylogenetic likelihood function available. Our method is generic and can seamlessly be integrated into any phylogenetic likelihood implementation.

1 Introduction

In phylogenetic tree analyses, such as maximum likelihood (ML) based tree searches or Bayesian inference (BI), the by far most costly operation is the repeated evaluation of the *phylogenetic likelihood function* (PLF). It is already known, that many operations performed during the PLF evaluation in popular ML tools such as PhyML [9] and RAxML [15] or BI tools such as ExaBayes [2] or MrBayes [14], are redundant and can thus be omitted to accelerate the PLF. For instance, savings can be achieved by taking into account that (sub-)trees with identical leaf labels (in our case nucleotides), identical branch lengths and the same model parameters always yield the same likelihood score or conditional likelihood values. Therefore, we can save computations by detecting repeating site patterns in the *multiple sequence alignment* (MSA) for a given (sub-)tree topology. We will refer to those repeating site patterns as *repeats*. A commonly used method exploiting this property consists in only evaluating the likelihood of unique sites (columns) of a MSA. Assuming that only one set of model parameters is used for the entire MSA (i.e. unpartitioned analysis), identical sites yield the same likelihood. Therefore, the likelihood can still be accurately calculated by assigning a weight to each unique site. These weights correspond to the site frequency in the original MSA. Felsenstein refers to this method as *aliasing* in the documentation of PHYLIP [6] (also referred to frequently as *site patterns*). Another standard technique for accelerating the PLF at *inner* nodes whose descendants are *tips* (or *leaves*) is to precompute the conditional likelihood for any combination of two states. Since there is a small, finite number of character states, those precomputed entries can be stored in a lookup table, and queried when needed, instead of repeatedly re-computing them. These two techniques are standard methods and are incorporated in virtually all PLF implementations. The benefits are faster computation times and often, in the case of the first method, considerable memory savings in the order of $d \cdot s \cdot r \cdot (t-2) \cdot c$, where d is the number of duplicate sites, s the number of states, r the number of rate categories, t the number of taxa (tips), and c a constant size for storing a conditional likelihood entry (typically 8 bytes for double precision). For example, on a phylogeny of 200 taxa with 100 000 duplicate sites, 4 states (nucleotide data) and 4 rate categories, the memory savings could be as high as 2.5 gigabytes, not to mention the savings in PLF computations.

In this paper we aim to minimize the amount of operations and memory usage for the PLF, by detecting all conditional likelihood entries at any node in the tree, that yield identical likelihood values. Computing these entries only once is sufficient to calculate the overall tree likelihood or any of the omitted (duplicate) entries. To be practically applicable, such an algorithm must exhibit certain properties. First, the overhead incurred by finding repeats must be relatively small such that the overall PLF execution is faster. Furthermore, hardware related issues such as non-linear cache accesses also need to be considered. For that reason, the speed of a new algorithm should be measured against a highly optimized software for PLF calculations and not a toy implementations. Second, the book-keeping overhead must be small such that it does not increase the PLF memory footprint. Third, the algorithm, and the corresponding data structures must be flexible enough to allow for so-called *partial* tree traversals. When proposing new tree topologies via some tree rearrangement (e.g., *nearest neighbor joining*, *subtree pruning and re-grafting*), not all conditional likelihood vectors need to be updated. An efficient method for calculating repeats must take this into account and analogously only update the necessary data structures for the partial traversal (i.e. subset of conditional likelihoods). Thus, the overall goal is to minimize the book-keeping cost for detecting repeats such that the trade-off is favourable.

Our results. We present a new, *simple* algorithm that generalizes the common PLF optimization techniques explained above and satisfies the efficiency properties; it detects identical sites at *any* node of the phylogenetic tree and not only at the (selected) root, and thus minimizes the number of operations required for likelihood evaluation. It is based on our linear-time and linear-space (on the size of tree) algorithm for detecting repeating patterns in general, rooted, non-phylogenetic trees [8]. In order to obtain the desired run-time improvements, we present an adapted version of this algorithm for the PLF that reduces book-keeping overhead and relies on two additional properties of phylogenies as opposed to general *multifurcating* (or $n-$ary) trees. First, we assume a *bifurcating* (binary) tree. This assumption can be relaxed to allow multifurcating trees by using a bifurcating tree that arbitrarily resolves the multifurcations. Second, the calculation of the so-called conditional likelihood depends on the transition probability of one state to another. These probabilities are not generally the same for different branches in the tree. Thus, we only consider identical nucleotide patterns to be repeats if they appear at the tips of the same (ordered) subtree. We show that a prototype implementation of the PLF, that makes use of our method, consistently outperforms one of the most efficient PLF implementations available by a factor of 2 to 10. In addition, the memory requirements are always significantly lower than for all widely used PLF implementations. In some cases up to 4 times **less** memory is required. For the theoretical part of this paper and for the sake of simplicity, we assume that genetic sequences only contain the four DNA bases (i.e., `A`, `C`, `G`, `T`). The approach we present can be easily adapted to any other number of states (e.g., degenerate DNA characters with gaps or protein sequence data). The data sets we use for benchmarking our method in Section 3 are empirical DNA data sets that *do* contain gaps and ambiguous characters.

Related work. Sumner *et al.* presented a method that relies on so-called partial likelihood tensors [17]. There, for each site of the alignment, the nucleotides at each tip node are iteratively included in the calculations. Let s_i be the nucleotide for site s at tip node i. The values are first calculated for (s_1), then (s_1, s_2), (s_1, s_2, s_3) and so on, until $(s_1, s_2, s_3, \ldots, s_m)$ has been processed, where m is the number of tip nodes. If the likelihood for another site s' with $s'_1 = s_1$, $s'_2 = s_2$ and $s'_3 \neq s_3$ is to be computed, the results for s restricted to the first two tip nodes (s_1, s_2) can be reused for this site. A lexicographical sorting of the sites is applied in order to approximately maximize the number of operations that can be saved with this method. The authors report run-time improvements for data sets with up to 16 taxa. For more than 16 taxa, the performance of the method is reported to degrade significantly. In addition, the authors measured the relative speedup of the PLF with respect to their own, unoptimized implementation and not the absolute speedup with respect to the fastest implementation available at that time. In [11] the idea of using general subtree site repeats for avoiding redundant PLF operations is mentioned, but dismissed as not practical because of the high book-keeping overhead. Instead, only repeating subtree patterns consisting entirely of gaps are considered since they can be easily identified by using and updating bit vectors, that is, the book-keeping overhead is low. In so-called gappy MSAs, with a high percentage of gaps, the authors report a speedup of 25-40% and 65% resp. 68% memory savings on gappy alignments consisting of 81.53% resp. 83.4% gaps (missing data). The underlying data structure used for identifying such repeating subtree sites is called subtree equality vector (SEV) and was originally introduced in [16]. There, only homogeneous subtree columns are considered. That is, a repeat is only stored as such, if all nucleotides in this subtree column are identical. This is done to avoid the perceived complexity associated with finding general (heterogeneous) subtree site repeats. In [16] a speedup of 19-22% is reported for the PLF computation. Similar to [17], the authors of [13] devised a method for accelerating the likelihood computation of a site by storing and reusing the results obtained for a preceding site. Since only the results for one single site (the preceding site) are retained, an appropriate sorting of the sites is required. This column sorting approach is reported to yield speedups in settings where the PLF is evaluated multiple times for the same topology. The authors showed that, sorting the sites in order to maximize the saving potential, can lead to run-time reductions of roughly 10% to over 80%. This corresponds to a more than 5-fold speedup. However, the authors also note, that an ideal algorithm for PLF calculations would reuse all previously computed values from all sites and not just the neighboring ones. Furthermore, the optimal column sorting relies on solving the NP-hard traveling salesman problem and relies on the tree topology. Thus, in order to construct a polynomial time algorithm, a search heuristic — that may yield sub-optimal results — is used. This means that, the proposed column sorting may not yield the maximum amount of savings. The most similar method to what we describe here also deploys a flavour of subtree repeats to accelerate the PLF has been presented in [18]. There, the PLF is used for a positive selection test. However, the authors focus on the performance for a fixed tree topology only that is repeatedly traversed. Thus, the overhead for detecting repeats is negligible, since repeats need to be computed only once. Here, we present a general method for dynamically changing trees. Performance was tested against the well known CODEML software of the PAML package [20].

2 Algorithm

First, we introduce the notation which we will use throughout the paper. A tree $T = (V, E)$ is a connected acyclic graph, where V is the set of nodes and E the set of *edges* (or *branches*), such that $E \subset V \times V$. We use the notation $(u, v) \in E$ to denote an edge with end-points $u, v \in V$ and $\ell_{u,v}$ to denote the associated branch length. The set $L(T)$ comprises the tip nodes. We use T_u to denote a subtree of a (rooted) tree T rooted at node u.

2.1 The phylogenetic likelihood function

Before we introduce our method, it is necessary to give a brief description of PLF computations. The likelihood is a function of the states σ, the transition probabilities P for all branches, and the equilibrium frequencies of the states $\pi = \bigcup_{\forall s \in \sigma} \{\pi_s\}$. The PLF can be further extended with additional parameters such as variable rates (denoted $\mathcal{C}$) of substitution across sites (see for instance [19]). In his seminal paper [4], Felsenstein introduced the *pruning* algorithm for computing the PLF, which is a dynamic programming approach for computing the likelihood of a given tree T. The method iteratively computes all *conditional* (or *partial*) likelihoods $L_u^{i,j}(s)$, that is, the

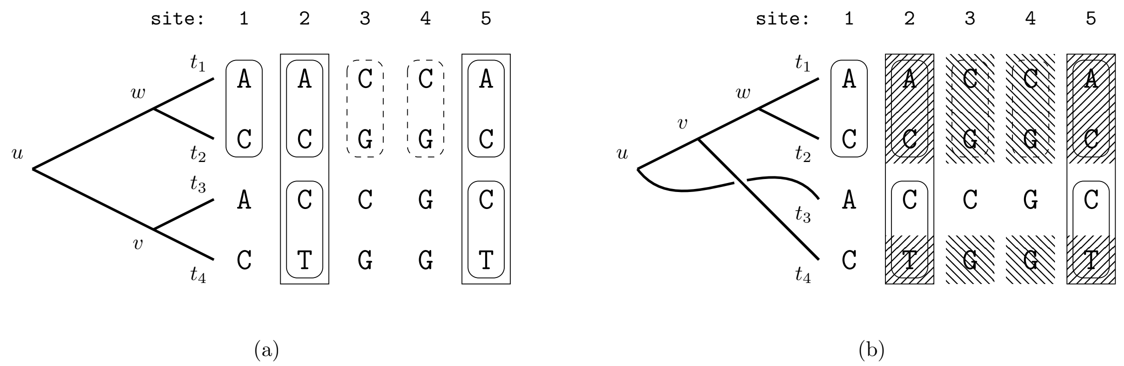

Figure 1: (a) Sites 1,2 and 5 form repeats at node w as they share the same pattern AC.Another repeating pattern is located at sites 3 and 4 (CG) for the same node. Note that, node u also induces a subtree with pattern AC at the tips. However, since branch lengths can be different than for the subtree rooted at node w, the conditional likelihoods may differ as well. Analogously, sites 2 and 5 are site repeats for node v as they have the same pattern CT, and hence the conditional likelihood is the same for those two sites. Finally, sites 2 and 5 form repeats for node u (ACCT). (b) Repeats are not necessarily substrings of MSA sites. For this particular tree topology, node v has two sets of repeats: sites 2 and 5 (ACT) and sites 3 and 4 (CGG). The repeats are not contiguous in the alignment columns.

likelihood for a subtree rooted at node u, for site i and rate j, assuming the state at node u is s, via a post-order (or bottom-up) traversal of the tree. Such a traversal always performs the first computation at a tip node and conducts computations at a node only after both its children were visited. Now, let us assume an MSA of n sites (columns) and m sequences constructed from s states (e.g., 4 for nucleotide data). The conditional likelihood $L_u^{i,j}(s)$ of any tip node $u \in L(T)$ with the sequence $x = x_1x_2 \dots x_n$ is defined as

$$L_u^{i,j}(s) = \begin{cases} 1 & : \quad s = x_i \\ 0 & : \quad s \neq x_i \end{cases}$$

In the case of data with ambiguities or gaps, we replace the conditions $s = x_i$ resp. $s \neq x_i$ with $s \in x_i$ resp. $s \notin x_i$. In the case of *inner* nodes, i.e. $u \notin L(T)$, the conditional likelihood for site i and rate j is defined as

$$L_u^{i,j}(s) = \left(\sum_{\forall s_v \in \sigma} P_{s \mapsto s_v}(j \cdot \ell_{u,v}) L_v^{i,j}(s_v) \right) \left(\sum_{\forall s_w \in \sigma} P_{s \mapsto s_w}(j \cdot \ell_{u,w}) L_w^{i,j}(s_w) \right).$$

where $P_{x \mapsto y}(z)$ is the probability of state x changing to y in z units of time, and v and w are the two descendants of u. We write the conditional likelihood vector (CLV) entries for all rates and all possible states at a particular site i of node u as

$$\mathcal{L}^u(i) = \bigcup_{\forall j \in \mathcal{C}} \bigcup_{\forall s \in \sigma} L_u^{i,j}(s).$$

Finally, the overall likelihood L for a rooted tree T with root node r is computed as

$$L = \prod_{i \leftarrow 1}^{n} \frac{1}{|\mathcal{C}|} \sum_{\forall j \in \mathcal{C}} \sum_{\forall s \in \sigma} L_r^{i,j}(s) \pi_s.$$

We evaluate the overall likelihood of an unrooted binary tree at branch (u, v) as

$$L = \prod_{i \leftarrow 1}^{n} \frac{1}{|\mathcal{C}|} \sum_{j \in \mathcal{C}} \left(\sum_{\forall s_u \in \sigma} L_u^{i,j}(s_u) \pi_{s_u} \sum_{\forall s_v \in \sigma} P_{s_u \mapsto s_v}(j \cdot \ell_{u,v}) L_v^{i,j}(s_v) \right).$$

For more details on the PLF, see [5].

		t_2			
		A	C	G	T
t_1	A		1		
	C			2	
	G				
	T				

1	2
$\mathcal{L}^w(1)$	$\mathcal{L}^w(3)$
AC	CG

(a)

		t_4			
		A	C	G	T
t_3	A		1		
	C			3	2
	G			4	
	T				

1	2	3	4
$\mathcal{L}^v(1)$	$\mathcal{L}^v(2)$	$\mathcal{L}^v(3)$	$\mathcal{L}^v(4)$
AC	CT	CG	GG

(b)

		w	
		1	2
v	1	1	
	2	2	
	3		3
	4		4

1	2	3	4
$\mathcal{L}^u(1)$	$\mathcal{L}^u(2)$	$\mathcal{L}^u(3)$	$\mathcal{L}^u(4)$
AC AC	AC CT	CG CG	CG GG

(c)

Figure 2: Identifier associations of nodes w (a),v (b), and u (c) for the tree from Figure 1a. The respective lists at the bottom store the corresponding CLVs that must be computed for each unique identifier. Table (a) shows that two likelihood computations need to be performed for node w (sites 1 and 3), while the rest of the sites are repeats of those two. Tables (b) and (c) show the corresponding information for nodes v and u.

2.2 Site repeats

We now introduce *site repeats*. Let T_u be a subtree of T rooted at node u, which represents the relations among $|L(T_u)|$ taxa (tip nodes). We denote the sequence of the i-th taxon $x^i = x^i_1 x^i_2 \ldots x^i_n$. Two sites j and k are called *repeats* of one another *iff* $x^i_j = x^i_k$ for all taxa i, $1 \leq i \leq |L(T_u)|$, in T_u. Next, we make two observations.

Observation 1 *If two sites j and k are not repeats in some tree T_u, then they are not repeats in any tree that has T_u as a subtree.*

Observation 2 *Let u be a node whose two direct descendants (children) are nodes v and w. If two sites j and k are repeats in both T_v and T_w, then j and k are also repeats in T_u.*

With these two observations we can formulate the algorithm for detecting site repeats in binary phylogenetic trees. Before we formalize the algorithm though, let us consider Figure 1 again. From Observations 1 and 2, we see that the only repeating sites at the root node (node u), are sites 2 and 5. This is obviously correct, since both have the nucleotide pattern `ACCT` at the tips.

2.3 Calculating Repeats

The method we propose identifies site repeats at each node via a bottom-up (post-order) traversal of the tree, meaning that a node can only be processed once the repeats for both its two children have been determined. Tip nodes only have the trivial repeats of all sites that show a common character (for DNA, `A`, `C`, `G`, or `T`, respectively). Therefore, the method always starts at an inner node whose two children are tip nodes. By construction, such a node always exists in any binary tree and assuming four nucleotide states, there are 16 possible combinations of homologous nucleotide pairs in the sequences of its two child nodes. We use a bijective mapping $\tau : \hat{\sigma} \times \hat{\sigma} \rightarrow \{1, 2, \ldots, \hat{\sigma}^2\}$ to assign a unique identifier to each nucleotide pair. The problem of identifying repeats is thus reduced to filling, and querying the corresponding entries in a list of CLVs. Note that, $\hat{\sigma}$ corresponds to the set of observed states (4 for nucleotides, or 16 when considering ambiguities and gaps).

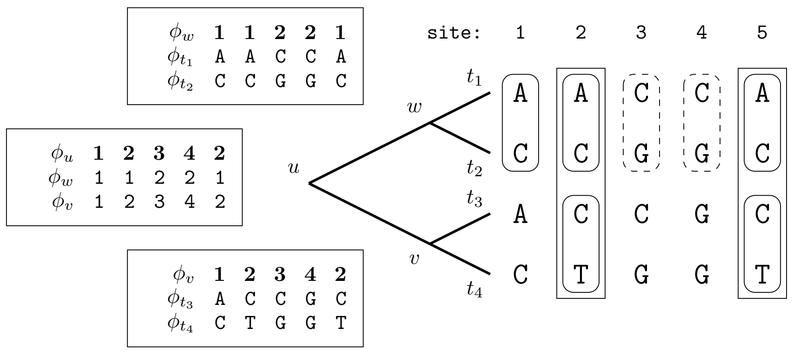

Figure 3: Identifiers (here ϕ_x) are shown for every site of the alignment at every node in the tree. As we have already observed, sites 2 and 5 are repeats at node u, and thus, have been assigned the same identifier. For simplicity, identifiers at tip nodes are represented as nucleotide bases.

Tip–Tip case. Assuming that x^v resp. x^w are the sequences at the two children v and w of the parent node u, site i of u is assigned the *identifier* $\phi_u(i) = \tau(x_i^v, x_i^w)$. This function assigns the same identifier to sites which are repeats in T_u. Figure 2 illustrates the assignment of identifiers to combinations of nucleotides at the tips for the example given in Figure 1. The CLV entries are computed only once for each identifier (for example, the first time it is encountered) at the parent node u. By Observation 2, if a site i is a repeat of site j, that is, they were assigned the same identifier, then the method can either (a) copy the CLV from site j (run-time saving), or (b) completely omit the likelihood value, since it can always be retrieved from site j (run-time *and* memory saving). Furthermore, by Observation 1, we know that each repeat is identified by this method.

Tip–Inner and Inner–Inner cases. We proceed analogously to detect repeats at nodes for which at least one child node is not a tip. Again, let u be the parent node and v and w the two child nodes for which all repeats have already been computed. Further, let $\phi_v(i)$ and $\phi_w(i)$ be the respective identifiers of v and w at site i. We define the maximum over all $\phi_v(i)$ and $\phi_w(i)$ as *vmax* and *wmax* respectively. The values *vmax* and *wmax* are also the numbers of unique repeats at nodes v and w. Now, finding repeats at u is again simply a matter of filling the appropriate lists/matrices. Given *vmax* and *wmax*, there are at most *vmax* $\times$ *wmax* combinations at the sites. See Figure 2c for the identifier calculation at node u for the example tree and MSA in Figure 1. Figure 3 shows the combined overall result.

Figure 4 outlines algorithm REPEATS(u, v, w, ϕ), which calculates the CLV for a given node u, with child nodes v and w. Using algorithm REPEATS we can now design the overall method by conducting a post-order traversal on all nodes of a tree T. For this, the tip nodes t_j can be assigned constant identifier sequences that correspond to their respective DNA sequences. The actual nucleotides A, C, G, and T can simply be mapped to integers. Note that, in most (if not all) phylogenetic inference tools, nucleotides are encoded using the *one-hot* (also called *1 out of N*) encoding, which ensures that the binary representations of their identifiers have exactly one bit set (e.g., $\mathtt{A} \mapsto 1$, $\mathtt{C} \mapsto 2$, $\mathtt{G} \mapsto 4$ and $\mathtt{T} \mapsto 8$). This is beneficial because the identifiers of ambiguities which are typically represented as disjoint unions of nucleotide codes, can be encoded as the bit-wise OR of the identifiers of the respective nucleotides. To simplify the method description, we discard ambiguities and consider only the four nucleotide bases. Hence, we use the encoding

$$\mathtt{A} \to 1, \mathtt{C} \to 2, \mathtt{G} \to 3, \mathtt{T} \to 4.$$

Lookup Table Since our focus is on an efficient implementation of the algorithm, we need to consider some technical issues in more detail. First, matrix M (defined in algorithm REPEATS) can,

```
REPEATS($u, v, w, \phi$)
        ▷ Initialization
 1.     $vmax \leftarrow 0$
 2.     $wmax \leftarrow 0$
 3.     for $i \leftarrow 1$ to $n$ do                          Get max identifier of node $v$
 4.         if $\phi_v(i) > vmax$ then $vmax \leftarrow \phi_v(i)$
 5.     for $i \leftarrow 1$ to $n$ do                          Get max identifier of node $w$
 6.         if $\phi_w(i) > wmax$ then $wmax \leftarrow \phi_w(i)$
 7.     $M \leftarrow \{0\}^{vmax \times wmax}$                 Initialize matrix $M$
 8.     $LH \leftarrow \{0\}^n$
 9.     $ident \leftarrow 0$
        ▷ Computation
10.     for $i \leftarrow 1$ to $n$ do                          Iterate over sites
11.         if $M[\phi_v(i), \phi_w(i)] = 0$ then               Check if site is a repeat
12.             $ident \leftarrow ident + 1$                    Increase identifier count
13.             $M[\phi_v(i), \phi_w(i)] \leftarrow ident$      Set an identifier for the site
14.             $LH[ident] \leftarrow \mathcal{L}^u(i)$         Compute likelihood entries for site
15.             $CLV[i] \leftarrow LH[ident]$                   Place likelihood entry in CLV
16.         else                                                Site is a repeat
17.             $CLV[i] \leftarrow LH[M[\phi_v(i), \phi_u(i)]]$ Copy likelihood entry from repeat
18.         $\phi_u(i) \leftarrow M[\phi_v(i), \phi_w(i)]$      Set site identifier
19.     return $CLV$                                            return CLV
```

Figure 4: Algorithm to compute the CLV of a parent node p. The most costly operation is the calculation of the CLVs, here, denoted by $\mathcal{L}^u(i)$. The algorithm thus minimizes the number of calls to this function.

in the worst case, become quadratic in size with respect to the number of sites in the alignment. This is unfortunate, since filling M affects overall asymptotic run-time. However, in terms of practical space requirements, M needs to be allocated only once and can be re-used for each inner node. For that, a linear list *clean* with one entry per MSA site, can be used to keep track of which entries are *valid*, that is, contain identifiers assigned to sites of the current node, and which entries are *invalid* and contain identifiers assigned to the sites of a preceding node. After assigning an identifier i to a site of a node u, which we store in the array M, for example at position d, we also store the pair (d, u) in array *clean* at position i. Later on, when we process a different node, say v, and by chance, decide to assign the same identifier i to some site, and again, by chance, the location for which we have to query matrix M is d, the element *clean*$[i]$ helps us to distinguish between valid and invalid records in M. Invalid records are equivalent to empty records and are overwritten. Further, in the actual implementation we limit the size of M to a constant maximum size. We implement this limit to avoid the impact of the quadratic complexity for filling M. Table 2 of Section 3 gives an overview of the size of M for different data sets. Since dynamically tuning the size of M to the data set can have a negative impact on the run-time and memory performance, the size of M is an input parameter. In addition, as M grows larger (i.e., we move closer to the root of the tree), it is less likely to encounter repeats in the alignment. Note that, at the CLV of the root, there will be no repeats at all, since they have already been removed by compressing MSA sites into patterns during MSA pre-processing. One may also consider the following alternative view. If M is an $n \times n$ matrix, where n is the number of sites in the alignment, there can be no repeats, as every site must, by construction, have a unique identifier. If at least two sites were repeats of another, the maximal identifier would be strictly less that n and thus, M would not be a $n \times n$ matrix. Thus, if the product of maximum identifiers for two child nodes at some node u (that is, $vmax \times wmax$) exceeds our threshold for the size of M, we do not calculate repeats any more. Instead, the CLV entries are calculated separately for all sites as in

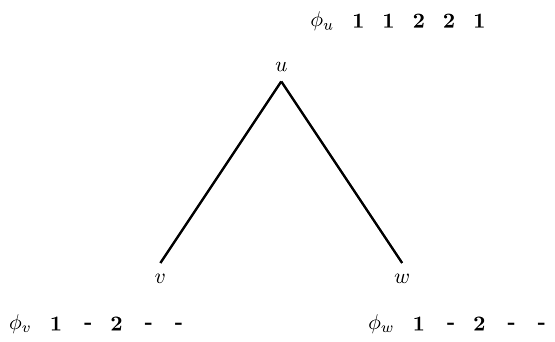

Figure 5: Not all sites are needed for the likelihood calculation at parent node u. According to the identifiers of this example, sites 2 and 5 are repeats of site 1, and site 4 is a repeat of site 3. Therefore, the CLVs at sites 2, 4, and 5 do not need to be computed nor stored, as the CLV for sites 2 and 5, and site 4, of node u can be copied from sites 1, and 3, respectively.

standard PLF implementations. In other words, if calculating repeats becomes disadvantageous, repeat calculations are omitted. This allows to trade-off repeat detection overhead, against PLF efficiency.

Memory savings Notice that, given algorithm Repeats, not all entries in the CLVs of the child nodes v and w are needed to calculate the CLV at the parent node u. In particular, the CLV entries at site i for nodes v *and* w are only needed if the CLV at site i must be computed for u (see Figure 5). In fact, only the CLVs in array LH of algorithm Repeats must be stored. Let $LH[u, j]$ be the CLV computed by Algorithm 4 for node u and the site with identifier j. Then, the CLV for any site i is simply $LH[\phi_u(i)]$. In practice, this observation enables us to reduce the memory footprint of the PLF. Each CLV entry stores more than one single or double precision floating point values. For example, RAxML stores one double precision floating point number per DNA character and per evolutionary rate for each CLV entry. Typically, the Γ model of rate heterogeneity is used (see [19]) with 4 rate categories. Thus, the memory footprint of a standard PLF algorithm for a MSA with n sequences of length m is $8 \times 4 \times 4 \times (n-2) \times m$ bytes. On the other hand, storing the site identifiers at each node only requires a single, unsigned integer per site. Thus, the memory required for storing CLVs without compression is $4 \cdot 4 = 16$ times higher than that of the site identifier list.

Thus, despite the fact that, we need additional data structures, and hence space for keeping track of the site identifiers at nodes, the memory requirements (if we do not store unnecessary CLV entries) are smaller than those of standard production level tools [7, 15]. While the identifiers are not the only additional data structures required for the actual implementation of the algorithm, the above argument illustrates that storing fewer CLV entries can help to save substantial amounts of RAM.

The overall algorithm, with memory savings and a bounded M, is given by algorithm Repeats-Full in Figure 6. One main difference to the snippet of Figure 4, is the introduction of a new array ($maxid$) which stores the maximal identifier assigned to each of the $2m-1$ nodes of the rooted tree T (assuming T has m tip nodes). Thereby, we eliminate the run time $\mathcal{O}(n)$ required for finding the maximal identifiers of the two children nodes (lines 3-6 in Figure 4) at the cost of $\Theta(m)$ memory. The second difference is that, we can no longer use the original set $\mathcal{L}^u(i)$ for the CLV entries of a site i at a node u. This is due to the memory saving technique which omits the computation and storage of unnecessary CLVs as illustrated in Figure 5. The problem is that the CLV of the two children may not reside at entries i because repeats might have occurred.

```
REPEATS-FULL($T, \tau, tsize, x, n, m$)
 ▷  Initialization
 1.  $M \leftarrow \{0\}^{tsize}$                          Initialize matrix $M$
 2.  $clean \leftarrow \{0,0\}^n$                          Initialize $clean$ array
 3.  $maxid \leftarrow \{0\}^{2m-1}$                       Initialize $maxid$ array
 4.  $P \leftarrow \{u_1, u_2, \ldots, u_{m-1}\}$          Post-order traversal of inner nodes
 5.  ▷ Map nucleotides at tips to integers
 6.  for $u$ in $L(T)$ do                                  Iterate over all tip nodes $u$
 7.     for $i \leftarrow 1$ to $n$ do                     Iterate over all sites of sequence $x^u$
 8.        $\phi_u(i) \leftarrow \tau(x_i^u)$              Use mapping $\tau$ to encode nucleotide $x_i^u$
 ▷  Traverse all inner nodes in post-order
 9.  for $u$ in $P$ do                                     Iterate through inner nodes
10.     $v \leftarrow$ LEFT-CHILD($u$)                     Set $v$ as the left child of $u$
11.     $w \leftarrow$ RIGHT-CHILD($u$)                    Set $w$ as the right child of $u$
12.     $vmax \leftarrow maxid(v)$                         Get maximal identifier of $v$
13.     $wmax \leftarrow maxid(w)$                         Get maximal identifier of $w$
14.     if $vmax \times wmax > tsize$ then                 Check if table size reached
15.        for $i \leftarrow 1$ to $n$ do
16.           $CLV[u,i] \leftarrow \hat{\mathcal{L}}^u(i)$            $\hat{\mathcal{L}}^u(i)$ uses $CLV[v,\phi_v(i)]$ and $CLV[w,\phi_w(i)]$
17.           $\phi_u(i) \leftarrow i$
18.     else                                               we can still use site repeats
19.        $ident \leftarrow 0$
20.        for $i \leftarrow 1$ to $n$ do
21.           $mpos \leftarrow (\phi_v(i) - 1) \times wmax + \phi_w(i)$    linearize two coordinates into one index
22.           if $M[mpos] = 0$ or                          If matrix entry is empty or contains invalid
                 $clean[M[mpos]] \neq (mpos, u)$ then      data, then compute likelihood from scratch
23.              $ident \leftarrow ident + 1$
24.              $M[mpos] \leftarrow ident$
25.              $clean[M[mpos]] \leftarrow (mpos, u)$
26.              $CLV[u, ident] \leftarrow \hat{\mathcal{L}}^u(i)$     $\hat{\mathcal{L}}^u(i)$ uses $CLV[v,\phi_v(i)]$ and $CLV[w,\phi_w(i)]$
27.           $\phi_u(i) \leftarrow M[mpos]$
28.        $maxid(u) \leftarrow ident$                     Store max identifier for $u$
29.  return $CLV$                                          return CLV
```

Figure 6: Full description for computing all CLVs of a tree T with the memory saving technique and site repeat detection. Input parameters are the tree T of m taxa, the sequences (of size n) for each of the m taxa (denoted x^u for the sequence at tip node u), a mapping τ for encoding the MSA data to integer values, and the size $tsize$ of the matrix used for computing site repeats. The algorithm computes only the necessary CLVs required for evaluating the likelihood of tree T, avoiding PLF calls on site repeats.

Therefore, the new set $\hat{\mathcal{L}}^u(i)$ is defined as

$$\hat{\mathcal{L}}^u(i) = \bigcup_{\forall j \in \mathcal{C}} \bigcup_{\forall s \in \sigma} \left(\sum_{\forall s_v \in \sigma} p_{s \mapsto s_v}(j\ell_{u,v}) L_v^{\phi_v(i),j}(s_v) \right) \left(\sum_{\forall s_w \in \sigma} p_{s \mapsto s_w}(j\ell_{u,w}) L_w^{\phi_w(i),j}(s_w) \right)$$

and the CLVs $L_v^{\phi_v(i),j}$ resp. $L_w^{\phi_w(i),j}$ for all rates $j \in \mathcal{C}$ can be obtained from $CLV[v, \phi_v(i)]$ resp. $CLV[w, \phi_w(i)]$.

Observation 3 (Runtime) *Algorithm 6 computes all subtree repeats, and the corresponding CLVs, in linear time, with respect to the size of the alignment (number of sites times number of nodes).*

This trivially holds by inspection.

3 Computational Results

We can now compare the performance of the PLF implementation using our algorithm, against a standard implementation for this task. We implemented a prototype of our algorithm in a new, low-level implementation of the *Phylogenetic Likelihood Library* PLL [7] (which we refer to as LLPLL) that does not make use of the highly optimized PLF of PLL, but allows for a straightforward implementation of our method. We used two implementations of our method. First, SRDT which computes the site repeats assuming a dynamically changing topology, that is, repeats are pre-computed before each likelihood computation. The second variant (SRCT) assumes a constant tree topology and therefore pre-computes all site repeats once and uses that information every time the likelihood function is called without re-computing repeats. We compare the performance of our method against the AVX-vectorized, sequential PLF implementation from PLL which uses the same, highly optimized PLF as RAxML. We selected the PLL/RAxML because (i) it is our own code and (ii) it is currently among the fastest and most optimized PLF implementations available. This guarantees a fair comparison, and ensures that our method can truly be used in practice for speeding up state-of-the-art inference tools. We use two flavors of PLL in our experiments; the plain version (we refer to it as PLL) and the memory saving SEV-based implementation of PLF (accessible using the `-U` switch in RAxML) which we refer to as PLL-SEV. To obtain an accurate speedup estimate, we also used AVX intrinsics in our low-level repeats implementation (LLPLL). However, it is still a prototype as PLL is faster by a factor of approximately 1.40 - 1.45 as we show further.

Experimental setup. We performed four types of experiments for assessing the performance of our method. The results indicate that the prototype implementation of our method outperforms the PLL likelihood function by up to a factor of 10. The four experiments cover the typical PLF use cases. First, we exhaustively assess the performance of *full traversals* for all possible rootings of the trees on two data sets. Second, we assess the performance of full traversals on all selected data sets for a limited number of rootings drawn at random. Third, we evaluate the performance for *partial* traversals. Finally, we assess PLF performance for fixed tree topologies. In this setting, preprocessing of site repeats is done only once and not for each likelihood evaluation. For the experiments we used a 4-core Intel i7-2600 multi-core system with 16 GB RAM. To eradicate the potential impact of server-side events such as context switching or performance peaks of running processes, we always executed several (usually 10 000) independent likelihood computations.

Data sets. We used a mixture of empirical and simulated nucleotide data sets which are summarized in Table 1. The data sets contain gaps and ambiguous DNA characters. Table 1 also reports the percentages of gaps and site repeats in the alignments. The number of gaps is important, since it affects the performance of the PLL-SEV implementation. The percentages of site repeats are given for an arbitrary root of the parsimony trees calculated for the data sets using RAxML [15]. The data set with 2 000 taxa has the lowest percentage of repeats, however, this data set still produces 86.95% repeats (which directly translate to identical conditional likelihood entries). We want to emphasize that we did not choose these data sets for their high numbers of repeats. In fact, the fraction of site repeats for each data set was previously unknown to us. Our implementation, as well as the data sets used for testing, are available on an on-line repository[1]. For the run-time comparisons we focus purely on the PLF evaluation. Branch lengths and model parameters are fixed, and remain unchanged as they do not impact the run-time of PLF. The underlying trees are parsimony trees inferred with RAxML [15]. Since the calculation of the PLF takes up up to 85%-98% of the total run-time [3] of ML phylogenetic tree inferences, accelerating the performance of the PLF significantly impacts the overall execution time of ML analyses.

[1]`https://github.com/stamatak/test-Datasets/`

Sequences [-]	59	128	354	404	500	994	1 512	2 000	3 782	7 764
Sites [-]	6 951	29 198	460	13 158	1 398	5 533	1 577	1 251	1 371	851
Repeats [%]	92.04	91.78	94.65	96.49	89.43	94.63	90.09	86.95	94.18	87.62
Gaps [%]	44.24	32.48	14.71	78.92	2.25	71.39	3.02	12.65	2.70	20.60

Table 1: Nucleotide data sets summary. Sequences denotes the number of taxa in the data set. Sites is the length of the provided MSA. Repeats denotes the amount (in percentage) of sites in the MSA which are repeats of another at any node, and can thus be copied or omitted. This amount depends on the chosen root of the tree structure and the tree topology itself. The (unrooted) trees were obtained by running a maximum parsimony tree search for each of the data sets, and we chose one random node as the root to estimate the number and the table indicates the amount of repeats for that particular rooting. Gaps indicates the amount of gaps in the alignment.

Sequences	59	128	354	404	500	994	1 512	2 000	3 782	7 764
Sites	6951	29 198	460	13 158	1 398	5533	1 577	1 251	1 371	851
Memory PLL [MB]	53	474	24.5	680	93	707	312	328	678	875
Memory PLL-SEV [MB]	46	403	21.5	326	93	256	308	297	674	819
Memory SRCT [MB]	32	303	7.5	202	34	164	104	120	171	298
Table size	5.3	220.1	0.07	23.5	0.81	6.6	2.9	2.4	2.8	0.86

Table 2: Memory requirements for the different methods. Table size denotes the size of the lookup table M of Algorithm Repeats-Full, in millions of entries, that are needed to compute *all* possible repeats. Memory requirements for M, in MB, are thus four times as high as the presented numbers, since all entries are *unsigned integers*.

The memory savings due to site repeats, as well as the actual size of the lookup table for repeats, are presented in Table 2. The size of the lookup table was bounded by 200 MB. This corresponds to roughly 50 million entries (namely *unsigned integer* values). The actual memory for the lookup table was only allocated as needed. For most data sets, less than 200 MB RAM were required (confer Table 2). The notable exception is the data set containing 128 taxa. For this data set, 220.1 million entries (roughly 880 MB) in M are needed in the worst case. Since we bound the size of M to 200 MB, this means that not all repeats were found when analyzing this particular data set.

3.1 Exhaustive evaluation of all rooting

First, we evaluate the run-time impact of distinct rootings. The two data sets we used for this experiment have 59 and 354 taxa. Our implementation with site repeats enabled (*SRDT*), with site repeats disabled (*LLPLL*), as well as PLL and PLL-SEV were executed to perform PLF calculations (full tree traversals) for rootings on tip nodes. All of these implementations make use of an AVX-based function for calculating conditional likelihoods. This choice of rootings was selected because PLL requires that likelihood evaluations based on full traversals of unrooted trees start at *terminal* edges, that is, edges whose one end-point is a tip node. Hence in this experiment we exhaustively evaluate the PLF for all possible rootings on tip nodes. For each rooting, we executed 10 000 independent PLF computations. The size of matrix M is limited to 200 MB for all data sets. As we can see in Table 2, this is sufficient to find all repeats for these two data sets. This initial analysis helps us understand the effect of root placement on the number of site repeats present in a full tree traversal and as a consequence on run-times.

For the 354 taxon data set, SRDT had an average run-time of 11.714 seconds (for 10 000 iterations) and reached maximum and minimum run-times of 15.207 and 10.211 seconds, respectively. The standard deviation of run-times among all rootings was 0.94. PLL needed, on average, 58.112 seconds for this data set with maximum and minimum run-times of 61.067 and 57.573 respectively and a standard deviation of 0.54. Enabling the SEV method, PLL-SEV required 54.449 seconds,

Summary of speedups obtained using SRDT for a sample of rootings										
Data set	59	128	354	404	500	994	1512	2000	3782	7764
Speedup over PLL	3.46	3.27	4.96	5.31	2.78	4.5	3.03	2.47	4.06	2.63
Speedup over PLL-SEV	3.15	2.97	4.74	2.99	2.93	1.91	3.18	2.39	4.25	2.65

Table 3: Speedup obtained when using SRDT over PLL and PLL-SEV for each of the ten data sets. SRDT is consistently faster than both methods.

with minimum and maximum run-times of 55.128 and 57.299 seconds respectively. The standard deviation decreased to 0.411162.

For the 59 taxon data set, SRDT had an average runtime of 37.491 seconds. The respective maximum and minimum run-times were 44.787 and 31.515 seconds, and the standard deviation 3.197. PLL needed, on average, 134.769 seconds with a maximum run-time of 141.031 and minimum of 132.727 seconds. The standard deviation was 1.66. The average run-time of PLL-SEV was 124.733 seconds, with minimum and maximum run-times at 122.558 and 132.341, respectively, and a standard deviation of 2.05294. From this we see that, while the mean was higher for the PLL than for the SRDT method, the standard deviation was lower for the PLL.

To obtain initial estimate of how the original LLPLL implementation performs in comparison to PLL (and PLL-SEV) we measured its run-times by disabling site repeats. For the 354 taxon data set, the implementation averaged to 81.283 seconds, and 194.932 seconds for the 59 taxon data set. The standard deviation was 0.231 and 0.783 for the two data sets, respectively. For these two data sets (354 and 59), PLL and PLL-SEV are on average faster by a factor of 1.4 and 1.45 (for PLL) and a factor of 1.49 and 1.56 (for PLL-SEV) respectively. The differences in speed between LLPLL and PLL can be explained by two factors. First of all, PLL is a highly optimized software for PLF calculations that was directly derived from RAxML, which has been developed and optimized for over 10 years. Second, the standard optimization method explained in the introduction, namely, the lookup table for tip-tip cases is not implemented in LLPLL yet. The reason for this is that the lookup at tip-tip node is replaced by the general repeats method implemented in SRDT.

3.2 Evaluation of a sample of rootings

For the actual comparison of run-times for full tree traversals between SRDT and PLL, we use nucleotide data sets with taxon numbers ranging from 59 taxa to 7764 taxa (see Table 1). The run-times were measured for 10 different rootings. These rootings were randomly chosen, and are not necessarily the same for the SRDT and PLL methods. Given the standard deviation, as demonstrated above, for different run-times of the PLL under different rootings, this is a reasonable comparison. For the PLL method, the nodes for the different rootings were again restricted to be tip nodes only. For each rooting, we again conducted 10 000 full tree traversals and calculated the ratio of the time needed by SRDT, divided by the time needed by the PLL. The presented overall speedup of our new method per data set is then the average speedup over all 10 rootings. Table 3 shows the run-time improvements. As we can see, the SRDT implementation is always at least more than twice as fast as the PLL. In fact, the lowest observed average speedup (over the general PLL method) was 2.47. The maximal speed up was obtained for the data set containing 404 taxa. Here, the SRDT implementation was 5.31 times faster than the PLL. In table 1 we also see that this particular data set has the highest relative number of repeats among all analyzed data sets. This reinforces the initial intuitive assumption that the amount of repeats positively influences the runtime improvement. On the other hand, the largest decrease in speedup when comparing to PLL-SEV was for data sets 404 and 994 which contain over 70% gaps. Note also, that for data sets with a low amount of gaps, run-times for PLL-SEV increased compared to PLL (data sets 1512, 3782 and 7764).

3.3 Partial traversal performance

In phylogenetic inferences, to calculate the overall likelihood of the tree, it is not always necessary to conduct full tree traversals, in particular when conducting BI or ML tree searches that deploy local topological updates using, for instance nearest neighbor interchange (NNI) or subtree pruning and re-grafting (SPR) moves. We need to assess the performance of our approach for this type of partial CLV updates as well, since less CLVs are updated and they might be located at the inner part of the tree where the number of repeats is lower. Therefore, we also assess performance, by emulating partial CLV updates. To this end, for each root at which we evaluate the overall likelihood, we pick a random path into two directions away from it. At each node on this path we take a randomized decision on whether to stop the traversal (with probability $(1-p)$), or continue to a randomly chosen child node with probability p. The traversal stops if both directions of the path have either been stopped with probability p, or a tip node has been reached. This pattern of CLV updates emulates the topological moves described in [12] for BI. As mentioned before, additionally to the time spent in the PLF, other factors such as optimizing branch lengths and model parameters for ML, also contribute to the overall execution time. Here we concentrate only on measuring the time for calculating the PLF. Figure 7 shows the run-time improvements of the

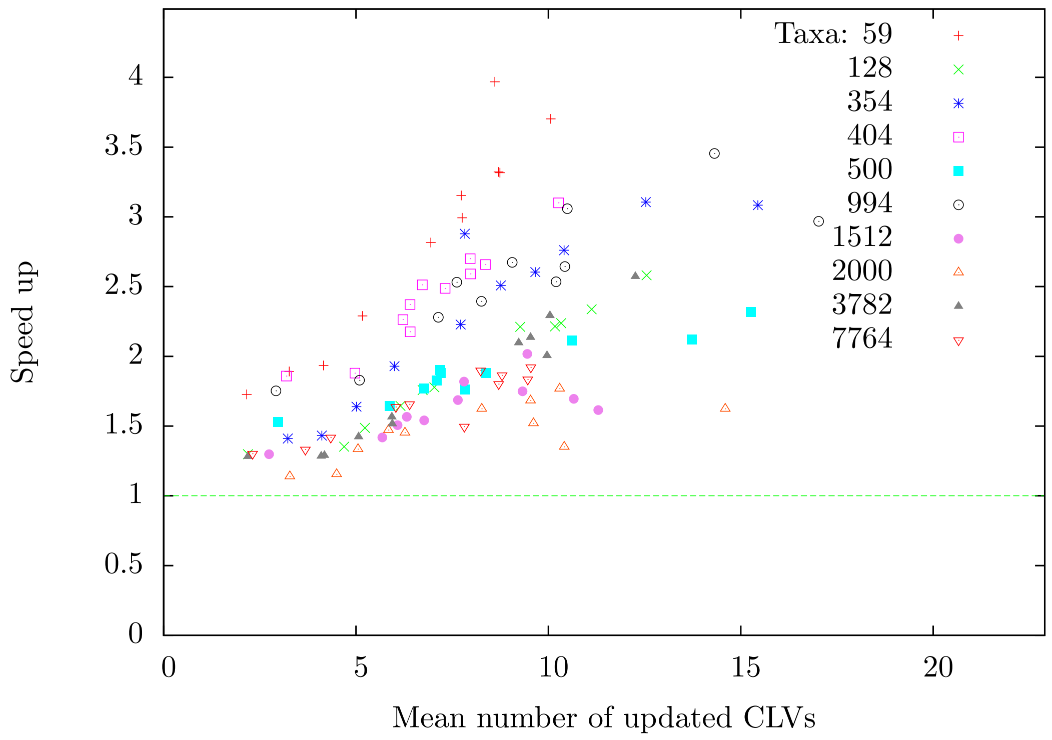

Figure 7: Plotted are the runtime improvements of the SRDT method over the LLPLL method against the average number of updated CLVs. The colors distinguish the different data sets. Each data set is represented by eleven measurements for eleven different nodes.

SRDT method over LLPLL. For each data set, eleven rootings are chosen at random. For each of those nodes, 10 000 partial updates are simulated and timed by recalculating the CLVs along the path chosen by the above method (with $p = 0.95$). We present the individual average speed-ups , plotted against the average number of nodes that are updated for this particular rooting. We choose to only compare the SRDT method to LLPLL, since for comparing it to the PLL it is very hard to implement exactly identical partial traversals because of the different internal structure of PLL. Thus, the speedup for the partial updates is *not* the absolute speedup for PLF

Summary of speedups obtained using SRCT when considering fixed topologies										
Sequences	59	128	354	404	500	994	1512	2000	3782	7764
Sites	6951	29198	460	13158	1398	5533	1577	1251	1371	851
Speedup over PLL	5.71	4.64	8.59	9.96	4.29	8.16	4.66	3.62	6.86	3.91
Speedup over PLL-SEV	5.22	4.23	8.22	9.96	5.44	3.30	4.88	3.48	7.18	3.93

Table 4: Speedups obtained using the SRCT method which considers a fixed topology over the PLL and PLL-SEV.

implementations. Instead, our results demonstrate the relative speedup that can be achieved by incorporating repeats into any PLF implementation. For full traversals over all possible rootings, the PLL was 1.4-1.45 times faster than the LLPLL method (see page 12). Assuming that these values are representative, the SRDT method still allows for faster PLF computations than the PLL for most, if not all, data sets, under this setting. Furthermore, as we discussed before, the speed difference is partially due to the lack of a dedicated tip-tip evaluation scheme for the LLPLL method. However, given the experimental set up here, a node for which both children are tips is included in the path, which is to be updated, at most twice. Thus, it remains to be evaluated, whether the speed difference between the LLPLL and PLL method persist under this setting.

3.4 Performance on fixed topologies

Many phylogenetic tools use a fixed tree topology on which the likelihood is repeatedly calculated. Divergence time estimates [10] and model selection [1] are examples of this. With fixed tree topologies repeats can be pre-computed once and then reused for subsequent PLF invocations. Table 4 shows the run-time improvement of the SRCT method over PLL and PLL-SEV under this setting.

4 Conclusion

Taking into account repeating site patterns in the alignment *does* matter for an efficient PLF implementation. We have introduced a method for quickly identifying all repeating site patterns, and consequently, minimize the number of operations required for evaluating the PLF. We have compared a prototype implementation of our method against one of the fastest and most optimized implementations of the PLF available using 10 real and simulated data sets. When setting a fixed tree topology (e.g., model selection or divergence time estimation), site repeats are only computed once in a pre-processing step and we obtain up to 10 fold run-time improvements. On dynamically changing tree structures where site repeats need to be re-computed on the fly, we still observe a faster execution of the PLF of more than five times. Moreover, the speedups are achieved without requiring more memory than standard production level software for calculating the PLF. On the contrary, we have shown that the memory requirements are even smaller by up to 78% than other standard implementations.

References

[1] Federico Abascal, Rafael Zardoya, and David Posada. ProtTest: selection of best-fit models of protein evolution. *Bioinformatics*, 21(9):2104–2105, 2005.

[2] Andre J Aberer, Kassian Kobert, and Alexandros Stamatakis. ExaBayes: Massively Parallel Bayesian Tree Inference for the Whole-Genome Era. *Molecular Biology and Evolution*, 31(10):2553–2556, 2014.

[3] N. Alachiotis and A. Stamatakis. A generic and versatile architecture for inference of evolutionary trees under maximum likelihood. In *Signals, Systems and Computers (ASILOMAR),*

2010 Conference Record of the Forty Fourth Asilomar Conference on, pages 829–835. Institute of Electrical and Electronics Engineers (IEEE), 2010.

[4] J Felsenstein. Evolutionary trees from DNA sequences: a maximum likelihood approach. *J. Mol. Evol.*, 17(6):368–376, 1981.

[5] J. Felsenstein. *Inferring phylogenies*. Sinauer Associates, 2003.

[6] Joe Felsenstein. PHYLIP (Phylogeny Inference Package) version 3.5c, 1993.

[7] T. Flouri, F. Izquierdo-Carrasco, D. Darriba, A.J. Aberer, L.-T. Nguyen, B.Q. Minh, A. von Haeseler, and A. Stamatakis. The phylogenetic likelihood library. *Systematic Biology*, 2014.

[8] Tomáš Flouri, Kassian Kobert, Solon P Pissis, and Alexandros Stamatakis. An optimal algorithm for computing all subtree repeats in trees. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 372(2016):20130140, 2014.

[9] Stephane Guindon, Jean-Francois Dufayard, Vincent Lefort, Maria Anisimova, Wim Hordijk, and Olivier Gascuel. New Algorithms and Methods to Estimate Maximum-Likelihood Phylogenies: Assessing the Performance of PhyML 3.0. *Systematic Biology*, 59(3):307–321, 2010.

[10] Tracy A. Heath, Mark T. Holder, and John P. Huelsenbeck. A dirichlet process prior for estimating lineage-specific substitution rates. *Molecular Biology and Evolution*, 2011.

[11] Fernando Izquierdo-Carrasco, StephenA Smith, and Alexandros Stamatakis. Algorithms, data structures, and numerics for likelihood-based phylogenetic inference of huge trees. *BMC Bioinformatics*, 12(1), 2011.

[12] Clemens Lakner, Paul van der Mark, John P. Huelsenbeck, Bret Larget, and Fredrik Ronquist. Efficiency of Markov Chain Monte Carlo Tree Proposals in Bayesian Phylogenetics. *Systematic Biology*, 57(1):86–103, 2008.

[13] Sergei L. Kosakovsky Pond and Spencer V. Muse. Column Sorting: Rapid Calculation of the Phylogenetic Likelihood Function. *Systematic Biology*, 53(5):685–692, 2004.

[14] Fredrik Ronquist, Maxim Teslenko, Paul van der Mark, Daniel L Ayres, Aaron Darling, Sebastian Höhna, Bret Larget, Liang Liu, Marc a Suchard, and John P Huelsenbeck. MrBayes 3.2: efficient Bayesian phylogenetic inference and model choice across a large model space. *Systematic Biology*, 61(3):539–42, 2012.

[15] Alexandros Stamatakis. RAxML Version 8: A tool for Phylogenetic Analysis and Post-Analysis of Large Phylogenies. *Bioinformatics*, 2014.

[16] A.P. Stamatakis, T. Ludwig, H. Meier, and M.J. Wolf. AxML: a fast program for sequential and parallel phylogenetic tree calculations based on the maximum likelihood method. In *Bioinformatics Conference, 2002. Proceedings. IEEE Computer Society*, pages 21–28, 2002.

[17] J.G. Sumner and M.A. Charleston. Phylogenetic estimation with partial likelihood tensors. *Journal of Theoretical Biology*, 262(3):413 – 424, 2010.

[18] Mario Valle, Hannes Schabauer, Christoph Pacher, Heinz Stockinger, Alexandros Stamatakis, Marc Robinson-Rechavi, and Nicolas Salamin. Optimization strategies for fast detection of positive selection on phylogenetic trees. *Bioinformatics*, 30(8):1129–1137, 2014.

[19] Ziheng Yang. Maximum likelihood phylogenetic estimation from dna sequences with variable rates over sites: Approximate methods. *J. Mol. Evol.*, 39(3):306–314, 1994.

[20] Ziheng Yang. Paml 4: Phylogenetic analysis by maximum likelihood. *Molecular Biology and Evolution*, 24(8):1586–1591, 2007.