Novel and Efficient Approaches for Exploring Phylogenetic Tree Space

Zur Erlangung des akademischen Grades eines
Doktors der Ingenieurwissenschaften

von der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Anastasios Togkousidis

Tag der mündlichen Prüfung:	24.06.2026
1. Referent:	Prof. Dr. Alexandros Stamatakis
2. Referentin:	Prof. Dr. Maria Anisimova

Hiermit erkläre ich, dass ich diese Arbeit selbständig angefertigt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht habe. Ich habe die Satzung des Karlsruher Institutes für Technologie (KIT) zur Sicherung guter wissenschaftlicher Praxis beachtet.

Heidelberg, 24.06.2026

. . . . . .

(Anastasios Togkousidis)

Zusammenfassung

Die Inferenz phylogenetischer Bäume anhand des Maximum-Likelihood-(ML)-Kriteriums ist $\mathcal{NP}$-schwer. Daher setzen Werkzeuge zur ML-basierten Stammbaumberechnung Heuristiken ein, um den immensen Suchraum möglichst effizient zu explorieren. Für ein gegebenes multiples Sequenzalignment (MSA) zielen phylogenetische Inferenzwerkzeuge darauf ab, diejenigen statistischen Parameter, welche Baumtopologie, Astlängen, sowie Parametern des Evolutionsmodells umfassen, zu bestimmen, welche die phylogenetische Likelihood-Funktion maximieren. Die Form des Likelihood-Raumes variiert jedoch erheblich zwischen verschiedenen Datensätzen. Für MSAs mit starkem phylogenetischem Signal weist der Raum der Stammbäume ein einzelnes Likelihood-Maximum auf, welches einer einzigen Topologie zugeordnet werden kann, zu der die meisten, wenn auch nicht alle, unabhängigen ML-Baumsuchen rasch konvergieren. Für andere Datensätze können unterschiedliche Suchläufe widersprüchliche—jedoch gleichermaßen wahrscheinliche bzw. statistisch hinsichtlich ihrer Sequenz in solchen Fällen nicht unterscheidbare—Topologien liefern, was einem zerklüfteten Raum mit mehreren lokalen Optima entspricht.

Gleichzeitig sind die Eingabesequenzen eines MSA inhärent verrauscht. Dies erhöht die Wahrscheinlichkeit, dass ML-Werkzeuge zu Overfitting tendieren, wobei in solchen Fällen die beste gefundene Topologie fälschlicherweise neben dem tatsächlichen phylogenetischen Signal auch das Rauschen modelliert. Selbst wenn kein Overfitting auftritt, wirft das vorhandene Rauschen die Frage auf, ob eine (zu) ausgiebige Optimierung gerechtfertigt ist. In vielen Fällen reicht eine hinreichend "gute" und statistisch äquivalente (zur bestgefundenen) Lösung aus, zumal gründliche ML-Inferenzwerkzeuge wie RAxML-NG häufig lediglich marginale und statistisch nicht unterscheidbare Likelihood-Verbesserungen während der finalen Optimierungsphasen erzielen.

Eine orthogonale Herausforderung besteht wenn sogenannte Multi-Gen-Alignments fehlende Daten aufweisen, die potenziell das Phänomen sogenannter Stände im phylogenetischen Baumraum induzieren können. Stände sind Mengen von Bäumen, die mit den unvollständigen Bäumen der einzelnen Gene kompatibel sind. Anhand der üblicherweise verwendeten Bewertungskriterien weisen die Bäume auf einem Stand identische Bewertungen aus und bilden somit eine sogenannte Terrasse. Terrassen vergrößern die Anzahl zu berücksichtigender plausibler Bäume und untergraben damit die Zuverlässigkeit phylogenetischer Analysen.

Alle bislang genannten Herausforderungen offenbaren die intrinsische Komplexität des ML Bauminferenzproblems, sowohl aus ingenieurwissenschaftlicher als auch aus biologischer Perspektive. Ferner werden die Anforderungen an ML-basierte Methoden immer höher, da moderne Sequenzierungstechnologien mit superexponentieller Rate Sequenzdaten generieren und die Größe zu analyiserender MSAs somit stetig wächst.Aufgrund der Vielfalt dieser internen und externen Herausforderungen ist die Entwicklung neuartiger und adaptiver datenbasiert Heuristiken, die plausible Bäume schneller identifizieren

können notwendig.

Ziel dieser Dissertation ist es, ein verbessertes Verständnis der Struktur des Likelihood Baumraums zu erlangen sowie die Effizienz von ML-Bauminferenzheuristiken zu verbessern. Mein erster Beitrag besteht in der Entwicklung und Implementierung einer adaptiven Baumsuchheuristik in RAxML-NG, welche die Gründlichkeit der Suche als Funktion der von Pythia prognostizierten sogenannten "Schwierigkeit" anpasst. Basierend auf Methoden des maschinellen Lernens sagt Pythia das phylogenetische Signal eines gegebenen MSAs präzise vorher. Die adaptive RAxML-NG Version kann in 99.7% aller untersuchten Fälle plausible Bäume berechnen und erzielt Beschleunigungen von bis zu 16× im Vergleich zu RAxML-NG v1.1, insbesondere bei leicht und schwer zu analysierenden MSAs.

Der zweite Beitrag besteht in der Integration des Kishino-Hasegawa (KH) Tests in RAxML-NG, welcher anschließend als zuverlässiges und schnell zu berechnendes Abbruchkriterium der Baumsuche verwendet wird, um einer übermäßigen sowie rechenintensiven Überoptimierung vorzubeugen. Die Baumsuche wird vorzeitig beendet, wenn weitere Verbesserungen statistisch insignifikant werden, und passt sich damit einhergehend an das Konvergenzverhalten eines gegebenen Datensatzes an. Der Abbruch führt zu bis zu 5× Beschleunigungen, während die statistische Plausibilität in bis zu 98% der untersuchten Fälle im Vergleich zu RAxML-NG v1.2 erhalten bleibt.

Drittens führe ich eine systematische Untersuchung von Overfitting in ML Bauminferenzwerkzeugen durch, mit einem Schwerpunkt hinsichtlich der potenziellen Auswirkungen eines Overfittings auf die Baumtopologie. Die Ergebnisse zeigen, dass ML-Werkzeuge—und insbesondere die gründlichen—kein Overfitting aufweisen. Konkret wurden Tendenzen zum Overfitting für weniger als 1% der getesteten Datensätze festgestellt. Darüber hinaus bieten Holdout-Validierungstechniken, die im Deep Learning häufig eingesetzt werden, im Kontext der ML-basierten Bauminferenz keinen praktischen Vorteil.

Viertens schlage ich die Standard-Suchheuristik für RAxML-NG v2.0 vor, indem ich die Erkenntnisse aus den Entwicklungen von Adaptive RAxML-NG und dem Frühabbruch sowie das vertiefte Verständnis des heuristischen Verhaltens aus der Overfitting-Untersuchung zusammenführe. Neben der Standardheuristik schlage ich eine beschleunigte „schnelle" heuristische Alternative vor, die Anwender einsetzen können, wenn sie sehr große Phylogenien mit Tausenden von Taxa analysieren, bei denen gründliche Inferenzen in Bezug auf die Rechenzeit und notwendigen Ressourcen unerschwinglich werden. Die vorgeschlagene schnelle Heuristik erreicht eine deutlich höhere Genauigkeit als konkurrierende Werkzeuge mit vergleichbaren Laufzeiten, wie beispielsweise FastTree.

Abschließend stelle ich eine effiziente Parallelisierung von Gentrius vor, einem Branch-and-Bound-Algorithmus, der alle Stand-Bäume eines gegebenen partitionierten MSA mit fehlenden Daten aufzählt. Die Parallelisierung nutzt einen Thread-Pooling-Mechanismus zur Optimierung der Lastverteilung und erreicht dadurch eine hohe parallele Effizienz mit linearer Skalierbarkeit auf bis zu 16 Kernen.

Abstract

Inferring phylogenies under the Maximum Likelihood (ML) criterion is $\mathcal{NP}$-hard. Therefore, ML tree inference tools deploy heuristics to navigate through the vast tree space. For a given multiple sequence alignment (MSA), phylogenetic inference tools strive to determine the statistical model comprising the tree topology, its branch lengths, and the evolutionary model parameters, that maximize the phylogenetic likelihood function. The shape of the likelihood space, however, varies substantially across datasets. For MSAs with strong phylogenetic signal, the tree space exhibits a single likelihood peak, associated with a unique topology, to which most, if not all, independent ML tree searches rapidly converge. For other datasets, distinct searches may yield contradicting—yet equally likely or statistically indistinguishable—topologies, reflecting a rugged space with multiple local optima.

At the same time, MSA input sequences are inherently noisy. This increases the chances that ML tools may experience overfitting, whereby the best-found topology incorrectly models noise alongside the true phylogenetic signal. However, even if overfitting does not occur, the *de facto* presence of noise raises the question whether thorough optimization is justified. In many cases, a reasonably "good" and statistically equivalent (to the best-found) solution may suffice, especially since thorough ML inference tools, such as RAxML-NG, often only attain marginal and statistically indistinguishable likelihood improvements during the final optimization stages.

An orthogonal challenge arises from multi-gene alignments with missing data, which occasionally give rise to the phenomenon of so-called stands in the phylogenetic tree space. Stands are sets of trees that are compatible with the incomplete per-gene trees. Under standard tree scoring criteria that are typically used, the trees on a stand have identical scores, and thereby constitute a terrace. Terraces magnify the number of plausible trees to be considered and undermine the robustness of a phylogenetic analysis.

All issues mentioned hitherto reveal the internal complexity of the ML tree inference problem, both from an engineering as well as a biological perspective. On top of this, modern sequencing technologies generate sequence data at a super-exponential rate, thereby imposing an additional, external pressure on ML-based methods, namely, the continuous increase in size of MSAs to be analyzed. Due to the miscellany of internal and external challenges, the need to devise novel, adaptive, data-informed heuristics that can identify equally plausible trees more rapidly has become imperative.

The purpose of this thesis is to offer an improved understanding of the structure of the likelihood-tree space, and of how ML tree inference heuristics operate on it. My first contribution is the development and implementation of an adaptive tree-search heuristic in RAxML-NG, which adjusts the thoroughness of the search as a function of the difficulty score predicted by Pythia. Pythia is a machine learning predictor that accurately estimates the phylogenetic signal in a given input MSA. Adaptive RAxML-NG

infers plausible trees in 99.7% of cases, and achieves speedups of up to 16× compared to RAxML-NG v1.1, particularly on easy- and difficult-to-analyze MSAs.

The second contribution is the integration of the Kishino-Hasegawa (KH) test into RAxML-NG, which is then used as a reliable and fast-to-compute early stopping criterion, to effectively evade excessive and compute-intensive over-optimization. The tree search terminates early when improvements become statistically insignificant, thereby adapting to the convergence behavior of an inference in a dataset-specific manner. Early stopping yields speedups of up to 5×, while preserving statistical plausibility in up to 98% of the cases, compared to RAxML-NG v1.2.

Third, I conduct a systematic investigation of overfitting in ML tree inference tools, specifically focusing on the potential impact of overfitting the tree topology. The results show that ML tools, and, more importantly, the thorough ones, do not exhibit overfitting. Specifically, overfitting trends are detected in less than 1% of the tested datasets. Further, holdout-validation techniques that are commonly used in deep learning offer no practical benefit in the context of ML-based tree inference.

Fourth, I propose the default tree search heuristic for RAxML-NG v2.0, by synthesizing the insights attained during the Adaptive RAxML-NG and early stopping developments, as well as the deeper understanding of heuristic behavior obtained from the overfitting investigation. Alongside the default heuristic, I propose an accelerated "fast" heuristic alternative that practitioners may invoke when analyzing very large phylogenies comprising thousands of taxa, on which thorough inferences become prohibitive. The proposed fast heuristic achieves substantially higher accuracy than competing tools with comparable runtimes, such as FastTree.

Finally, I provide an efficient parallelization of Gentrius, a branch-and-bound algorithm that enumerates all stand trees of a given partitioned MSA with missing data. The parallelization deploys a thread-pooling mechanism to reduce load imbalance, and thereby achieves high parallel efficiency with linear scalability on up to 16 cores.

Acknowledgements

Foremost, I would like to thank my PhD supervisor, Prof. Alexandros Stamatakis. He has been extremely supportive and trusting during my entire PhD journey. I am grateful for his valuable feedback, guidance, and for opening new research directions, while providing me the flexibility to choose which projects to engage. With his support, my PhD felt like a smooth and pleasant journey. Alexis is undoubtedly a mastermind in the field of phylogenetics and an incredible boss in all respects.

Second, I thank my second main collaborator (one would also say my informal co-supervisor), Dr. Olivier Gascuel, for hosting me during my research visit in Paris. His constructive feedback and suggestions were instrumental in the two projects we collaborated on. I definitely consider him a leading researcher in phylogenetics, and it was truly an honor for me to work with him.

Third, I deeply thank Dr. Paschalia Kapli and Dr. Tomas Flouri. Beyond being excellent scientists, they are also very good friends. I feel very proud of them for trying to establish their own research group. I am grateful to them for hosting me during my research visit in London, for their ideas and collaboration, and for their friendship, which I deeply value.

I would also like to warmly thank Prof. Dr. Maria Anisimova for agreeing to review this work.

Further, I would like to thank the Klaus Tschira Foundation for funding my research, and the Heidelberg Institute for Theoretical Studies (HITS), which was definitely an ideal working place over the past 4.5 years. I also thank the Karlsruhe Institute of Technology for accepting me as a PhD student, and for providing me the opportunity to defend this work.

Beyond formalities (sorry Alexis), I want to express my deepest gratitude to my family. My mother Sotiria, my father Ilias, and my twin brother Vassilis. They have all been extremely supportive throughout my entire life, and they are partly the reason why I chose the academic path so far. I feel gratitude and love for them, and I know they will always stand by my side. Thank you.

Moreover, I want to thank my colleagues and friends in Heidelberg, for being next to me throughout these years. I thank Dimitri, the very first person that I met in the lab, and a great friend. We had multiple philosophical discussions together at the beginning, which I really miss. I thank Marilena, my very first Greek friend in Heidelberg. I really miss those days when we used to live together in the WG. I thank Oleksiy, my friend also known as the RAxML-NG guy; I still have to read his book gift. I thank Luise, a great friend and definitely a creative and artistic mind. I thank Benoit, who was sitting next to me for the first two years; he has now moved to industry and is now rich, and I am happy for him. I thank Julia, Lukas, and Ben, very good friends and collaborators, all of them very very smart. I also thank the newcomers in the lab, Johannes and Christoph.

Both of them very funny and very sharp. I wish I had more time to collaborate with them. I thank Stelios and Marietta, Andreas and Matina, and Valantis and Elena. They are Greek friends I met during my final year in Heidelberg. Special thanks to Andreas who was driving the bus to HITS.

I should not forget my friends around the globe: Vaggelis, Theo, Apostolis, Batzolis, Mitsos, Giannis (Tass), Giannis (Reppas), Christos, Stefanos, Elina, Vicky, Balaouras, Tsitselas, Tzikas (although he is ghosting us), and Vanella. With all of you I shared wonderful moments, spanning many countries, which alleviated my PhD pressure and existential pain in general. Thank you all.

Finally, a very deep, personal, and special thank you to my girlfriend, Iliana. She has been the one person who directly stood by me every single day throughout this journey. She shared my joyful moments with genuine happiness, and supported me through my most stressful days. She was by my side during the long period when I was ill, something that I will never forget. And all of this while trying to balance her own studies and work. I feel very proud of her. For your endless support and true love, I love you.

Contents

List of Figures

List of Tables

List of Acronyms

AA Amino-Acid

AIC Akaike Information Criterion

AU Approximately Unbiased

BIC Bayesian Information Criterion

BLO Branch-Length Optimization

CLV Conditional Likelihood Vector

CTMC Continuous-Time Markov Chain

CTS Candidate Tree Set

DAG Directed Acyclic Graph

DNA Deoxyribonucleic Acid

DNN Deep Neural Network

ELW Expected Likelihood Weight

ES Early Stopping

GTR General Time Reversible

HV Holdout Validation

KH Kishino-Hasegawa

LBA Long Branch Attraction

LCM Left-Child Move

ML Maximum Likelihood

MP Maximum Parsimony

MPI Message Passing Interface

MPO Model-Parameter Optimization

MSA Multiple Sequence Alignment

NJ Neighbor Joining

NNI Nearest Neighbor Interchange

PAM Presence-Absence Matrix

PLF Phylogenetic Likelihood Function

PoT Patterns-over-Taxa

PTS Plausible Tree Set

RBM Root-Back Move

RCM Right-Child Move

RF Robinson-Foulds

RHAS Rate Heterogeneity Across Sites

RNA Ryribonucleic Acid

SH Shimodaira-Hasegawa

SoT Sites-over-Taxa

SP Sum of Pairs

SPR Subtree Prune and Regraft

TBR Tree Bisection and Reconnection

1. Introduction

1.1 Background and Motivation

Evolution plays a central role in modern biology [41]. It serves as the unifying principle that explains both the diversity of life and the similarities among living organisms. The evolutionary history of a set of organisms follows a hierarchical structure called *phylogeny*, which schematically represents the temporal occurrence of *speciation* events. The idea that organisms share a common, hierarchically structured phylogeny dates back to the early 19th century [35, 78, 114, 153], when contemporary scholars proposed evolutionary hypotheses (*cladograms*) based on empirical observations. Charles Darwin synthesized the ideas of his time into a unifying theory of evolution. In the book *On the Origin of Species* [35], he proposed natural selection as the driving mechanism of evolution and provided substantial empirical evidence supporting his theory. For these reasons, Darwin is regarded as the founder of evolutionary biology.

Until the 1950s, experts had been proposing phylogenies solely based on morphological and phenotypic traits observed across organisms [67, 185, 206]. Evidently, such hypotheses entailed subjective biases. The advent of protein sequencing techniques in the 1950s [44, 168, 170] marked the first time that molecular sequence data became available. A few years later, the discovery of DNA revolutionized molecular biology [59, 221], and in the next decade, sequencing technologies for nucleic acids (DNA/RNA) became available [169]. This opened the possibility of inferring phylogenies directly from molecular sequence data via mathematical and computational methods. Mathematicians developed mechanistic models of sequence evolution, and computer scientists developed and implemented algorithms to infer phylogenies from sequences using these methods.

With the advent of next-generation (NGS; [13, 127]) and third-generation (TGS; [29, 45]) sequencing technologies, DNA sequencing cost decreased exponentially [222], leading to an explosive growth in the available sequence data volume [126]. Computational approaches therefore became the only realistic way to infer phylogenies at scale. The two primary methods developed for phylogenetic inference from sequence data (which are still widely used) are the Maximum Parsimony (MP) [56] and the Maximum Likelihood (ML) [52] criteria. However, both problems are $\mathcal{NP}$-hard [36, 157], rendering tree searches on large multiple sequence alignments (MSAs) computationally intractable. Therefore, tools such as RAxML-NG [111], IQ-TREE [131], FastTree [151], and PhyML [68] indispensably rely on *heuristics* to obtain "good" solutions within reasonable time.

In addition to the inherent computational complexity of ML inference, factual challenges arise when the problem is viewed through a biological lens. The heuristics implemented in modern ML tools are pseudo-random: given identical starting conditions, they

deterministically[1] infer the same output tree. To assess the robustness of the inferred topology, users—or the tools themselves, implicitly—modify the starting conditions and repeat the analysis. Ideally, an input MSA should contain sufficient *phylogenetic signal* such that, even when starting conditions differ, the resulting trees remain identical, or at least highly similar. For some datasets, the former scenario prevails, while for others—with weak signal—different starting conditions yield highly contradictive topologies. The Pythia tool [73], developed by fellow lab member Julia Haag, is a machine learning predictor which quantifies this behavior. For each MSA, Pythia returns a floating-point number in $[0.0, 1.0]$ that reflects the phylogenetic signal strength of the MSA. This naturally leads to the idea of using the predicted Pythia score of the input MSA to effectively *adapt* the tree-search heuristic, with the goal to attain runtime improvements while preserving accuracy.

Moreover, sequence data are inherently noisy for a plethora of reasons, such as sequencing and alignment errors. This indicates that thorough optimization may not be necessary in practice. Indeed, if the MSAs are noisy, then a "reasonably good" solution should suffice, as the mathematically optimal one might overfit the noise rather than reflect the true phylogenetic signal. Hence, I explore the possibility of further accelerating heuristic searches by integrating *early stopping* strategies. I always stipulate the statistical equivalence of the trees inferred via the accelerated heuristics, compared to those obtained via some thorough (and potentially over-)optimization, as necessary condition for accepting such heuristic modifications. There is a fine line between effective early stopping and premature termination.

It is important, though, to distinguish between the phenomena of over-optimization and overfitting. For instance, an accelerated, early stopping version of RAxML-NG may infer ML trees that are statistically indistinguishable from those obtained via the standard heuristic. However, this does not in itself imply the presence of overfitting. It may instead indicate that the default RAxML-NG heuristic is overly stringent, and invokes unnecessary optimization rounds toward the end of the search, that yield negligible improvements. A characteristic example is the adjustment of the log-likelihood convergence threshold (ε) that was introduced between RAxML-NG v1.1 and v1.2. The ε-threshold was relaxed from 0.1 to 10, owing to the results of a systematic study [74], which showed that the former value was excessively strict. Building on this, I argue that the ε-threshold should not remain fixed across datasets, but should instead be dynamically determined according to the convergence behavior of an individual tree inference and MSA.

At the same time, even if early stopping strategies succeed in accelerating thorough ML heuristics, the question of *overfitting* still remains unanswered. ML-based tree inference is a typical optimization problem in statistical learning. The statistical model being optimized is parameter-rich, comprising both discrete (the tree topology) and continuous parameters (e.g., branch lengths and substitution model parameters) [228]. In analogy to the training process in deep learning models, the optimization process can be decomposed into distinct iterations, each one corresponding to a single topological move. Moreover, since the fundamental assumption of the ML criterion is that sites evolve independently, we can split the input data (i.e., the MSA sites) into training and testing sites. Therefore, in analogy to standard deep learning techniques, we can investigate the potential impact of overfitting in ML tree inference. We may pose the question whether thorough ML

[1] In fact, ML tools are deterministic when they are provided with identical starting conditions, and are executed sequentially. In parallel execution, minor variation may occur due to the non-associativity of floating-point arithmetic in parallel reduction (sum) operations [63, 84].

tools are susceptible to overfitting and, if so, whether overfitting primarily affects the tree topology, which is the biologically most meaningful parameter. Further, we may examine whether deep learning techniques to evade overfitting, such as *holdout-validation*, offer practical benefits in phylogenetic inference. These are important questions that, to the best of my knowledge, have not been sufficiently addressed in the current literature, and may provide a better understanding of the behavior of ML heuristics and tools.

Orthogonal to all of the above, there exists another phenomenon that further magnifies the number of plausible trees to be considered as valid solutions. This is the phenomenon of *terraces*, that is, a set of distinct trees with the exact same analytical likelihood or parsimony score [166, 167]. Terraces are closely related to the phenomenon of *stands*, which may arise under the following conditions. Partitioned MSAs, comprising several genes, are frequently used to infer phylogenies, instead of using single-gene alignments. These datasets are constructed by concatenating single-gene MSAs. Missing per-gene (or per-locus) sequences may give rise to stands, that is, sets of trees that are all compatible with the incomplete—due to missing sequences—per-locus trees. Under many common criteria, the trees from a stand reside on a terrace. One main challenge is that the number of trees within a stand can grow exponentially with the number of species in the partitioned MSA. Therefore, identifying stands and efficiently enumerating the trees they comprise is crucial for conducting robust phylogenetic analyses.

In summary, phylogenetic tree inference is a typical optimization problem, yet with prohibitive computational complexity. The expected runtime for finding the optimal solution scales super-exponentially with respect to the number of MSA sequences (under the assumption that $\mathcal{P} \neq \mathcal{NP}$). Moreover, the biological interpretation of the inferred solution is equally important and induces an additional layer of complexity. Practitioners ultimately seek tools that are both efficient and reliable, often taking the credibility of the inferred tree for granted. In this thesis, I aim to provide a better understanding of the properties of phylogenetic tree space and on how ML tree inference heuristics operate within it. Building on this, I introduce adaptive, data-informed modifications based on the intrinsic properties of the input MSAs. The overall objective is to revise the current heuristic that RAxML-NG 1.x deploys, such that we achieve substantial runtime improvements while preserving statistical equivalence among the inferred ML trees, thereby effectively establishing the default tree search heuristic of RAxML-NG v2.0.

1.2 Thesis Structure and Contributions

This thesis is structured as follows. In **Chapter 2**, I introduce the fundamental concepts of phylogenetics which are relevant for understanding this work. I place particular emphasis on ML tree inference with a focus on RAxML-NG, which is the tool into which I integrate the methods developed in this thesis. In addition to the fundamental principles of phylogenetic tree inference, I also provide the necessary background information on several related topics: (i) the statistical tests used to establish the equivalence of distinct tree topologies within the ML framework, (ii) the Pythia tool, (iii) sources of inherent noise in MSAs and the concept of overfitting in statistical learning, and (iv) the concepts of stands and terraces.

In **Chapter 3**, I present Adaptive RAxML-NG, a tool which modifies the thoroughness of the tree search heuristic as a function of the difficulty score predicted by the Pythia tool. Experimental results on 9,515 empirical and 5,000 simulated MSAs demonstrate

substantial speedups compared to RAxML-NG v1.1, particularly on easy and difficult datasets, where they attain an up to 16× runtime improvement.

At the same time, I explored the possibility of integrating early, yet adequate, stopping criteria into RAxML-NG, to evade over-optimization. Hence, in **Chapter 4**, I describe how I integrated the Kishino-Hasegawa (KH) test [106] into RAxML-NG as a fast and reliable early-stopping method. The tree search terminates early when further improvements become statistically insignificant, thereby adapting to the convergence behavior of each individual tree search and MSA. An important by-product of this study was the development of a simplified heuristic for RAxML-NG (sRAxML-NG), which I proposed as a necessary simplification in order to apply any form of early stopping criterion. In conjunction with sRAxML-NG, the KH-based early stopping criterion achieves an up to 5× speedup on DNA, and 3.9× on protein MSAs compared to RAxML-NG v1.2, when benchmarked on 300 large empirical MSAs. At the same time, it infers statistically plausible ML trees for up to 98% of empirical DNA datasets, while performance on protein MSAs is slightly worse (∼94%).

In **Chapter 5**, I conduct a systematic investigation of overfitting in ML tree inference tools that focuses on the potential impact of overfitting on the inferred tree topology. I examine overfitting trends in three widely used ML tools: RAxML-NG, IQ-TREE [131], and FastTree [151]. Specifically, I employ a 10-fold Monte Carlo cross-validation scheme, by partitioning 9,062 empirical and 6,342 simulated MSAs into training (80%) and testing (20%) sites. I conduct ML tree inferences using the aforementioned tools and store all intermediate improved topologies, that is, those which were visited and retained by the tools due to higher log-likelihood scores. I evaluate the recorded trees on the testing sites to construct testing curves, which I subsequently analyze. The results show that topological overfitting is virtually absent: for instance, RAxML-NG and IQ-TREE exhibit overfitting trends in less than 1% of empirical MSAs. To complement these findings, I develop and benchmark a site-based holdout-validation (HV) variant of RAxML-NG. The results further confirm the absence of overfitting and indicate that excluding MSA sites substantially reduces phylogenetic signal.

The development of Adaptive RAxML-NG (Chapter 3) marked my first own development of ML tree inference heuristics. Although the contribution is undoubtedly important, in retrospect the heuristic that Adaptive RAxML-NG deploys is most likely too stringent. Meanwhile, sRAxML-NG provided an intuition on how the default RAxML-NG heuristic should be redesigned to achieve substantial runtime improvements without compromising accuracy. However, sRAxML-NG with ES requires further tuning, as in some cases (especially on protein MSAs) it tends to terminate prematurely.

After extensive benchmarking and fine-tuning, I propose the default heuristic of RAxML-NG v2.0 in **Chapter 6**. This new heuristic synthesizes ideas from both the Adaptive and the ES versions. I call it the Hegelian heuristic, as it represents a tradeoff between speed and accuracy, echoing Hegel's philosophy of synthesis between opposing tensions. Specifically, the default RAxML-NG v2.0 heuristic is on average 7× faster than RAxML-NG v1.1, while it infers statistically equivalent, or even better ML trees. Alongside this default heuristic, I introduce an additional fast mode for RAxML-NG v2.0, that enables rapid, yet still sufficiently accurate, inference of very large phylogenies. This fast heuristic exhibits comparable accuracy to that of IQ-TREE, and substantially outperforms superficial tree search alternatives (e.g., FastTree) both in terms of accuracy and speed.

Chapter 7 is orthogonal, and concerns the parallelization of the Gentrius algorithm [28].

Gentrius is a branch-and-bound algorithm which enumerates all stand trees for a given dataset. However, its workflow tree cannot be determined *a priori*, and the number of stand trees can grow exponentially with the number of taxa [19]. These properties render effective parallelization and load balancing particularly challenging. I describe how I overcome these difficulties by employing a thread-pooling approach, that achieves linear scalability up to 16 cores. Chronologically, this was the first project of my PhD, but I present it in the final chapter because it is somewhat independent of, yet conceptually related to, the remainder of this work. I conclude and discuss future work in **Chapter 8**.

2. Preliminaries

This Chapter contains text and figures from the following publications:

- **Anastasis Togkousidis**, Olga Chernomor, and Alexandros Stamatakis. "Parallel Inference of Phylogenetic Stands with Gentrius." *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (pp. 139-148). IEEE.
- **Anastasis Togkousidis**, Oleksiy M. Kozlov, Julia Haag, Dimitri Höhler, and Alexandros Stamatakis. "Adaptive RAxML-NG: Accelerating Phylogenetic Inference under Maximum Likelihood using Dataset Difficulty." *Molecular Biology and Evolution*, Volume 40, Issue 10, October 2023. `https://doi.org/10.1093/molbev/msad227`
- **Anastasis Togkousidis**, Alexandros Stamatakis, and Olivier Gascuel. "Accelerating Maximum Likelihood Phylogenetic Inference via Early Stopping to Evade (Over-)optimization." *Systematic Biology*, 2025. `https://doi.org/10.1093/sysbio/syaf043`
- **Anastasis Togkousidis**, Olivier Gascuel, and Alexandros Stamatakis. "A Systematic Investigation of Overfitting in Maximum Likelihood Phylogenetic Inference." *biorxiv*, 2025. `https://doi.org/10.1101/2025.10.07.680876`

In this chapter, the material I include from the above publications serves only as background information, and does not refer to the methods or results reported therein. The first three publications are peer-reviewed. The fourth publication is currently available as a preprint on *biorxiv*, and has been submitted for peer-review.

Phylogenetics is the study of the evolutionary history and relationships among biological entities, such as species, individuals, or specific genes. We refer to those entities as *taxa*. For a given a set of taxa, the goal is to infer the underlying evolutionary history, which, in the context of this work, follows a treelike structure. The ultimate goal of the field is to reconstruct the, so called, universal *Tree of Life* [95, 225, 226]. Historically, experts inferred phylogenies based on morphological and phenotypic traits observed across specimens [206]. Today, phylogenetic analyses heavily rely on computational methods. Input taxa are now represented by nucleotide or protein sequences, typically sampled from distinct species. Modern approaches use a mathematical framework which describes how the input sequences evolved. In this chapter, I outline the fundamentals of the computational methods that are used to infer phylogenies from sequence data. These serve as prerequisites to understand my research contribution. Throughout this thesis, all references to phylogenetic tree inference refer to computational methods.

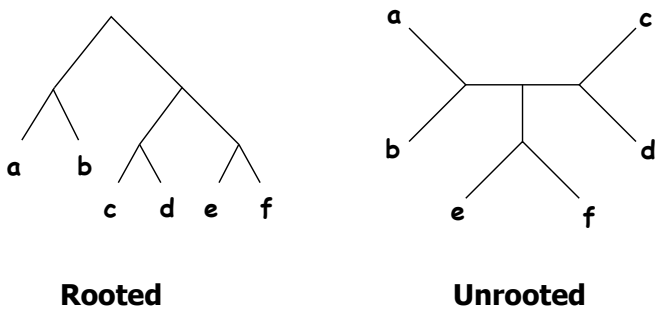


Figure 2.1: An example of a rooted (left), and an unrooted (right) phylogenetic tree. The tree on the right is the unrooted version of the tree on the left.

2.1 Phylogenetic Trees

2.1.1 The Phylogenetic Tree as a Graph

With respect to graph theory, phylogenetic trees are unordered, strictly binary trees, with labeled *tip nodes* (*leaves*). These trees are also called strictly *bifurcating*. There is a bijective mapping between the tip nodes of the phylogenetic tree, and the set of taxa included in the analysis. Hence, tip `x` refers to the tip node to which taxon `x` is assigned. Phylogenetic trees can either be *rooted* or *unrooted*. In the former, the *root* node corresponds to the most recent common ancestor of the taxa at the tips. A rooted phylogenetic tree is a Directed Acyclic Graph (DAG). *Inner nodes* signify speciation events and, in analogy to the root, correspond to the most recent common ancestor of the subset of taxa contained in the descending *subtree*. All inner nodes, except for the root, have a degree of 3. They are connected, through an incoming edge, to their *ancestor* (or *parent* node), and they give rise to exactly two *descendants* (or *child* nodes), to which they link via outgoing edges. The root node has a degree of 2, as it has no ancestor. Tip nodes have a degree of 1 (only one incoming edge). The edges of the tree are also called *branches*. *External branches* are those connected to the tip nodes, while all other branches are *internal* (or *inner*).

We can convert a rooted phylogenetic tree to an unrooted one, by removing its root node, and merging the two outgoing edges into a single edge (see Figure 2.1). Unrooted phylogenetic trees are *undirected*. They primarily capture relationships between taxa, that is, they describe how taxa cluster together. This will become more evident in the next Section (Section 2.1.2), where I introduce the concepts of *splits* and *quartets* on unrooted trees. All inner nodes of an unrooted tree have a degree of 3.

We assign *branch lengths* to the edges of a phylogeny to reflect evolutionary distances between speciation events. As discussed in Section 2.2, taxa correspond to nucleotide/protein sequences. Therefore, branch lengths quantify the amount of genetic change that occurred between speciation events; the longer the branches, the greater the evolutionary divergence between the corresponding tree nodes. Moreover, branch lengths are an important set of parameters of likelihood-based models (introduced in Section 2.5). Usually, the term *tree topology* (or simply *topology*) is used to refer to the graph structure of a phylogenetic tree, that is, its branching pattern independent of branch lengths. In the current thesis, when referring solely to the graph structure of a phylogeny, I use the terms

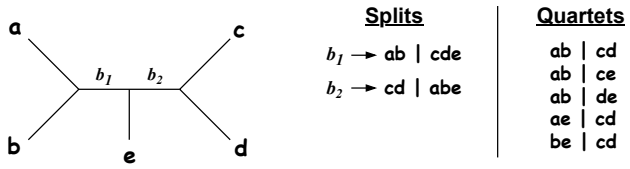


Figure 2.2: An example of the non-trivial splits and quartets of an unrooted phylogenetic tree, with a taxon set $\{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}, \mathsf{e}\}$. The inner branches b_1 and b_2 define the non-trivial splits $\mathsf{ab} \mid \mathsf{cde}$ and $\mathsf{cd} \mid \mathsf{abe}$, respectively. The induced quartets are: $\mathsf{ab} \mid \mathsf{cd}$, $\mathsf{ab} \mid \mathsf{ce}$, $\mathsf{ab} \mid \mathsf{de}$, $\mathsf{ae} \mid \mathsf{cd}$, and $\mathsf{be} \mid \mathsf{cd}$.

"tree" and "topology" interchangeably. However, only the term "tree" may additionally entail branch lengths.

Throughout this work, I focus exclusively on unrooted phylogenetic trees. Unless otherwise stated, all trees are assumed to be unrooted. Additionally, all tree inference tools I use and develop operate on unrooted trees.

2.1.2 Splits, Quartets, and Compatibility on Unrooted Trees

Every branch of an unrooted tree uniquely defines a *bipartition* of the taxon set. If the branch is removed, the taxa are grouped into two disjoint sets, corresponding to the two emerging, disconnected subtrees. Bipartitions are also called *splits*. For example, the unrooted phylogenetic tree in Figure 2.2 has the taxon set $\{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}, \mathsf{e}\}$. Its inner branch b_1 implicitly defines the split $\{\{\mathsf{a}, \mathsf{b}\}, \{\mathsf{c}, \mathsf{d}, \mathsf{e}\}\}$, which we denote as $\mathsf{ab} \mid \mathsf{cde}$. If one of the two disjoint subsets only comprises a single taxon (e.g, $\mathsf{a} \mid \mathsf{bcde}$), the split is called trivial. Trivial splits correspond to tip branches. On the other hand, if both subsets comprise at least two taxa (e.g., $\mathsf{ab} \mid \mathsf{cde}$), the split is called non-trivial. Non-trivial splits correspond to inner branches. For a given tree T, we denote its set of non-trivial splits as $\mathcal{S}(T)$.

Let T be a tree with taxon set Y, and let $Y' \subset Y$ be a subset of taxa. The *restriction* T' of T to Y' is called the *induced* subtree of T on Y'. By restriction, I refer to the stepwise process, in which: for every taxon $\mathsf{y} \in Y \setminus Y'$ we (a) prune the leaf y from T together with the adjacent tip branch, and (b) remove the emerging *suppressible*, degree-2 (inner) node and merge the two adjacent branches. We say that T *displays* T', and write $T' = T \mid Y'$.

A *quartet* is an unrooted tree with four taxa. Each quartet contains a single inner branch, and we can fully determine its topology by the non-trivial split associated with that branch. For example, for the taxon set $\{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}\}$, the three possible quartets are: $\mathsf{ab} \mid \mathsf{cd}$, $\mathsf{ac} \mid \mathsf{bd}$, and $\mathsf{ad} \mid \mathsf{bc}$. A quartet of an unrooted tree T is the induced subtree obtained by restricting T to any four of its taxa.

Let T_1, T_2 be unrooted, bifurcating trees on Y_1, Y_2, respectively, such that $Y_1 \cap Y_2 \neq \emptyset$. We call those trees *compatible* if there is a tree T which displays both of them. Such a tree exists *if and only if* the two trees have identical induced subtrees for the taxa they share: $Y_1 \cap Y_2$ [66].

Splits, quartets, and compatibility are closely related concepts [9, 27, 94, 174]. For example, compatibility can be defined in terms of splits [27], and the "Quadraple condition for compatibility" (see Lemma 5.9.1 in [94]) establishes a connection between quartets and the former two. I do not further address these topics here, as they are mostly outside

the scope of this thesis. I do revisit the notion of compatibility in Section 2.11 though, where I introduce the concept of *stands* in phylogenetic tree space.

2.1.3 Distance Metrics for Phylogenetic Trees

Let T_1, T_2 be two trees on the *same* taxon set Y. The *Robinson-Foulds (RF)* distance [155] between the two tree topologies is the cardinality of the symmetric difference between their two non-trivial[1] split sets $\mathcal{S}(T_1)$ and $\mathcal{S}(T_2)$, respectively. Hence:

$$d_{RF}(T_1, T_2) = |\mathcal{S}(T_1) \triangle \mathcal{S}(T_2)| \tag{2.1}$$

where $d_{RF}(\cdot)$ denotes the distance metric, and $\triangle$ the symmetric difference operator. Equation (2.1) describes an absolute distance metric, as it does not account for the input tree sizes. In many contexts, especially when summarizing distances across trees of varying sizes, it is preferable to use a normalized version of the RF distance. Since an unrooted tree with n taxa comprises $(n-3)$ inner branches which determine the number of non-trivial splits, the maximum RF value between any two trees is $2(n-3)$. Therefore, the *normalized* or *relative* RF distance is defined as:

$$d_{nRF}(T_1, T_2) = \frac{d_{RF}(T_1, T_2)}{2(n-3)} \tag{2.2}$$

where $d_{nRF}(\cdot)$ is the normalized distance metric, and n the number of taxa in T_1, T_2.

Further, a *quadraple* on Y is a four-taxon subset of Y. As $\mathcal{Q}(Y)$ we define the set of all possible $\binom{n}{4}$ quadraples on Y. The *quartet distance* [46] between two trees T_1, T_2 on Y, is the number of incongruent quartets induced by T_1 and T_2, for each possible quadraple in $\mathcal{Q}(Y)$. Hence:

$$d_Q(T_1, T_2) = |\{q \mid q \in \mathcal{Q}(Y) \text{ and } T_1|q \neq T_2|q\}| \tag{2.3}$$

where $d_Q(\cdot)$ is the (absolute) quartet distance metric, and $T_1|q \neq T_2|q$ implies topological dissimilarity. In analogy to Equation (2.2), and since the maximum quartet distance value between any two trees is $\binom{n}{4}$, the normalized or relative quartet distance is defined as:

$$d_{nQ}(T_1, T_2) = \frac{d_Q(T_1, T_2)}{\binom{n}{4}} \tag{2.4}$$

The RF distance is the most widely used tree comparison metric and can be computed efficiently in $\mathcal{O}(n)$. It is an intuitive distance metric, especially when the trees being compared are topologically similar [199]. The quartet distance, on the other hand, is more refined, as it yields a wider range of values for trees on the same taxon set. It can be used to accentuate slight topological differences. However, it is slower to compute. Since there are $\binom{n}{4}$ possible quadraples in a taxon set Y comprising n taxa, a brute-force algorithm would require $\mathcal{O}(n^4)$ steps to compute the quartet distance, since it examines each quadraple individually. More efficient algorithms reduced the time complexity, first to $\mathcal{O}(n^3)$ steps by Steel and Penny [198], and then to $\mathcal{O}(n^2)$ steps by Bryant *et al.* [26]. Finally, Brodal *et al.* proposed a dynamic programming algorithm that computes the

[1] Trivial splits are identical across all trees on the same taxon set. They provide no information about the branching structure of a given tree, and they do not contribute to topological differences. They are, therefore, excluded from the RF distance definition.

quartet distance in $\mathcal{O}(n \log n)$ time [25]. This algorithm is based on a hierarchical decomposition of the two trees under comparison and the aggregation of shared quartet counts using auxiliary counters at the tree nodes. These counters are computed recursively from the counters of their corresponding subtrees, and the hierarchical decomposition allows them to be updated efficiently in logarithmic time.

The quartet distance distribution, as compared to RF, is harder to interpret [199], as it is more sensitive to local topological changes. For example, a single Nearest Neighbor Interchange (NNI) rearrangement (see Section 2.6.2) produces neighboring topologies with an absolute RF distance of 2, regardless of the size of subtrees being interchanged. In contrast, the quartet distance between two NNI neighbors varies proportionally to the product of the four subtree sizes, that are incident to the central inner branch around which the NNI is performed; therefore, it depends both on the overall tree size and the balance of the taxon distribution among those subtrees.

The introduced metrics completely neglect branch lengths, and they only account for topological differences. Adaptations that incorporate branch lengths (e.g., the weighted RF distance) have been proposed in the literature [6, 121, 156]. Alternatively, one can use bootstrap support values [51] as edge weights in the weighted versions of the RF or quartet distance. Extensions also exist that relax the requirement of identical taxon sets, such as the generalized RF distance by Kuhner and Felsenstein [113], which handle trees on different, yet overlapping, taxon sets. In this work, I only use the standard definitions in Equations (2.1)–(2.4).

2.2 Multiple Sequence Alignments

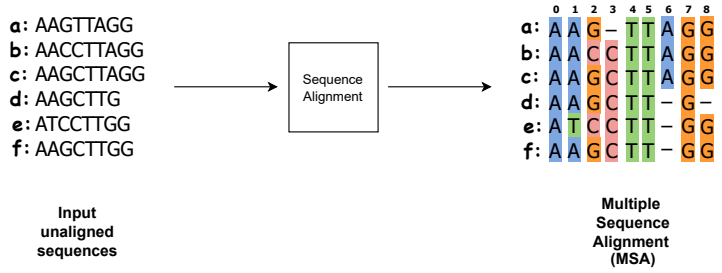


Figure 2.3: The problem of sequence alignment in its generalized form, where the input data is a set of unaligned biological sequences (here DNA), and the output is an MSA. MSA coloring is a standard visualization technique to highlight site homogeneity and mismatches. Here, I apply the color scheme used in ClustalΩ [182].

Tip nodes on unrooted trees correspond to taxa. Taxa are represented by biological sequences, sampled from distinct individuals that belong to different species. "Biological sequences" is used as a generic term to either refer to genetic DNA or protein sequences. Genetic (or nucleotide) sequences represent Deoxyribonucleic acid (DNA), or Ribonucleic acid (RNA) molecules/strands. These molecules are composed of nucleotides, and the sequence depicts the order of nucleotide bases along the molecule. Amino-acids (AAs), on the other hand, compose proteins, and the respective sequences capture the linear order of AAs. We treat those sequences as *strings*. Hence, DNA sequences are finite strings over the characters of the DNA alphabet {`A`, `C`, `G`, `T`}, which stand for the four nucleotide

bases: adenine (A), cytosine (C), guanine (G), and thymine (T). In the RNA alphabet we replace thymine (T) by uracil (U). Likewise, protein sequences are strings over the AA alphabet that comprises letters for the 20 standard AAs.

In short, we obtain sequences via processing biological samples (such as blood, tissue, etc.), from which we extract the innate genetic material. Several preprocessing steps follow, the most important of which are genome sequencing, assembly, and annotation. Genome annotation assigns functional information to the assembled sequences and identifies genomic features, such as genes and coding regions. Following annotation, the next important step for phylogenetic analysis is orthology assignment. For tree inference, it is typically assumed that the input sequences correspond to the same gene (or genomic locus) across taxa. In other words, the sequences should be *orthologous*, meaning that they diverged through speciation events, rather than gene duplication events.

Due to possible uncertainties emerging as a by-product of wet-lab sequencing and assembly, the standard DNA/RNA alphabets are often extended by ambiguous characters [31]. For example, the ambiguous character N marks a position where the nucleotide could be any of A, C, G, or T (for DNA)[1]. To obtain protein sequences, an additional translation step is performed on annotated coding regions, in which nucleotide triplets (*codons*) are mapped to their corresponding AAs according to the genetic code [32, 142].

The final step before obtaining the input data for a phylogenetic tree inference is *sequence alignment*. Initially, the preprocessed input sequences vary in length, and are therefore *unaligned*. The length differences arise from genetic changes that accumulate over time: nucleotide *mutations* (or *substitutions*), *insertions*, and *deletions*. One goal of sequence alignment is to identify sequence regions with varying degrees of genetic variation/conservation.

The problem of sequence alignment consists in inserting an appropriate number of gap characters ("-") into each of the n input sequences, such that:

1. All resulting sequences have equal length, forming a *Multiple Sequence Alignment (MSA)*, comprising n *rows* (taxa)[2], and s *columns* (or *sites*).
2. No column contains only gaps.
3. The resulting MSA optimizes a predefined criterion.

I provide a toy example of the sequence alignment problem in Figure 2.3. The resulting MSA comprises $n = 6$ taxa, and $s = 9$ sites. Repeated sites, that is, sites with identical content, are called *patterns*. For example, in Figure 2.3, sites at positions 4 and 5 are identical, and constitute a single pattern. The *length* of the MSA refers to the number of columns it contains. Sequence characters that fall within the same MSA column (excluding gaps) are called *homologous*, for they are assumed to have evolved from the same character in the ancestral sequence. Gaps indicate insertion or deletion events (*indels*).

Regarding optimality criteria (see point 3 above), the sum of pairs (SP) score is a standard MSA criterion [71]. The goal is to find the MSA that maximizes the SP score. The SP-score considers all $\binom{n}{2}$ pairwise alignments of the MSA, i.e., pairs of rows exactly as they appear on the given alignment, and computes a rudimentary score by adding: a positive constant for every column-match (e.g., AA or TT are exemplary matches), and a negative constant for every column-mismatch (e.g., AG or G-). It typically excludes gap-gap pairs (--) from the summation by assigning them a score of zero [72]. Then, it

[1] For example, due to sequencing failure.

[2] I use the symbol n to denote the number of rows/taxa in an MSA, which is consistent with the use of the same symbol to quantify the number of tip nodes on a tree.

aggregates the pairwise scores over all sequence pairs to extract to overall SP score of the alignment.

For the special case of $n = 2$, there exist efficient dynamic programming algorithms (e.g., the Needleman-Wunsch algorithm [143]) which solve the problem in $\mathcal{O}(s_1 \cdot s_2)$ steps, where s_1 and s_2 are the two unaligned sequence lengths. In the general case of arbitrary n, the MSA problem is $\mathcal{NP}$-hard [100]. MSA tools, such as MAFFT [101], MUSCLE [43], and ClustalΩ [182] deploy heuristics to approximate a "good" alignment.

2.3 The Phylogenetic Tree Inference Problem

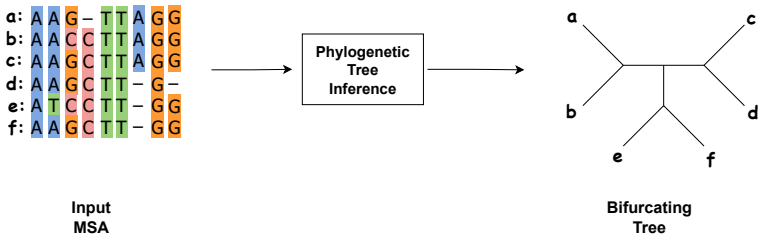


Figure 2.4: The problem of phylogenetic tree inference in its generalized form, where the input is an MSA, and the output is a bifurcating tree.

Figure 2.4 presents the phylogenetic tree inference problem in its general form. The input is an MSA, and tree inference methods aim to infer the bifurcating tree which maximizes a predefined optimality criterion. The core of this thesis deals with phylogenetic tree inference, and in particular, with *Maximum Likelihood (ML)* methods. Before introducing the likelihood function in Section 2.5, I first explicate a parsimony-based approach for tree inference. Both the parsimony and the likelihood methods are called *character-based*, in contradistinction to distance-based methods. An example of a distance-based method for phylogenetic inference is the Neighbor Joining algorithm (NJ; [164]). However, I do not address distance-based methods in this thesis.

2.4 A Parsimonious Approach to Infer Phylogenies

One approach to infer phylogenies is to apply the *Maximum Parsimony (MP)* criterion [56, 79]. The underlying principle is that, for a given MSA, we select the tree requiring the least amount of character substitutions (i.e., nucleotide or AA mutations) to explain the observed MSA sequences at its tip nodes. As I discuss below, this approach completely neglects branch lengths, and solely focuses on topologies. The parsimony score on a given, fixed topology can be computed via a dynamic programming algorithm [171] with $\mathcal{O}(n \cdot s)$ operations. A schematic description of this algorithm is provided in Figure 2.5. This illustration is rather implementation-oriented, since it begins by encoding the characters of the tip sequences as bit-vectors[1]. The advantage that this description

[1]The encoding step, however, is not strictly necessary. Instead, we can keep the sequences unchanged and compute the parsimony score by comparing the per-site sequence characters. For instance, if the two child sequences are `ACG` and `AAG`, the mismatch at the second site-position can be represented as `{A or G}` in the reconstructed ancestral sequence.

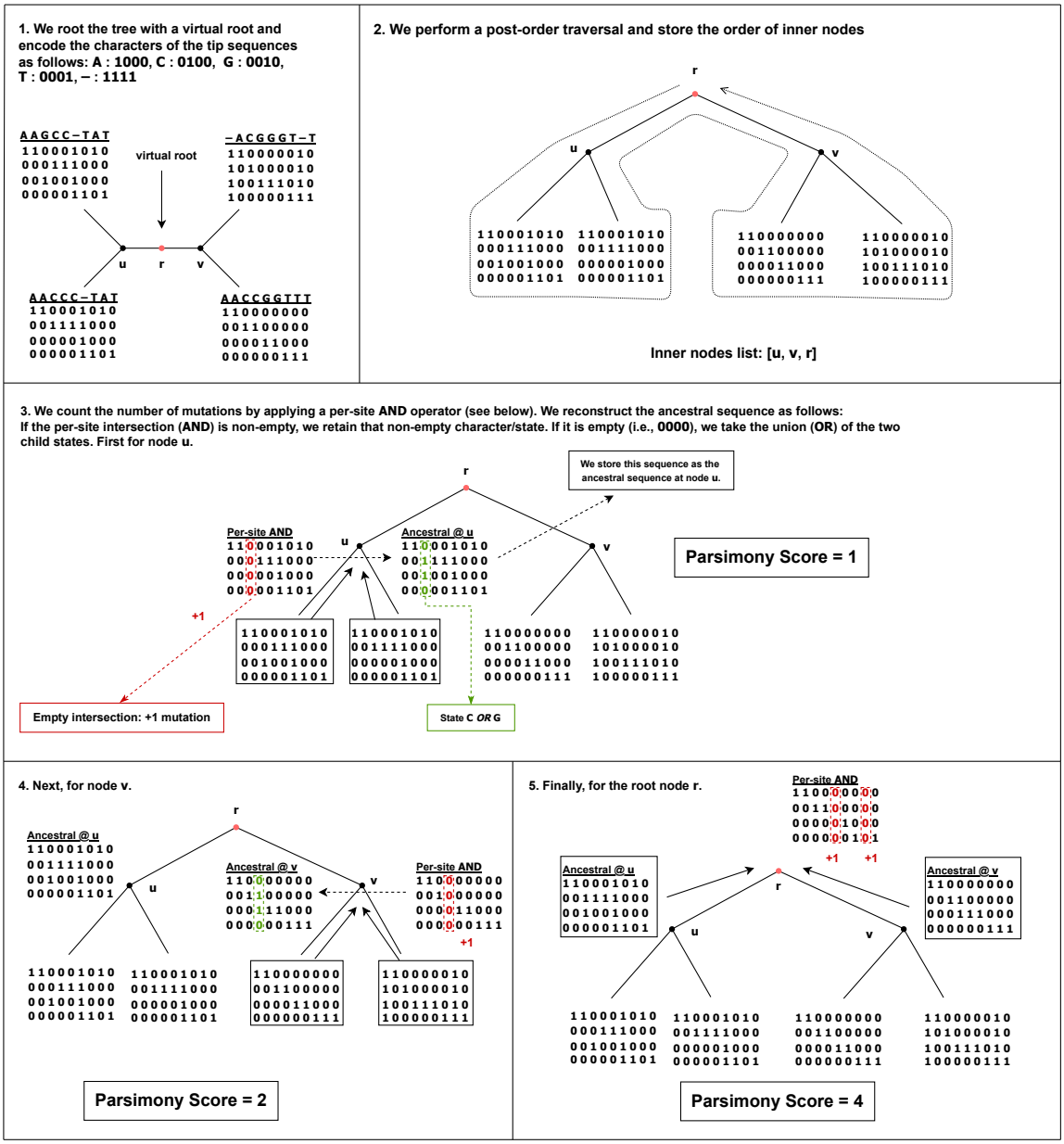


Figure 2.5: A schematic description of the dynamic programming algorithm used to compute the parsimony score of a given tree: 1. We assign a virtual root to the tree and encode the tip-sequence characters as bit vectors. **2.** We perform a post-order traversal and record the inner nodes in the order in which they are encountered **3–5.** We count the number of mutations and reconstruct the ancestral sequences for the inner nodes in the recorder order. At the root node (step **5**) the ancestral state sequence is not required, and hence not computed.

offers—and explains why it is also endorsed by optimized implementations—is that it enables using efficient bitwise arithmetic and vector instructions, which substantially reduce the constant factor overhead in the $\mathcal{O}(n \cdot s)$ complexity notation. Nonetheless, the problem of finding the MP tree is $\mathcal{NP}$-hard [36].

The parsimony score computation relies on several assumptions. I briefly mention

two of them. First, we assume that the MSA-sites evolve independently. This is, in fact, a universal assumption for character-based methods throughout this thesis, as it also applies to likelihood-based calculations (see Section 2.5). Second, MP assumes that along the branches of the phylogeny, at most one substitution per sequence-site can occur. Hence, this method completely neglects branch lengths, since regardless of how long a branch is, only at most one mutation per site and branch is admitted and counted. MP does not account for intermediate (hidden) mutations, which are, however, effectively modeled by the probabilistic, likelihood-based approaches.

Further, MP has important limitations. It is statistically inconsistent under certain conditions [50, 53], and a well-known artifact of the method is *long branch attraction (LBA)*, also referred to in the literature as the "Felsenstein's zone" [93]. Under LBA, highly distant taxa are incorrectly inferred to be grouped together, regardless of their true evolutionary history.

Another limitation of MP is that distinct topologies may have identical parsimony scores. Such a collection of equally scoring trees is called a *terrace* [166]. Terraces may arise both in the parsimony and the likelihood space of a given MSA (see Section 2.11). However, the underlying causes in ML and MP differ. In likelihood-based inference, terraces typically arise due to missing data in partitioned MSAs (see Section 2.5.4). In parsimony-based inference, on the other hand, it is rather possible for distinct topologies to have identical scores without requiring any underlying condition to hold, due to the discrete nature of the parsimony score itself.

2.5 Maximum Likelihood Tree Inference

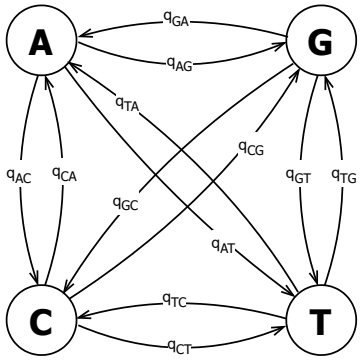
A more advanced approach to infer phylogenies is to maximize the phylogenetic likelihood function (PLF). This method is called *Maximum Likelihood (ML)* [52] tree inference. The general expression of the PLF is:

$$L\left(MSA \mid T, \bar{b}, \bar{\theta}\right) \tag{2.5}$$

where L denotes the probability that the given MSA was generated by the tree topology T, with branch lengths $\bar{b}$, under a given, *fixed* model of stochastic sequence evolution with model parameters $\bar{\theta}$. The topology, the branch lengths, and the model of sequence evolution jointly define the probabilistic model that describes how sequences evolved [228, 230]. The ML tree inference problem is $\mathcal{NP}$-hard [36]. Therefore, ML tools (presented in Section 2.6) deploy heuristics to find a "good" solution, which is typically a local optimum in the phylogenetic tree space [187]. In this section, I explicate the mathematical and computational framework underlying the PLF calculations on a given tree, with fixed continuous model parameters $(\bar{b}, \bar{\theta})$. In the next section, I describe the heuristics that ML tools deploy to navigate through phylogenetic tree space, as well as the numerical algorithms they use for continuous parameter optimization.

2.5.1 Models of Sequence Evolution

We assume that the sites of the MSA evolve independently. Each site evolves under a continuous-time Markov Chain (CTMC) model, with as many states as the respective alphabet of the sequence type. For DNA sequences, the model comprises 4 states: `A`, `C`, `G`, and `T`, corresponding to the four nucleotides; in protein sequences, the number of



$$Q = \begin{pmatrix} q_{AA} & q_{AC} & q_{AG} & q_{AT} \\ q_{CA} & q_{CC} & q_{CG} & q_{CT} \\ q_{GA} & q_{GC} & q_{GG} & q_{GT} \\ q_{TA} & q_{TC} & q_{TG} & q_{TT} \end{pmatrix}$$

$$\text{where} \begin{cases} q_{ij} \geq 0 & i \neq j \\ q_{ii} = -\sum_{j \neq i} q_{ij} & i, j \in \{A, C, G, T\} \end{cases}$$

Figure 2.6: (Left) CTMC model for nucleotide substitutions, represented as a state-transition graph. **(Right) The Q-matrix representation of the same model.** All non-diagonal entries of the Q-matrix are non-negative. The main diagonal entries are determined by the constraint that row sums must equal zero.

states equals the number of amino-acids (i.e., 20). The CMTC process is defined by an instantaneous transition rate matrix Q, whose elements q_{ij} denote the (instantaneous) substitution rates from state i to state j. Figure 2.6 illustrates the CTMC state-transition graph for nucleotides, along with the corresponding 4×4 Q-matrix. We assume the CTMC process to be *stationary*, that is, the Q-matrix remains static and does not change as a function of evolutionary time t. After an elapsed time t, the transition probability matrix $P(t)$ is obtained via the following matrix exponentiation:

$$P(t) = e^{Qt} \tag{2.6}$$

The element $P_{i,j}(t)$ of $P(t)$ quantifies the probability that a sequence site, initially in state i, will be observed to be in state j after time t. Unlike parsimony, this probability integrates over all possible intermediate (hidden) substitution paths [228]. For instance, $P_{A,T}(t)$ is the probability that, given an initial nucleotide `A`, the site evolves into `T` after time t, accounting for all possible intermediate scenarios, such as `A`→`T`, `A`→`C`→`T`, `A`→`C`→`A`→`T`, etc. Here, the branch lengths directly express relative evolutionary times between speciation events. For a branch of length b, the corresponding transition probability matrix is $P(t) \mid_{t=b} = e^{Qb}$. We slightly modify the former equation in Section 2.5.3, when we introduce the idea of *rate heterogeneity among sites (RHAS)*. Importantly, we assume that the Q-matrix is the same across all sites[1] [53, 174, 228].

From the definition of the Q-matrix in Figure 2.5, it follows[2] that the CTMC process reaches equilibrium as $t \to \infty$. The equilibrium is characterized by a stationary distribution vector $\boldsymbol{\pi}$, whose length equals the number of states. Each element π_i represents the time that a single site spends in state i at equilibrium. Consequently, across an evolving sequence, the expected proportion of characters being in state i equals π_i. For DNA sequences the stationary distribution vector takes the form $\boldsymbol{\pi} = [\pi_A, \pi_C, \pi_G, \pi_T]^\intercal$. We can compute the stationary distribution by solving the system:

$$\boldsymbol{\pi}^\intercal Q = 0$$

[1] Or more precisely, across large groups of sites called *partitions* (See Section 2.5.4).
[2] The mathematical conditions are met, although not described here in detail.

with the additional constraint:

$$\sum_i \pi_i = 1$$

The elements π_i of $\boldsymbol{\pi}$ are called *stationary base frequencies*.

Further, the CTMC process is *time reversible* if it satisfies the detailed balance condition:

$$\pi_i q_{ij} = \pi_j q_{ji},\ \forall i \neq j \tag{2.7}$$

Time reversibility is an important property of the Q-matrix, because it substantially simplifies likelihood computations on phylogenetic trees. In fact, the standard models that are used in ML phylogenetic inference (see below) are time reversible. Throughout this thesis, I assume time reversibility, and therefore use time-reversible models.

We can specifically select Q and $\boldsymbol{\pi}$ to satisfy Equation (2.7) as follows. Let $R = \{r_{ij}\}$ be a symmetric matrix (i.e., $r_{ij} = r_{ji}$ for $i \neq j$), with positive non-diagonal entries. Moreover, let $\boldsymbol{\pi}$ be a distribution vector, that is, with non-negative elements that sum to 1. We then define the Q-matrix as:

$$Q = R \cdot diag(\pi_i)$$

where $diag(\pi_i)$ is a diagonal matrix whose entries are the elements of $\boldsymbol{\pi}$. The resulting Q-matrix satisfies Equation (2.7), and is therefore time reversible. For DNA sequences we have:

$$\begin{aligned} Q &= R \cdot diag(\pi_i) \\ &= \begin{pmatrix} r_{AA} & \alpha & \beta & \gamma \\ \alpha & r_{CC} & \delta & \varepsilon \\ \beta & \delta & r_{GG} & \zeta \\ \gamma & \varepsilon & \zeta & r_{TT} \end{pmatrix} \begin{pmatrix} \pi_A & 0 & 0 & 0 \\ 0 & \pi_C & 0 & 0 \\ 0 & 0 & \pi_G & 0 \\ 0 & 0 & 0 & \pi_T \end{pmatrix} \\ &= \begin{pmatrix} r_{AA}\pi_A & \alpha\pi_C & \beta\pi_G & \gamma\pi_T \\ \alpha\pi_A & r_{CC}\pi_C & \delta\pi_G & \varepsilon\pi_T \\ \beta\pi_A & \delta\pi_C & r_{GG}\pi_G & \zeta\pi_T \\ \gamma\pi_A & \varepsilon\pi_C & \zeta\pi_G & r_{TT}\pi_T \end{pmatrix} \end{aligned}$$

The above Q-matrix corresponds to the general time reversible (GTR) model [205] of DNA sequence evolution. We compute the main diagonal entries based on the condition that the row elements of any Q-matrix must sum to 0. The model comprises a total of 8 free parameters: (a) 5—out of 6—relative substitution rates, because we typically set $\zeta = 1.0$ for normalization, and (b) 3—out of 4—stationary frequencies, for they must sum to 1. A second example of a DNA substitution model is the Jukes-Cantor model (JC69; [97]). Unlike GTR, the JC model is rather simplistic with no free parameters. All substitution rates are set to 1.0, and all stationary frequencies to 0.25.

In AA sequences, the number of character/states in the Markov model increases from 4 to 20, yielding a quadratic increase of free parameters. Specifically, the AA-based GTR model comprises 208 free parameters. This parameter increase introduces a substantial computational overhead in model estimation. In conjunction to this, the inferred model may be over-parametrized with respect to the available sequence data (MSA). The practical solution is to use empirically derived substitution models, such as the LG [116] or the WAG [223] model, that are estimated from very large collections of protein alignments.

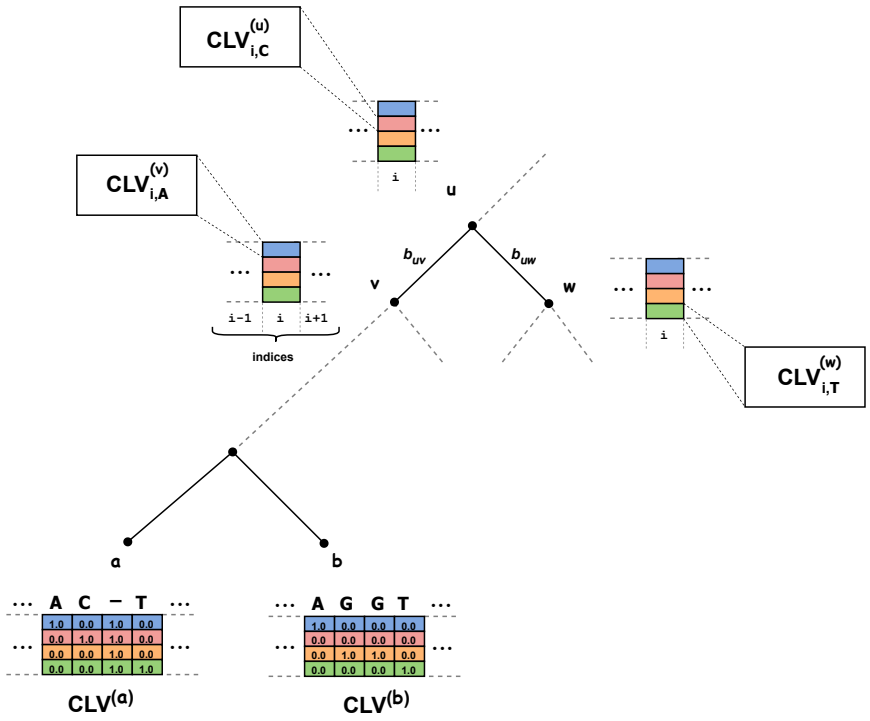


Figure 2.7: Illustration of Conditional Likelihood Vectors (CLVs) for tip sequences and inner nodes, on DNA sequences.

Returning to Equation (2.6), for a branch of length b, the transition probability matrix $P(b)$ is computed via the stated matrix exponential. We compute this exponential by applying an eigendecomposition of matrix Q, that is, we factorize Q as follows:

$$Q = U\Lambda U^{-1}$$

where U is a square matrix whose columns are the eigenvectors of Q, and Λ is a diagonal matrix, whose diagonal elements are the respective eigenvalues. Substituting this decomposition into Equation (2.6) gives:

$$P(b) = e^{U\Lambda U^{-1}b} = U \cdot diag(e^{\lambda_i b}) \cdot U^{-1} \tag{2.8}$$

where $diag(e^{\lambda_i b})$ is the diagonal matrix whose entries are $e^{\lambda_i b}$, with λ_i denoting the i-th eigenvalue of Q.

2.5.2 Likelihood Computation

We compute the PLF of a given tree by applying the Felsenstein's pruning algorithm [55]. The algorithm first assigns a virtual root to the tree. Owing to time reversibility, we can place the root on any inner branch, and at any point along that branch. In Section 2.6.5, I present how we benefit from this property in the implementation of Felsenstein's algorithm. Next, the algorithm performs a post-order traversal, recording all inner tree nodes in the order they are visited. For each inner node, it computes a *Conditional Likelihood Vector (CLV)*. A CLV stores, for every site and character, the probability of

observing that character at the given site, at the the respective inner node. Therefore, it summarizes the information from the node's descending subtree, since it stores the conditional likelihood for each possible ancestral state, given the descending subtree and tip sequences (sub-alignment) contained therein.

In the following, I demonstrate the CLV computations by an example of DNA sequences. The procedure for AA sequences is analogous, as it only differs in the state space size.

At the tip sequences, we initialize trivial CLVs for each site, according to the observed sequence character at that site. Specifically, the per-site CLVs for each possible nucleotide (including the gap character "-") are:

$$\begin{aligned} \texttt{A} &\rightarrow (1,0,0,0) \\ \texttt{C} &\rightarrow (0,1,0,0) \\ \texttt{G} &\rightarrow (0,0,1,0) \\ \texttt{T} &\rightarrow (0,0,0,1) \\ \texttt{-} &\rightarrow (1,1,1,1) \end{aligned}$$

I provide an exemplary illustration of the arrangement of CLVs on a fixed tree in Figure 2.7. To reference a unique CLV entry at a specific tree node, I use the notation $CLV_{i,q}^{(u)}$, where u is the node of interest, i is the index of the MSA site (i.e., $i \in \{1, 2, \ldots, s\}$), and q represents the character state (i.e., $q \in \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$). For an inner node u with child nodes v and w (see Figure 2.7), we compute the CLV entries as follows:

$$CLV_{i,q}^{(u)} = \Big(\sum_{q_1 \in \{A,C,G,T\}} P_{q,q_1}(b_{uv}) \cdot CLV_{i,q_1}^{(v)} \Big) \Big(\sum_{q_2 \in \{A,C,G,T\}} P_{q,q_2}(b_{uw}) \cdot CLV_{i,q_2}^{(w)} \Big) \tag{2.9}$$

where b_{uv} and b_{uw} are the lengths of the branches (u, v) and (u, w), respectively. Having computed the $CLV^{(r)}$ for the root node r, we obtain the *per-site likelihoods* of the given tree by calculating the inner product of the per-site CLVs with the stationary base frequencies vector $\boldsymbol{\pi} = [\pi_A, \pi_C, \pi_G, \pi_T]^\intercal$. Specifically, for each site i, the per-site likelihood is:

$$L_i = \sum_{q \in \{A,C,G,T\}} \pi_q \cdot CLV_{i,q}^{(r)} \tag{2.10}$$

Since we assume that sites evolve independently, the overall likelihood of the given tree is the product over the per-site likelihoods, that is, $L = \prod_{i=1}^{s} L_i$. To avoid numerical overflow, it is a common practice to use the natural logarithm of the likelihood (*log-likelihood*):

$$\log L = \sum_{i=1}^{s} \log L_i \tag{2.11}$$

2.5.3 Rate Heterogeneity Among Sites: The Γ Model

Different regions of the MSA evolve at different speeds, due to varying degrees of evolutionary pressure they undergo. The CTMC models of sequence evolution, described thus far, assume a uniform rate of evolution across all sites, and do not account for such variability. To incorporate evolutionary rate variation, several models of *rate heterogeneity among sites (RHAS)* have been proposed. A commonly used RHAS model is the Γ-model [229], which I describe below.

The assumption is that each substitution rate is a random variable X which follows a Γ distribution, with probability density function:

$$f_\Gamma(x;\alpha,\beta) = \frac{\beta^\alpha}{\Gamma(\alpha)} x^{\alpha-1} e^{-\beta x} \text{ for } x > 0 \text{ and } \alpha,\beta > 0 \tag{2.12}$$

where α,β are the shape and the inverse scale parameters, respectively, and $\Gamma(\cdot)$ is the gamma function. A common numerical practice is to approximate the continuous Γ distribution via K distinct rate categories. To achieve this, we divide the space of possible values $r \in [0,\infty)$ into K intervals $I_1, I_2, \ldots, I_K$ with equal probability mass, i.e.:

$$\begin{aligned}
\int_0^{x_1} f_\Gamma(x;\alpha,\beta)dx &= \frac{1}{K} \;\rightarrow\; I_1 = [0, x_1) \\
\int_{x_1}^{x_2} f_\Gamma(x;\alpha,\beta)dx &= \frac{1}{K} \;\rightarrow\; I_2 = [x_1, x_2) \\
&\vdots \qquad\qquad\qquad \vdots \\
\int_{x_{K-1}}^{\infty} f_\Gamma(x;\alpha,\beta)dx &= \frac{1}{K} \;\rightarrow I_K = [x_{K-1}, \infty)
\end{aligned}$$

For each interval I_k, we select a representative rate r_k, which is either the mean or the median of that interval. Given the selected rates $r_k, k = 1, 2, \ldots, K$, we adjust the P-matrix calculation described in Equation (2.6) as follows:

$$P(t, r_k) = e^{r_k Q t} \tag{2.13}$$

CLVs are then computed separately for each rate category. This increases the memory requirement for storing CLVs by a factor of K. The number of floating-point operations (FLOPs) also increases by the same factor, since the likelihood needs to be computed independently for each rate category. To denote the per-category CLVs, I extend the CLV notation introduced in Section 2.5.2. The extended notation is $CLV^{(u)}_{i,q,k}$, where the additional subscript index k identifies the gamma rate r_k associated with the indicated CLV entry. The rates-informed version of Equation (2.9), which describes the recursive CLV computation, is (for DNA sequences):

$$\begin{aligned}
CLV^{(u)}_{i,q,k} = \Big(\sum_{q_1 \in \{A,C,G,T\}} P_{q,q_1}(b_{uv}, r_k) \cdot CLV^{(v)}_{i,q_1,k} \Big) \times \\
\Big(\sum_{q_2 \in \{A,C,G,T\}} P_{q,q_2}(b_{uw}, r_k) \cdot CLV^{(w)}_{i,q_2,k} \Big)
\end{aligned} \tag{2.14}$$

Following the same logic, the per-category per-site likelihoods at the root node are computed for each rate category separately:

$$L_{i,k} = \sum_{q \in \{A,C,G,T\}} \pi_q \cdot CLV^{(r)}_{i,q,k}$$

The overall per-site likelihood at the root is then defined as the average likelihood over the K categories:

$$L_i = \frac{1}{K} \sum_{k=1}^{K} L_{i,k}$$

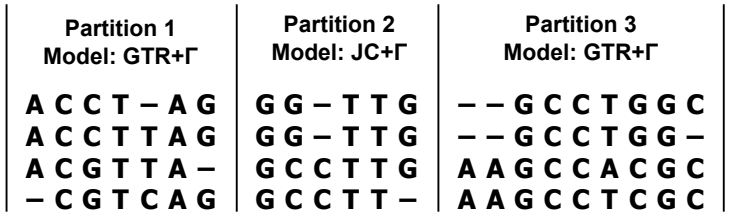


Figure 2.8: A toy example of a partitioned MSA, comprising three partitions.

Finally, we compute the log-likelihood of the given tree via Equation (2.11).

The parameters α and β of the Γ-model require optimization. We can eliminate one of them by applying a simple adjustment. For the probability density function $f_\Gamma(x; \alpha, \beta)$ of the Γ distribution (Equation (2.12)), the expected value is $E[X] = \beta/\alpha$. Further, when substituting $r_k = 1$ in Equation (2.13), the resulting P-matrix is identical to its rate-free version in Equation (2.6). Hence, we can center the Γ distribution around 1.0, by setting $E[X] = 1 \Rightarrow \alpha = \beta$, such that the per-rate likelihoods are directly comparable to the rate-free case. Consequently, in the Γ model we only need to optimize the shape parameter α.

In standard implementations, the default number of discreet Γ rate categories is $K := 4$, following the recommendations by Z. Yang [229]. I adopt this default setting ($K := 4$) throughout this thesis. To indicate the use of a substitution model in combination with Γ rates, I use the standard notation and append a "+Γ" suffix to the model name (e.g., GTR+Γ, LG+Γ, etc.).

2.5.4 Multi-partitioned MSAs

Thus far in likelihood computations, we have assumed that all sites of the MSA evolve under the same substitution rate matrix Q, whose parameters are optimized based on the entire alignment. Further, we accounted for evolutionary rate heterogeneity only through the RHAS Γ-model, whose shape parameter α is likewise optimized on the full MSA. Under this setup, the MSA is treated as being *unpartitioned*. There are scenarios, however, where distinct groups of sites exhibit highly divergent evolutionary patterns. A common example is when separate per-gene MSA are concatenated into a larger alignment (see below). In such cases, separate substitution models should be applied to, and optimized for, the respective site groups. A common technique among practitioners is, therefore, to divide the MSA into disjoint *partitions*, and assign a partition-specific Q-matrix to each, possibly accompanied by a partition-specific Γ-model of RHAS. The resulting alignments are called *partitioned* MSAs.

Figure 2.8 illustrates a toy example of a partitioned MSA, comprising three partitions. We assign the GTR+Γ model to the first partition, the JC+Γ model to the second, and (once more) the GTR+Γ model to the third. Importantly, the Q-matrices of each partition, and the corresponding α-shape parameters of the partition-specific Γ models, are optimized solely on the respective partition sites. In terms of the substitution and RHAS models:

- Partition 1 comprises 8 free parameters for the substitution model (GTR), and 1 free shape parameter for the Γ-model of RHAS;
- Partition 2 comprises 0 free parameters for the substitution model (JC), and 1 free

shape parameter for the Γ-model of RHAS;
- Partition 3 comprises 8 free parameters for the substitution model (GTR), and 1 free shape parameter for the Γ-model of RHAS.

The total number of free parameters across all substitution and RHAS models is 19.

MSA partitions typically represent either groups of sites, or genes/loci within genomes (see below), which *appresent* their own idiosyncratic evolutionary dynamics, yet do share a common tree topology. This evolutionary *ipseity* can also be modeled in the way branch lengths are handled among partitions[1]. Specifically, there are three standard ways to estimate branch lengths when analyzing partitioned MSAs:

- **Linked branch lengths**: Branch length are shared among partitions, and per-partition log-likelihoods are computed using these shared branch lengths.
- **Unlinked branch lengths**: Branch lengths are treated as free parameters on a per-partition basis. Each partition stores its own branch length values, which are optimized to maximize the per-partition log-likelihood (and consequently the overall score). There are two main drawbacks associated with unlinked branches. First, this approach substantially increases the number of free parameters, potentially leading to over-parametrization. Specifically, the number of branch lengths to be optimized is multiplied by the number of partitions. Second, analyzing partitioned MSAs using unlinked branch lengths may give rise to *terraces*, that is, sets of trees that yield identical log-likelihood scores, if the given partitioned MSA exhibits missing data (i.e, missing partition entries; see Section 2.11).
- **Scaled branch lengths**: This approach offers a compromise between the linked and unlinked approaches. The branch lengths of the underlying topology are shared among partitions, yet each partition has an additional scaling parameter, that uniformly scales all branch lengths of the topology. Hence, the only additional parameters are the partition scaling factors.

The standard approach to compute the likelihood of a given tree on a partitioned MSA is the *concatenation* (or *supermatrix*) method. In this configuration, we assemble all disjoint partitions into a single large supermatrix, and compute the overall log-likelihood score by aggregating the per-partition log-likelihoods, each computed under the partition-specific substitution and RHAS models.

In practice, users construct partitioned MSAs in two ways. First, they infer partitions implicitly, for example when analyzing MSAs with no specific prior knowledge (e.g., structural or functional information), and the goal is to identify meaningful partitions to improve the accuracy of phylogenetic reconstruction. For this purpose they employ tools such as the PartitionFinder [115] to individuate partitions in an optimal way. This tool evaluates a wide range of possible partitioning schemes accompanied by substitution (and possibly RHAS) models, and selects the best-fitting combination. The optimal fit is expressed as a trade-off between model-fit (i.e., log-likelihood scores) and model complexity (i.e., the number of free parameters), in order to avoid over-parametrization. Appropriate model selection criteria, which quantify the aforementioned trade-off, are the Akaike information criterion (AIC; [4]), and the Bayesian information criterion (BIC; [173]). In the second scenario, practitioners construct partitioned MSA by concatenating distinct MSAs, typically corresponding to separate genes or loci within the genomes of the or-

[1]The verb *appresent* is inspired by the Husserlian concept of appresentation (German: Appräsentation), which refers to the indirect presentation of something through what is directly given. The term *ipseity* (from Latin *ipse*, meaning "self") is also borrowed from Philosophy and denotes selfhood or intrinsic identity.

ganisms/individuals under study. In this case, partitions are defined explicitly. Model selection, though, can still be performed on the fixed partitions [34, 149].

2.6 Heuristics and Tools

In this Section, I provide a detailed overview of the heuristics that ML tree inference tools deploy to identify trees with "good" log-likelihood scores. Those trees typically represent local optima in the phylogenetic tree space. I place particular emphasis on RAxML-NG [111], since it is the main tool that our lab actively develops and maintains. Therefore, RAxML-NG is the software I most frequently use throughout the thesis, and into which I integrate my methods. I also refer to old classic RAxML [193], the predecessor of RAxML-NG. Other ML tools that I use, compare against, or simply refer to, are: IQ-TREE [131], FastTree [151], and PhyML [68].

I begin by describing a simple heuristic implemented in RAxML-NG to infer parsimony trees, i.e., trees with a "good" parsimony score. We might also consider this heuristic as a part of the ML tree inference process, for which parsimony trees typically serve as starting points. This is because they constitute reasonable initial estimates, for they tend to have relatively high, yet suboptimal, log-likelihood scores [133, 193]. Next, I describe the basic topological moves that ML tools apply to navigate through tree space. I briefly mention the numerical methods that RAxML-NG employs to optimize continuous parameters, such as the branch lengths and the substitution model parameters. I also provide an overview of the heuristics that the RAxML-NG, IQ-TREE, and FastTree tools deploy to infer their best ML trees, again with particular emphasis on RAxML-NG. Finally, I discuss specific implementation details of RAxML-NG, along with optimizations and shortcuts, which I consider useful for the reader.

2.6.1 Stepwise Taxon Addition to Infer Parsimony Trees

This heuristic is a rather simplistic and straightforward approach to infer "good" parsimony starting trees. RAxML-NG begins by randomly shuffling the list of input taxa. From the first three taxa in the shuffled list it builds the star topology, that is, the unique unrooted topology comprising three taxa, all of them being connected into a single inner node. For each subsequent taxon in the list, RAxML-NG (a) iteratively attaches the taxon to every branch of the current unrooted tree, and computes the parsimony scores of the resulting topologies; (b) retains the topology with the lowest parsimony score and proceeds to the next taxon. The algorithm continues in this manner until all taxa have been added to the tree.

The stepwise taxon addition heuristic operates in $\mathcal{O}(n^2)$ steps, where n is the number of taxa. Since the parsimony score calculations are highly optimized, owing to the use of bitwise arithmetic and vector instructions (see Section 2.4), the runtime required to build parsimony starting trees using this approach only constitutes a small fraction of the overall ML tree inference time. Importantly, this heuristic does not guarantee that the inferred topology is a local parsimony optimum. That is, there may exist SPR- or NNI-neighbors (see next Section for definitions), with lower parsimony scores. RAxML version 8 performs additional parsimony-based SPR optimization rounds (see Section 2.6.4) to further refine the parsimony topology and identify a local optimum. However, empirical observations and experiments in RAxML-NG have shown that such parsimony-based

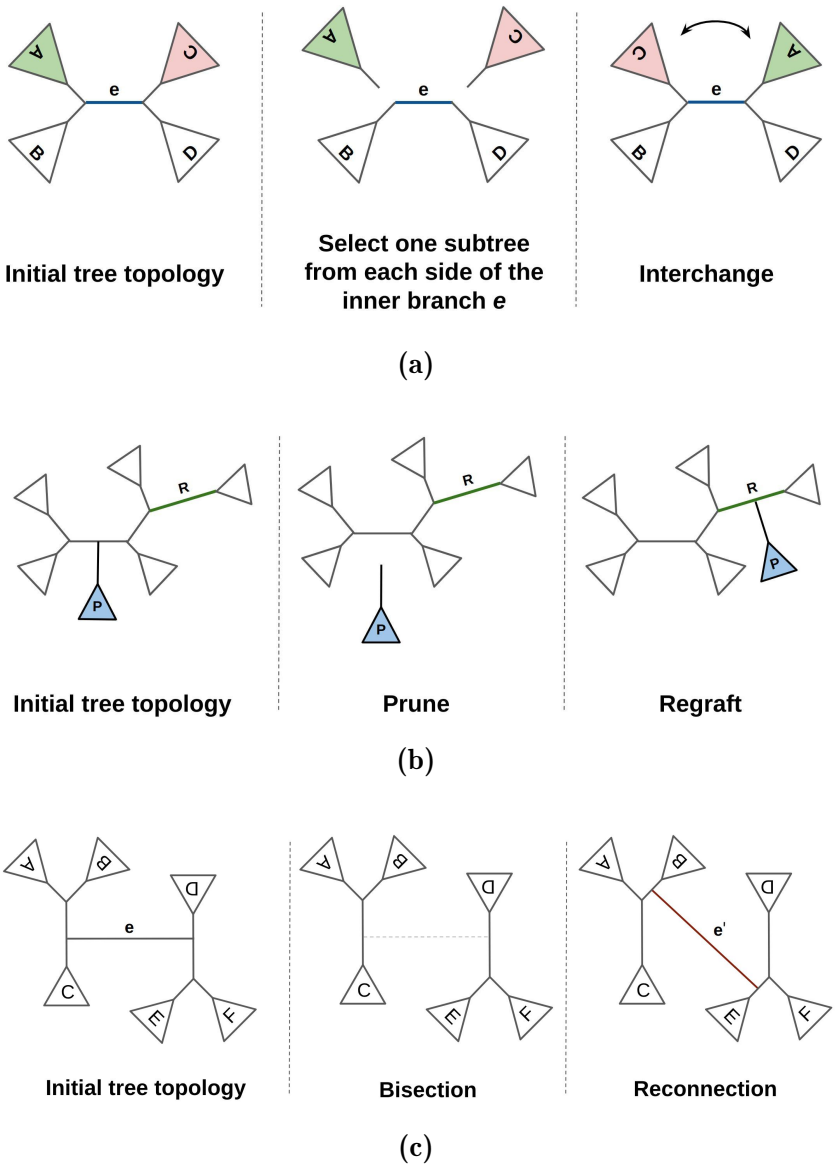


Figure 2.9: (a) An NNI move example around an inner branch e**.** Two subtrees, one from each side of the inner branch, are selected, pruned, and interchanged. The NNI move is a special case of the SPR move. **(b) An SPR move example.** The subtree P is pruned from the initial comprehensive tree and regrafted into branch R. The SPR move is a special case of the TBR move. **(c) A TBR move example.** An inner branch e is selected and pruned, bisecting the tree into two disconnected subtrees. A new branch e' reconnects the two subtrees, resulting in a new topology (Subfigures **(a)** and **(b)** are reproduced from [210]).

topological optimizations do not provide any substantial benefit regarding the final ML inference result [110, 111], while adding unnecessary computational overhead. Therefore, RAxML-NG only uses the stepwise heuristic for parsimony starting tree inference.

2.6.2 Topological Moves

ML tools explore the tree space by applying three types of topological rearrangements or moves: *Nearest Neighbor Interchange (NNI)*, *Subtree Prune and Regraft (SPR)*, and *Tree*

Bisection and Reconnection (TBR) moves. I provide a schematic illustration of those moves in Figure 2.9. NNI moves are a special case of SPR moves, which in turn are a special case of TBR moves.

RAxML and RAxML-NG v1.x (i.e., versions 1.1 and 1.2) exclusively employ SPR moves to optimize the tree topology. One of my contributions that I present in this thesis is the implementation of an NNI-based heuristic in `coraxlib`, that is, the COre RAXml LIBrary [47] which we actively develop as the successor to `libpll` [58]. Both `coraxlib` and `libpll` implement the likelihood functions used in RAxML. I integrated this NNI-based heuristic into Adaptive RAxML-NG [210], that employs NNI moves to infer ML trees on easier-to-analyze MSAs. I elaborate on this contribution in Chapter 3. On the other hand, IQ-TREE solely relies on NNI moves for tree space exploration. FastTree and PhyML use a combination of both SPR and NNI rearrangements.

The advantage of using NNIs instead of SPRs is computational efficiency: for a fixed topology, the number of NNI neighbors to *evaluate* (i.e., to compute the log-likelihood score for) is $\mathcal{O}(n)$, whereas the SPR ones are $\mathcal{O}(n^2)$ [82]. Hence, in principle, an NNI-based heuristic may reach a local optimum faster than an SPR-based one.

However, in practice, NNI moves fall short of SPRs in two aspects. First, SPRs potentially induce larger log-likelihood improvements than NNIs, yielding greater "jumps" during the tree inference process. Thus, when starting from suboptimal trees (e.g., random trees), a well-adjusted SPR heuristic (see Section 2.6.4) may actually converge to a local optimum faster than an NNI-based one, particularly so on large trees[1]. The second drawback of NNIs is that they tend to get stuck in worse local optima. Therefore, the quality of the inferred trees with respect to their log-likelihood score favors SPR-based heuristics.

2.6.3 Optimization of Continuous Parameters in RAxML-NG

On a fixed tree topology T, RAxML-NG optimizes the branch lengths $\bar{b}$ and model parameters $\bar{\theta}$ (e.g., substitution rates, equilibrium frequencies, gamma rates, etc.) using general purpose numerical optimization algorithms. These continuous parameters are optimized iteratively. That is, RAxML-NG optimizes one parameter at a time, while keeping the others fixed. It iterates over the specified set of continuous parameters (depending on the invoked routine; see next paragraph), and repeats this process for multiple rounds, until convergence. For the optimization of branch lengths, RAxML-NG uses the Newton-Raphson method, whereas for model parameter optimization it employs the Brent [24], and the Broyden-Fletcher-Goldfarb-Shanno (BGFS; [57]) algorithms.

Within the RAxML-NG workflow (see Figure 2.13 below), we distinguish between two main continuous parameter optimization blocks: the Branch-Length Optimization (BLO) round, and the Model-Parameter Optimization (MPO) round. During BLO rounds, RAxML-NG only optimizes the branch lengths, whereas MPO rounds optimize *all* continuous parameters, that is, both branch lengths $\bar{b}$ and model parameters $\bar{\theta}$. More details on the continuous parameter optimizations in RAxML-NG can be found in [110].

2.6.4 SPR and NNI Rounds in RAxML-NG

An SPR round is the core topological optimization block in RAxML-NG. The optimization heuristic is based on the greedy hill-climbing algorithm previously introduced in

[1]Large trees are trees with many taxa.

RAxML [193], with minor modifications [110]. The SPR round routine accepts several input parameters; at this stage, it suffices to mention r_{min} and r_{max}, which denote the minimum and maximum *reinsertion* (or *regrafting*) *radii*, respectively.

A single invocation of the SPR round routine proceeds as follows. Initially, the algorithm stores pointers to all inner nodes (each one being represented by a unique virtual node, see below) of the initial tree topology. For every pointer in the list, RAxML-NG prunes the corresponding subtree (the one that extends in the direction of Left-Child and Right-Child traversal moves, see below) from the thus far, best-scoring tree, and evaluates its possible reinsertions into neighboring branches, whose distance from the pruning branch lies within $[r_{min}, r_{max}]$. The move that yields the highest likelihood improvement, if such a move exists, is accepted and the algorithm proceeds to the next element in the node list. When SPR moves for all pointers/nodes have been evaluated, and therefore all corresponding subtrees have been pruned and regrafted once, the SPR round is completed.

When RAxML-NG prunes a subtree during an SPR move, it also removes the emerging degree-2 (inner) node from the pruning edge, and merges the two adjacent branches by summing their lengths. Similarly, when regrafting, it reinserts the pruned subtree at the midpoint of the target edge, thereby splitting it into two branches of equal-length. There are two types of SPR rounds in RAxML and RAxML-NG: *FAST* and *SLOW SPR* rounds. The main difference is that, in FAST SPR rounds, RAxML-NG evaluates each tree topology generated by an SPR move using the existing branch lengths, whereas in SLOW SPR rounds, the lengths of the three branches adjacent to the subtree insertion node are being reoptimized. RAxML (but not RAxML-NG) further optimizes the pruning branch in SLOW SPR rounds; RAxML-NG leaves this branch unchanged.

In addition to SPR moves, I implemented an NNI-based heuristic in RAxML-NG, for the purpose of integrating it into the Adaptive RAxML-NG tree search strategy (see Chapter 3). This NNI heuristic is analogous to the NNI round implementation in RAxML v8.2 (`-f J` option; [195]). The algorithm starts from an inner branch and compares the three alternative NNI-neighboring topologies around that central inner branch, in terms of their likelihood score. For computing the likelihood, the algorithm optimizes the five branches which are most affected by an NNI move. These are, the central inner branch itself, and the four adjacent branches. Once an NNI evaluation has been executed, the algorithm accepts the topology with the highest likelihood score and proceeds to the adjacent inner branches. When all inner branches have been visited once, the algorithm returns to the initial inner branch and repeats the process for multiple iterations, until it reaches an NNI-optimal tree (i.e., a tree for which the application of any additional NNI move will not further improve the likelihood).

A parameter of the NNI round routine that requires special reference is, the so-called, NNI tolerance, denoted here as ε_{NNI}. The role of this parameter is to determine convergence. That is, the NNI round iterations continue as long as the log-likelihood improvement between successive iterations exceeds ε_{NNI}. In practice, ε_{NNI} is set to a value that is sufficiently large to tolerate numerical imprecisions in log-likelihood calculations [176, 200], and yet sufficiently small to provide a good approximation of the local optimum. For example, Adaptive RAxML-NG uses a default value of $\varepsilon_{NNI} := 0.1$ (see Chapter 3), whereas in subsequent improvements of the RAxML-NG heuristic, I increased this default value to $\varepsilon_{NNI} := 1.0$ (see Chapter 6).

Another distinction between SPR and NNI rounds lies in the iteration schemes. As described above, the SPR round consists in a single traversal of all inner nodes, that is, a

single iteration. By contrast, the NNI round repeats the process over multiple iterations until it reaches an NNI-local optimum. Consequently, a single SPR round on its own is not guaranteed to find an SPR optimum. This means that there may exist SPR neighbors within the regrafting radius range $[r_{min}, r_{max}]$ with a higher log-likelihood score. For this reason, both RAxML and RAxML-NG perform multiple successive invocations of the SPR round routine to ensure convergence to an SPR optimum.

2.6.5 RAxML and RAxML-NG

In this Section, I present specific implementation details of the RAxML and RAxML-NG tools, including the data structure the tools utilize to store the tree and CLVs, the ML heuristic they deploy, optimization and parallelization strategies, and further shortcuts they apply in their ML tree inference heuristics. The description is primarily based on RAxML-NG software. However, most of these details are shared between the tools [188, 197].

Tree and CLVs Implementations in `coraxlib`

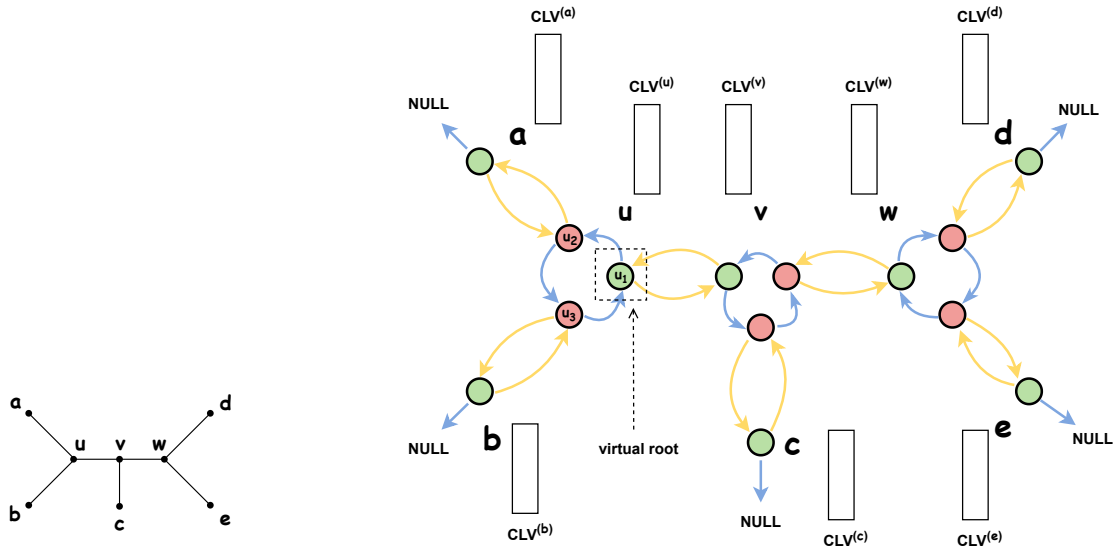
The `coraxlib` data structure which RAxML-NG uses to store the tree and the CLVs, is shown in Figure 2.10. The rudimentary building elements of this structure are the so-called *virtual nodes*. The tree graph is essentially an arrangement of virtual nodes connected via pointers. Each virtual node is equipped with two pointers: a `next` and a `back` pointer. Every (actual) inner node of the tree comprises three virtual nodes, connected in a circular arrangement via their `next` pointers. The `back` pointers, in turn, represent the branches that connect adjacent nodes in the tree. That is, they link virtual nodes belonging to distinct (actual) tree nodes. Tip nodes are represented by a single virtual node, whose `next` pointer is set to `NULL`. This property can thus be used to identify tip nodes, by checking whether their `next` pointer is `NULL`. Owing to time reversibility, the virtual root of the tree can be assigned to any of the three virtual nodes of any inner node (see Figure 2.10).

Along with the `next` and `back` pointers, each virtual node also stores a boolean variable called `clv_valid`. This variable indicates which of the three virtual nodes of an (actual) inner node holds the computed/updated CLV. In order to understand its utility, we first need to explain how `coraxlib` (and consequently RAxML-NG) performs a full post-order tree traversal.

Assume that we begin from the virtual node $\mathtt{u}_1$ (of tree node `u`) of the data structure shown in Figure 2.10. From this node, there are two possible traversal moves.

- The **Right-Child Move (RCM)** is performed by invoking the `next` pointer once, followed by the `back` pointer. Thus, the algorithm reaches the right-child node of $\mathtt{u}_1$ through $\mathtt{u}_1 \rightarrow \mathtt{next} \rightarrow \mathtt{back}$.
- The **Left-Child Move (LCM)** is performed by double invocation of the `next` pointer, followed by the `back` pointer. Thus, the algorithm reaches the left-child node of $\mathtt{u}_1$ through $\mathtt{u}_1 \rightarrow \mathtt{next} \rightarrow \mathtt{next} \rightarrow \mathtt{back}$.

From $\mathtt{u}_1$'s child nodes, the algorithm re-applies the RCM and LCM moves recursively until it reaches the tip nodes. However, if the starting node $\mathtt{u}_1$ is the virtual root, the algorithm must also traverse the subtree on opposite to $\mathtt{u}_1$. That is, the subtree descending from the opposite side of the root edge, which is accessible through $\mathtt{u}_1$'s `back` pointer. In this case only, the algorithm invokes the back pointer once to reach the opposite subtree, and then



(a) (b)

Figure 2.10: Illustration of the basic data structure used to store the tree and CLVs in `coraxlib`. (a) An example of a simple unrooted tree T on $\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}\}$. The inner nodes are conventionally labeled as `u`, `v`, and `w`; the latter labels are auxiliary to refer to inner nodes. **(b) The representation of** T **within `coraxlib`**. Each inner node (e.g., `u`) corresponds to a triplet of *virtual* nodes that are drawn as black circles (with green or red fill colors). For example, the inner node `u` is represented by the triplet of virtual nodes $\{\mathtt{u}_1, \mathtt{u}_2, \mathtt{u}_3\}$. Each virtual node is equipped with a `next` (blue arrow) and a `back` (yellow arrow) pointer. The three virtual nodes representing an inner node are connected circularly via `next` pointers, whereas `back` pointers indicate the branches connecting the nodes of T. Tip nodes consist of a single virtual node, whose `next` pointer is set to `NULL`. Each virtual node also stores a boolean variable (`clv_valid`), indicated here by green (for `TRUE`) and red (for `FALSE`) fill colors. This variable marks which virtual node within a triplet holds the respective up-to-date CLV (see text). The virtual root is set to one of the three virtual nodes of an inner node (here $\mathtt{u}_1$).

again applies the RCM and LCM moves recursively. We call this one-time back-move from the virtual root the **Root-Back Move (RBM)**.

Figure 2.11 schematically illustrates how Felsenstein's pruning algorithm [55] is implemented in `coraxlib`. Initially, the `clv_valid` variables of *all* inner virtual nodes are set to `FALSE`. This initialization step is called CLV *invalidation*, and indicates that either (a) the CLVs have to be computed for the first time, or (b) they have to be recomputed/updated (see below). Second, the algorithm performs a post-order traversal by recursively applying RCM and LCM moves, along with the one-time RBM move from the virtual root node. Having reached the tip nodes, the algorithm iteratively computes the CLVs of inner virtual nodes, as defined by the post-order traversal. Each time a CLV is computed/updated for an inner virtual node, its `clv_valid` variable is set to `TRUE`, that is, the CLV is *validated*. Hence, a `clv_valid` value of `TRUE` for an inner virtual node $\mathtt{u}_0$ indicates that the corresponding CLV entries are up-to-date with respect to the descending subtree, defined by the recursive application of RCM and LCM moves *only*.

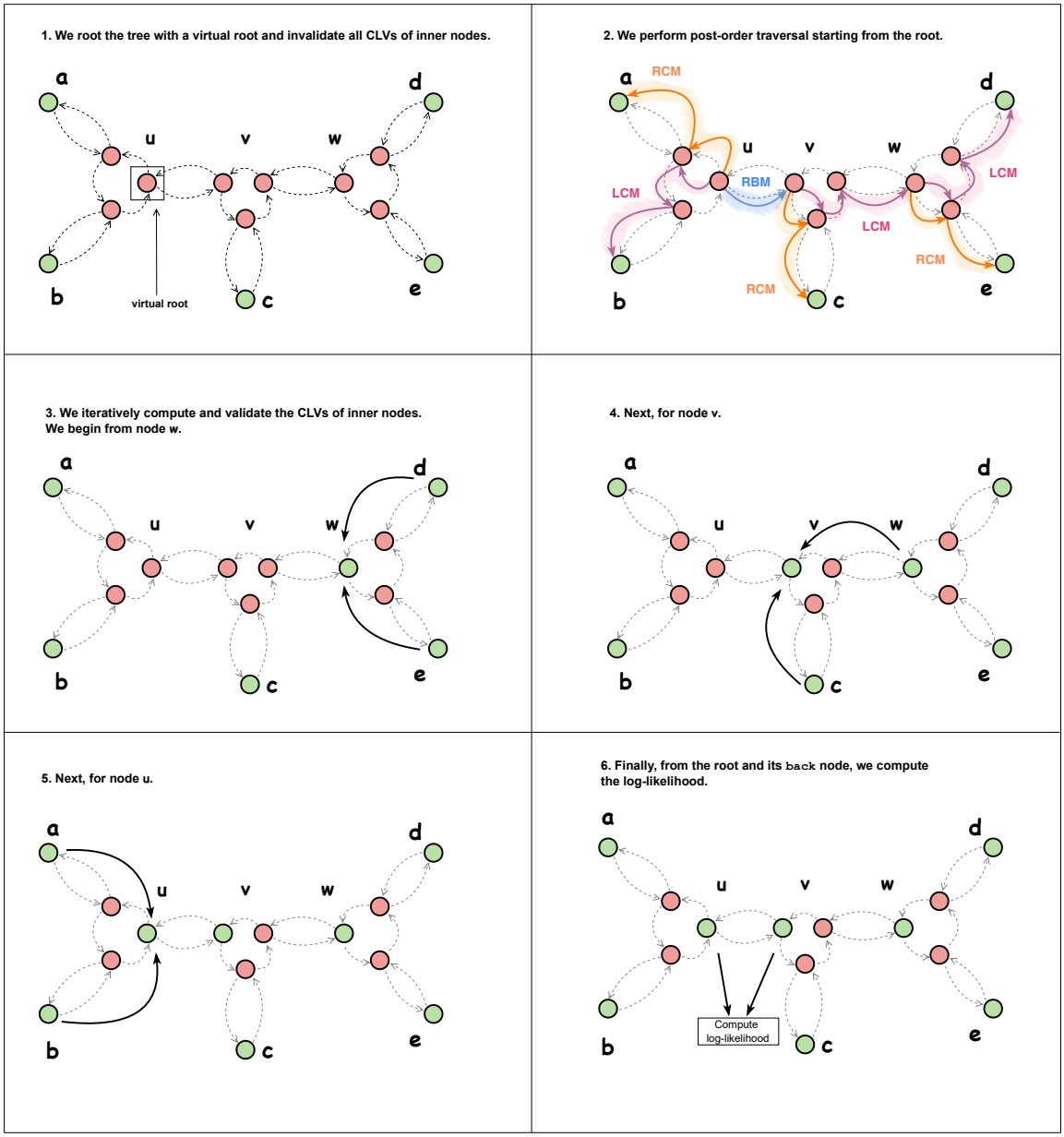
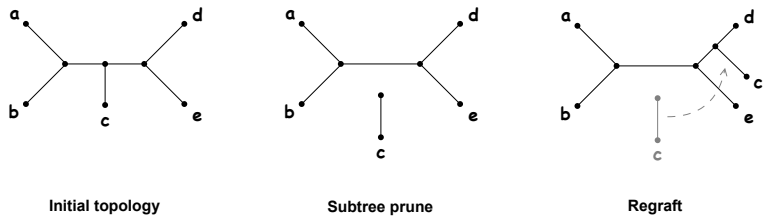


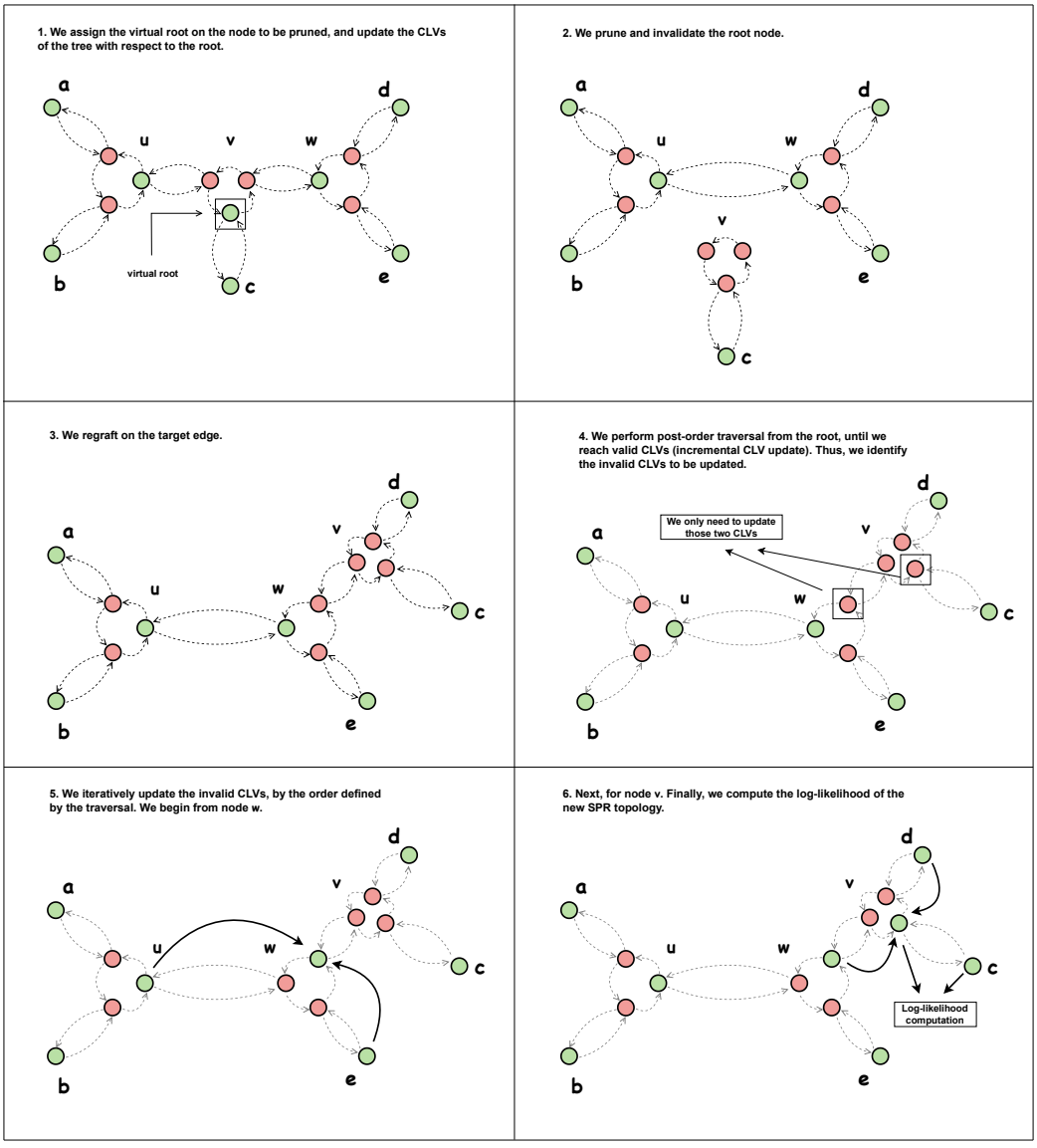
Figure 2.11: Implementation of Felsenstein's pruning algorithm in `coraxlib`: 1. We assign a virtual root and invalidate all CLVs of all inner virtual nodes. **2.** We perform a post-order traversal by recursively applying RCM and LCM moves, along with a one-time RBM from the virtual root node. A single RCM move is depicted by two consecutive orange arrows; a single LCM move by three consecutive purple arrows; and the RBM move by single blue arrow. **3–5.** We iteratively compute CLVs for inner virtual nodes in the order defined by the traversal, and validate each inner node upon update. **6.** Compute the overall log-likelihood score across the root edge.

Moreover, this property holds recursively for all virtual nodes reachable from u_0 through RCMs and LCMs, for which `clv_valid == True`. Evidently, for each triplet of virtual nodes per inner node, at most one virtual node can have `clv_valid == True`.

After all CLVs for the inner virtual nodes have been computed/updated, the two adjacent virtual nodes on the root edge (i.e., the virtual root and its opposite node)



(a)



(b)

Figure 2.12: Demonstration of incremental CLV update in `coraxlib`, in the context of an SPR move. (a) An example of a simple SPR move. (b) Internal operations within `coraxlib` to execute the SPR move: 1. The virtual root is moved to the node that is pruned, and the CLVs are updated with respect to the new root. **2–3.** The root node is pruned, invalidated, and regrafted onto the target edge. **4.** A post-order traversal is performed, starting from the root, until the algorithm reaches valid CLVs. **5–6.** Incremental updates of invalid CLVs, in order to compute the likelihood of the resulting SPR topology.

are both valid; that is, `clv_valid == True` holds for both. This is, in fact, the only edge in the data structure where both adjacent virtual nodes are valid (see Steps 5–6 in Figure 2.11). To compute the log-likelihood of the tree, we need to compute an additional "root" CLV (not shown in Figure 2.11), corresponding to the virtual root of the tree. Since the virtual root node `r` is identical with `u`, the root CLV is computed using an equation analogous to Equation (2.9), by setting $b_{ru} = 0$ (assuming DNA sequences):

$$CLV_{i,q}^{(r)} = \sum_{q_1 \in \{A,C,G,T\}} \sum_{q_2 \in \{A,C,G,T\}} CLV_{i,q_1}^{(u)} \cdot P_{q,q_2}(b_{rv}) \cdot CLV_{i,q_2}^{(v)}$$

The overall log-likelihood of the tree is computed according to Equations (2.10)–(2.11). Evidently, when using models that accommodate Γ rate heterogeneity, the CLV update follows the rates-informed Equation (2.14).

The CLV computation described so far corresponds to what is referred to as *full* CLV update in `coraxlib`. This procedure is necessary when RAxML-NG populates the CLV entries for the first time, for example when computing the CLVs and the log-likelihood score of a starting tree. In practice, however, RAxML-NG rarely needs to recompute all CLVs. Instead, it performs the so-called *incremental* CLV updates, an algorithmic optimization that comes at no additional cost, and substantially accelerates tree inference. Incremental CLV updates are crucial, since full CLV updates on every encountered topology would be computationally intractable and unnecessary. For example, in an SPR move, only a small subset of CLVs must be recomputed to evaluate the log-likelihood of the SPR topology. The specific CLVs that require update are directly identified through the value of the `clv_valid` variable of the respective virtual node on the updated tree. Figure 2.12 illustrates how RAxML-NG benefits from this incremental CLV update in the context of an SPR move.

Incremental CLV updates have broad applicability throughout the tree inference process. For instance, they are performed when re-rooting the tree to update the CLVs with respect to the new root, or during BLOs when the root is placed on the branch to be optimized. In the latter case, the branch length itself may change due to the optimization, but the CLVs on either side of the branch remain unaffected. On the other hand, a characteristic case where incremental CLV updates cannot be applied is during MPOs. Even a change of a single substitution parameter requires recomputing all P-matrices across *all* branches of the tree, which in turn requires a full CLV update to compute the log-likelihood of the tree under the new parameter value.

RAxML-NG v1.x Tree Inference Heuristic

Figure 2.13 provides a schematic representation of the RAxML-NG v1.x tree inference heuristic. The heuristic can be divided into three distinct stages. In each stage, the algorithm performs successive SPR rounds of the same type, while the type of SPR rounds differs across stages. The three stages correspond to the three closed loops of Figure 2.13. The SPR rounds are denoted by the following functions in the workflow:

1. `AUTODETECT FAST SPR`(r_{min}, r_{max}): This function takes as input the minimum (r_{min}) and maximum (r_{max}) SPR radius parameters, which denote the minimum and maximum range—in terms of number of nodes away from the original branch where the subtree was pruned—of regrafting distances of an SPR move on the tree topology (see Section 2.6.4). The initial values of r_{min} and r_{max} parameters are 1 and 5, respectively, while the initial value of the r_{best} parameter is 5. AUTODETECT

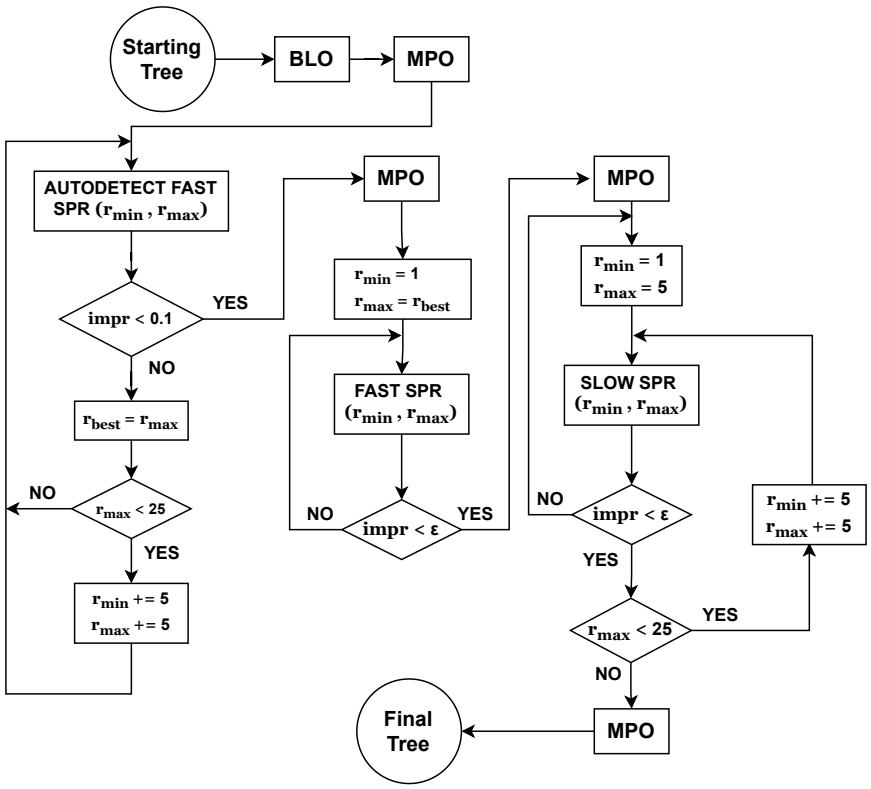


Figure 2.13: A schematic representation of the RAxML-NG v1.x tree inference heuristic (Figure adapted from the Supplementary Material of [211]).

SPR rounds are essentially FAST SPR rounds (see below). The purpose of this initial sequence of SPR rounds is to automatically and adaptively determine the r_{best} parameter for the dataset at hand, which is used by the subsequent FAST SPR rounds as the maximum regrafting distance. The algorithm successively increases the minimum and maximum radius parameters by 5 units after each SPR round, and the sequence terminates when the log-likelihood improvement (in Figure 2.13 denoted as `impr`) is less than 0.1 log-likelihood units, or when the global maximum radius setting of 25 units has been reached.

2. `FAST SPR`(r_{min}, r_{max}): This function takes as input the minimum regrafting distance, which is always set to 1, and the maximum regrafting distance as given by the r_{best} parameter and determined via the preceding AUTODETECT FAST SPR rounds sequence. During FAST SPR rounds, RAxML-NG evaluates each tree topology generated by an SPR move (i.e., each subtree insertion) using the existing branch lengths. The insertion node is placed in the middle of the regrafting branch, with the total length being unchanged. The sequence of consecutive FAST SPR rounds terminates when the log-likelihood improvement is less than the ε-threshold parameter (discussed below).

3. `SLOW SPR`(r_{min}, r_{max}): In analogy to the preceding SPR rounds, this function also takes as input the minimum and maximum regrafting distances, that are initially set to 1 and 5, respectively. The main difference between FAST and SLOW rounds is that, during slow rounds, RAxML-NG evaluates each tree topology resulting from

an SPR move by re-optimizing the lengths of the three adjacent branches around the insertion node (see Section 2.6.4). If, after an SPR round, the log-likelihood improvement is below the ε-threshold, the minimum and maximum parameters are both increased by 5; otherwise, in the next SPR round the minimum and maximum radii are reset to 1 and 5, respectively. The sequence of SLOW SPR rounds terminates when the log-likelihood improvement is below the ε-threshold for five consecutive SLOW SPR round invocations, with increasing radius intervals (1–5, 6–10, 11–15, 16–20, 21–25).

After each SPR round, RAxML-NG conducts a full BLO on the output tree. For simplicity, this post-SPR BLO step is not shown in Figure 2.13, but we consider it as being an implicit part of the SPR round block in the diagram. Another detail omitted from Figure 2.13 is that, within each SPR round, and after all subtrees have been pruned and regrafted once, a BLO is conducted on the X best-scoring trees encountered during the round. The algorithm then compares the log-likelihoods of these candidate trees with that of the currently best tree, replacing the latter if a higher score is found. In AUTODETECT SPR rounds, RAxML-NG deactivates this feature ($X := 0$), while in FAST and SLOW SPR rounds it sets $X := 20$.

Multiple Starting Trees

Evidently, the heuristic shown in Figure 2.13 corresponds to a single ML tree search. This means that the algorithm begins with a starting tree and operates on the same tree-object throughout the entire process, by either optimizing the continuous parameters of the (statistical) model, or by conducting SPR moves to identify topologies with higher likelihoods. The succession of all improved topologies, that is, those visited and accepted by RAxML-NG because they yield higher log-likelihood scores, designates the tree search path. This path, however, constitutes only a single (and potentially biased) perspective of the geometry of the log-likelihood surface. Ideally, since ML tree inference is $\mathcal{NP}$-hard and evaluating all possible topologies is prohibitive, it is desirable to conduct as many independent tree inferences as possible. This pluralism of ML inferences offers several benefits, such as: (a) selecting and printing the best-found ML tree among the ML local optima identified across the independent searches, thereby increasing the chances of approaching the *global* optimum; (b) statistically comparing the ML local optima against the best-found ML tree to identify equally *plausible* solutions (see Section 2.7); (c) critically assessing the topological variation observed among plausible evolutionary scenarios; and (d) investigating how different starting tree types (e.g., random vs. parsimony) affect the plausibility of the respective inferred ML tree.

A correlative concept to point (c) is what the literature refers to as *phylogenetic signal* [18, 73, 85, 103, 133, 154, 210, 212], a distinctive *quale* of the given MSA, reflecting the intensity of its latent evolutionary affinities. Broadly, it describes the degree to which the input data support a robust and/or consistent phylogenetic analysis. It is a generic term, because it can be interpreted from different perspectives. For example, strong phylogenetic signal may imply clear algorithmic convergence of independent ML tree inferences into a single peak in the log-likelihood space, with little or no topological variation being observed among the inferred evolutionary scenarios [73, 210]. It may also indicate that the data resolve inner nodes effectively (i.e., few or no polytomies[1] [212]),

[1] ML tools typically collapse very short branches, yielding multifurcating nodes, or polytomies.

or that inner branches exhibit high bootstrap support [51]. The former behaviors are closely related indicators for sufficient signal with respect to analytical robustness. On the other hand, with regard to consistency, the term phylogenetic signal is used to describe statistical dependency between specimen's traits (e.g., phenotypic, ecological, behavioral) and their phylogenetic proximity, relative to a presupposed evolutionary history [18, 103, 154]. Here, we adopt the former interpretation, that is, the ability of independent ML tree inferences to converge toward a single likelihood peak. Since the true tree is unknown, algorithmic convergence remains our most reliable indicator of phylogenetic robustness.

RAxML-NG v1.x conduct 20 independent tree inferences by default, 10 initiated on random, and 10 on parsimony starting trees. Users can also specify their preferred number of starting trees. The rationale in RAxML-NG v1.x is to address point (a) above. Namely, the tool conducts 10+10 independent ML tree inferences to increase the probability of identifying the global optimum. This multiplicity of searches, however, coupled with subsequent post-analyses (e.g., bootstrap support estimation, plausibility assessment, etc.) [133], has provided a diligent introspection of the geometry of the log-likelihood surface across MSAs with diverse characteristics, such as the number of sites, taxa, patterns, and site entropies [175]. The cumulative empirical observations reported over the years led Haag *et al.* [73] to define the *difficulty* (or *Pythia*) *score* of an MSA, as a measure to quantify the phylogenetic signal contained in the data. One of my main contributions in this thesis is to effectively reduce the number of independent tree searches (i.e., starting trees) performed by RAxML-NG, based on the difficulty score *predicted* by the Pythia tool (see Section 2.8).

In the following, we refer to local ML optima inferred by independent ML tree inferences as ML trees. The best out of them is referred to as the best-found ML tree, and is the evolutionary scenario that RAxML-NG gives as output to the user.

Subtree Cutoff

A shortcut that RAxML-NG applies to further reduce the number of topologies evaluated during an SPR round, with the rationale to omit evaluating topologies which are likely suboptimal, is the so-called *subtree cutoff* feature [197]. When RAxML-NG prunes a subtree during an SPR round and evaluates its possible reinsertions into neighboring branches, a subset of the resulting SPR-neighbors will inevitably yield lower log-likelihood scores. If this is the case for a specific regrafting branch R, and if the difference between the original log-likelihood L (before pruning) and the regrafting log-likelihood $\hat{L}$ exceeds a certain threshold ε_{cutoff}, RAxML-NG entirely omits evaluating possible reinsertion points within the subtree descending from the branch R, i.e., it will not descend beyond branch R.

The ε_{cutoff}-threshold is dynamically defined. It begins with an initial value ε^0_{cutoff}. Let ε^n_{cutoff} denote the cutoff threshold at the n-th SPR round. During the next $n+1$-th SPR round, RAxML-NG records all log-likelihood differences $(L-\hat{L}^{(i)}), i = 1, 2, \ldots, m$ for all m alternative rearranged topologies where $\hat{L}^{(i)} < L$. As mentioned, if for a regrafting topology i' it holds that $L - \hat{L}^{(i')} \geq \varepsilon^n_{cutoff}$, the reinsertion points within the subtree descending from the regrafting branch are entirely omitted. For the next iteration, the subtree cutoff threshold is updated as:

$$\varepsilon^{n+1}_{cutoff} = \frac{\sum_{i=1}^{m} L - \hat{L}^{(i)}}{m}$$

RAxML sets the initial value of the cutoff threshold to be $\varepsilon^0_{cutoff} = \infty$, meaning that during the first SPR round it evaluates all possible reinsertions. In RAxML-NG, the initial value is set to $\varepsilon^0_{cutoff} = -L_0 \ / \ 1,000$, where L_0 denotes the initial log-likelihood of the tree before the first FAST SPR round. I emphasize "FAST", because in AUTODETECT SPR rounds the subtree cutoff feature is deactivated. Further, there is no cutoff reset between FAST and SLOW SPR rounds in RAxML-NG. Thus, the first SLOW SPR round inherits the threshold value determined during the final FAST SPR round.

Convergence Threshold (ε)

In Figure 2.13 shows that the log-likelihood improvement (`impr`) induced by FAST and SLOW SPR rounds is assessed via an ε parameter. This comparison is illustrated in the corresponding decision blocks (rhombus shapes), where the algorithm checks whether `impr` $< \varepsilon$. The ε parameter serves as the convergence threshold in RAxML-NG and is one of the decisive criteria for terminating the tree search. However, this is not the only terminating factor. When the log-likelihood improvement from an SPR round falls below the ε-threshold, the algorithm does not instantly terminate. Instead, RAxML-NG continues the tree search by adjusting SPR-specific parameters for the subsequent SPR rounds. For example, it increases the minimum/maximum SPR radii, or switches from FAST to SLOW SPRs, after conducting an MPO. Nevertheless, the ε-threshold triggers adjustments which indicate that the algorithm is approaching convergence.

In RAxML-NG v1.1, the default convergence threshold is $\varepsilon := 0.1$, a rather stringent setting. This default value was relaxed in v1.2 to $\varepsilon := 10$, owing to the results of a systematic study [74], which showed that the original value was unnecessarily strict. In Chapter 4, I introduce a method to dynamically determine this ε-threshold in RAxML-NG. Instead of relying on a fixed value, my approach statistically evaluates the pre- and post-SPR trees of each SPR round of the tree search, and derives an ε value that reflects inference-specific convergence dynamics.

Pattern Compression and Site Repeats

Additional algorithmic optimizations that RAxML-NG accommodates (similar to the incremental CLV updates described in Figure 2.12) are the *pattern compression* and *site repeats* techniques. Pattern compression [54] collapses identical site-patterns in the MSA (see Section 2.2) and assigns frequency counts (weights) to each site of the compressed alignment. The compressed MSA only comprises unique patterns, and RAxML-NG computes the per-site log-likelihoods only once per unique pattern. The overall log-likelihood is, then, obtained as the weighted sum over these per-site log-likelihoods. Extending the pattern compression logic, the site repeats technique further exploits repeating subpatterns across subtrees to avoid recomputing identical CLVs [108]. Both pattern compression and site repeats optimizations substantially reduce runtime and memory usage (exact numbers are not reported here), without affecting numerical calculations or results.

Vectorization and Parallelization

RAxML-NG employs vectorization to accelerate likelihood computations at the instruction level. By using SIMD extensions such as AVX/AVX2, it computes multiple per-site likelihoods via a single processor instruction [110].

RAxML-NG supports two levels of parallelization on shared-memory machines. In the *fine-grained* parallelization mode, the alignment sites are distributed across the available CPU cores. Each core independently computes the per-site log-likelihoods for its assigned subset of sites. Synchronization between cores only occurs when the overall log-likelihood of the tree is required. At this point, the partial log-likelihoods computed by each core are aggregated using an all-reduce operation (summation). In fine-grained mode, each core maintains its own object of the tree class, which is an exact copy of the other cores' tree-objects. Hence, during tree searches (e.g., while testing SPR moves), all cores evaluate the same set of candidate topologies and arrive at identical log-likelihood values, since the total likelihood calculations are synchronized and communicated globally. As a result, all cores make the same decision regarding which topological move to accept or reject

In contrast, under *coarse-grained* parallelization, each independent tree inference is assigned to a separate CPU core. RAxML-NG essentially employs a hybrid parallelization scheme, in which the available CPU cores are assigned to groups. Each group of cores/threads performs one independent tree inference at the coarse-grained level, while the cores within a group operate in fine-grained parallelization mode. Determining the optimal division of threads into groups is a non-trivial task and is handled automatically by RAxML-NG using MSA-specific load-balancing algorithms [110, 134].

Finally, RAxML-NG supports parallelization on distributed-memory systems via the Message Passing Interface (MPI). In MPI mode, RAxML-NG employs essentially the same hybrid parallelization scheme as on shared-memory machines, combining both fine- and coarse-grained parallelization. Groups of cores are confined to the same physical node, meaning that if RAxML-NG initiates multiple groups (i.e., performs several independent tree inferences concurrently), each inference runs exclusively on a single machine. However, for very long genomic alignments, fine-grained parallelization is typically far more efficient than its coarse-grained complement. In such cases, RAxML-NG only initializes a single group and executes the independent tree inferences sequentially, while each inference itself is parallelized at the fine-grained level. In this configuration, cores from distinct nodes may collaborate on the same tree inference, and synchronization as well as overall likelihood calculation occurs across multiple machines.

2.6.6 IQ-TREE Heuristic

IQ-TREE exclusively uses NNI moves to navigate through tree space. During an NNI round, IQ-TREE visits and evaluates all NNI-neighboring topologies of the current tree, and sorts them based on their score improvements. Starting from the highest-ranked move, the algorithm sequentially applies all *concordant*[1] NNI moves to the current tree. A set of NNI moves is concordant, if the central branches between any two moves in the set are non-adjacent. When applying those concordant NNIs, the log-likelihoods of intermediate occurring topologies are not computed; only the final topology is properly evaluated including BLO. If the final likelihood exceeds the score of the tree induced by applying only the highest-ranked NNI move on the initial list, the resulting topology is accepted; otherwise, the applied NNI moves are reversed, except for the first move, which guarantees to obtain the highest score improvement. This process is repeated iteratively, until no further NNI moves yield score improvements. This heuristic was initially introduced in [70], and is referred to as the hill-climbing NNI round in the

[1] In the original IQ-TREE paper [144], these are referred to as *compatible* moves. Here, I use the term *concordant* to distinguish it from tree compatibility, as defined in Section 2.1.2.

original IQ-TREE paper [144].

NNI-based heuristics, however, are particularly prone to become trapped in local optima. To overcome this, IQ-TREE employs two additional mechanisms; the candidate tree set (CTS) and the perturbation step. The CTS is a fixed-size population of locally (sub-)optimal trees maintained by the algorithm. By default, the CTS is initialized with 99 parsimony, and one BioNJ [61] tree[1], unless the user provides an alternative set of starting trees. At the beginning of every round, the algorithm randomly draws a tree from the CTS and applies random NNI moves to it (perturbation), to escape from the local optimum. The perturbed tree serves as input for the next hill-climbing NNI round. After each NNI round, both, the CTS, and the best-found ML tree are updated accordingly. The process is repeated and the algorithm terminates only after the best-found topology has remained unchanged for a predefined number of consecutive rounds (100 by default).

2.6.7 FastTree Heuristic

FastTree implements a rather superficial tree search strategy. It is an "approximately maximum-likelihood" [151] alternative to infer phylogenies. It applies multiple shortcuts in both, discrete topological, and continuous parameter optimizations. It begins with an NJ tree. During the first phase, the algorithm conducts minimum-evolution (ME; [163])-based NNI and SPR rounds, similar to FastME [38, 118], albeit using profiles instead of true distances to further accelerate computation. Additional heuristics confine the SPR search space from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$ (where n is the number of taxa), and the total number of ME-based topological optimization rounds to $4\log_2(n)$ NNIs, and at most 2 SPRs, respectively. In the second phase, FastTree performs ML-based NNI rounds that entail shortcuts in likelihood computations and confinements in NNI-space. FastTree also limits the total number of ML-NNI rounds to $2\log_2(n)$. Partial BLOs under ML are conducted sporadically, while only a single full MPO is conducted, between the ME- and ML-based topological optimization rounds. Furthermore, FastTree does not account for gamma-distributed rate heterogeneity during tree search. Instead, it employs a variant of the CAT approximation [150], which assigns each site to a fixed, precomputed category (from up to 20 categories). I write "a variant", because the FastTree CAT approximation substantially differs from the original implementation in [190]. Further, the algorithm offers an option to optimize gamma rates (`-gamma` option), yet this is only applied to the output tree topology, and not during the topological optimization rounds.

2.7 Statistical Tests and Plausibility

A scenario that practitioners commonly encounter in phylogenetic analysis is when they end up with a plethora of ML trees, which correspond either to the results of independent ML tree inferences conducted by a single tool (e.g., RAxML-NG), or to the best-found ML trees across tools (e.g., RAxML-NG vs. IQ-TREE). The question arises how the user should decide which of the inferred trees represent reasonable solutions. Relying on the log-likelihood score *tout court* is inadequate, for several reasons. First, the log-likelihood calculations are implemented differently across tools, and thus one cannot simply compare the printed log-likelihood values reported in the log files. Even if all candidate ML trees

[1]BioNJ is a tool which implements a weighted, biologically-informed version of the Neighbor Joining (NJ; [164]) algorithm.

are re-evaluated—by conducting thorough BLOs and MPOs—using a single tool, the resulting log-likelihoods still cannot be used as an absolute metric for comparison. One reason is because log-likelihood calculations are sensitive to numerical errors [176, 200]. A second and more substantial reason, as I describe in the following Section, is that MSA sequences are inherently noisy; specifically, *sampling bias* is the noise type which concerns us here. If the given MSA comprised a finite number of taxa of infinite length, the optimal tree would converge to the true evolutionary history [161, 220], "unless the assumed substitution model is extremely misspecified" [178]. In practice, however, the sequences are finite, and due to sampling error, the best-found ML tree may appear optimal only incidentally without necessarily being the true tree.

The decision problem is typically resolved by applying likelihood-based statistical tests [64]. These tests operate on the per-site log-likelihoods of the candidate trees under comparison. Ideally, to approximate the underlying, infinite-sample distribution of the per-site log-likelihoods for the true tree, one would apply the non-parametric Felsenstein bootstrap [51]. That is, one would generate B bootstrap MSA replicates, of equal length to the original MSA, by randomly sampling sites with replacement. A reference ML tool (e.g., RAxML-NG) would then evaluate all candidate trees on each bootstrap replicate, conducting thorough BLOs and MPOs. The resulting per-site log-likelihood distributions, after a centering step, would serve to accept or reject the respective null hypotheses of the tests referenced below. In practice, this re-optimization of continuous parameters for each bootstrap replicate and candidate tree is computationally expensive. Therefore, standard implementations employ the resampling estimated log-likelihoods (RELL)-approximation [107]. Under RELL, the continuous parameters are optimized only once per candidate tree for the original MSA. For each bootstrap replicate, the tests re-use the precomputed per-site log-likelihoods as an approximation to the per-replicate optimized distribution, thereby circumventing repeated optimizations.

Statistical tests for ML tree comparison are implemented in phylogenetic tools, such as IQ-TREE [141] and CONSEL [179]. The shared tests in both tools are: the Kishino-Hasegawa (KH) test [106], the Shimodaira-Hasegawa (SH) test [177], their weighted variants (WKH and WSH; [180]), and the approximately unbiased (AU) test [178]. In addition, IQ-TREE reports bootstrap proportion probabilities (bp-RELL) using a confidence interval-based decision rule (see below), and applies the Expected Likelihood Weight (ELW) test [201] in a similar manner. The shared statistical tests—i.e., (weighted-)KH, (weighted-)SH, and the AU—assign a p-value to each candidate tree. The trees with p-values ≥ 0.05 are considered as being *plausible* under the respective test, and are retained in the set of statistically indistinguishable solutions. Conversely, a p-value < 0.05 implies that the candidate tree is *significantly worse*, and is therefore excluded from the *plausible tree set (PTS)*.

An approach to determine statistical significance is to require for a candidate tree to be plausible under *all* statistical tests implemented in IQ-TREE or CONSEL. Morel *et al.* applied this approach in [133] using IQ-TREE, and refer to it as the "conservative" method. I initially adopted this strategy in the benchmarking of Adaptive RAxML-NG [210] described in Chapter 3. In retrospect, it became evident that this approach is rather problematic in two ways. First, it neglects the individual biases of each statistical test, which can substantially distort the results. For example, the KH test was originally designed to compare only two trees [106], and therefore has no correction for multiple testing. It is also prone to selection bias [64, 177, 178, 180], and is theoretically valid only when the two candidate trees are specified *a priori*, rather than being inferred from

the data at hand. The SH test [177] claims to fix those issues. However, a recent benchmark study by Markowski and Susko [128] shows that the SH underperforms in cases of extreme selection bias, and the authors explicitly advocate for the latter test be abandoned. Moreover, the SH test rejects fewer trees than expected when many candidates are compared [64, 178]. The AU test further adjusts the p-value calibration to correct this relaxed behavior, and is considered the most reliable test for comparing multiple ML trees [128, 130, 178].

Second, the "conservative" approach, as applied in [133, 210], is also problematic because it treats the bp-RELL and ELW values reported in the IQ-TREE log as if they were p-values, which they are not. In fact, the bp-RELL and ELW tests in IQ-TREE are substantially different from the KH, SH, and AU tests: the values they infer are posterior weights (resembling posterior probabilities) and should not be interpreted as p-values. Nonetheless, IQ-TREE accepts or rejects candidate trees based on a 95% confidence interval rule. It sorts the candidate trees by their posterior weights (from highest to lowest), computes the cumulative distribution, and rejects all remaining trees once the cumulative probability reaches 95%. Therefore, bp-RELL and ELW are not null hypothesis-based significance tests, but only measures of relative support, and it is rather debatable whether they should be equated with KH, SH, and AU.

Upon that, my lab mates Julia Haag and Dimitri Höhler spotted certain anomalies in IQ-TREE's statistical tests implementation. For instance, a striking observation was that the p-value of a candidate tree is highly sensitive to the order of the input trees. Permuting the input list of candidate trees often yields noticeably different p-values. These observations are documented in the Supplementary Material of [74].

For all of the above reasons, in Chapters 4 and 5 I adopt the *liberal* approach (as opposed to the "conservative" approach mentioned above)[1], that is, I exclusively rely on the AU test as implemented in the CONSEL tool for ML tree plausibility assessment. The AU test resolves the biases of KH and SH, and the CONSEL implementation is empirically stable and robust. Accordingly, I repeat the benchmarking from [210] (see Chapter 3) using the AU test in CONSEL, and report the latter results in juxtaposition to those obtained via IQ-TREE's statistical tests under the "conservative" approach. Finally, in Chapter 4, I describe an efficient implementation of the KH test that I integrated into RAxML-NG to ascertain convergence during the ML inference process and trigger Early Stopping. In this way, RAxML-NG utilizes the KH test as a convergence criterion, thereby achieving substantial speedups.

2.8 The Pythia Tool

As already discussed, heuristics do not guarantee that one will find the globally optimal ML tree. Empirical observations [137, 191] suggest that, on certain datasets, independent ML tree searches converge to a single—or topologically highly similar—tree(s), while on other datasets, they yield multiple topologically highly distinct trees with almost identical likelihood scores [133]. In the latter case, standard phylogenetic significance tests (KH, SH, or AU), usually report no statistically significant difference among the

[1]Here, I implicitly refer to the historical conservatism-liberalism dipole of the early 19th century Germany, where conservatism was the political movement favoring a regression toward a theocratic regime, while liberalism defended individual freedom and the democratic organization of the state. For more context, see [207].

majority of the inferred, highly contradicting, topologies. Thus, "easy" MSAs exhibit a single, well-distinguishable, globally optimal peak on their likelihood surface, associated with a single tree topology. In contrast, "difficult" MSAs exhibit a rugged likelihood surface, with multiple locally optimal and statistically indistinguishable peaks induced by contradicting topologies.

Pythia (developed in our research group by Julia Haag) is a machine learning[1] tool which predicts the difficulty of the phylogenetic analysis of an MSA *prior to* initiating ML-based tree inferences. Following the version naming used in the most recent Pythia preprint [75], we refer to the model version accompanying the initial publication [73] as Pythia v0.0, and to the recently released version as Pythia v2.0. In the Pythia v0.0 publication, the ground-truth inference difficulty of empirical MSAs is calculated by conducting 100 ML tree searches on each dataset using RAxML-NG. The Pythia score (difficulty) is defined as follows:

$$\text{difficulty} = \frac{1}{5} \cdot (RF_{all} + RF_{pl} + \frac{N^*_{all}}{N_{all}} + \frac{N^*_{pl}}{N_{pl}} + (1 - \frac{N_{pl}}{N_{all}})) \tag{2.15}$$

The five terms included in Equation (2.15) are the following, in order of appearance:

- RF_{all}: The average nRF distance between all pairs of trees among the 100 ML trees.
- RF_{pl}: The average nRF distance between all pairs of trees in the PTS[2].
- $\frac{N^*_{all}}{N_{all}}$: The number of unique tree topologies in the output ML tree set (N^*_{all}) divided by the total number of trees in the same set (e.g., given that Haag *et al.* carried out 100 ML tree searches in [73], this parameter would be $N_{all} = 100$).
- $\frac{N^*_{pl}}{N_{pl}}$: The number of unique tree topologies in the PTS (N^*_{pl}) divided by the total number trees in the same set (N_{pl}).
- $\frac{N_{pl}}{N_{all}}$: The number of trees in the PTS (N_{pl}) divided by the total number of trees in the output ML tree set ($N_{all} = 100$). This ratio is subtracted from 1, leading to the full expression of the last term $(1 - \frac{N_{pl}}{N_{all}})$.

Pythia v0.0 uses a Random Forest regressor [22], trained and tested on a subset of MSAs in TreeBASE database [147], with the difficulty scores (computed as described above) serving as ground-truth labels. Each MSA is represented by an 8-dimensional feature vector. Six of these features are simple, fast to compute, and only rely on MSA attributes: the patterns-over-taxa (PoT) ratio, the sites-over-taxa (SoT) ratio, the proportion of gaps, the proportion of invariant sites, the site entropy [175], and the Bollback Multinomial metric [20]. The remaining two features are derived from 100 Maximum Parsimony (MP) inferences, which are orders of magnitude faster than ML searches. From these 100 MP trees, Pythia v0.0 extracts (i) the proportion of unique MP topologies, and (ii) the average nRF distance among all MP trees.

Since its initial publication, Pythia has undergone substantial improvements. The newest version, Pythia v2.0 [75], replaces the Random Forest regressor by a Gradient Boosted Tree model [60, 102]. The training dataset was expanded to include the full TreeBASE database (rather than a subset thereof), and the RAxML Grove database [90]. In addition to the original eight features, Pythia v2.0 employs two additional features: the pattern-over-sites ratio, and the pattern entropy. Furthermore, the number of MP searches used to compute the MP-based features was reduced from 100 to 24, based on the

[1] I assume the reader is already familiar with basic concepts from statistical or machine learning. For a broader overview of those fields, the reader may refer to sources such as [11, 80, 132].

[2] In the original Pythia v0.0 publication, plausibility was established using IQ-TREE tests.

results of an extensive analysis described in [75]. These modifications yielded substantial improvements in prediction accuracy and runtime. A detailed description of the evolution of Pythia versions, along with exact accuracy and speedup metrics, can be found in [76].

The Pythia tool is particularly useful for providing a fast and accurate quantification of the phylogenetic signal present in an MSA. For example, the original Pythia paper [73] reports that a prediction is, on average, 5× faster than a single RAxML-NG ML tree inference. Given that RAxML-NG v1.x conducts 20 independent tree inferences by default (see Section 2.6.5), the Pythia score gives a rapid estimate of how easy or difficult the dataset is to analyze. Moreover, Pythia v2.0 achieves an additional 2.4× average speedup over Pythia v0.0. In Chapter 3, I describe how I use the predicted Pythia score in Adaptive RAxML-NG [210]. Based on the predicted score, I adjust the number of independent ML tree searches as well as the tree search heuristic itself, achieving on average up to 16× speedups on both easy and difficult MSAs, while maintaining accuracy. The Pythia tool has now become an indispensable built-in module of RAxML-NG, as we move toward the release of the second major version of the tool (RAxML-NG v2.0; see Chapter 6).

Finally, the current definition of the Pythia score, as given in Equation (2.15), entails a minor drawback, which I also report in the Supplementary Material of [210]. Each term in Equation (2.15) yields a value between 0.0 and 1.0. Since all five terms are weighted equally, each can contribute at most 0.2 units to the overall score. As we will observe in Chapters 3–6, only a small fraction of MSAs in the studied datasets exceed a Pythia score of 0.9. The main reason for this is the last term in the parentheses, $(1 - \frac{N_{pl}}{N_{all}})$. On easy datasets, almost all independent ML tree searches yield trees with similar topologies and likelihood scores. Therefore, the vast majority of the final ML trees is included in the PTS. In this case, the ratio $\frac{N_{pl}}{N_{all}} \approx 1.0$, and consequently $(1 - \frac{N_{pl}}{N_{all}}) \approx 0.0$. However, the majority of ML trees inferred on difficult MSAs are also plausible. While the topologies are highly incongruent, their likelihood scores are almost equal, and it, again, holds that $(1 - \frac{N_{pl}}{N_{all}}) \approx 0.0$. Hence, on difficult MSAs the last term most likely contributes less than 0.1 units (out of 0.2), yielding an upper "bound" of 0.9 for the overall score which is rarely exceeded. The four remaining terms, however, contribute 0.2 units each, and thus, the problem is somewhat alleviated for difficult MSAs. This happens because, in such cases, both the nRF distance terms and the unique-topology ratios are close to 1.0, implying a high number of distinct and incongruent topologies in the final ML tree set.

Overall, the difficulty score for "hopeless-to-analyze" MSAs is usually underestimated by the current definition in Equation (2.15). Intuitively, the difficulty score of such datasets should be close to 1.0, but since the contribution of the last term in the parenthesis is low (for the reasons outlined above), the calculated value usually lies within the $[0.8, 0.9]$ range. On the other hand, due to the high contribution of the remaining four terms, the difficulty scores of difficult MSAs are rarely below 0.8, rendering the Pythia score in its current state fully functional. Still, a reformulation of the Pythia score could help to distribute the difficult-to-analyze MSAs more uniformly across the upper part of the range $[0.7, 1]$.

2.9 Noise in MSAs

I intentionally distinguish between two basic types of noise in MSAs: (a) noise within the available (non-missing) MSA sequences, and (b) noise due to missing data in partitioned

MSAs. This distinction is important, because each noise type gives rise to a different problem that I address in this thesis.

2.9.1 Noise in Available MSA sequences

MSA sequences are subject to noise [212] stemming from both *stochastic* and *systematic* sources. Evolution, which is stochastic in nature, induces a form of randomness in the sequences, reflecting deviations between observed data distributions and theoretical expectations [86, 140]. Upon that, sampling noise is superimposed [230], as phylogenetic analyses typically rely on sequences that represent only a small fraction of the corresponding genomes, raising concerns about their adequacy in approximating the original distribution. Systematic noise, that is more prevalent in empirical MSAs, is introduced by factors such as sequencing errors [138], alignment errors [42], homoplasy [158], and gene tree-species tree discordance in supermatrix MSAs [124]. Because of this intrinsic noise in sequence data, there is a risk that thorough ML optimization may lead the inferred statistical model to overfit, capturing both the phylogenetic signal and the noise, as discussed in Section 2.10.

2.9.2 Missing Data in Partitioned MSAs

Partitioned (or multi-locus) datasets often exhibit patches of missing data. That is, a species may have no data present at a specific locus, either due to sampling issues, or because the target locus is simply absent from the species' genome. As I describe in Section 2.11, the pattern and amount of such missing entries may induce two adverse phenomena in phylogenetic tree space, *stands* and *terraces*, which both complicate the identification of a clear, optimal (local or global) peak. Missing data are rather common in multi-locus empirical datasets. For example, in RAxML Grove v0.7 database [90], we counted $7,295$ empirical, partitioned multi-gene datasets, $4,959$ (68%) of which had a non-zero proportion of missing data and $1,390$ (19%) a missing data proportion exceeding 30%.

2.10 Statistical Learning and Overfitting

In statistical learning the term *training* refers to the process of estimating the parameters of a statistical model, such that they optimally fit the input data according to a given criterion/loss-function. For example, in deep neural networks (DNNs), training typically involves iterative methods [162], which efficiently adjust the DNN weights (parameters) to progressively minimize prediction error. The input dataset is typically split into training and testing data, with a larger portion assigned to training (e.g., 80%–20% split). The training set is used to optimize model parameters (via training), while the testing set is used for model evaluation.

Overfitting occurs when the model being trained does not only capture the underlying pattern, but also the inherent noise in the training data [62, 80, 132]. It manifests itself as a degradation in the model-fit on the testing data after a certain point during the iterative training process. Moreover, the testing data model-fit may deteriorate progressively with every iteration, as the noise-fit iteratively increases on the training data. Overfitting is often a consequence of *over-training* an *over-parameterized* model, relative to the size or the intrinsic properties of the training data [17, 87]. A numerical example of

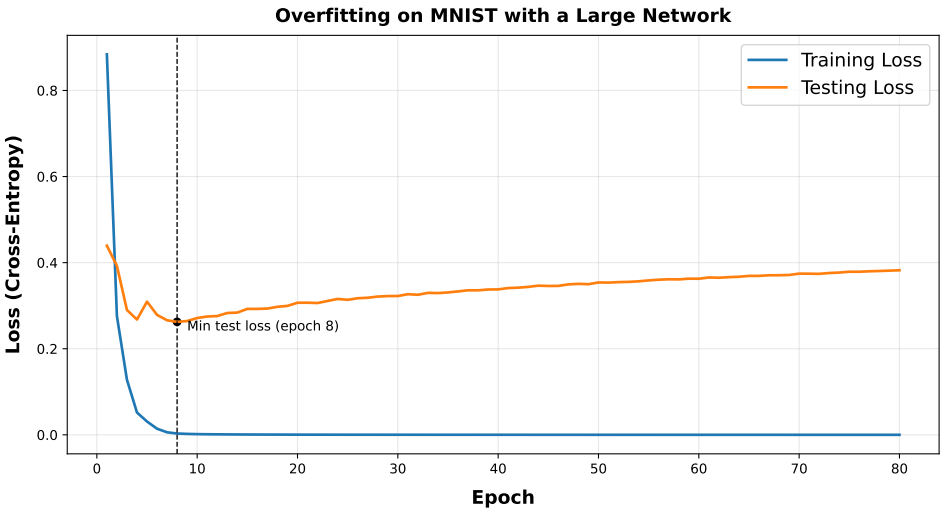


Figure 2.14: Numerical example of overfitting emerging during the training of a parameter-rich DNN. The plot corresponds to empirically derived learning curves obtained from training a DNN on a small, random subsample (1,600 and 400 data points for training and testing, respectively) of the MNIST dataset [117]. The DNN has four fully connected layers with widths of 784, 1024, 1024, and 10. I train the DNN via the Adam optimizer [105], yet disable any form of regularization or early stopping. After iteration/epoch 8, the testing loss function progressively increases and diverges from the training one, which continues to decrease and eventually attains zero loss (Figure reproduced from [209]).

overfitting is shown in Figure 2.14. In this example, the DNN is trained to minimize the cross-entropy loss function. Initially, both training and testing loss decrease; however, after iteration 8, the testing loss begins to progressively increase and diverges from the training loss, which continues to decrease and eventually reaches zero. Common strategies to mitigate overfitting in deep learning include holdout-based early stopping [186], or penalizing redundant or excessively large parameters [88]. It should be noted, however, that extensive model training by pushing the optimization to its limits, despite strong indications of convergence in the training score/error, does not necessarily induce overfitting [12, 234].

In Chapter 5, I draw an analogy between standard training procedures in statistical learning and the heuristics that ML inference tools deploy. Both approaches aim to optimize the parameters of statistical models, and the optimization procedures can be decomposed into distinct iterations. This perspective allows to systematically investigate overfitting in ML tree inference tools. Accordingly, I provide an answer to an important question which has been insufficiently examined in current literature: whether standard ML tools (especially the thorough ones) experience overfitting [209].

2.11 Stands and Terraces

In partitioned MSAs, there exist two alternative approaches to reconstruct phylogenies, the so-called supermatrix (see Section 2.5.4) and supertree methods. In the first case, per-gene MSAs are assembled/concatenated into a large supermatrix that is typically divided into disjoint partitions, that often represent the distinct genes/loci. The phylogenetic tree

is then inferred from this supermatrix. Each gene is typically assumed to evolve under its own model of evolution, while all genes share the same underlying tree topology. Second, in supertree methods, separate gene trees are initially inferred for each gene and subsequently reconciled into a species tree by maximizing appropriate criteria, such as the reconciliation likelihood [135, 136] or the quartet-consistency score [152, 232].

Multi-locus datasets typically exhibit patches of missing data (see Section 2.9.2). The presence or absence of data can be captured by a *presence-absence matrix (PAM)*. Its elements are 1's and 0's, indicating the presence or absence of data for a specific locus/species pair. The pattern and amount of missing data in a PAM can lead to the so-called *stands*, that is., collections of all species trees that are compatible with a set of corresponding induced per locus subtrees. Specifically, let Y be the full set of species/taxon labels in the PAM, and $Y_i \subseteq Y$ be the subset of available taxa for locus i. Hence, $Y = \bigcup_{i=1}^{p} Y_i$, where p is the number of loci. Further, let T_i be a tree on Y_i, which may be inferred, for example, via ML methods. A stand is defined as the set of all trees on Y that are *compatible* (see Section 2.1.2) with all trees T_i, each one defined on Y_i for locus i. Hence, for any tree T on the stand, it holds that $T|Y_i = T_i$ for all $i = 1, 2, \ldots, p$, that is, T displays T_i.

Under many scoring criteria typically used in phylogenetics (e.g. likelihood, parsimony in supermatrix, or quartet-consistency score in supertree approaches), all trees on the stand have identical scores. In such cases, a stand is called a *terrace*. For example, when evaluating the trees on a stand using the log-likelihood score with unlinked branch lengths (see Section 2.5.4), all stand trees yield identical scores and thus form a terrace [166]. The concept of terraces was first used implicitly in [196], and was explicitly described in [166, 167] for supermatrix, and in [48, 77] for supertree approaches.

Note that, it is not a necessary condition for trees to belong to a stand in order to form a terrace. In fact, equally scoring trees may occur even for complete datasets with no missing data. This phenomenon has been documented in the literature for discrete scoring functions/criteria, such as parsimony and supertree methods [48, 65, 77, 129, 215]. Moreover, although distinct ML trees on complete datasets (with no missing entries) do not have analytically identical log-likelihood scores, in practice, due to constraints in numerical calculations, they may appear to have exactly the same (or only marginally different) log-likelihood scores [23, 73, 210]. In such cases, the practitioner has little control over the scoring landscape generated by the input MSA, and therefore in the robustness of the final result. By contrast, in the case of phylogenetic terraces induced by stands and caused by missing data, the user can reduce their adverse effect on inferences, for example by adding/removing species or loci to the dataset.

Therefore, identifying stands and determining their sizes is crucial for a robust phylogenetic analysis. Chernomor *et al.* developed Gentrius [28], a branch-and-bound algorithm that enumerates all stand trees given a set of unrooted incomplete locus trees. However, the computational complexity of building stands for unrooted trees is intractable [21], and the number of trees in a stand can be exponential with respect to the number of input taxa [19]. For this reason, I developed an efficient parallelization of the Gentrius algorithm [208], which I describe in Chapter 7.

3. Adaptive RAxML-NG

This Chapter is based on the following peer-reviewed article:

- **Anastasis Togkousidis**, Oleksiy M. Kozlov, Julia Haag, Dimitri Höhler, and Alexandros Stamatakis. "Adaptive RAxML-NG: Accelerating Phylogenetic Inference under Maximum Likelihood using Dataset Difficulty." *Molecular Biology and Evolution*, Volume 40, Issue 10, October 2023. `https://doi.org/10.1093/molbev/msad227`

Contributions: Anastasis Togkousidis designed the Adaptive RAxML-NG heuristic, integrated it into the RAxML-NG software, and conducted the performance evaluation. Julia Haag developed the Pythia predictor, which Adaptive RAxML-NG directly uses. Dimitri Höhler generated the simulated datasets used in the performance evaluation. Oleksiy Kozlov and Alexandros Stamatakis supervised the project and helped with the writing of the paper.

In Chapter 2, I described the concept of the phylogenetic signal in an MSA. I also introduced the Pythia tool [73], which is a machine learning-based predictor that estimates the difficulty of analyzing a given MSA *prior* to an ML tree inference. The estimated difficulty score is a real number which ranges between 0.0 to 1.0, and reflects the ruggedness of the phylogenetic tree space. Easy MSAs, with a difficulty score close to 0.0, exhibit a single, potentially globally optimal peak on their likelihood surface, while difficult ones, with a score close to 1.0, exhibit multiple locally optimal peaks. The difficulty score does not only capture these two extreme cases, but also the entire spectrum of intermediate cases. Hence, one can roughly classify MSAs as being easy, intermediate, and hard-to-analyze. In this Chapter, I present how I utilize the predicted MSA difficulty score to devise an Adaptive heuristic for ML tree inference, which I integrate into RAxML-NG [210].

3.1 Related Work

Properly evaluating and comparing different heuristics/tools for ML-based phylogenetic inference constitutes a challenging task when introducing novel heuristics. According to a comparative study conducted by fellow lab member Dimitri Höhler [89], there is no standard benchmark dataset for assessing/comparing tools. Most benchmark studies typically rely on *ad hoc* sampled dataset collections. These collections are sometimes biased towards specific MSA properties, which might distort the overall picture and even yield contradictory results among studies (see below). In this section, I examine some major issues reported in Höhler *et al.*, as well as in previous benchmark studies. I outline the connection between the Pythia score of an MSA on the one hand, and the respective

convergence speed of the ML heuristic on the other, as well as the accuracy and robustness of the result.

FastTree [151] is one of the fastest and most widely used tools for ML inference; yet, its heuristic is considered as being rather superficial. The original FastTree publication [151] reports speedups of 100–1,000× relative to PhyML [68] and RAxML [193], but the trees inferred by the latter tools are substantially more accurate, due to the thoroughness of the tree search heuristics they deploy. The suboptimal performance of FastTree was later confirmed by an independent study [235], in which the authors found that RAxML, IQ-TREE [144], and PhyML perform similarly in terms of topological accuracy on single-gene datasets, while FastTree performs substantially worse. In contrast to the aforementioned studies, the results from a third comparative study between RAxML and FastTree only [123], show that, although RAxML generally infers more accurate topologies than FastTree, the differences diminish as the sites-over-taxa (SoT) ratio decreases. This observation is confirmed by Höhler *et al.*

A similar pattern was described by Morel *et al.* [133], in the context of analyzing SARS-CoV-2 data. Due to the comparatively low nucleotide substitution rate of SARS-CoV-2 [216], the four MSAs analyzed in [133] are characterized by a high proportion of invariable sites, and the patterns-over-taxa (PoT) ratio is ≤ 1. Morel *et al.* conducted 100 ML tree searches using RAxML-NG and extracted the plausible tree set (PTS). In all four MSAs, each independent search yielded a unique topology, and more than 70% of those 100 trees were plausible. Further, the average normalized RF distance between all pairs of 100 ML trees was $\sim$0.78, indicating the prevalence of topologically highly distinct trees.

The seemingly inconsistent conclusions in the aforementioned studies can be reconciled by reflecting on the concept of the difficulty score. Starting from the latter study, the 100 output ML trees with contradicting topologies imply an extremely rugged likelihood surface that exhibits a plethora of local optima. This hypothesis is confirmed by calculating the difficulty score on the full SARS-CoV-2 dataset; with a score of 0.84 (estimated by Pythia v0.0), the dataset is virtually "hopeless-to-analyze". In other words, analyzing such a dataset is not expected to yield a single, well-defined, strictly bifurcating tree, that is, a clear peak. Additional indicators of high difficulty include low SoT and PoT ratios, both of which are used by Pythia as prediction features. First, Rosenberg and Kumar [159] showed that higher SoT ratios generally imply stronger phylogenetic signal and thus more accurate inference. This observation provides a sufficient explanation for the results of the aforementioned study by Liu *et al.* [123]: as the number of taxa grows, the SoT ratio of the MSA decreases, and consequently its Pythia score increases as well. Hence, both RAxML and FastTree infer (potentially) distinct locally optimal trees with similar likelihood scores.

Further, Höhler *et al.* [89] establish a strong negative correlation between the PoT ratio and the difficulty of empirical MSAs. They show that the difficulty decreases substantially when the PoT ratio exceeds 10. This provides an additional explanation for the high difficulty of the SARS-CoV-2 dataset, where the PoT ratio is close to 1.0. In the same study, RAxML-NG and IQ-TREE perform substantially better than FastTree in terms of topological accuracy on empirical datasets with difficulty scores ranging between 0.2 and 0.6 (easy-to-intermediate), which is in agreement with the results of earlier comparative studies [151, 235]. For easy datasets (difficulty < 0.2), FastTree performs similarly in terms of likelihood score and topological accuracy.

3.2 Method

3.2.1 Empirical Observations

The initial dataset collection comprised 10,389 empirical MSAs from TreeBASE [147] and 5,000 simulated MSAs (see Section 3.3.1) with varying degree of difficulty. I divided datasets into three classes: (a) *easy* datasets with a difficulty score below 0.3, (b) *intermediate* datasets with a difficulty between 0.3 and 0.7, and (c) *difficult* datasets with a difficulty exceeding 0.7. The development of the Adaptive RAxML-NG tree search heuristic was based upon three observations:

1. The majority of independent ML tree searches on easy MSAs either converges to a single, or topologically highly similar, tree(s). Moreover, fast and thorough heuristics (FastTree and standard RAxML-NG) perform similarly in terms of topological accuracy and likelihood score, especially when the difficulty score is close to 0.0.

2. Independent ML tree searches on difficult MSAs yield topologically highly distinct trees, with most of them being equally likely and, therefore, statistically indistinguishable. Fast and thorough heuristics perform similarly in this difficulty range with respect to the likelihood score. Overanalyzing such datasets is hopeless. Thus, it suffices to rapidly infer only a few, out of the many almost equally likely, topologies, to reduce overall execution time.

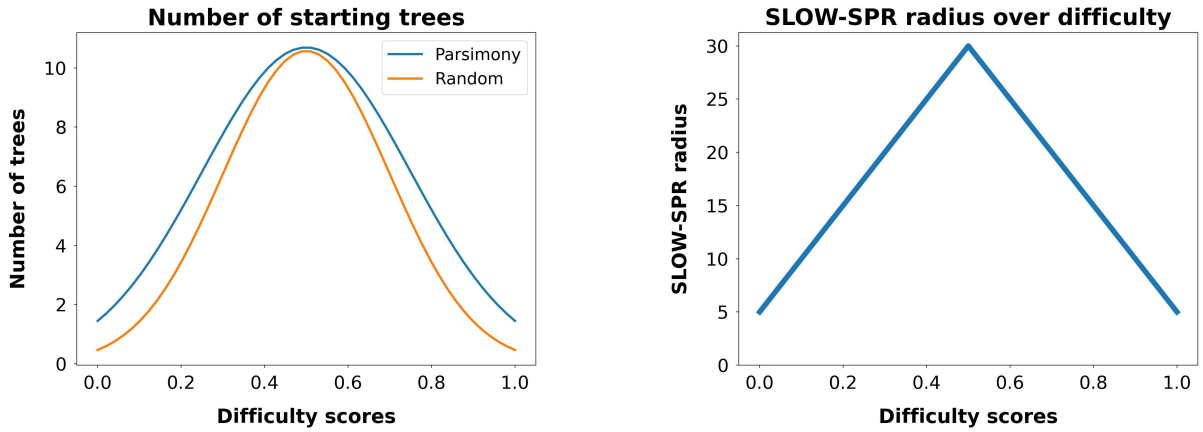
3. On datasets of intermediate difficulty, thorough heuristics yield statistically better ML trees than superficial ones. This is because superficial heuristics are more prone to become stuck in local optima which correspond to suboptimal trees. In general, intermediate MSAs exhibit fewer peaks on their likelihood surface than the difficult ones. However, only a small proportion of these peaks is significantly better and, therefore, more thorough search strategies yield significantly better trees.

3.2.2 The Adaptive Heuristic

Pythia was originally implemented in Python [98], and was later integrated into the RAxML-NG framework as a `C` module [99] within `coraxlib` [47]. Adaptive RAxML-NG begins by invoking Pythia, which predicts the difficulty score of the input MSA.

Next, the algorithm determines the required thoroughness of the tree space exploration via two factors: the number of independent tree inferences that the tool initiates, and the thoroughness of each individual search. Regarding the former, RAxML-NG v1.x by default executes ML tree inferences on 10 random and 10 parsimony starting trees. Adaptive RAxML-NG, on the other hand, modifies the number of independent tree inferences based on the difficulty score of the input MSA. The functions to determine the number of random/parsimony starting trees, given the input MSA difficulty score, are shown in Figure 3.1a. Both functions are Gaussian curves with a peak of 10 trees when the difficulty score is 0.5. The curve for the number of parsimony trees is intentionally wider, such that the heuristic uses more parsimony than random starting trees on easy and difficult datasets, owing to an observation made by Morel *et al.* [133] on the SARS-CoV-2 dataset. The authors observed that tree searches initiated on parsimony starting trees consistently yielded phylogenies with substantially better log-likelihood scores.

Regarding the second factor, Adaptive RAxML-NG modifies the thoroughness of each individual search by adjusting the maximum SLOW SPR regrafting radius, that is, the



(a) (b)

Figure 3.1: (a) Number of random and parsimony starting trees that are used by the Adaptive RAxML-NG to conduct independent tree inferences, as a function of the difficulty score. **(b) The maximum SLOW SPR regrafting radius** parameter over the difficulty score (Figure reproduced from [210]).

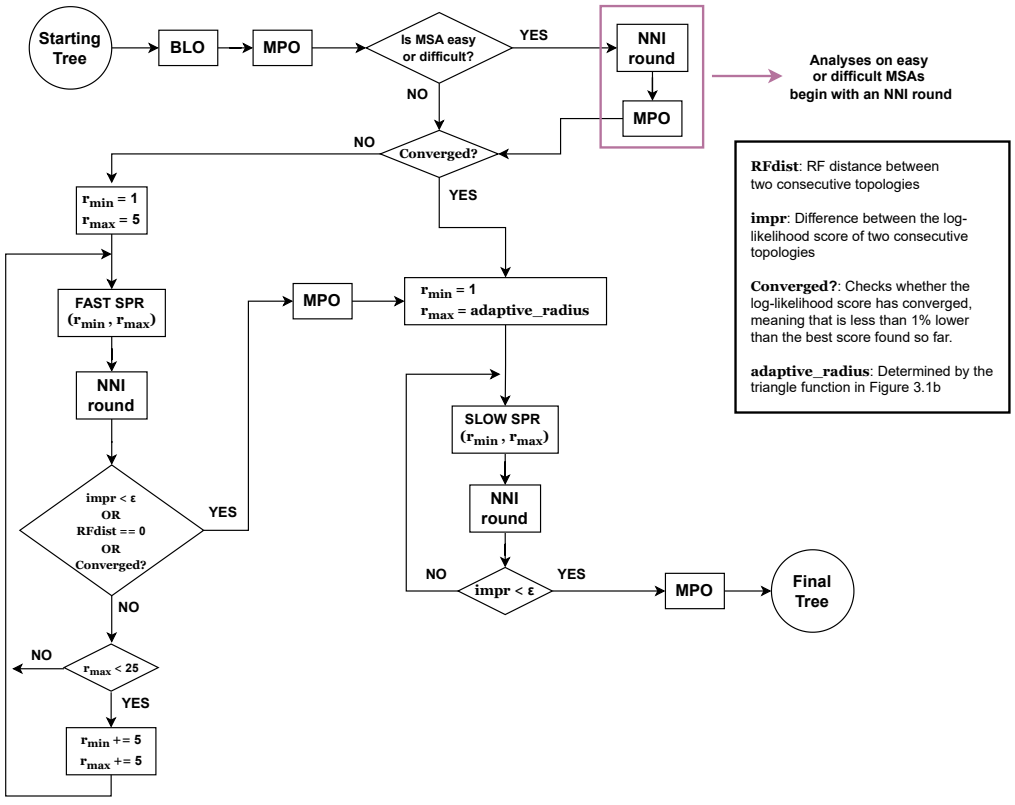


Figure 3.2: A schematic representation of the Adaptive RAxML-NG tree inference heuristic (Figure adapted from [210]).

maximum distance between the pruning and regrafting points of an SPR move. This parameter directly affects the number of alternative topologies that are explored per SPR round (see Section 2.6.4). Figure 3.1b shows the function to determine the maximum SLOW SPR regrafting radius given the input MSA's difficulty. A triangle function is used, which starts from a radius of 5 when the difficulty score is either 0.0 or 1.0, and attains a peak radius of 30 for a difficulty of 0.5.

Figure 3.2 provides a schematic representation of the tree search heuristic that Adaptive RAxML-NG deploys. In contrast to standard RAxML-NG v1.x, which exclusively searches the tree space via SPR moves, Adaptive RAxML-NG also employs NNI moves in addition to SPRs. I provide an efficient NNI round implementation in `coraxlib`, which is analogous to the NNI round implemented in RAxML v8.2 (`-f J` option; [195]). The rationale for using NNIs is that, on easy and difficult MSAs, inferences tend to converge rapidly, even when applying superficial heuristics (e.g., FastTree). Evidently, NNI-based heuristics are less thorough than SPR-based ones[1]. Hence, as shown in Figure 3.2, on easy- and difficult-to-analyze MSAs, Adaptive RAxML-NG begins with an NNI round followed by an MPO, as the probability to find a local ML peak only via this NNI round is comparatively high.

After the initial BLO and MPO conducted on the starting tree, as well as the NNI round for easy and difficult MSAs, Adaptive RAxML-NG executes two phases. Before proceeding, though, the algorithm evaluates whether the current-best tree has attained what I refer to as "first-stage convergence": that is, the score of the currently-best tree is less than 1% worse than the score of the best ML tree inferred so far from previous, already completed inferences (if any). In case no ML inference has been conducted yet, the first-stage convergence check returns `FALSE`. This check is conducted for all MSAs, that is, not only for the easy and difficult ones where the algorithm begins with an NNI round, but also for datasets of intermediate difficulty, where only BLO and MPO rounds are performed on the starting tree. This is because, it may be the case that the initial parsimony starting trees have log-likelihood scores within 1% of the best score inferred so far. In such cases, the algorithm skips the first, FAST-SPR phase entirely, and directly proceeds to the second, SLOW-SPR phase (see Figure 3.2).

In both phases, SPR rounds are alternating with NNI ones. During the first phase, FAST SPR rounds alternate with NNI rounds, until either of the three conditions is met: (a) the RF distance between two consecutive tree topologies is zero[2], (b) the log-likelihood improvement between two consecutive topologies does not exceed a fixed threshold $\varepsilon := 0.1$ (the same ε-threshold as in standard RAxML-NG v1.1), or (c) the first phase has attained "first-stage convergence" (see above).

In the second phase, SLOW SPR rounds alternate with NNIs until the log-likelihood improvement ε-threshold is reached again. Further, as in the standard heuristic (illustrated in Figure 2.13), Adaptive RAxML-NG conducts full BLOs on the starting tree, and after each FAST/SLOW SPR round. The post-SPR BLO rounds are not displayed in Figure 3.2, but we consider them as being an implicit part of the SPR round blocks. MPO is conducted on the starting tree (after the BLO), in-between the first and second phase, and on the final tree topology. On easy and difficult MSAs, an extra MPO is conducted after the additional NNI round invoked at the beginning.

[1] For a discussion on NNI- vs. SPR-based heuristics, see Sections 2.6.2 and 2.6.4.

[2] In this context, two tree topologies are consecutive if they are separated by exactly one SPR plus one NNI round.

Parameter Configuraion

The initial values of the r_{min} and r_{max} parameters, that is, the minimum and maximum SPR regrafting radii, are 1 and 5, respectively. Both radii are increased by 5 units at each invocation of the FAST SPR round until r_{max} reaches its upper limit of 25. Once this limit is reached, and if the algorithm remains in the first stage, the radii stay fixed at $r_{min} := 21$ and $r_{max} := 25$. During FAST SPR rounds, the subtree cutoff feature (see Section 2.6.5) is disabled. In addition, FAST SPR rounds do not maintain a list of the X best-scoring topologies encountered during the search; instead, they set $X := 0$ (see Section 2.6.5).

In SLOW SPR rounds, the values of r_{min} and r_{max} remain constant. The minimum regrafting radius is always set to $r_{min} := 1$, whereas r_{max} is determined by the triangular function shown in Figure 3.1b. The subtree cutoff feature is enabled, with a default initial value $\varepsilon^0_{cutoff} = -L_0 \ / \ 1,000$, similar to RAxML-NG v1.x. The number of X best scoring topologies stored during the SLOW SPR round—on which a full BLO is conducted at the end of the SLOW SPR round in order to be compared against the current best tree—is set to $X := 20$. Finally, the NNI tolerance parameter is set to $\varepsilon_{NNI} := 0.1$ (see Section 2.6.4), and the log-likelihood convergence ε-threshold is set to $\varepsilon := 0.1$, following the settings used in RAxML-NG v1.1, that is, the most up-to-date RAxML-NG version at the time of developing the Adaptive heuristic.

3.3 Experimental Setup

3.3.1 Benchmark Pipeline

In this benchmark, I compare Adaptive RAxML-NG against standard RAxML-NG v1.1. For each empirical/simulated MSA (see Section 3.3.2), both versions are executed using their default settings. This means that each version conducts multiple independent tree inferences. By default, standard RAxML-NG conducts 20 tree searches using 10 random, and 10 parsimony starting trees. The number of independent tree inferences performed by Adaptive RAxML-NG is determined as a function of the input MSA difficulty score (see Figure 3.1a). I compare standard and Adaptive RAxML-NG by explicitly focusing on the best ML trees inferred by each version on each MSA. I refer to the best ML tree inferred via RAxML-NG v1.1 as the *standard tree*, and the one inferred via the Adaptive version as the *adaptive tree*.

In the Adaptive RAxML-NG paper [210], I statistically evaluate the standard and adaptive trees for each MSA using the significance tests implemented in IQ-TREE [131, 141]. I consider an ML tree as being *plausible* if it passes *all* statistical tests implemented in IQ-TREE. Therefore, standard and adaptive trees are statistically indistinguishable only when they both pass *all* tests. As discussed in Section 2.7, this "conservative" approach is rather problematic for a plethora of reasons. Hence, in this Chapter I repeat the comparison between standard and adaptive trees from [210] by adopting a more liberal approach. Specifically, I apply the AU test [178] as implemented in the CONSEL tool [179], and consider ML trees with a p-value ≥ 0.05 as being plausible. Moreover, I require that the absolute log-likelihood difference between the two ML trees under comparison exceeds a minimum threshold of 0.5 log-likelihood units, before even considering the AU p-value reported in the CONSEL log file. If the log-likelihood difference falls below this threshold, the two trees are considered statistically equivalent *ipso facto*. Using

this minimum threshold is necessary, because CONSEL may encounter numerical issues when the log-likelihood difference between the candidate trees is very small. I refer to this combined criterion (i.e., AU test supplemented by a minimum threshold for correction) as the *neo-liberal* approach. I henceforth adopt this evaluation criterion throughout the remainder of this thesis for plausibility assessments. In Section 3.4, I report the plausibility assessment results obtained via CONSEL in juxtaposition to those obtained via IQ-TREE.

For each MSA, I calculate the overall speedup of Adaptive RAxML-NG relative to the standard version, by dividing the standard runtime by the adaptive one. To obtain accurate runtime measurements, I executed both versions in sequential mode. Due to the large number of MSAs (see Section 3.3.2), I set an upper execution time limit of 24h for both versions. I additionally define and compute the per-search speedup (PS):

$$PS = \frac{T_S/N_S}{T_A/N_A}$$

where T_S and T_A are the overall execution times of standard and Adaptive RAxML-NG, and N_S and N_A are the number of independent tree searches conducted by each RAxML-NG version, respectively (hence $N_S := 20$). I present summary results for both speedups and per-search speedups across all MSAs.

Finally, for each MSA, I compute the relative/normalized RF distance (nRF; see Section 2.1.3) between the standard and adaptive tree. For simulated MSAs, I further compute the relative RF distances between the best ML trees inferred by each version and the true reference tree, that is, the tree used to generate the simulated sequences.

3.3.2 Datasets

I conducted experiments on both empirical and simulated datasets. For the empirical analyses, I initially collected all 10,389 MSAs available in TreeBASE. Several filtering steps were then applied. Specifically, I excluded MSAs that either exceeded the 24h runtime limit, or on which the corresponding IQ-TREE statistical tests execution failed. Such failures typically arose from unfavorable MSA properties (e.g., duplicate sequences or sequences only comprising gaps) or from invalid characters in taxon names, which triggered errors in IQ-TREE when importing the trees inferred by RAxML-NG. The final empirical dataset collection comprises 9,515 MSAs that successfully passed all filtering stages. Out of those datasets, 8,052 are unpartitioned DNA MSAs (UP-DNA), 638 are partitioned DNA (P-DNA), 817 are unpartitioned AA (UP-AA), and 8 are partitioned AA alignments.

I also used 5,000 simulated MSAs. Specifically, I randomly sub-sampled 4,500 simulated DNA MSAs used in the benchmark study by Höhler *et al.* [89]. Dimitri Höhler, who is also a co-author of the Adaptive RAxML-NG paper, generated an additional 500 simulated AA MSAs, based on a sample from the RAxML Grove [91] database. RAxML Grove datasets contain files with inferred trees, their respective estimated substitution model parameters, and statistical information about the analyzed MSA. In order to avoid simulating extremely large MSAs, datasets with an MSA number of taxa and patterns below the 95^{th} percentile were selected. Next, D. Höhler used the trees and the substitution models that were selected by the users of the RAxML web servers to specify the model parameters, which were used for MSA simulation with AliSim [213]. Overall, the following substitution models for AA MSA simulations were

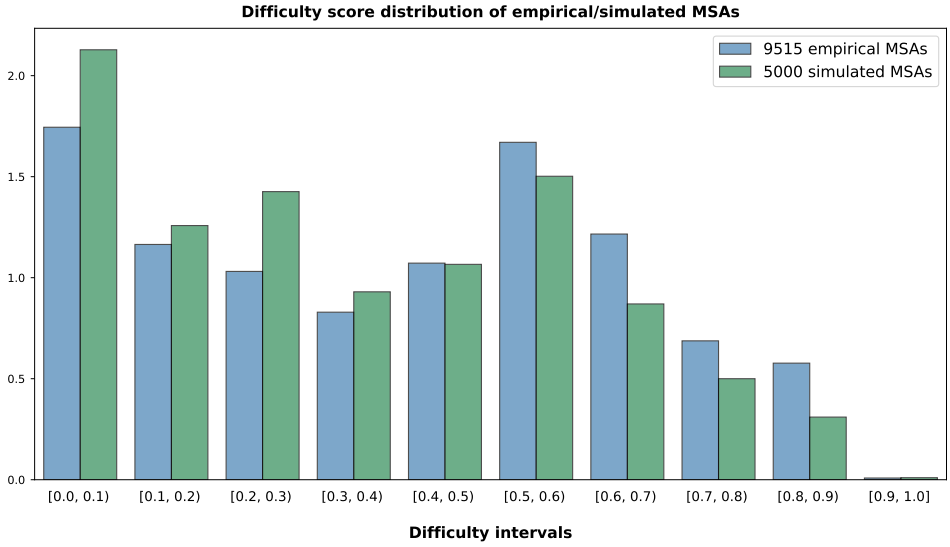


Figure 3.3: Difficulty score distributions of 9,515 empirical and 5,000 simulated MSAs, as computed via Pythia v0.0 (Figure adapted from [210]).

used: JTT+Γ (29.79%; [96]), LG+Γ (28.64%; [116]), Dayhoff+Γ (26.70%; [37]), WAG+Γ (7.37%; [224]), Blosum62+Γ (4%; [83]), MtArt+Γ (2.71%; [1]), VT+Γ (2.34%; [139]), GTR+Γ (1.57%; [205]), MtREV+Γ (1.22%; [2]), MtMAM+Γ (1.0%; [231]), rtREV+Γ (0.85%; [40]), MtZOA+Γ (0.65%; [160]), cpREV+Γ (0.47%; [3]), PMB+Γ (0.22%; [217]), HIVw+Γ (0.1%; [145]), FLU+Γ (0.07%; [33]), DCMut+Γ (0.07%; [109]), and HIVb+Γ (0.07%; [145]). Among the 5,000 simulated MSAs, 4,482 were generated as UP-DNA, 18 as P-DNA, 487 as UP-AA, and 13 as P-AA datasets. All simulated MSAs successfully passed all filtering steps.

On empirical MSAs, unless an explicit partitioning scheme with model assignments was specified by the user (as is the case for some TreeBASE MSAs), we treat all the remaining MSAs as unpartitioned. In such cases, I invoked both RAxML-NG versions using the GTR+Γ model for DNA, and the LG model for protein MSAs, respectively. Further, I treated all simulated MSAs as unpartitioned, using the GTR+Γ model for DNA, and the LG model for AA MSAs, respectively. The difficulty score distribution for both empirical and simulated MSAs is shown in Figure 3.3.

Importantly, to reduce the computational complexity and the CO_2 footprint when analyzing protein MSAs, I did not account for RHAS with the Γ-model. In this benchmark, omitting Γ rates does not affect the comparison between standard and Adaptive RAxML-NG, because the MPO routines are identical among the two RAxML-NG versions. This becomes more evident based on the following observation. When comparing the standard and adaptive trees via CONSEL, the respective per-site log-likelihood distributions are computed by evaluating the two trees with standard RAxML-NG, that is, by performing a full BLO followed by MPO on the given tree(s). Whether or not Γ rates are used during this re-evaluation is irrelevant, since the MPO routines are identical, and hence the overall plausibility assessment remains unaffected.

3.3.3 Commands and Compute Cluster Details

Adaptive RAxML-NG is integrated into RAxML-NG and is available under the GNU GLP license. A snapshot of the Adaptive RAxML-NG version which was used for the benchmarks reported here (as well as the corresponding paper [210]) is archived as a tagged release in my personal GitHub repository: `https://github.com/togkousa/raxml-ng/releases/tag/v1.0`. Further, I continued developing Adaptive RAxML-NG after its publication, in the `adaptive` branch of a forked RAxML-NG repository: `https://github.com/togkousa/raxml-ng/tree/adaptive`. These subsequent improvements are already integrated into the second major version of RAxML-NG (see Chapter 6).

Users can invoke the Adaptive RAxML-NG version using the `--adaptive` option when executing RAxML-NG. The results can be reproduced by running the following commands:

Standard RAxML-NG v1.1[1]**:**

```
./raxml-ng --threads 1 --msa {msa} --model {model} --seed 0
```

Adaptive RAxML-NG:

```
./raxml-ng --adaptive --threads 1 --msa {msa} --model {model} --seed 0
--lh-epsilon 0.1
```

In the Adaptive version, it is important that the user explicitly specifies an ε-threshold of 0.1 (using `--lh-epsilon 0.1`). This is because the tagged release incorporates minor subsequent improvements made between RAxML-NG versions 1.1 and 1.2. One of those was to increase the default ε-threshold from 0.1 to 10, owing to the results of a systematic study [74].

I executed the experiments on the Haswell/URZ Cluster, located at the Computing Center of the University of Heidelberg. It comprises 224 nodes with Intel Haswell CPUs (E5-2630v3 running at 2.40GHz). Each node contains 2 CPUs and each CPU has 8 physical cores. The operating system is CentOS Linux 7 (Core).

3.4 Results

Figure 3.4 summarizes the plausibility assessment results obtained via the IQ-TREE significance tests, which were conducted on all standard-adaptive tree pairs, over all datasets. The best scoring tree of each pair serves as the reference tree. In 95.6% of empirical, and 93.4% of simulated MSAs, the two trees are statistically indistinguishable, while in 0.84% of the cases, Adaptive RAxML-NG infers a significantly better ML tree.

The results obtained via IQ-TREE under the "conservative" approach are already encouraging for the Adaptive heuristic. They suggest that adjusting the tree search thoroughness based on the input MSA's difficulty score predicted by Pythia constitutes an effective tree search strategy. Only on 2.64% of the empirical, and 4.88% of the simulated MSAs, does RAxML-NG v1.1 infer a significantly better ML tree. This emerging trend is even more pronounced when evaluating the two versions via CONSEL's AU test (under the neo-liberal approach). Figure 3.5 shows that, under the second comparative framework, on 99.74% of empirical, and 99.96% of simulated MSAs, the standard and Adaptive versions yield statistically indistinguishable ML trees.

[1] https://github.com/amkozlov/raxml-ng/releases/tag/1.1.0

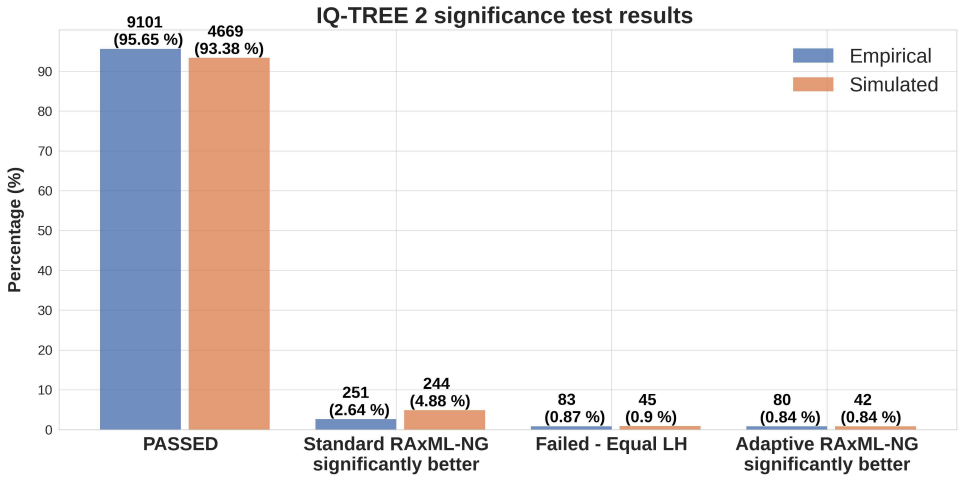


Figure 3.4: Plausibility assessment based on the statistical tests implemented in IQ-TREE. Label `PASSED` denotes that both the standard and the adaptive tree passed *all* statistical tests, and are hence statistically indistinguishable (Figure reproduced from [210]).

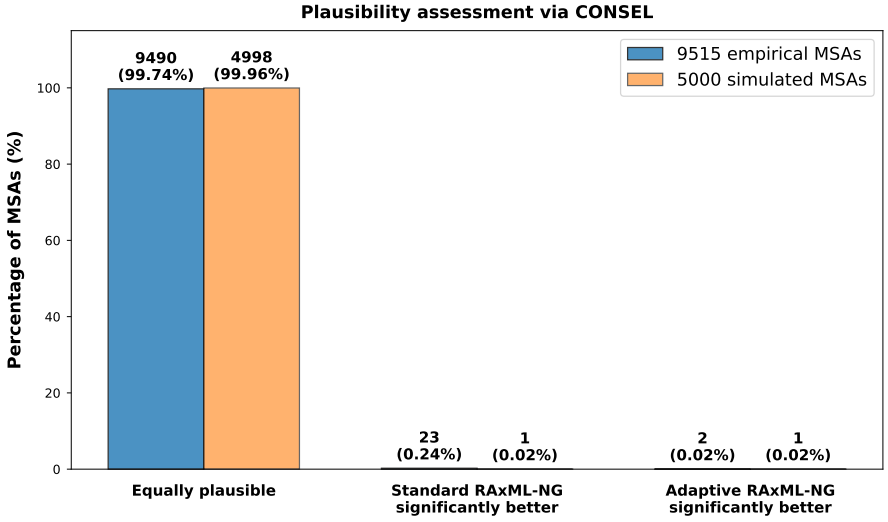


Figure 3.5: Plausibility assessment using the AU test, as implemented in the CONSEL tool. ML trees with a p-value ≥ 0.05 are considered to be plausible.

Next, Figure 3.6 summarizes the speedup distributions of Adaptive relative to standard RAxML-NG, on empirical and simulated MSAs, over 10 difficulty intervals. As expected, I attained substantial speedups (exceeding 5×) on easy and difficult datasets, since the number of independent tree searches performed by Adaptive RAxML-NG is lower for these difficulty intervals. Table 3.1 provides analogous summary statistics, including average speedups and standard deviations, for both the overall and per-search speedup distributions across the 10 difficulty intervals. Further, I define the accumulated speedup to be the ratio of the overall accumulated execution times of the two RAxML-NG versions under comparison. The accumulated speedups on empirical and simulated MSAs are 3.11× and 4.27×, respectively.

Regarding topological accuracy, Figure 3.7 illustrates the relative RF distances between the best ML trees inferred by each version over all standard-adaptive tree pairs. For MSAs with a difficulty score below 0.5, the standard-adaptive pairs generally exhibit

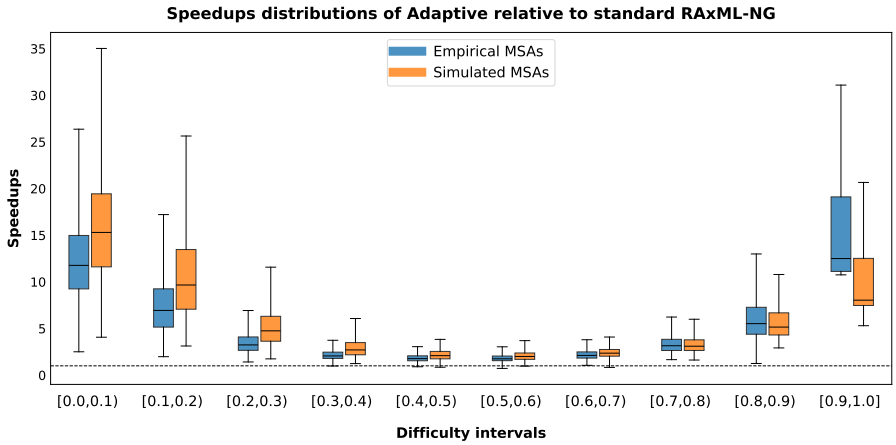


Figure 3.6: Speedup distributions of Adaptive vs. standard RAxML-NG, for empirical/simulated MSAs over 10 difficulty intervals. The dashed line at the bottom corresponds to a speedup of 1× (Figure adapted from [210]).

	Empirical				Simulated			
Difficulty	Av.S	Std.S	Av.PS	Std.PS	Av.S	Std.S	Av.PS	Std.PS
[0.0, 0.1)	12.58	4.77	1.55	0.5	16.06	6.02	1.96	0.81
[0.1, 0.2)	7.48	3.08	1.88	0.57	10.71	4.74	2.72	1.05
[0.2, 0.3)	3.48	1.15	1.82	0.49	5.13	2.17	2.69	1.02
[0.3, 0.4)	2.2	0.61	1.79	0.46	3.02	1.2	2.44	0.87
[0.4, 0.5)	1.85	0.45	1.84	0.44	2.2	0.63	2.18	0.61
[0.5, 0.6)	1.84	0.47	1.83	0.47	2.1	0.54	2.09	0.53
[0.6, 0.7)	2.15	0.55	1.81	0.42	2.36	0.63	2.0	0.5
[0.7, 0.8)	3.29	1.09	1.8	0.49	3.24	0.97	1.75	0.42
[0.8, 0.9)	6.43	3.48	1.85	0.75	5.71	2.01	1.68	0.45
[0.9, 1.0)	16.06	6.69	2.7	0.85	10.8	5.46	1.95	0.74

Table 3.1: Average value and standard deviation of speedups and per-search speedups of Adaptive vs. standard RAxML-NG, over 10 difficulty intervals. Av.S: average speedup, Std.S: standard deviation of speedups, Av.PS: average Per-search speedup, Std.PS: standard deviation of Per-search speedup (Table reproduced from [210]).

relative RF distances below 0.2, indicating that both versions infer topologically similar trees when the phylogenetic signal is sufficiently strong. Finally, Figure 3.8 presents the relative RF distance distributions between the inferred (standard/adaptive) trees and the true tree, on simulated MSAs. As expected, the distance to the true tree increases for both methods with increasing MSA difficulty. Importantly, the relative RF distance distributions for the two versions are nearly identical, demonstrating that Adaptive RAxML-NG maintains the same level of topological accuracy as the standard version, across the entire difficulty spectrum.

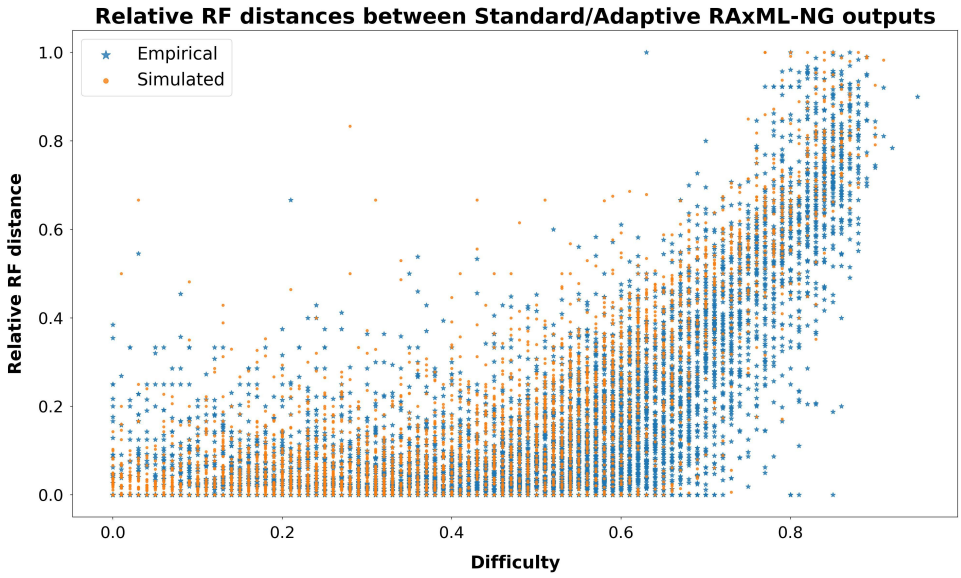


Figure 3.7: Relative RF distances between the standard and adaptive tree across all empirical and simulated MSAs. (Figure reproduced from [210]).

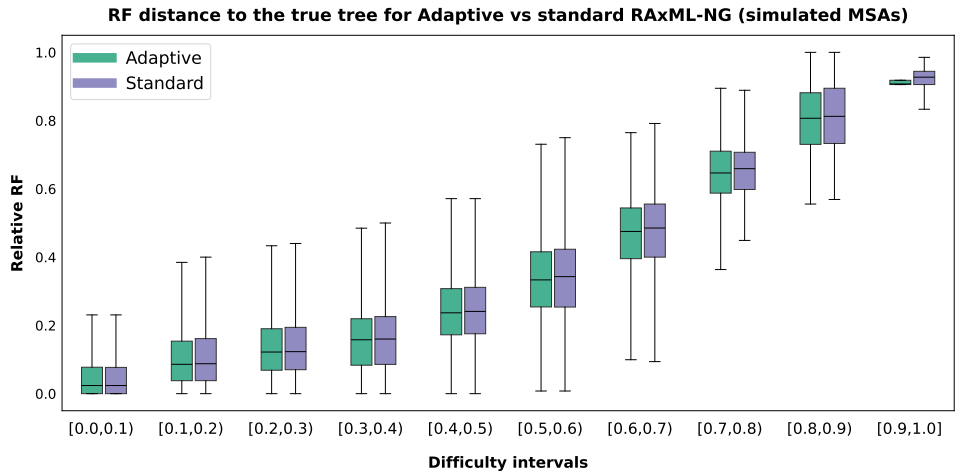


Figure 3.8: Relative RF distances between the standard/adaptive trees and the true tree, over all simulated MSAs.

3.5 Discussion

By introducing Pythia into phylogenetic search engines, users can obtain an *a priori* estimate of the expected robustness of the final result, since the difficulty score directly reflects the amount of phylogenetic signal in the input MSA. The benefits of analyzing easy MSAs with the Adaptive version are the substantial speedups, and the robustness of the final result on easy- and intermediate-to-analyze MSAs. On the other hand, users should be aware that any individual tree inferred on a difficult MSA with a rugged likelihood surface is most likely non-informative. On difficult MSAs, Adaptive RAxML-NG executes a fast heuristic to rapidly infer just a few, out of the many equally likely, yet incongruent, tree topologies. In such cases, the software issues a warning in its log file, informing the users about the insufficient phylogenetic signal in the input MSA.

The development of Adaptive RAxML-NG is an important contribution to the field. It demonstrates that the predicted Pythia score can serve as a reliable estimate of the quality of the phylogenetic signal that the MSA contains, and that the tree search heuristic can be adjusted accordingly, using faster heuristics on easy and difficult MSAs, and more thorough strategies on intermediate ones. Nonetheless, as any novel method, it exhibits certain limitations that became more evident after its publication, and motivated my further research. I briefly outline a few of these considerations here.

First, although the Adaptive heuristic is relatively straight-forward (compared to RAxML-NG v1.1), it is still stringent and can be adjusted even further (e.g., see Chapter 4). Second, the application of an initial NNI round to easy and difficult MSAs, regardless of the starting tree type, proved to be suboptimal in retrospect. As discussed in Section 2.6.4, the NNI round iterates until it reaches an NNI-optimal tree. When applied to large MSAs (i.e., MSAs comprising thousands of taxa), the NNI round may substantially slow down the search when the search starts with a random tree. This was not observed in the Adaptive RAxML-NG benchmarks, because TreeBASE MSAs typically comprise up to 1,000 taxa. Therefore, the NNI round should be applied only under appropriate conditions, that is, on smaller MSAs (e.g., MSAs with less than 1,000 taxa), or when the current tree has already a relatively "good" log-likelihood score (e.g., a parsimony starting tree). Ultimately, SPR rounds yield larger log-likelihood "jumps" and allow for a more thorough exploration of the tree space, and are hence more appropriate as primary search mechanism.

Finally, the 1% "first-stage convergence" criterion introduced some additional challenges for parallelizing the heuristic. To ensure reproducibility, the current design can be parallelized in only two ways: (a) by employing fine-grained parallelization, or (b) by initially executing *only* the first ML tree inference in fine-grained mode while all the remaining searches wait, and then executing the remaining inferences via a hybrid fine- and coarse-grained parallelization scheme. In both scenarios, however, the computational resource utilization is suboptimal.

However, the above limitations do not undermine the importance of the Adaptive RAxML-NG contribution. On the contrary, they provided insights and motivated my further research on developing even more efficient heuristics, building on the knowledge gained from the Adaptive RAxML-NG project. As I write in the Conclusion of the Adaptive RAxML-NG paper:

> Lastly, we aim to test alternative heuristic tree search strategies so that we can design and implement improved heuristic approaches in adaptive RAxML-NG to further refine likelihood scores and reduce runtime. [210]

3.6 Data and Software Availability

As mentioned in Section 3.3.3, Adaptive RAxML-NG is integrated into RAxML-NG and is available under the GNU GLP license. A snapshot of the Adaptive RAxML-NG version which was used for the above benchmarks is archived as a tagged release in my personal GitHub repository: `https://github.com/togkousa/raxml-ng/releases/tag/v1.0`. All MSAs used for the experiments are available for download at `https://cme.h-its.org/exelixis/material/raxml_adaptive_data.tar.gz`.

4. Early Stopping in RAxML-NG

This Chapter is based on the following peer-reviewed article:

- **Anastasis Togkousidis**, Alexandros Stamatakis, and Olivier Gascuel. "Accelerating Maximum Likelihood Phylogenetic Inference via Early Stopping to Evade (Over-)optimization." *Systematic Biology*, 2025. `https://doi.org/10.1093/sysbio/syaf043`

Contributions: Anastasis Togkousidis designed and implemented the simplified RAxML-NG (sRAxML-NG) heuristic, integrated the Kishino-Hasegawa (KH) test into the RAxML-NG code, used it to implement the KH-based stopping criteria, and conducted the performance evaluation. Alexandros Stamatakis and Olivier Gascuel supervised the project and helped with the writing of the paper. Olivier Gascuel also proposed the idea of using the KH test as a means for Early Stopping in ML inference, as well as its corrected version for multiple testing.

As discussed in Section 2.9, MSA sequences are inherently noisy for a plethora of reasons (e.g., stochastic, systematic, or sampling noise). This increases the risk of overfitting in ML tools, whereby the output (statistical) model incorrectly captures the noise in addition to the true phylogenetic signal. But even if overfitting does not occur, over-optimization remains a recurrent issue in thorough ML tree inference tools. Indeed, RAxML-NG v1.x typically spends substantial computational time in long log-likelihood plateaux during the final stages of optimization, where the observed log-likelihood improvements introduced by successive topological rearrangements are negligible, and may be attributed to data noise rather than the true signal. Therefore, there exist compelling reasons to not push optimization to its limits, by means of early—yet adequate, in terms of yielding statistically plausible ML trees—stopping criteria.

In this Chapter, I initially introduce a simplified RAxML-NG heuristic (sRAxML-NG). This heuristic serves as an underlying Early Stopping method upon which I subsequently develop the stopping criteria. Next, I integrate the Kishino-Hasegawa (KH) test [106] into sRAxML-NG as a reliable, fast-to-compute method for Early Stopping. The sRAxML-NG tool uses the KH test to statistically assess the significance of differences between intermediate trees prior to and after each SPR round. The tree search terminates early when improvements are statistically insignificant. Further, I extend the standard KH test by employing a multiple testing correction, in order to improve the p-value approximation. As I describe below, the central idea is that the KH test can be used as a mechanism to dynamically adjust RAxML-NG's log-likelihood improvement threshold parameter (ε), based on the convergence behavior of each individual tree inference and MSA.

4.1 Background

A straight-forward solution to apply Early Stopping in ML tree inference is to employ more superficial, yet substantially faster, tree search heuristics, such as the one implemented in FastTree [151]. Benchmarking studies, however, have demonstrated that tools using more thorough heuristics (i.e., RAxML, RAxML-NG, IQ-TREE and PhyML) yield trees that exhibit higher statistical plausibility as well as bootstrap support [89, 120, 235]. Hence, the objective in this Chapter is to devise likelihood-based inference stopping criteria that circumvent unnecessary optimization steps, *without* compromising the quality of the inferred trees. I do so by considering the following.

Both RAxML and RAxML-NG use a fixed log-likelihood improvement ε-threshold to determine convergence during topological optimization[1] (SPR) rounds. The default RAxML v8 heuristic uses an ε-threshold of $\varepsilon_{RAxML} := 0.01$ [194], which is not user-configurable[2]. In RAxML-NG v1.1, the default ε-threshold was set to $\varepsilon := 0.1$. This value was relaxed to $\varepsilon := 10$ in v1.2, owing to the results of a systematic study [74], which showed that the original value of 0.1 was unnecessarily strict. An important limitation of using such fixed, and arbitrarily chosen, ε-thresholds—even when those are explicitly specified by users, as is possible in RAxML-NG via the `--lh-epsilon X` option—is that they may not adequately reflect the phylogenetic signal-to-noise ratio of a given MSA. They may also fail to capture the convergence dynamics of each individual tree inference, and thus unnecessarily prolong the search.

On the other hand, several (early) topological optimization stopping rules have been proposed, which do not rely on static ε-thresholds. RAxML v8, for instance, provides a stopping rule option that terminates the search when the RF distance between the trees obtained from two consecutive SPR rounds falls below 1% (`-D` option; [192, 195]). Earlier, IQPNNI [219], which is the predecessor of IQ-TREE, used to estimate the probability of having found the optimal tree during the tree inferences, by monitoring log-likelihood improvements over search algorithm iterations, and stopped the search once this probability attained 95% (using a Weibull distribution of recorded values). This method was later replaced by a more straightforward rule in IQ-TREE. The latter tool terminates the search if no better tree is found after a fixed number of iterations (default $N := 100$). In practice, the above described early stopping rules were not widely adopted by users.

Finally, another aspect to be discussed, before presenting the main methods, is the number of tree inferences conducted by ML tools in relation to the thoroughness of their heuristic. As already described (see Section 2.6), ML tools deploy hill-climbing heuristics [187] to find a "good" solution, which typically is a local optimum due to the inherent $\mathcal{NP}$-hardness of ML optimization [157]. RAxML-NG and IQ-TREE implement strategies which explore multiple tree search trajectories, although their specific approaches vary. By default, RAxML-NG v1.x conducts 20 independent ML tree inferences that are initiated on 10 parsimony and 10 random starting trees. The program provides the best ML tree inferred by these independent inferences as output. In contrast, IQ-TREE performs a single tree search while maintaining a pool of 100 candidate trees, initially

[1] Thresholds are also used to assess convergence in continuous parameter optimization routines, that is, during BLO and MPO rounds. In this project, however, I retain the default RAxML-NG v1.x parameter configuration for BLO and MPO routines, and do not modify them.

[2] In fact, I extracted this parameter value by examining the RAxML open source code [194]. To my knowledge, this ε-threshold is neither reported in the RAxML v8 log-files, nor in the RAxML manual [195], nor in the respective paper describing the default rapid hill-climbing heuristic [197].

populated with 99 parsimony, and one BioNJ [61], starting trees (see Section 2.6.6). A recent benchmark study [122] recommends performing at least 10 independent ML tree inferences when using RAxML-NG or IQ-TREE. In the case of IQ-TREE, this translates into executing independent runs where each one initiates 100 starting candidate trees. PhyML, by comparison, follows a single search trajectory, starting from a BioNJ tree, and employs a hill-climbing heuristic (analogous to RAxML-NG) to optimize the topology. This raises the important question of whether phylogenetic inference tools should prioritize more exhaustive heuristics with fewer starting trees, or favor a broader exploration of tree space by conducting multiple, yet potentially less thorough, independent searches. In this Chapter, I advocate for the latter approach. Specifically, I argue that when using RAxML-NG with multiple (e.g., 10 parsimony) starting trees, while employing an accelerated, strategically refined, Early Stopping-enhanced heuristic, one can achieve comparable accuracy while substantially reducing runtime and associated CO_2 emissions.

4.2 Methods

I first introduce the sRAxML-NG heuristic and explain the rationale for developing this heuristic simplification, which is necessary for integrating and evaluating the KH-based stopping criteria into RAxML-NG. Next, I describe the application of the KH test within the RAxML-NG framework. Specifically, the KH test is used to statistically assess the significance of log-likelihood differences between tree topologies *prior to* and *after* each SPR round. The optimization process terminates early when improvements between such subsequent tree topologies are statistically insignificant. Further, I extend the KH test by a multiple testing correction to more accurately approximate p-values.

4.2.1 The sRAxML-NG Heuristic

Section 2.6.5 provides a full description of the RAxML-NG v1.x heuristic, along with a schematic overview of the algorithm in Figure 2.13. From this description, it becomes evident that RAxML-NG v1.2 terminates when several distinct conditions are met. One critical factor is the ε-threshold parameter. However, termination also depends on other factors, such as the completion of three distinct sequences of increasingly thorough SPR rounds (i.e., AUTODETECT, FAST, and SLOW SPRs). Ultimately, during the final sequence of SLOW SPR rounds, the algorithm only terminates if the log-likelihood improvement does not exceed the ε-threshold for five consecutive SPRs with increasing minimum and maximum regrafting radii values. This thorough heuristic was originally designed by my fellow lab member Oleksiy Kozlov to maximize the likelihood of the inferred tree topology to the best possible degree [110, 111]. However, the original objective of the *Early Stopping in ML inference* project was to propose and evaluate methods which dynamically determine this ε-threshold parameter, based on the convergence behavior observed during each individual tree inference. Within the existing RAxML-NG v1.2 optimization framework, the role of the ε-parameter with respect to termination is confounded by other stopping conditions, and the impact of modifying the ε-threshold is difficult to assess. Therefore, I needed to simplify the RAxML-NG v1.2 heuristic, such that termination is predominantly controlled by the ε-threshold parameter.

The accelerated tree search heuristic I proposed is called *Simplified RAxML-NG (sRAxML-NG)* and is illustrated in Figure 4.1. It conducts a sequence of FAST SPR

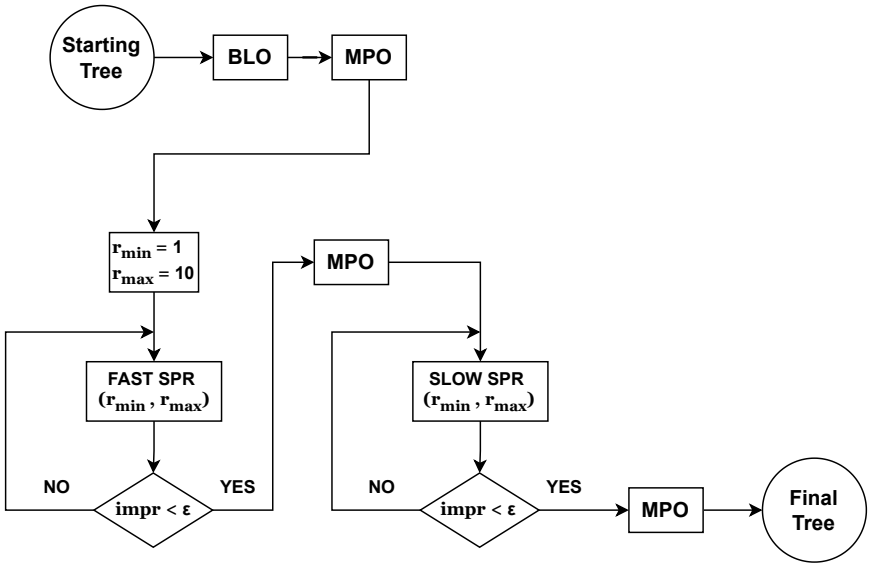


Figure 4.1: A schematic representation of the sRAxML-NG tree inference heuristic (Figure adapted from the Supplementary Material of [211]).

rounds followed by a sequence of SLOW SPR rounds. Each sequence halts when the log-likelihood improvement drops below a fixed default threshold of $\varepsilon := 10$ log-likelihood units, as in RAxML-NG v1.2. Unlike in the standard search strategy, sRAxML-NG simply executes the FAST and SLOW SPR rounds with fixed minimum and maximum subtree reinsertion radii parameters: $r_{min} := 1$ and $r_{max} := 10$, respectively. All remaining SPR parameters/settings (e.g., the subtree cutoff threshold) are identical to those used in the FAST and SLOW SPR rounds of RAxML-NG v1.x (see Section 2.6.5). Full BLO rounds are conducted on the initial tree and after each FAST and each SLOW SPR round (as in RAxML-NG v1.x). MPO rounds are invoked on the initial tree topology (after BLO), in-between the series of FAST and SLOW SPR rounds, and on the final tree before termination (after the last SLOW SPR round).

This simplified heuristic substantially accelerates inferences and already yields an initial, yet ad-hoc, Early Stopping implementation. In fact, it is a noteworthy by-product of this study, as I mainly developed sRAxML-NG to seamlessly assess the accuracy of dynamic adaptations of the ε convergence threshold parameter. The KH-based methods described below build upon the sRAxML-NG heuristic, by dynamically adjusting the ε-threshold after each SPR round.

4.2.2 Kishino-Hasegawa (KH) Test in RAxML-NG

I deploy the KH test to compare subsequent best trees prior to and after each SPR round. Based on the per-site log-likelihood differences of the two trees, an ε-value can be derived, which depends on the observed variations of the per-site log-likelihood distributions that correspond to the SPR round under consideration. Consequently, the ε-threshold parameter varies over SPR rounds, and automatically adapts to the convergence dynamics of each independent ML tree search and MSA.

The KH test is a paired test that compares the log-likelihood of each site in an MSA for two different phylogenetic trees. Typically, the KH test assesses whether the likelihood

difference between two trees is significant for a given MSA. To avoid optimization bias, it is generally advised that neither tree in the list was inferred on the MSA under study [128]. This is because, if one tree represents a commonly accepted hypothesis about the studied species, while the other tree has been inferred on the given MSA, the KH test would inherently favor the latter. In this application, *both* trees are inferred using the studied MSA, thus eliminating bias in favor of one tree over the other. However, since the algorithm tests (evaluates) many trees during a single SPR round, we are faced with a scenario where multiple testing would be beneficial (see below).

Several versions of the KH test are described in [64]. I adopt the fastest version, as implemented in PAUP* [203] and PHYLIP [49], which yields results that are comparable to the more computationally intensive resampling estimated log-likelihood (RELL; [107]) method. Moreover, this faster approach substantially simplifies the multiple-test implementation.

For an MSA with s sites, suppose that the per-site log-likelihood vectors before ($\mathbb{L}$) and after ($\mathbb{L}'$) any given SPR round, are:

$$\mathbb{L} = (\mathcal{L}_1, \mathcal{L}_2, ..., \mathcal{L}_s)$$

$$\mathbb{L}' = (\mathcal{L}'_1, \mathcal{L}'_2, ..., \mathcal{L}'_s)$$

And the corresponding log-likelihoods are:

$$L := \sum_{i=1}^{s} \mathcal{L}_i$$

$$L' := \sum_{i=1}^{s} \mathcal{L}'_i$$

The KH test can be applied as follows:

1. We compute the distribution of the (paired) differences between the vector coordinates $(\mathcal{L}'_i - \mathcal{L}_i), i = 1, 2, ..., s$. These differences can be either positive or negative, even if $L' - L > 0$.

2. We compute the standard deviation σ_{KH} of the distribution of the s differences. Assuming that the per-site differences are i.i.d., the standard deviation of the difference $(L' - L)$ is $\sigma_{KH} \cdot \sqrt{s}$.

3. In the fast version of the KH test [64], it is assumed that the difference $(L' - L)$ is normally distributed. This constitutes a consistent approximation due to the law of large numbers, and because we are summing over log-likelihood differences and not the log-likelihood values themselves. In this respect, under the null hypothesis $(L' \equiv L)$, the term $\frac{L'-L}{\sigma_{KH}\sqrt{s}}$ follows a normal distribution $N(0, 1)$ with a cumulative distribution function $\Phi(\cdot)$. The tree search continues (i.e., we reject the null hypothesis that $L' \equiv L$) with 95% confidence if:

$$1 - \Phi\left(\frac{L' - L}{\sigma_{KH}\sqrt{s}}\right) \leq 0.05 \quad \Leftrightarrow$$

$$\varepsilon = 1.645 \cdot \sigma_{KH}\sqrt{s} \tag{4.1}$$

4.2.3 KH Test with Multiple Correction

Since the standard KH test does not incorporate adjustments for multiple testing, it may unnecessarily prolong the tree search. Note that an SPR round comprises multiple iterations, where each iteration involves pruning and regrafting each subtree of the current tree (see Section 2.6.4). During each SPR iteration, numerous potential regrafted topologies are evaluated (i.e., topologies generated by SPR moves performed on a given subtree within a specified radius), and only the best move is ultimately selected. This is a situation that requires accounting for the multiplicity of the tests. However, the majority of the candidate topologies visited during an SPR round exhibit a decreased fit (lower likelihood) to the input MSA, especially toward the end of the tree search, when the algorithm approximates the (local) optimum. To account for this, I propose a solution that does neither require additional calculations nor data storage for each SPR move that is being performed during an SPR round, as required by more sophisticated approaches (e.g., Holm-Bonferroni, AU). Instead, during an SPR round, I track the number of SPR topologies, denoted as N_t, which improve the likelihood relative to the best tree identified prior to the current iteration. This count is accumulated across all pruning-regrafting iterations during the SPR round (see Section 2.6.4). After the end of the SPR round, I apply a Bonferroni correction to the p-value of the best tree generated thereby. The adjusted ε-threshold is then derived as follows:

$$1 - \Phi\left(\frac{L' - L}{\sigma_{KH}\sqrt{s}}\right) \leq \frac{0.05}{N_t} \quad \Leftrightarrow$$

$$\varepsilon = \sigma_{KH}\sqrt{s} \cdot \Phi^{-1}\left(1 - \frac{0.05}{N_t}\right) \tag{4.2}$$

4.2.4 Boundary Cases

Certain boundary cases require dedicated measures when implementing the KH-based versions. These cases are listed below.

- If the log-likelihood improvement $(L' - L)$ after an SPR round is very small (down to numerical round-off error), it is expected that the standard deviation σ_{KH} will also be very small. Consequently, the quantity $\frac{L'-L}{\sigma_{KH}\sqrt{s}}$, used as an argument in the cumulative distribution function $\Phi(\cdot)$ of the standard normal distribution (see Equations (4.1)–(4.2)), becomes undefined, inducing numerical instability. In fact, in cases when $(L' - L)$ is very small, the KH test, or any statistical test for that matter, should not be used for comparison. To avoid this, when $L' - L < 10$, the algorithm exits the current sequence of SPR rounds. Essentially, the minimum value that the ε-threshold can take in both KH-based methods is 10. This prevents the KH-based methods from assigning a value to the ε-threshold that is lower than the one used in RAxML-NG v1.2 (and hence unnecessarily prolonging the search), which has been experimentally determined to suffice for inferring high-quality trees on a large collection of datasets [74].

- If the quantity $\frac{L'-L}{\sigma_{KH}\sqrt{s}}$ exceeds 3.5, I assume that the Null Hypothesis is rejected (in both KH-based versions), as this value is typically the maximum in normal distribution tables. In this case, I conventionally set $\varepsilon := L' - L - 1$ (hence $L' - L > \varepsilon$), and the algorithm proceeds to the next SPR round.

- Occasionally, $L' - L > 0$ may occur even if the topology remains unaltered prior to and after the SPR round, due to branch-length optimizations within and after the SPR round, again due to round-off error propagation. In such instances, I conventionally set $\varepsilon := L' - L + 1$ (hence $L' - L < \varepsilon$) and the algorithm exits the current SPR round sequence.

4.3 Experimental Setup

4.3.1 Benchmark Pipeline

I conducted experiments on 300 large empirical MSAs (222 DNA and 78 AA) sampled from the TreeBASE database [147], as well as on 1,076 simulated DNA MSAs sampled from the datasets used in two recent benchmark studies, conducted by our group and colleagues [89, 214]. I provide details on the sampling criteria and additional dataset features (e.g., sizes, Pythia scores) in Section 4.3.2.

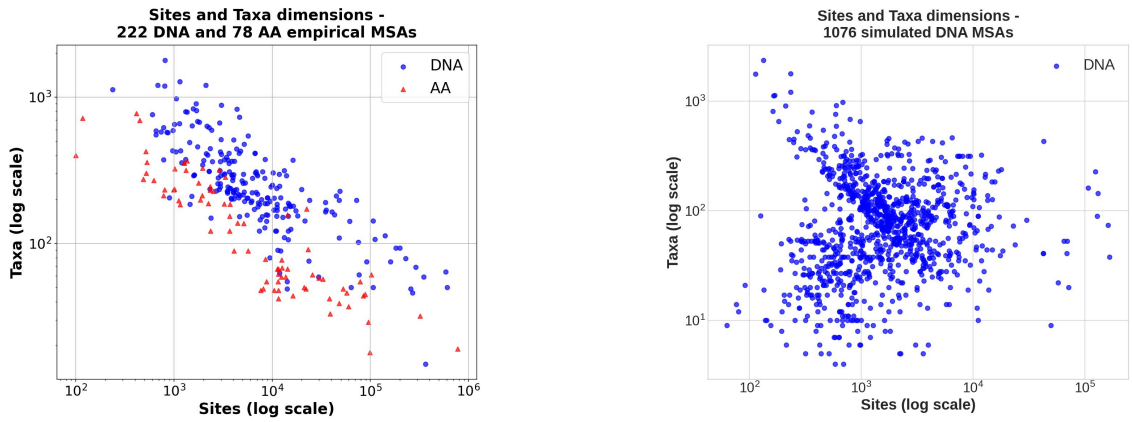
As mentioned in the Methods section, the KH-based stopping criteria have been integrated into sRAxML-NG. In the following experiments, I benchmark three methods—to which I refer as the *Early Stopping* (ES) versions/criteria—against RAxML-NG v1.2:

(a) sRAxML-NG alone (with $\varepsilon := 10$);
(b) sRAxML-NG combined with the simple KH test (KH-simple);
(c) sRAxML-NG combined with KH-multiple correction test (KH-mult).

Specifically, I compare all ES versions against RAxML-NG v1.2, as well as the KH-based versions against sRAxML-NG. For each MSA and program version I conducted two executions: one using 10 parsimony, and one using 10 random starting trees. Therefore, each program version conducts 10 independent tree inferences per execution. All RAxML-NG versions use the exact same set of random and parsimony starting trees, thereby ensuring identical initial conditions across experiments and providing a fair basis for comparison. I refer to the 10 output trees from each execution as the inferred *ML trees*.

The evaluation includes plausibility assessments of the inferred ML trees, runtime comparisons, and (on simulated MSAs) topological accuracy values. Regarding plausibility assessment, I adopt two evaluation schemes. In the first (scheme I), I collect all ML trees inferred by all RAxML-NG versions, for a given MSA and a fixed starting tree type (i.e., 10 ML trees per version, yielding 40 ML trees in total), and conduct the AU test [178] using the CONSEL tool [179]. This corresponds to the "neo-liberal" approach described in Section 3.3.1. I consider all ML trees whose p-value exceeds 0.05 as being plausible. Under the second evaluation scheme (scheme II), I jointly compare all 80 ML trees inferred by all versions for a given MSA (20 ML trees per version), again using the AU test with a p-value threshold of 0.05. This second setup resembles the default execution of RAxML-NG v1.2, where 10 random and 10 parsimony starting trees are used.

I adopt these two comparison schemes because it is important to benchmark the ES versions against RAxML-NG v1.2, both under the default execution conditions (that is, using both starting tree types), and on each starting tree type separately. Parsimony starting trees typically exhibit relatively high initial log-likelihood scores and are therefore closer to a local optimum. As I will show, though, the impact of ES is more pronounced when using random starting trees. At the same time, it is important to assess the overall impact of random starting trees on the ML inference process in general, as well as specifically in conjunction with ES. The results in Section 4.4 show that random starting trees



(a) **(b)**

Figure 4.2: Scatter plot showing the number of taxa and sites for each dataset in **(a) 300 large empirical MSAs** and **(b) 1,076 simulated DNA MSAs** (Figure reproduced from the Supplementary Material of [211]).

only offer negligible benefit in ES versions, which suggests that ES should not be invoked on random starting trees. More broadly, the results also question the overall utility of random starting trees even for the standard RAxML-NG heuristic. I extensively discuss that matter in Section 4.5.

To obtain accurate runtime measurements, I executed all program versions sequentially. Parallelization, however, has been implemented and tested for the ES versions. I report sequential speedups for random and parsimony starting tree executions separately, so as to illustrate that random starting trees also offer suboptimal runtime improvements compared to parsimony ones. Finally, for simulated MSAs, I conduct topological accuracy measurements, by computing the RF-distance between the best ML trees inferred by each tool version and the true reference tree. I summarize the results for each version, across all MSAs.

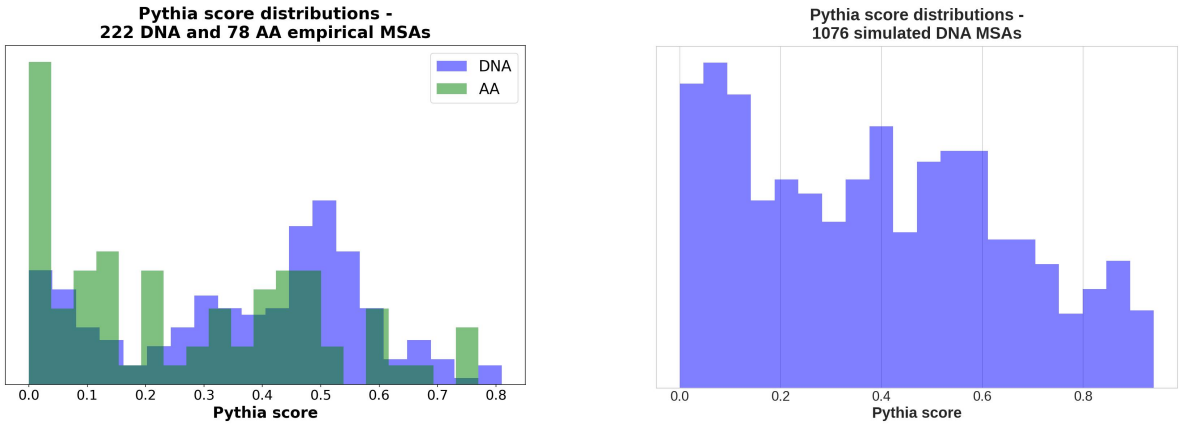
4.3.2 Datasets

As stated above, I conducted experiments on 300 large empirical MSAs (222 DNA and 78 AA) sampled from the TreeBASE database [147]. These MSAs were drawn from the set of 9,515 empirical TreeBASE alignments used to benchmark the Adaptive version of RAxML-NG (see Section 3.3.2). I ranked the full set of 9,515 MSAs according to the $n \cdot s$ product, where n denotes the number of taxa, and s the number of sites[1] in each MSA. From this ranking I sampled the top 300 MSAs.

In addition, I conducted experiments on 1,076 simulated MSAs, sampled from two recent benchmark studies [89, 214]. Specifically, I sampled 306 MSAs from Höhler *et al.* [89] and 770 MSAs from Trost *et al.* [214]. I drew a random sample of MSAs, with the only restriction to maintain an approximately uniform distribution of Pythia MSA scores.

For the analysis of the empirical and simulated DNA MSAs I used the GTR+Γ

[1]Although likelihood computations scale with the number of distinct MSA patterns (instead of sites), the number of sites serves as a proxy for the alignment size and, in many practical instances, correlates with the number of patterns.



(a) **(b)**

Figure 4.3: Distributions of Pythia v0.0 scores on **(a) 300 large empirical MSAs** and **(b) 1,076 simulated DNA MSAs** (Figure reproduced from the Supplementary Material of [211]).

model [205], and for the empirical AA datasets I used the LG+Γ model [116]. The scatter plots in Figure 4.2 illustrate the MSA dimensions, while the corresponding Pythia difficulty score distributions [73] (computed via Pythia v0.0) are shown in Figure 4.3.

4.3.3 Commands and Compute Cluster Details

The stopping criteria are implemented in the `stopping-criteria` branch of a forked RAxML-NG repository: `https://github.com/togkousa/raxml-ng/tree/stopping-criteria`. The sRAxML-NG heuristic can be invoked via `--extra simplified-on`. Stopping criteria are selected using the `--stopping-criterion` argument; available options are: `{KH | KH-mult}`, which correspond to the simple KH test and the KH test with multiple testing correction, respectively. When stopping criteria are specified, the algorithm automatically executes the sRAxML-NG heuristic search, hence `--extra simplified-on` does not need to be explicitly specified. The commands I used to invoke each version and generate the results presented in Section 4.4 are the following (e.g., using 10 parsimony starting trees):

RAxML-NG v1.2[1]**:**

```
./raxml-ng --threads 1 --msa {msa} --model {model} --tree pars{10}
--seed 0
```

sRAxML-NG:

```
./raxml-ng-adaptive --threads 1 --msa {msa} --model {model}
--tree pars{10} --seed 0 --extra simplified-on
```

KH-simple:

```
./raxml-ng-adaptive --threads 1 --msa {msa} --model {model}
--tree pars{10} --seed 0 --stopping-criterion KH
```

[1]https://github.com/amkozlov/raxml-ng/releases/tag/1.2.0

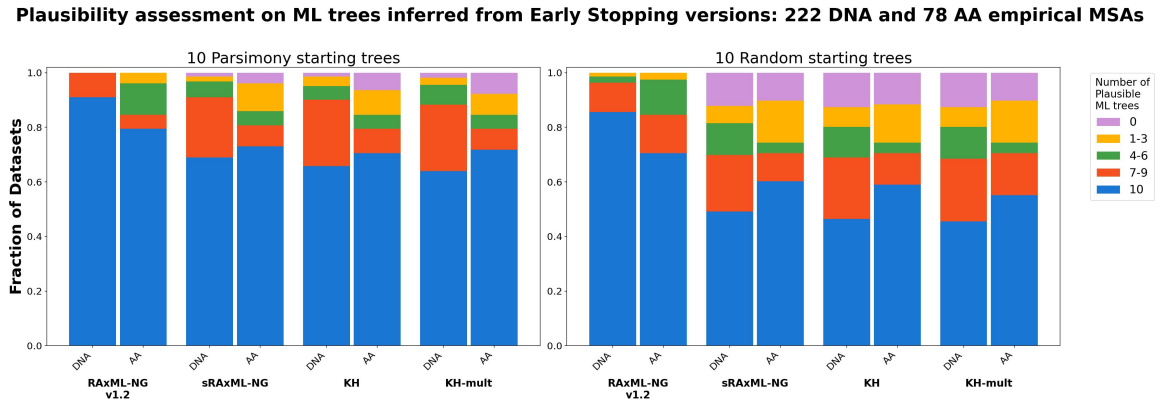


Figure 4.4: Plausibility test results (scheme I) for ML trees inferred via RAxML-NG v1.2 and ES versions across 222 DNA and 78 AA empirical MSAs. For each dataset, all RAxML-NG versions conduct 10 independent tree inferences using the same set of parsimony (left subfigure) or random (right subfigure) starting trees. I evaluated the inferred ML trees using the AU test (over 40 trees, for each type of starting trees) implemented in the CONSEL tool, considering ML trees with a p-value ≥ 0.05 as being plausible. (Figure reproduced from [211]).

KH-multiple testing:

```
./raxml-ng-adaptive --threads 1 --msa {msa} --model {model}
--tree pars{10} --seed 0 --stopping-criterion KH-mult
```

When using the above commands, all versions generate the exact same starting trees, since the seed is fixed among all invocations and the introduced modifications only affect the likelihood-based tree optimization steps.

I executed the experiments on the bwForCluster Helix, located at the Heidelberg University Computing Centre. I utilized the `cpu-single` cluster partition, which allocates AMD nodes for submitted jobs. Each AMD node is equipped with two AMD Milan EPYC 7513 Processors, running at 2.60GHz, providing a total of 64 physical cores per node. The operating system is RedHat Linux.

4.4 Results

4.4.1 Results on 300 Empirical MSAs

Plausibility Assessment

Figure 4.4 illustrates the fraction of datasets for which each version under comparison inferred x (out of 10) plausible trees, where x represents specific counts or intervals. The left subfigure corresponds to executions using 10 parsimony starting trees, and the right one to 10 random starting trees. A substantial part of the plausibility assessment analysis focuses on the fraction of MSAs for which a given version infers at least one ($x = 1$) plausible ML tree. In fact, this is a minimal, yet sufficient criterion to ascertain statistical equivalence between versions. When at least one plausible tree is found, the output (i.e., the best ML tree inferred by the given version) is statistically equivalent to the best tree found by the four versions under comparison. In Figure 4.4, the inferred ML

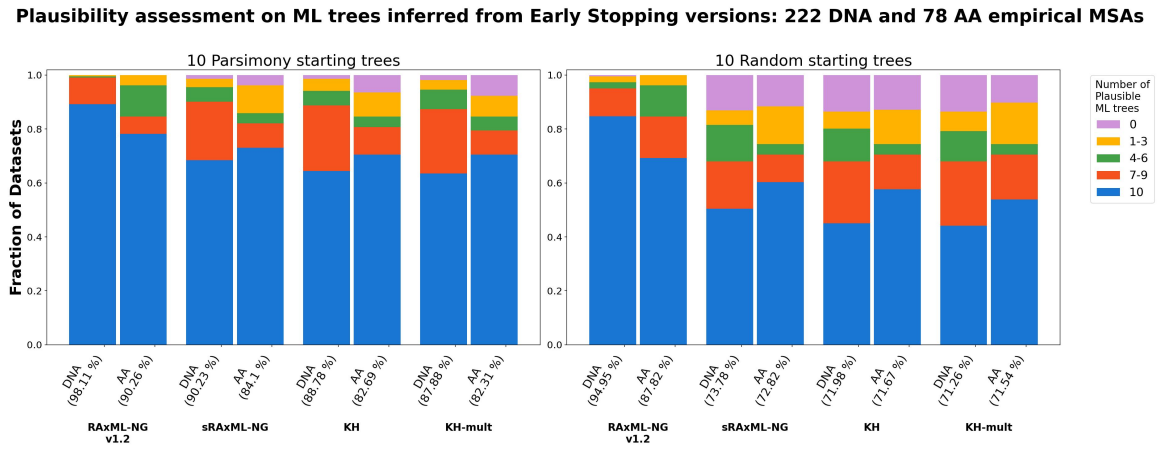


Figure 4.5: Plausibility test results (scheme II) for ML trees inferred via RAxML-NG v1.2 and ES versions across 222 DNA and 78 AA empirical MSAs. For each dataset, all RAxML-NG versions conduct 10 independent tree inferences using the same set of parsimony (left subfigure) or random (right subfigure) starting trees. Similar to Figure 4.4, the inferred ML trees are evaluated using the AU test (over all 80 trees from both starting tree types) implemented in the CONSEL tool. The percentage shown in parentheses next to each x-axis label represents the proportion of plausible trees for each method, calculated according to Equation (4.3) (Figure reproduced from the Supplementary Material of [211]).

trees on random and parsimony starting trees are evaluated independently (scheme I), as described in the Figure caption.

The results in Figure 4.4 indicate that ES versions perform substantially better in terms of output ML tree quality when parsimony starting trees are being used. This outcome is expected, as parsimony trees provide a reasonable, non-random starting point for tree searches. Starting from a random tree is suboptimal (in fact, the worst possible choice), and it increases the chances of terminating the search prematurely when applying ES. Specifically, the fraction of DNA MSAs for which the ES versions infer at least one (out of 10) plausible ML trees, when searches are initiated on parsimony starting trees, is 98.2% across all versions. For AA MSAs, the corresponding fractions with parsimony starting trees are 96.2% for sRAxML-NG, 93.6% for KH-simple, and 92.4% for KH-mult versions. In contrast, when random starting trees are being used, the corresponding fractions are approximately 87% for DNA, and ∼90% for AA MSAs across all ES versions, respectively. RAxML-NG v1.2 infers at least one plausible tree across all datasets (100%) when using either parsimony or random starting trees.

Additionally, in executions with parsimony starting trees, the fractions of DNA MSAs for which *all* output ML trees (10 out of 10) are plausible are 91% for RAxML-NG v1.2, 69% for sRAxML-NG, and ∼65% for the KH-based versions; for AA MSAs, the corresponding fractions are 80% for RAxML-NG v1.2 and ∼72% for the ES versions. In contrast, when using random starting trees, the fractions for DNA MSAs are 85% for RAxML-NG v1.2 but below 50% for the ES versions, while for AA MSAs, the fractions are ∼70% for RAxML-NG v1.2 and ∼58% for the ES versions. These results indicate that ES should primarily be used in conjunction with parsimony starting trees.

In addition to the plausibility assessment shown in Figure 4.4, I also jointly compared the ML trees resulting from both starting tree types (scheme II). For this joint compar-

ison, I performed a plausibility analysis analogous to that in Figure 4.4, but I now pool together all 80 ML trees obtained from the two starting tree types, across the four tool versions. As already mentioned, this comparison approach is effectively equivalent to having executed each RAxML-NG version using 10 random and 10 parsimony starting trees, which constitutes the default setting in RAxML-NG v1.2. The results from this comparison are depicted in Figure 4.5, and they show negligible differences to those in Figure 4.4. In fact, I present Figure 4.5 in juxtaposition to Figure 4.4, to illustrate the (only) marginal advantage induced by using random starting trees in combination with ES. For those MSAs where parsimony starting tree inferences failed to infer at least one (out of 10) plausible ML tree with the ES versions, random starting tree inferences did yield only a single (out of 10) plausible ML tree on a single protein MSA (22200_1). The observation for this specific dataset holds for all three ES versions. That is, none of the ES versions inferred plausible ML trees when exclusively using 10 parsimony starting trees, whereas all inferred a single plausible ML tree when they additionally used 10 random starting trees. Furthermore, the plausible ML tree is obtained for the exact same random starting tree across all three versions, suggesting that each version likely followed the same tree search trajectory.

Figure 4.5 also includes the proportion of plausible trees across *all* inferences for each method, MSA type (DNA or AA), and starting tree type (parsimony or random), that is calculated as follows:

$$P_{acc} = \frac{N_{pl}}{10N_{MSAs}} \cdot 100\% \tag{4.3}$$

Overall, both comparisons in Figures 4.4 and 4.5 show that: (i) as expected, optimization is more challenging (resulting in a lower number of plausible trees) when starting from random trees, across all methods; (ii) the difference is more pronounced for the three ES versions than for standard RAxML-NG; (iii) for ES, the difference is more pronounced for DNA than for AAs, as expected due to the limitations of parsimony for protein MSAs, which is rarely used for such data, unlike DNA [184].

Runtime Improvements

Figure 4.6 illustrates the speedup distributions across the empirical MSAs for ES versions compared to RAxML-NG v1.2, as well as the runtime improvements of the KH-based versions relative to sRAxML-NG. The figure presents speedups for both parsimony and random starting trees. The highest speedups are observed with parsimony starting trees, reinforcing the preference for parsimony starting trees when using ES. Specifically, in conjunction with sRAxML-NG, the average speedups of the KH-simple and KH-mult versions on parsimony starting trees over RAxML-NG v1.2 are 4.7× and 5× for DNA MSAs, and 3.6× and 3.9× for AA MSAs, respectively. With random starting trees, however, speedups are substantially lower at 2.6× and 2.8× for DNA MSAs, and 2.2× and 2.3× for AA MSAs, respectively. A substantial portion of the overall speedup can be attributed to the sRAxML-NG version itself, with KH-based versions providing additional runtime improvements, ranging from 10% to 50%. Specifically, on parsimony starting trees, the KH version is 29% and 35% faster than sRAxML-NG on DNA and AA MSAs, respectively, while the KH-mult version is 36% and 49% faster, respectively.

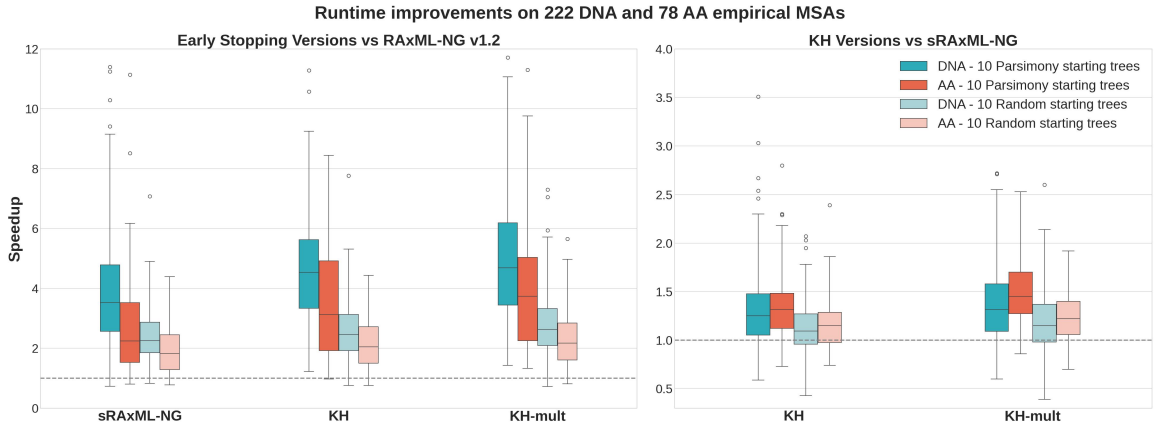


Figure 4.6: Speedup distributions of ES versions relative to RAxML v1.2 (left subfigure), as well as of the KH-based versions relative to sRAxML-NG (right subfigure), on **222 DNA and 78 AA large empirical MSAs.** The speedups refer to strictly sequential runtimes for either 10 parsimony or 10 random starting trees. The dashed line at the bottom of each subfigure corresponds to a speedup of 1× (Figure reproduced from [211]).

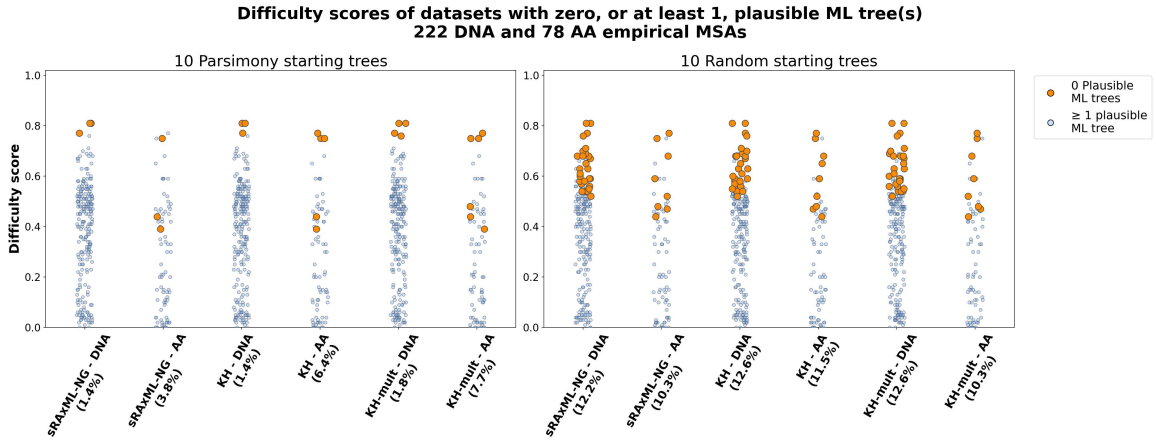


Figure 4.7: Stripplot illustrating correlation between the increased MSA difficulty scores and the inability of ES versions to infer at least one plausible ML tree, on 222 DNA and 78 AA large empirical MSAs. Each point represents a single dataset, with difficulty scores shown on the y-axis. The columns on the x-axis represent specific combinations of ES versions (sRAxML-NG, KH, and KH-mult) and dataset types (DNA, AA). The percentage next to each x-label indicates the fraction of MSAs for which not a single plausible ML trees could be inferred for the corresponding combination. The left and right subfigures display results for parsimony and random starting trees, respectively (scheme I). Datasets (points) where no plausible ML trees were inferred are highlighted more emphatically, while those where at least one (out of 10) plausible ML tree was inferred are indicated by lighter marks (Figure reproduced from [211]).

Set	Datasets	Count
IFP	22200_1 (AA), 15485_0 (DNA), 21973_10 (DNA), 21973_4 (DNA), 11762_4 (AA), 10436_9 (AA)	6
$FP_{sRAxML-NG} - IFP$	$\emptyset$	0
$FP_{KH} - IFP$	11762_7 (AA), 11762_6 (AA)	2
$FP_{KH-mult} - IFP$	22386_0 (DNA), 11762_7 (AA), 12097_1 (AA), 11762_6 (AA)	4

Table 4.1: Datasets on which ES versions inferred 0 (out of 10) plausible ML trees, when independent searches were initiated on 10 parsimony starting trees. $FP_{sRAxML-NG}$: Unsuccessful MSAs for the sRAxML-NG version; FP_{KH}: Unsuccessful MSAs for the KH-simple version; $FP_{KH-mult}$: Unsuccessful MSAs for the KH-mult version; IFP: Intersection of the above three sets, i.e., $IFP = FP_{sRAxML-NG} \cap FP_{KH} \cap FP_{KH-mult}$; $\big(FP_{sRAxML-NG} - IFP\big)$, $\big(FP_{KH} - IFP\big)$, $\big(FP_{KH-mult} - IFP\big)$: Set differences (Table reproduced from the Supplementary Material of [211]).

Unsuccessful MSAs vs. Difficutly

Returning to the first comparison framework (Figure 4.4; scheme I), there appears to be a correlation between higher Pythia scores in DNA MSAs and the inability of ES methods to infer at least one plausible ML tree (out of 10), irrespective of starting trees (random vs. parsimony). The stripplot in Figure 4.7 indicates that all DNA MSAs, for which no plausible trees were inferred by the ES versions, have difficulty scores above 0.5. For AA MSAs, however, this trend is less pronounced. Interestingly, the sets of MSAs for which ES methods failed to yield a single plausible tree are almost identical across all three versions. I refer to those MSAs as *unsuccessful MSAs* and report them in Tables 4.1 for parsimony, and 4.2 for random starting trees, respectively. The fact that the unsuccessful MSA sets are almost identical among the ES versions suggests that sRAxML-NG may have been over-simplified for such challenging datasets, which Pythia predicts to be difficult. KH-based approaches are not, or only marginally challenged by this case. Therefore, improving the thoroughness of sRAxML-NG is expected to improve the performance of all three ES versions on difficult datasets.

Progressive Increase of SPR-regrafting Radius

These additional experiments represent an attempt to improve the thoroughness of the sRAxML-NG heuristic, and to assess whether this improves the performance of the ES versions on difficult MSAs. The results are not entirely satisfactory. However, it is important to mention that sRAxML-NG already performs decently, and hence served as the starting point for the design of the RAxML-NG v2.0 heuristic, which I present in Chapter 6. The results reported below provide an initial indication of how sRAxML-NG could be improved, namely by increasing the SPR regrafting radius, which yields a noticeable performance improvement on DNA MSAs.

I incrementally increased the maximum SPR-regrafting radius parameter (r_{max}; see Section 2.6.4) of the sRAxML-NG heuristic in FAST and SLOW rounds, by setting the radius to $r_{max} \in \{12, 14, 16, 18, 20\}$. This is achieved by using the `--spr-radius X` option on the invocation command of ES versions. Note that the default SPR radius in sRAxML-

Set	Datasets	Count
IFR	11762_3 (AA), 18448_6 (DNA), 18883_4 (DNA), 18753_0 (DNA), 12003_0 (DNA), 26377_0 (DNA), 11902_0 (DNA), 14244_0 (AA), 19838_0 (DNA), 21973_4 (DNA), 19048_3 (DNA), 11762_4 (AA), 18577_0 (DNA), 11762_7 (AA), 17406_0 (DNA), 19048_2 (DNA), 17984_2 (DNA), 21973_10 (DNA), 22386_0 (DNA), 15847_0 (AA), 16630_0 (DNA), 10436_9 (AA), 11552_2 (DNA), 12097_0 (AA), 15485_0 (DNA), 13670_2 (DNA), 13670_0 (DNA), 12097_1 (AA), 11395_5 (DNA), 25268_0 (DNA), 17984_0 (DNA), 10707_0 (DNA), 27988_2 (DNA), 18330_0 (DNA)	34
$FR_{sRAxML-NG} - IFR$	24078_0 (DNA)	1
$FR_{KH} - IFR$	10986_0 (AA), 24078_0 (DNA), 24324_0 (DNA)	3
$FR_{KH-mult} - IFR$	20724_0 (DNA), 24324_0 (DNA)	2

Table 4.2: Datasets on which ES versions inferred 0 (out of 10) plausible ML trees, when independent searches were initiated on 10 random starting trees. $FR_{sRAxML-NG}$: Unsuccessful MSAs for the sRAxML-NG version; FR_{KH}: Unsuccessful MSAs for the KH version; $FR_{KH-mult}$: Unsuccessful MSAs for the KH-mult version; IFR: Intersection of the above three sets, i.e., $IFR = FR_{sRAxML-NG} \cap FR_{KH} \cap FR_{KH-mult}$; $\big(FR_{sRAxML-NG} - IFR\big)$, $\big(FR_{KH} - IFR\big)$, $\big(FR_{KH-mult} - IFR\big)$: Set differences (Table reproduced from the Supplementary Material of [211]).

NG is $r_{max} := 10$ (see Section 4.2.1). The hypothesis is that the increase of the regrafting radius enhances the thoroughness of the sRAxML-NG heuristic, thereby enabling it to more extensively explore the tree space and to identify better topologies.

I tested this hypothesis on sRAxML-NG and KH-mult versions. These additional experiments were conducted on the union of unsuccessful MSAs on either of the two versions, using both starting tree types. As a result, some MSAs classified as unsuccessful may correspond to only one version, while others are shared between the two (see Tables 4.1 and 4.2). Furthermore, on certain datasets, ES versions failed to infer plausible trees under both random and parsimony starting trees, whereas on others, unsuccessful MSAs occurred for one starting tree type only. Overall, I tested 10 MSAs using parsimony starting trees and 37 MSAs using random starting trees. As expected, the number of unsuccessful MSAs is higher for random starting trees.

The plausibility assessment results for the experiment described above are presented in Figure 4.8. Increasing the r_{max} parameter appears to have a notable impact on DNA MSAs when using random starting trees. In contrast, the behavior on AA data is less consistent, showing no clear relationship between larger SPR radii and plausibility, nor a strong correlation between difficulty scores and the inability to infer plausible trees.

Increasing the SPR radius does not appear to substantially impact the sequential

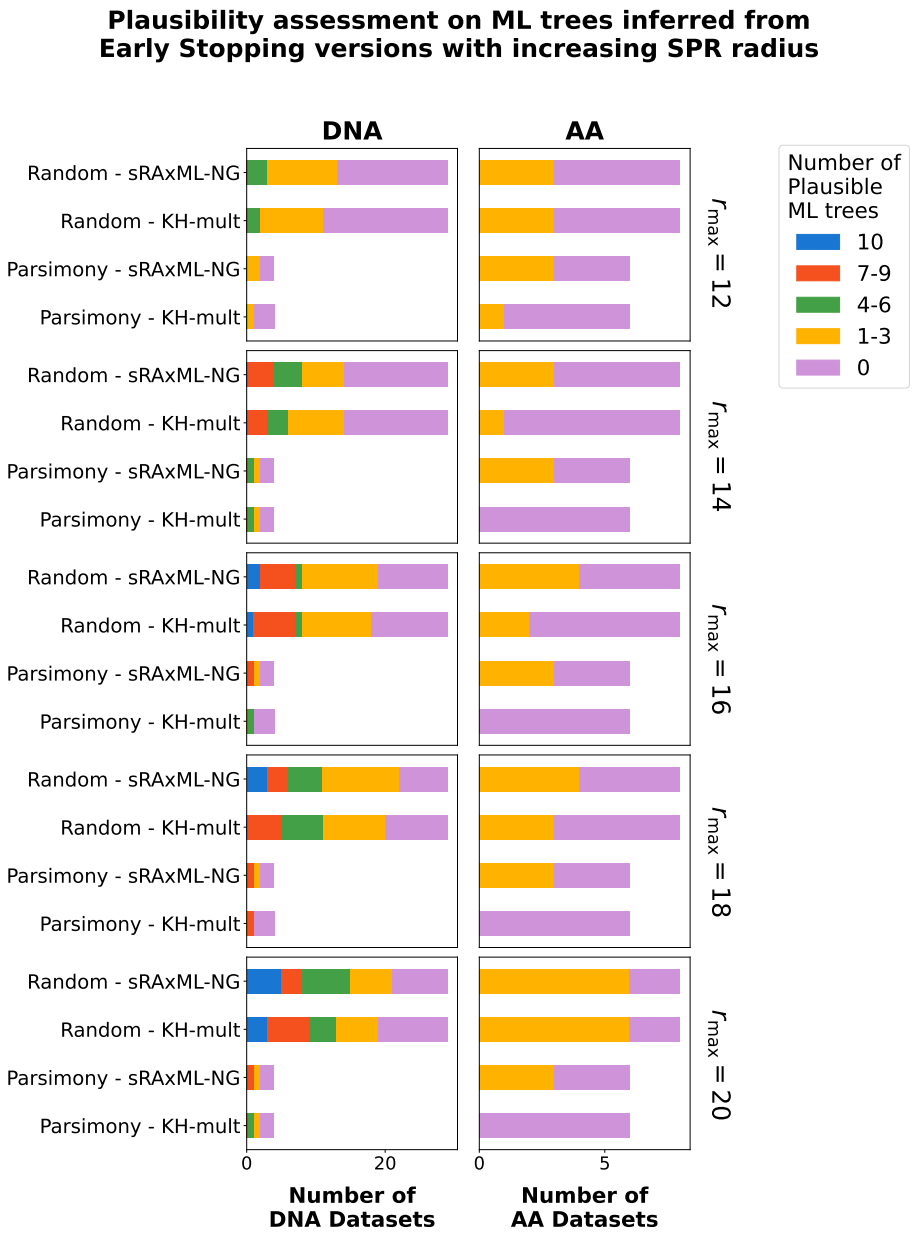


Figure 4.8: Plausibility assessment (scheme I) for the ML trees inferred from sRAxML-NG and KH-mult versions, with progressively increasing maximum SPR-regrafting radius parameter (r_{max}), taking values from the set $r_{max} \in \{12, 14, 16, 18, 20\}$. For each radius value and starting tree type (parsimony or random), the sRAxML-NG and KH-mult versions conduct 10 independent tree inferences. The 20 inferred trees from both versions with the same starting tree type are compared against the best ML tree—inferred from RAxML-NG v1.2—for the corresponding dataset and using the same starting tree type. The best ML tree was derived from the analysis depicted in Figure 4.4. For each starting tree type and radius value, I pool together the 10 ML trees inferred from sRAxML-NG, the 10 inferred from KH-mult, and the best ML tree (21 trees in total), and compare them using the AU test implemented in the CONSEL tool, considering ML trees with a p-value ≥ 0.05 as plausible (Figure reproduced from the Supplementary Material of [211]).

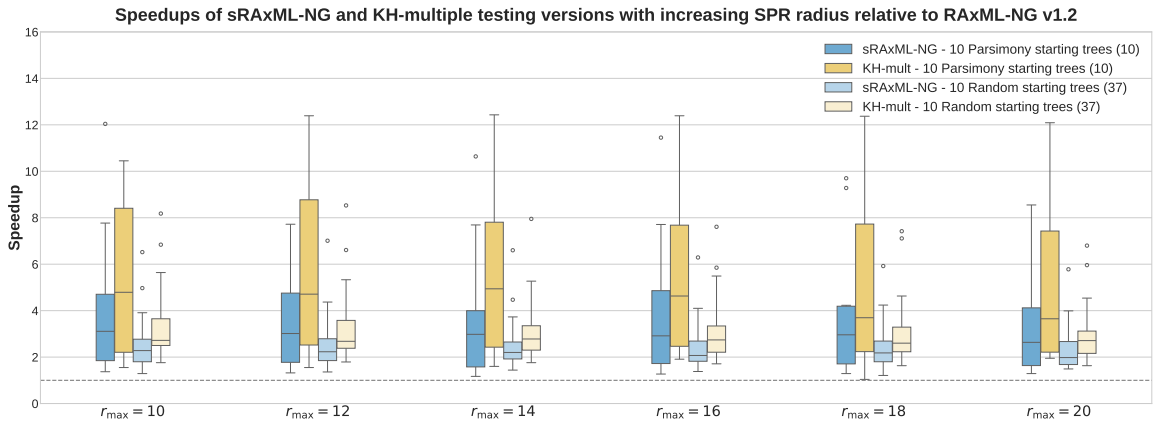


Figure 4.9: Speedup distributions for sRAxML-NG and KH-mult versions with progressively increasing maximum SPR-regrafting radius values (r_{max}), taking values from the set $r_{max} \in \{10, 12, 14, 16, 18, 20\}$. The distinct boxplot columns correspond to the different r_{max} values, while the boxplots within each column represent different versions (sRAxML-NG or KH-mult) and starting tree types (parsimony or random) (Figure reproduced from the Supplementary Material of [211]).

execution times on the selected, unsuccessful MSAs. Figure 4.9 shows the speedup distributions for the sRAxML-NG and KH-mult versions with progressively increasing r_{max}, compared to RAxML-NG v1.2. In this Figure, each boxplot column represents a distinct r_{max} value, while the four boxplots within each column differ by version (sRAxML-NG or KH-mult) and starting tree type (parsimony or random). No substantial changes in speedup distributions are observed as r_{max} increases. However, given the relatively small number of tested MSAs (10 for parsimony starting trees and 37 for random starting trees), these speedup measurements are less representative compared to the speedups derived from the full dataset (Figure 4.6).

4.4.2 Results on 1,076 Simulated MSAs

In this Section, I present benchmarking results for the ES versions on 1,076 simulated MSAs. Simulated MSAs are generally easier to analyze and tend to converge faster, due to their clearer and stronger phylogenetic signal [214]. Thus, I provide an overview of the plausibility assessment and runtime improvement plots without extensive numerical details. The primary focus of this section is on the topological accuracy plots that are presented at the end.

Figure 4.10 shows the plausibility assessment results on simulated MSAs, while Figure 4.11 shows the speedup distributions plot, in analogy to the respective Figures for empirical MSAs in Section 4.4.1 (Figures 4.4 and 4.6). In the plausibility assessment, all ES versions appear to infer high-quality trees on simulated MSAs when using parsimony starting trees. In terms of speedups, inferences starting with parsimony trees yield substantially higher average speedups than those starting with random ones.

Regarding topological accuracy, I compare the best ML trees inferred from all versions with the reference tree, using the RF distance metric [155]. Here, the reference tree is the "true" tree that was used to generate the simulated sequences. Figure 4.12 illustrates the distributions of RF distances between the best ML trees inferred from each RAxML-NG version and the reference tree topologies, when parsimony or random starting trees

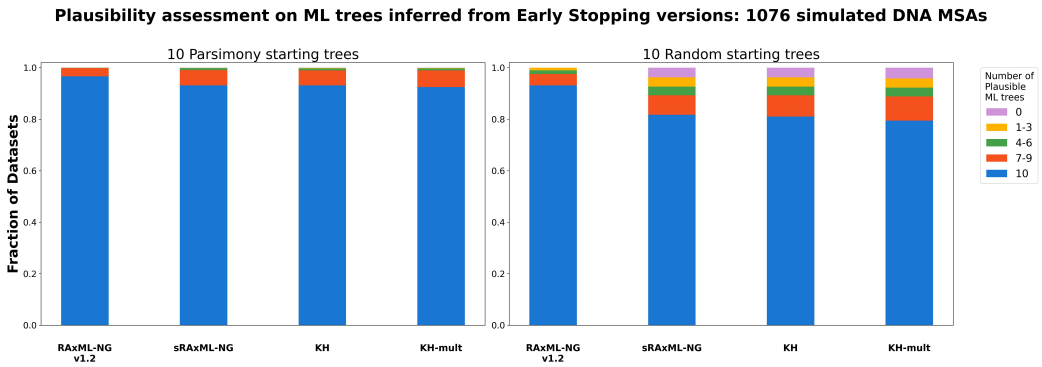


Figure 4.10: Plausibility test results (scheme I) for ML trees inferred by RAxML-NG v1.2 and the ES versions across 1,076 simulated DNA MSAs. For each dataset, all RAxML-NG versions conduct 10 independent tree inferences using the same set of parsimony (left subfigure) or random (right subfigure) starting trees. I evaluated the inferred ML trees using the AU test (over 40 trees, for each type of starting trees) implemented in the CONSEL tool, considering ML trees with a p-value ≥ 0.05 as plausible (Figure reproduced from the Supplementary Material of [211]).

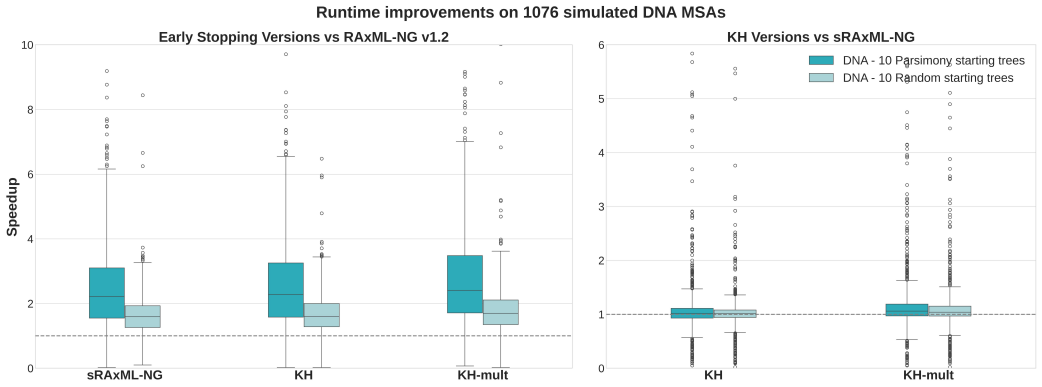


Figure 4.11: Speedup distributions of the ES versions relative to RAxML v1.2 (left subfigure), and of the KH-based versions relative to sRAxML-NG (right subfigure), on **1,076 simulated DNA MSAs**. The speedups refer to runtimes measured on sequential executions using 10 parsimony or 10 random starting trees. The dashed line at the bottom of each subfigure corresponds to a speedup of 1× (Figure reproduced from the Supplementary Material of [211]).

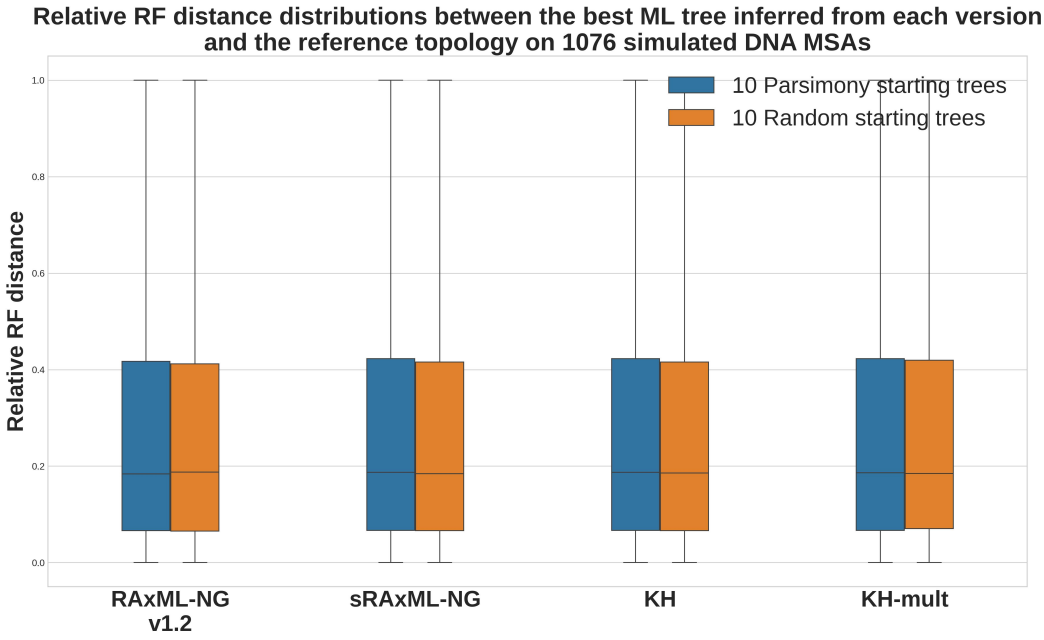


Figure 4.12: Relative RF distance distributions between the best ML trees inferred from each RAxML-NG version and the corresponding reference tree topologies, where each version conducted 10 independent tree inferences on 1,076 simulated DNA MSAs, using parsimony or random starting trees. The reference tree topology is the "true" tree used to generate the sequences (Figure reproduced from the Supplementary Material of [211]).

are used. The plot demonstrates that, under simulated data, there is no substantial difference in the distributions of relative RF distances between RAxML-NG v1.2 and the ES versions, in terms of topological accuracy. In fact, all RF distance distributions exhibit the same average value of 0.27, for both parsimony and random starting trees.

4.5 Discussion

I introduced Early Stopping (ES) criteria for ML-based phylogenetic inference and implemented them in RAxML-NG. The goal was to reduce computing time while maintaining inference accuracy. By integrating the KH test, and extending it via multiple testing correction, RAxML-NG can dynamically adapt the log-likelihood improvement ε-threshold in a dataset- and inference-specific manner. More specifically, the KH-based stopping criteria are built upon a simplified tree search heuristic, sRAxML-NG, which constitutes an important by-product of this study and already implements an initial form of ES. The experiments suggest that the standard RAxML-NG search strategy may be unnecessarily complex, especially on easy datasets.

Benchmark results on 300 large, empirical MSAs show that sRAxML-NG using the KH-multiple testing criterion (KH-mult version), when initiated with 10 parsimony starting trees, yielded plausible ML output trees for 98% of the DNA datasets and up to 96% of the AA datasets, compared to RAxML-NG v1.2. Under the same starting tree configuration, the KH-mult version achieved an, on average, 5× speedup for DNA datasets and a 3.9× speedup for AA datasets relative to RAxML-NG v1.2. As expected, these average

speedups are even more pronounced when comparing the ES versions using 10 parsimony starting trees, against the standard RAxML-NG v1.2 execution (using 10 parsimony and 10 random starting trees): 7.8× for sRAxML-NG, 9.6× for KH-simple, and 10.2× for KH-mult versions.

In contrast, random starting trees provide little to no benefit when combined with ES. Among the datasets where ES methods failed to infer a plausible ML tree when using parsimony starting trees alone, the addition of 10 random starting trees resulted in obtaining a plausible ML tree for only a single AA MSA. RAxML-NG v1.2 successfully inferred a plausible ML tree in all instances when only using parsimony starting trees, raising questions about the general utility of random starting trees in standard heuristics.

Based on these findings, as well as the results reported in Liu *et al.* [122], several conclusions can be drawn. First, random starting trees should be avoided when using ES. Therefore, I recommend executing the ES versions using parsimony starting trees only (ideally more than one). The Pythia score [73] is a good indicator for selecting the appropriate number of starting trees, as already discussed in Chapter 3. On easier datasets, fewer parsimony starting trees suffice, whereas more challenging datasets require a higher number. Further, random starting trees might be useful for difficult MSAs when the primary objective is to maximize the log-likelihood score via a thorough heuristic (e.g., RAxML-NG v1.2). In such cases, however, practitioners should keep in mind that the phylogenetic signal is typically weak, and over-optimizing the log-likelihood score is unlikely to yield meaningful biological insights.

The sRAxML-NG heuristic serves as a proof-of-concept approach to show that the default RAxML-NG heuristic can be substantially simplified. In fact, this constituted my starting point for redesigning the RAxML-NG v2.0 heuristic, which I present in Chapter 6. Analogous to sRAxML-NG, RAxML-NG v2.0 eliminates the AUTODETECT SPR rounds (see Section 2.6.5), and therefore only comprises two sequences of SPR rounds, FAST and SLOW. The default RAxML-NG v2.0 heuristic is tuned to be rather more thorough than sRAxML-NG, owing to extensive empirical evaluations. Random starting trees are still being used, but their number is substantially reduced compared to earlier versions.

Importantly, RAxML-NG v2.0 now features an alternative, very fast heuristic for rapid phylogenetic reconstruction. This fast heuristic is based upon the KH-mult version, but uses a single parsimony starting tree, and can be invoked via the `--fast` option. It is particularly well suited for large-scale phylogenetic analyses involving thousands of taxa, e.g., for analyzing viral MSAs [133]. The benchmarks presented in Chapter 6 demonstrate that this novel, very fast heuristic outperforms both FastTree and IQ-TREE's fast mode [227], with respect to both accuracy and runtime. Overall, RAxML-NG's fast heuristic is now even faster than the gold-standard tool for speed, that is FastTree, and provides a more reliable alternative for large-scale ML phylogenetic inference.

4.6 Data and Software Availability

The Early Stopping criteria are implemented in RAxML-NG and are available under GNU GPL on GitHub at `https://github.com/togkousa/raxml-ng/tree/stopping-criteria`. A code-snapshot of the ES versions that was used to generate the above results has been uploaded to Zenodo: `https://doi.org/10.5281/zenodo.14653326`. All datasets used are available for download from the Dryad Digital Repository: `https:`

//doi.org/10.5061/dryad.8gtht76zz.

5. Investigation of overfitting in ML phylogenetic inference

This Chapter is based on the following preprint, which is currently submitted under peer-review:

- **Anastasis Togkousidis**, Olivier Gascuel, and Alexandros Stamatakis. "A Systematic Investigation of Overfitting in Maximum Likelihood Phylogenetic Inference." *biorxiv*, 2025. `https://doi.org/10.1101/2025.10.07.680876`

Contributions: Anastasis Togkousidis designed the experimental pipeline for the overfitting analysis, and conducted the statistical evaluation of overfitting trends across widely used ML phylogenetic inference tools, namely RAxML-NG, IQ-TREE, and FastTree. To successfully perform the overfitting analysis, A.T. modified the source code of those tools. He also designed and implemented the Holdout-Validation (HV) version of RAxML-NG, and conducted benchmarks against the standard and Early Stopping (ES) versions of RAxML-NG. Olivier Gascuel and Alexandros Stamatakis supervised the project and helped with the writing of the paper.

MSA sequences are inherently noisy (see Section 2.9). This intrinsic noise increases the risk that ML-based phylogenetic inference tools may experience overfitting, whereby the inferred topology incorrectly models noise alongside the true phylogenetic signal. Within the ML phylogenetic context, overfitting has been extensively discussed in the literature with respect to substitution model selection. Numerous methods have been proposed to select the optimal model that (a) is not over-parametrized, and (b) is sufficiently supported by the data at hand. However, the role of the tree topology as a potential source of overfitting has received little attention, despite being the biologically most important parameter.

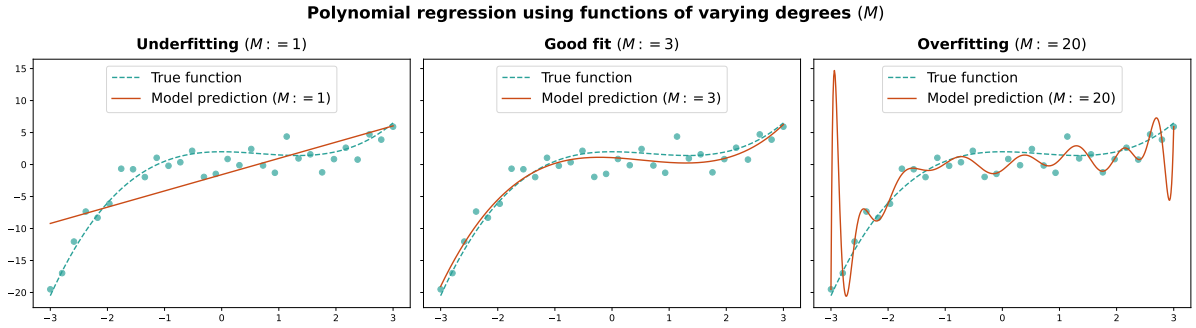
In this Chapter, I specifically investigate overfitting trends in ML tools with respect to the tree topology parameter. I begin by motivating this central research question, and explain its importance. To this end, I also restate the preliminary material introduced in Section 2.10 concerning overfitting in statistical inference. This content repetition is intentional: it allows me to draw a direct analogy between training in deep neural networks (DNNs) and the optimization process in ML phylogenetic tree inference. I juxtapose these two frameworks in order to clarify the analogy for the reader. Next, I examine topological overfitting using two complementary approaches. The first one statistically evaluates overfitting trends, to determine whether the ML tools—particularly the thorough ones—are prone to overfitting. The second approach involves benchmarking a Holdout-Validation (HV) version of RAxML-NG, which is analogous to overfitting mitigation strategies used in deep learning.

5.1 Background

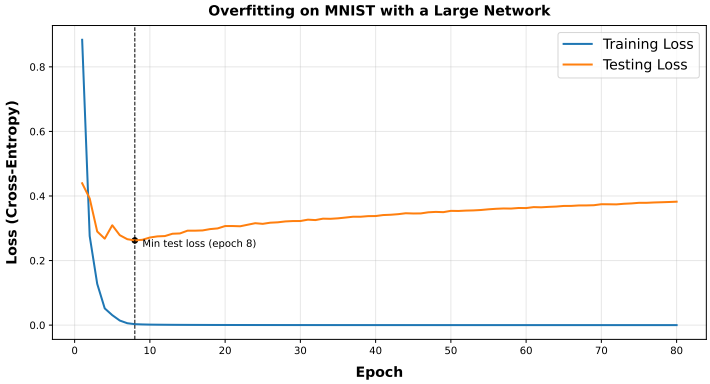
In machine learning, the term *training* refers to the process of estimating the parameters of a statistical model, such that they optimally fit the input data according to a given criterion/loss-function. For instance, in deep neural networks (DNNs), training typically involves iterative methods [162], which efficiently adjust the DNN weights (parameters) to progressively minimize prediction errors. *Overfitting* occurs when the model that is being trained does not solely capture the underlying pattern, but also the inherent noise in the data [62]. It is often a consequence of *over-training* an *over-parameterized* model (see Figure 5.1), relative to the size and the intrinsic properties of the input data [17, 87]. However, extensive training of the model, despite strong indications of convergence in the training score/error, does not necessarily induce overfitting [12, 234]. Whether or not overfitting occurs heavily depends on the statistical model and the specific dataset being analyzed.

ML-based phylogenetic tree inference aims to identify the unrooted binary tree that maximizes the likelihood of observing the input MSA sequences (i.e., the MSA sites) at the tip nodes of the inferred tree (see Section 2.5). This constitutes a standard statistical inference problem. Under ML, the statistical model being optimized comprises the binary tree topology, its branch lengths, and the substitution model parameters, that jointly describe the stochastic process under which the input MSA sites evolved. Importantly, the MSA sites correspond to the independent input data points of the ML framework. That is, each MSA site represents a distinct data point, and is analogous to a single point-dot in the regression example of Figure 5.1a. Heuristics strive to estimate the statistical model parameters, by optimizing the tree topology (a discrete parameter) and the continuous parameters (branch lengths and the substitution model parameters [228, 230]).

A key analogy between ML tree inference heuristics and deep learning methods is that they both employ iterative algorithms to optimize the model-fit to the input data, either by minimizing the error in DNNs, or by maximizing the log-likelihood score under ML. A second analogy is that both DNNs and likelihood-based phylogenetic models (discussed in the next paragraph) constitute parameter-rich models. As shown in the example of Figure 5.1, parameter-rich models are susceptible to overfitting. In the polynomial regression example of Figure 5.1a, overfitting occurs with a high degree polynomial ($M := 20$, where M indicates the polynomial degree). In contrast, a cubic-model ($M := 3$) suffices to capture the underlying pattern. Under overfitting ($M := 20$), the predictive accuracy on unseen (testing) data is suboptimal, as the model also fits the training data's noise on top of the true signal. This example directly applies to DNNs, which are used both for regression and classification problems. DNNs generally constitute parameter-rich models that comprise thousands or millions of parameters/weights [132]. The DNN parameter/weights initially have random values, and they are adjusted via successive training iterations. As shown in Figure 5.1b, overfitting typically emerges *during* training: after a certain point (iteration/epoch 8 in the example), the DNN begins to model the intrinsic noise in the training data, and therefore, its performance on unseen (testing) data deteriorates. A common technique to mitigate overfitting in deep learning is to apply holdout-based early stopping [186]. Under this approach, the input dataset is randomly split into model-training and validation data. During training, the model—with the currently inferred weights—is periodically evaluated on the validation set, and optimization halts once the validation error starts to increase.



(a)



(b)

Figure 5.1: (a) Numerical examples of underfitting, a good fit, and overfitting in polynomial regression. The example shows how intrinsic noise in the data may induce overfitting under parameter-rich models. The data points were generated via a cubic function (indicated as "True function" in the Figure legends) to which I injected Gaussian noise. I regress the input data using three polynomial functions of varying degrees (M). The linear model ($M := 1$) fails to capture the nonlinear data pattern and therefore underfits. The cubic model ($M := 3$) effectively captures the true pattern and yields a good fit. The parameter-rich model ($M := 20$) incorrectly captures noise alongside the true signal, resulting in overfitting. **(b) Numerical example of overfitting emerging during the training of a parameter-rich DNN.** The plot corresponds to empirically derived learning curves obtained from training a DNN on a small random subsample (1,600 and 400 data points for training and testing, respectively) of the MNIST dataset [117]. The DNN has four fully connected layers with widths of 784, 1024, 1024, and 10. I train the DNN via the Adam optimizer [105] but disable any form of regularization or early stopping. After iteration/epoch 8, the testing loss function progressively increases and diverges from the training one, which continues to decrease and eventually attains zero loss (Figure reproduced from [209]; subfigure **(b)** is also included in Section 2.10).

In analogy, likelihood-based phylogenetic tree models are also parameter-rich. Substitution models typically contain (only) a small number of free parameters (e.g., 9 for GTR+Γ4 [205]), and numerous methods/software have been proposed for selecting the best model [34, 119, 202] based on the AIC, [4], or the BIC [173] scores. Related work also addresses the risk of over-parameterization in profile mixture models [10]. The role of the *tree topology*, though, as a potential source of overfitting has received little attention, despite being the most biologically meaningful parameter. The phylogenetic tree

involves one parameter/length per branch (e.g., ∼2,000 edges in a tree comprising 1,000 tips), plus the tree topology itself that forms part of a vast, super-exponentially large space. ML tree inference, similar to DNN training, proceeds in discrete iterations, which correspond to single topological updates (see below). ML tools initiate tree searches either from a random or parsimony-based tree, and gradually improve with respect to the branch-lengths and tree topology. The process terminates when they reach a local likelihood optimum. The likelihood is computed on the input MSA, which is analogous to the training set in a DNN.

As a result, phylogenetic models may also become over-parameterized regarding the size, the amount of noise, and intrinsic properties of the input data [81]. On the other hand, ML inference and deep learning also exhibit a fundamental difference. In a fixed DNN graph, model parameters are continuous and typically homogeneous in nature and significance, whereas under ML, the discrete tree topology is biologically more important than the remaining parameters. Despite that, relatively few studies have explicitly addressed topological overfitting in ML tree inference [14, 218], that is, the tree topology itself as a potential source of overfitting. For instance, Berry *et al.* [14] evaluate overfitting within a static framework, by applying post-hoc overfitting-detection criteria to trees that had already been inferred. They argue that weakly supported branches may indicate over-parametrization, and they propose methods to collapse such branches into multifurcations. While this approach to mitigating over-parametrization is useful, it is orthogonal to the focus of this Chapter (see below). I nevertheless revisit the issue of overfitted branches in the Discussion section (see Section 5.5).

The analysis presented in this Chapter builds primarily on the study by de Vienne *et al.* [218], which—although published as a preprint only—served as a key reference point. Here, I also identify and discuss certain limitations of the latter study. Analogous to the approach in de Vienne *et al.*, I assume that ML tools operate on fully resolved bifurcating trees, which may indeed be over-parameterized given the input data, and examine the dynamics by which overfitting (potentially) emerges *during* the tree inference process. The question of interest is whether iterative heuristics implemented in ML tools reach a turning point during the optimization process, that is determined by the phylogenetic signal-to-noise ratio of the input MSA, beyond which, further iterations merely and increasingly fit noise (as in Figure 5.1b). I define iterations based on rudimentary updates to the current best-scoring tree topology; that is, each newly accepted bifurcating tree corresponds to a distinct iteration. My approach resembles dynamic overfitting detection in DNN training, as well as in other parameter-rich models (Figure 5.1b). On the other hand, because ML tools maximize the log-likelihood score, such a degradation manifests itself as a *decrease* in likelihood on unseen (testing) sites, in contrast to DNNs, where overfitting is observed as an *increase* in prediction error (see Figure 5.1b). Another major difference is that ML tools optimize in a space that is both discrete (topology) and continuous (branch lengths) [16]. As a consequence, the log-likelihood training/testing curves are inherently "bumpy" (Figure 5.3), since topological changes may induce abrupt likelihood increases. Hence, the curves are less smooth and thus more difficult to analyze.

I address overfitting via two complementary approaches. In the first part, I investigate systematic overfitting trends in three widely used phylogenetic inference tools: RAxML-NG [111], IQ-TREE [131], and FastTree [151]. In this comparison I also include the Early Stopping (ES) version of RAxML-NG [211], which I presented in Chapter 4. The term "systematic" indicates that I specifically investigate whether overfitting constitutes *the* statistically expected behavior of ML tools. This question is most relevant for RAxML-

NG and IQ-TREE, which deploy thorough tree search heuristics, whereas FastTree and RAxML-NG ES constitute faster alternatives. The main outcome of this analysis is that topological overfitting does not constitute a major issue in any of these tools.

The second part aims to confirm the conclusions of the first, and reinforces my findings. In this part, I benchmark a site-based holdout validation (HV) version of RAxML-NG that is analogous to approaches employed in deep learning. HV shows inferior performance relative to standard RAxML-NG, which conducts inferences on the entire MSA, even on long MSAs. This result is consistent with the findings from the first part of the study, namely that topological overfitting does not constitute a major issue. It also shows that the standard overfitting mitigation technique used in deep learning (HV) does not benefit ML-based tree inference.

5.2 Analysis

In this Section, I describe the methodology used to assess topological overfitting in ML phylogenetic inference tools. I provide a general overview of the implemented methods along with the experimental design and evaluation setup. Specifically, Section 5.2.1 outlines the pipeline used to evaluate systematic overfitting trends in RAxML-NG (standard and ES versions), IQ-TREE, and FastTree. Section 5.2.2 provides an overview of the holdout-validation (HV) version of RAxML-NG and describes its benchmarking pipeline. The results of the two complementary approaches are presented in Sections 5.4.1 and 5.4.2, respectively. I provide a more detailed description of the implemented methods in Section 5.6.

5.2.1 Statistical Evaluation of Topological Overfitting

A schematic illustration of the pipeline described below is shown in Figure 5.2. I employ a 10-fold Monte-Carlo cross-validation approach [146] by randomly splitting each empirical/simulated MSA into training and testing sites with a splitting ratio of 80%–20%. As is common in statistical modeling, the training set fraction is high (80%), so as to yield results that are representative of the entire MSA. On the contrary, the proportion of testing sites (20%) suffices to reliably estimate the cross-validated, unbiased value of the optimization criterion (i.e., the likelihood of unseen testing sites; see below). Importantly, this experiment is repeated 10 times (10-fold cross-validation) to statistically assess the significance of the results.

I conduct ML tree inferences on the training sites of each dataset-split using four tools: RAxML-NG, IQ-TREE, FastTree, and RAxML-NG ES[1] [211]. The ML tree inferences by the four tools mentioned above begin searching tree space from a parsimony starting tree (i.e., a tree with a "good" parsimony score), which I generate with RAxML-NG on the training MSA of the corresponding dataset-split. I use parsimony starting trees because they have relatively good (i.e., non-random), yet still suboptimal, log-likelihood scores [68, 193, 230]. This choice is appropriate, as I intend to assess whether overfitting occurs during the final optimization steps prior to convergence.

In the course of the ML tree inferences, I store *all* intermediate improved topologies that are accepted by each tool. That is, I store topologies that are visited and retained

[1]RAxML-NG ES corresponds to the sRAxML-NG version/heuristic that uses the KH-multiple testing criterion for Early Stopping. For a detailed description, see Chapter 4.

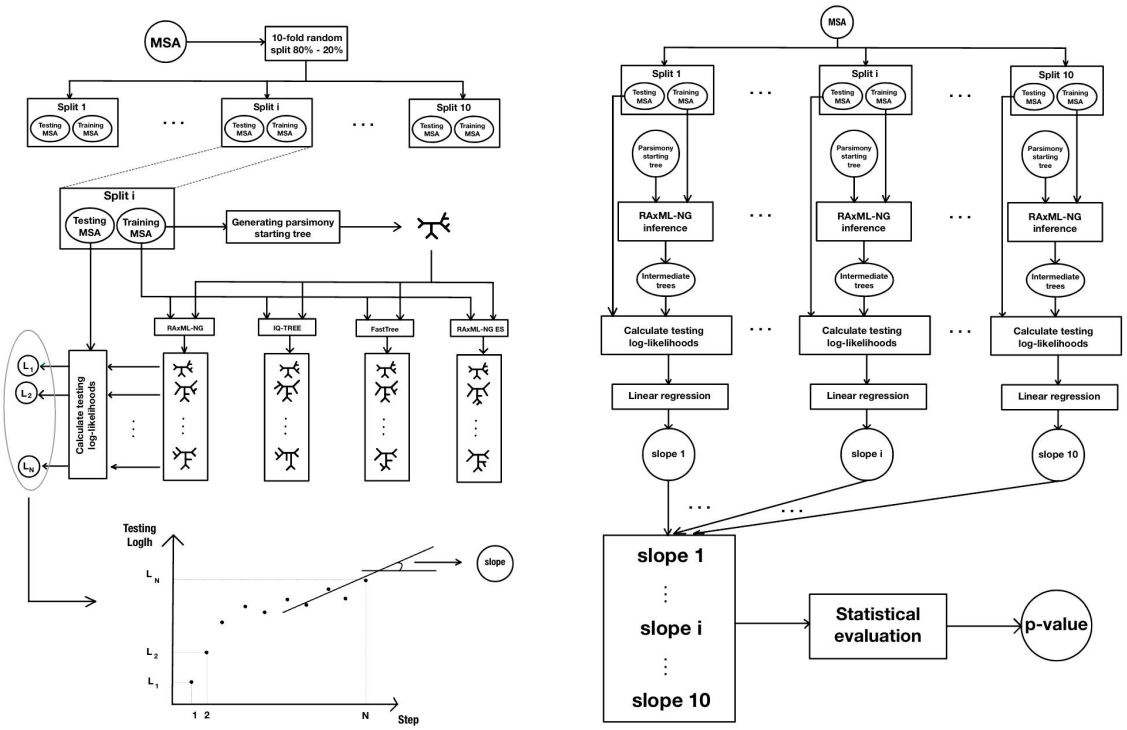


Figure 5.2: A schematic representation of the overfitting evaluation pipeline. (a) Each MSA is randomly partitioned into training and testing sites, applying a 10-fold Monte-Carlo cross-validation approach, with a splitting ratio of 80%–20%. For each dataset-split, RAxML-NG generates a parsimony starting tree from the training sites, which serves as the starting point for ML tools (i.e., RAxML-NG standard and ES, IQ-TREE, and FastTree) to conduct tree inferences on the training sites. During each individual ML inference, the tools store all intermediate improved (w.r.t. the log-likelihood score) topologies. Next, I evaluate the sampled topologies on the testing MSA to devise testing log-likelihood curves. I perform a linear regression on the final points of each testing curve and extract the slope of the fitted line. **(b)** A more abstract overview of the entire process. At the bottom, the 10 slopes of the regression lines (corresponding to the 10 datasets-splits) are collected and statistically assessed using the non-parametric sign test. The Figure illustrates the process for RAxML-NG only, but it applies analogously to IQ-TREE, FastTree, and RAxML-NG ES (Figure reproduced from the Supplementary Material of [209]).

because they improve the overall log-likelihood score. I adopted this comprehensive tree sampling strategy to capture even the smallest topology-induced log-likelihood improvements with respect to the currently best ML tree. This approach does not only increase the resolution of the learning curves. It also allows for assessing and comparing overfitting tendencies among distinct tools, and does not rely on arbitrarily recorded tree topology checkpoints. The challenges involved in implementing this fine-grained tree-sampling approach are described in Methods (see Section 5.6.2). Due to this setup, the training curves of all tools increase monotonically, and each point corresponds to an improved (w.r.t. its log-likelihood score) intermediate topology (see Figure 5.3).

Following the ML tree inferences with the four distinct tools, I subsequently evaluate

the ML score of all sampled topologies via RAxML-NG using the testing MSA to obtain testing curves for each inference and tool. For this evaluation, RAxML-NG performs full branch-length and model parameter optimizations. Thereby, I assess overfitting specifically *with respect to* the tree topology, that is, I focus on the tree topology as parameter of primary biological interest. Importantly, this setup does not assess the convergence dynamics of the sequence of intermediate trees toward some reference best-known tree topology. While I also present such a topological accuracy analysis in the Results (see Section 5.4.1; Figure 5.13), the objective here is different: I investigate the generalizability of each newly introduced topology, by comparing its likelihood on unseen (testing) sites, relative to its predecessor(s).

After constructing the testing curves, I perform linear regression on their final segments and extract the corresponding slope. I explore four alternative final segment definitions: the last two points, and the final 10%, 20%, or 30% of the points on each curve. In the following, I refer to these segment configurations as $\mathsf{n} = 2$, $\mathsf{n} = 10\%N$, $\mathsf{n} = 20\%N$, and $\mathsf{n} = 30\%N$, where n denotes the number of regression points and N the total number of points on the curve[1]. Slopes with an absolute value below 0.1 are set to 0, since such small differences can be attributed to numerical errors [176, 200]. I thereby derive 10 slopes per dataset, tool, and final segment length definition that correspond to the 10 distinct dataset-splits. I statistically assess these slopes via the non-parametric sign test, by applying a 95% confidence level (p-value < 0.05), to obtain an overall per-dataset and per-tool slope trend. For p-values below 0.05 and a majority of slopes being positive, the trend is classified as *Predominantly Positive* (or simply *positive* in the following) as this indicates a consistent increase in the final testing log-likelihoods. Conversely, if the majority of slopes are negative (and the p-value is below 0.05), the trend is classified as *Predominantly Negative* (or *negative*), as this implies a proclivity that the corresponding tool overfits on the specific dataset. If neither of these conditions is met, the overall trend is classified as non-significant. Overall, I systematically evaluate overfitting, that is, I investigate whether overfitting emerges as *the* statistically expected behavior when optimization is pushed to its limits. This question is particularly relevant (a) for RAxML-NG and IQ-TREE, which employ thorough heuristics, and (b) in a comparative context, that is, when comparing overfitting trends in thorough versus faster (RAxML-NG ES, FastTree) heuristics.

Figure 5.3 presents exemplary learning curves for the anecdotal dataset 15034_1 obtained from RAxML-NG inferences. In addition, the Figure displays the regression lines fitted to the final $\mathsf{n} = 30\%N$ points of the testing curves. Compared to the DNN learning curves shown in Figure 5.1b, which illustrate the cross-entropy loss during the DNN training example, the log-likelihood curves (especially the testing ones) are inherently more "bumpy". As discussed below, this behavior is rather expected.

Overall, the null model being used here (H_0 in statistical testing terminology) relies on two assumptions: First, we expect the same behavior (final curve) for the 10 holdout experiments (splits). This is well justified for comparatively long MSAs and supported by the random site selection approach, which ensures that each split constitutes a statistically representative sample of the MSA. Hence, the signal-to-noise ratio should remain unaltered across replicates, leading to analogous final curve behavior. Second, this behavior corresponds to a flat final curve (zero slope) that is perturbed by independent

[1]Note that in Chapter 2 I used the n symbol to denote the number of taxa in an MSA. To avoid ambiguity, in the expressions defining the final curve segments (e.g., $\mathsf{n} = 10\%N$) the n symbol is typeset in a different font and denotes the number of points used for regression.

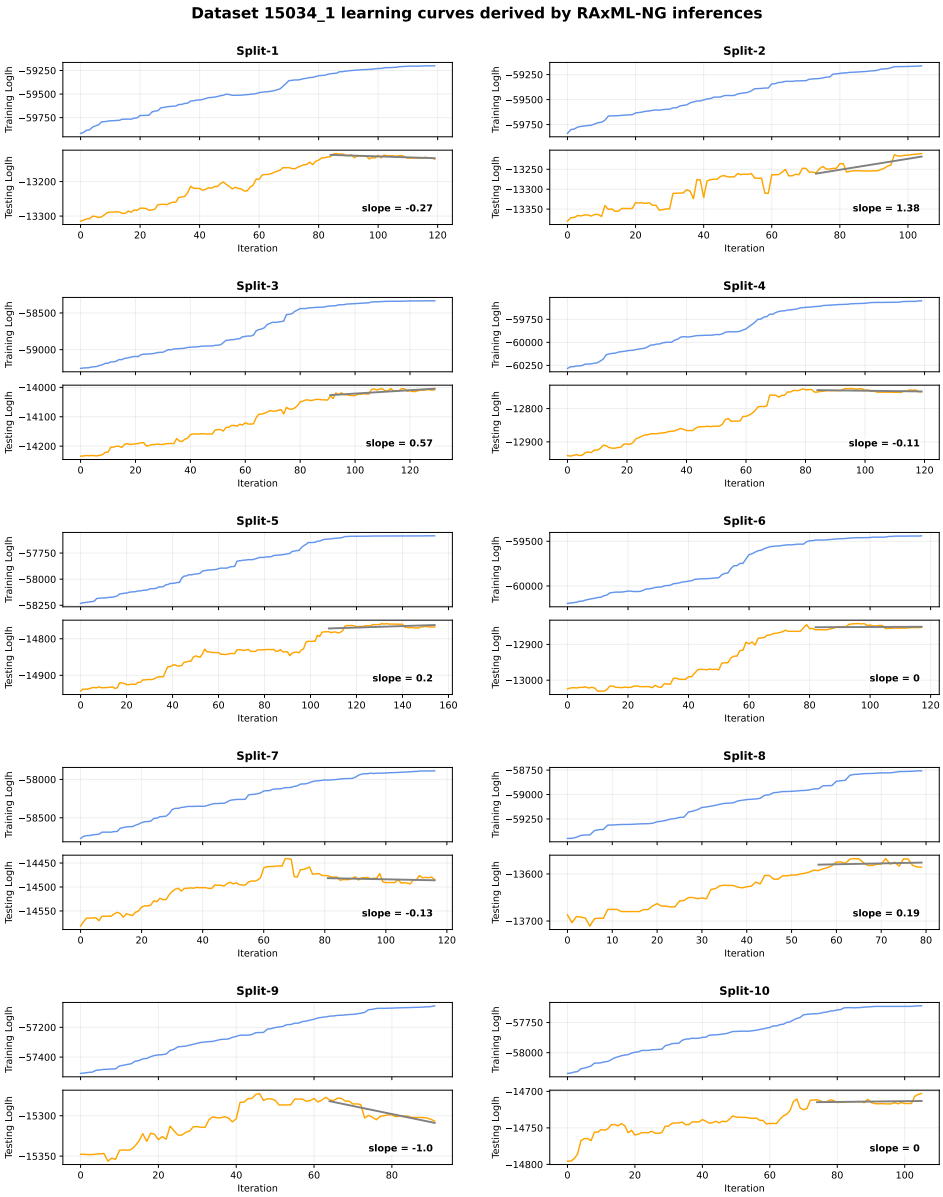


Figure 5.3: Exemplary learning curves corresponding to the anecdotal empirical DNA dataset 15034_1, comprising 222 taxa, 2,658 sites, and 1,237 patterns. The training log-likelihood curves (upper panel of each split-plot) increase monotonically and are comparatively smooth. The testing log-likelihood curves (lower panel) may be more "bumpy" and not necessarily monotonic. To capture the testing curves' behavior towards the end of the inference, I perform linear regression on the final curve-segments (here, $\mathsf{n} = 30\%N$). The fitted regression lines are shown in gray color, and the corresponding slope values are reported in the bottom-right corner of each testing log-likelihood panel. Based on the slope values, the final part of the testing curves may exhibit an increasing tendency (splits 2, 3, 5 and 8), form noisy plateaux (splits 6 and 10), or diverge from the curve's global maximum (splits 1, 4, 7, and 9). Overall, the sign-test yields a non-significant slope trend. In some cases, the fitted line may not fully capture the real testing curve dynamics. For instance, in split 7 the testing curve optimum appears too early, and the regression line gives a near-zero inflection (although, as reported, for $\mathsf{n} = 30\%N$ the inferred slope is negative and successfully reflects the model-fit decrease; yet, for the same dataset-split and for $\mathsf{n} = 20\%$, the estimated slope is 0). Analogously, in split 10 a late increase over the final few iterations is not captured by the fitted line, which yields a zero slope. In all 10 splits, the final and the optimal tree of the testing curve are statistically indistinguishable, when evaluated via the approximately unbiased (AU) test (see Section 5.5) on the corresponding testing sites (Figure reproduced from [209]).

and identically distributed (i.i.d.) noise, that has a null expectation and a symmetric distribution. This describes and captures the "bumpy" nature of the testing curves (Figure 5.3). Along the final curves of the ten experiments, we expect (under H_0) the slope to exhibit an equal probability of being positive or negative (i.e., the basis of the sign test, which can also accommodate zeros when the slope is almost flat). The sign test is based on the binomial distribution with a probability of 0.5 and 10 trials. Since the number of trials is low, only contrasted results will be significant (e.g., 8 negative slopes, 1 positive slope, and 1 flat slope corresponds to a left-sided p-value of $\sim$2%), while other cases will not be significant (e.g., 7, 2 and 1, respectively, p-value of $\sim$9%). For comparison, on empirical MSAs, I also use the one-sample t-test for slope trend evaluation (see Figure 5.6), and juxtapose the sign test vs. t-test results. No substantial difference between the two statistical tests is observed.

My approach focuses on the global trend in the final segment of the testing curve, which can be predominantly positive, negative, or non-significant. I specifically *do not* target the optimal point of the testing curve. This is because learning curves in deep learning are typically smooth, because they are based on continuous numerical functions. In contrast, log-likelihood testing curves are inherently non-smooth ("bumpy"), because topological moves on the discrete tree parameter induce major log-likelihood changes/jumps (Figure 5.3), and the optimal point itself is therefore of reduced value. Consequently, focusing on the optimal point and comparing it with the final point (i.e., the output tree, which has a negative or null difference with the optimum) may result in overestimating overfitting trends and hindering the detection of under-optimization trends that correspond to a positive slope. This is, in fact, the main limitation of the study by de Vienne *et al.* [218]. Specifically, the authors selected the best tree (in terms of testing log-likelihood) among the final ones, which they compared to the very last tree, the former being inevitably better (or at least not worse). This introduces a bias towards negative slopes, and thus over-estimates the overfitting signal, as it does not allow for yielding positive slopes, which are equally important when evaluating the model-fit dynamics of ML tools on testing data. Nevertheless, I do include such a two-point comparison between the optimal and the final trees in the Discussion section (see Figure 5.17).

For simulated MSAs, I also generate topological accuracy curves by computing the quartet distances [46] between each intermediate sampled topology (T_i), and the true tree topology (T_{ref}). I use the quartet distance, rather than the more standard RF distance [155], because it is more refined, that is, it generates more possible, distinct values, and can therefore better capture slight trends. I compute quartet distances ($d(T_i, T_{\text{ref}})$) using the tqDist tool [165], and define the topological accuracy of each intermediate topology as $1 - d(T_i, T_{\text{ref}})$. The devised topological accuracy curves are statistically evaluated via the same framework as above (linear regression followed by the sign test). This analysis, however, does not directly assess overfitting. Topologies that diverge from the true tree can indeed yield equal, or better, log-likelihood scores on the testing sites while not degrading the model-fit. Analogous observations were reported in our previous works [73, 89, 133, 210]. While not focusing on cross-validation, these studies examined and confirmed the statistical plausibility of highly divergent topologies. The results suggest that ML tree inference on difficult-to-analyze MSAs—as indicated by high Pythia scores—might yield topologically highly divergent, yet statistically equivalent topologies with respect to their likelihood score. The analysis of topological accuracy curves in the overfitting analysis context provides novel insights into whether the inferred trees systematically converge toward (or diverge from) the true topology during the optimization

process.

5.2.2 Holdout-Validation (HV) in RAxML-NG

RAxML-NG HV inherently splits the input MSA into *model-training* (80%) and *validation* (20%) sites that are selected uniformly at random. Next, it generates a parsimony starting tree and conducts an ML tree inference exclusively on the model-training part of the MSA, while utilizing the validation sites to assess convergence and eventually terminate. More specifically, HV computes the log-likelihood scores on the validation MSA for the currently best topology after each SPR round (as well as on the initial parsimony tree). It terminates when the *validation* log-likelihood either decreases, or when the improvement is negligible (based on an ε-threshold of 0.1), for k successive SPR rounds. Upon termination, the topology with the highest validation log-likelihood is retrieved and printed to file. This is analogous to approaches used in deep learning, where a validation set is used to terminate the optimization process and prevent overfitting [132]. In the following, I execute HV for $k := 1$ and $k := 5$, and refer to the corresponding runs as HV-1 and HV-5.

The benchmark pipeline also follows the setup of a typical deep learning benchmarking experiment [132], where the full dataset is partitioned into training and testing sets. The RAxML-NG versions under comparison (see below) use the training set for optimization, while the corresponding performance is evaluated on the testing set. The HV versions further account for overfitting, and they, in turn, split the training set into a model-training and a validation set, using the latter for convergence monitoring.

I provide details about the datasets used to benchmark RAxML-NG HV in Section 5.3.1. For each MSA, I re-use the same 10-fold split created to investigate topological overfitting, in the initial part of this work (see Section 5.2.1). Within each MSA-split, the RAxML-NG versions under comparison solely conduct inferences on the training MSA (using parsimony starting trees). The testing MSA exclusively serves as reference dataset for post-hoc evaluation. Specifically, for each individual MSA-split, I execute the following: (a) one standard RAxML-NG inference on the training MSA; (b) one ES inference on the training MSA; (c) one HV-1 inference, which further splits the training MSA into model-training (80%) and validation (20%) sites (the resulting MSAs comprise 64% and 16% of the original MSA sites, respectively); (d) one HV-5 inference, employing the same training MSA-partition as HV-1; and (e) one standard RAxML-NG inference on the model-training MSA (64%). A rationale for this specific evaluation setup is provided further below. The 5 inferred ML trees, within each MSA-split, are evaluated on the testing MSA via the AU test [178], as implemented in the CONSEL tool [179]. I thereby assess the output model-fit on unseen data. The trees with a p-value exceeding 0.05 are considered *plausible*. I aggregate the number of plausible ML trees inferred by each version across all splits, and record the number (out of 10) of plausible ML trees inferred by each version. This number is used as a metric to assess the overall performance across MSAs, that is, the higher the number of plausible ML trees inferred, the more robust the tool version is.

With the HV benchmark, I address two main questions. The first is whether HV effectively evades overfitting by only capturing the phylogenetic signal of the input (training) MSA, while avoiding fitting inherent noise. Intuitively, a thorough inference (standard RAxML-NG) on the training MSA may fit both, signal and noise, mitigating the generalizability of the inferred trees on the unseen (testing) sites. Further, this effect should

a fortiori be more pronounced when standard RAxML-NG is further restrained on the model-training MSA, which comprises only 64% of the original MSA sites. If this hypothesis is correct, HV is expected to yield trees with higher log-likelihood scores on the testing data, since it explicitly retains the tree with the highest validation log-likelihood, and thus maximizes the generalizability of the inferred model. Accordingly, a fraction of HV trees is expected to be statistically better compared to those inferred via standard RAxML-NG (on the training and model-training MSAs), when evaluated on the testing sites via the AU test.

The second question is whether HV provides a reliable strategy for Early Stopping in ML tree inference. To address this, I evaluate the relative performance of HV against that of ES, using standard RAxML-NG as the reference in both cases. The comparison considers both plausibility performance and runtime improvements.

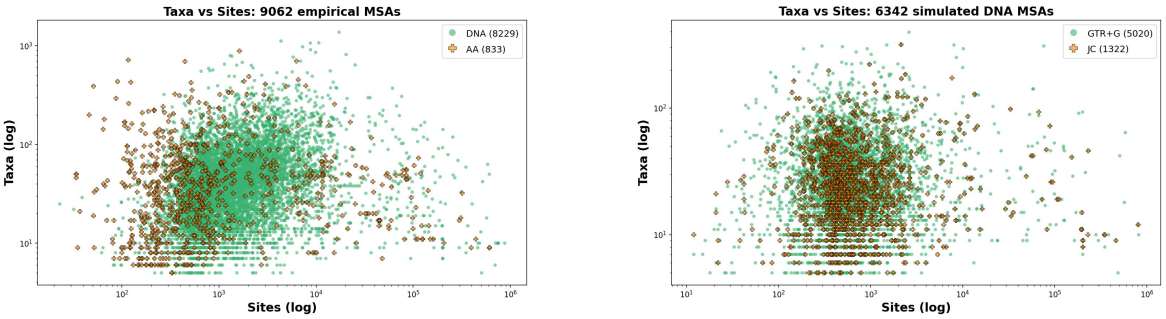
I also compute quartet distances using tqDist, between the ML trees inferred by each version and the reference tree. On simulated MSAs, the reference tree is the true tree used to generate the data. On empirical MSAs, I consider the ML tree inferred via standard RAxML-NG on the training MSA of each MSA-split as reference tree.

5.3 Experimental Setup

5.3.1 Datasets, Substitution Models, and Filtering

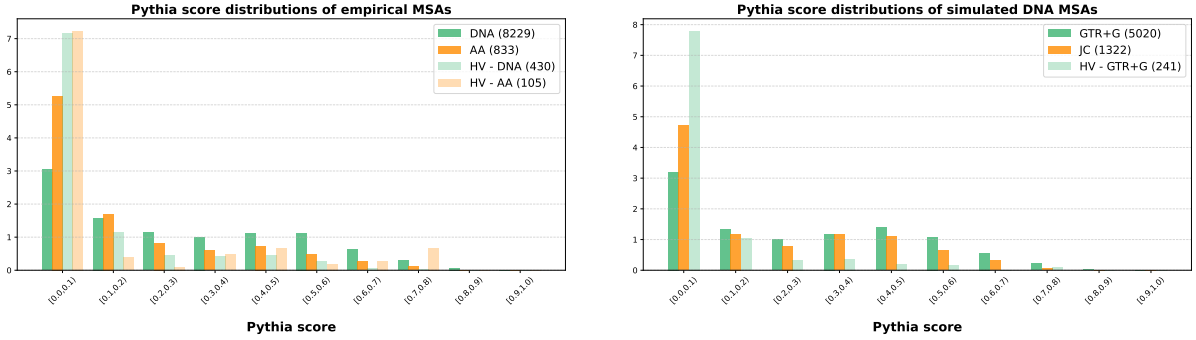
To statistically examine overfitting in ML phylogenetic tree inference (see Section 5.2.1), I conducted experiments on 9,062 empirical MSAs (8,229 DNA and 833 AA), sampled from TreeBASE [147]. No specific sampling criterion was applied, beyond retaining all MSAs which successfully completed all preprocessing and pipeline steps (see below). I further included 6,342 simulated DNA MSAs from a prior benchmark analysis on simulated data [214], comprising 5,020 alignments generated under the GTR+Γ substitution model [205] and 1,322 under the JC model [97], using the AliSim tool [213]. These simulated MSAs were selected at random, provided that they successfully completed all preprocessing and pipeline steps.

I applied a basic preprocessing step to all MSAs. First, I removed duplicate sequences to only retain unique sequences. Second, I removed gap-only MSA sites from the empirical datasets. This preprocessing step was necessary to ensure the smooth execution of all tools included in the pipeline (see Figure 5.2). This is because ML inference tools handle duplicate sequences and gap-only columns inconsistently. For example, IQ-TREE might fail to execute when gap-only columns are present. In addition, IQ-TREE, by default, automatically shrinks alignments to only contain unique sequences and therefore infers topologies with fewer taxa than expected. This subsequently leads to errors when these trees are being evaluated with RAxML-NG on the comprehensive MSA. On the other hand, while FastTree generally tolerates duplicate sequences, the sampled intermediate topologies for datasets comprising duplicates typically contain multifurcations. However, RAxML-NG fails to evaluate multifurcating trees, as it can only process strictly bifurcating trees, which causes the pipeline to terminate with an error. Thus, to avoid the need for tool-specific exception handling and ensure consistency across all analyses, I uniformly preprocessed all datasets by removing duplicate sequences and gap-only columns. Furthermore, during the 10-fold Monte-Carlo random split of MSAs into training and testing sets, I ensured that the training MSAs only contained unique sequences. If duplicate sequences occurred in the training subset due to the random splitting, the



(a) (b)

Figure 5.4: Scatter plots showing the number of taxa and sites for 9,062 empirical and 6,342 simulated MSAs. (a) Empirical MSAs: 8,229 DNA (circles) and 833 AA (crosses) datasets. **(b)** Simulated MSAs: 5,020 generated under the GTR+Γ model (circles), and 1,322 under the JC model (crosses) [Figure reproduced from the Supplementary Material of [209]].



(a) (b)

Figure 5.5: Pythia (v2.0) score distributions for 9,062 empirical and 6,342 simulated MSAs, as well as for the 535 empirical and 241 simulated MSAs selected to benchmark RAxML-NG HV. Datasets sampled for HV benchmarking are depicted via lighter-colored bars. **(a) Empirical MSAs:** 8,229 DNA and 833 AA datasets, along with 430 DNA and 105 AA MSAs selected for benchmarking RAxML-NG HV. **(b) Simulated MSAs:** 5,020 generated under the GTR+Γ model and 1,322 under the JC model. The 241 MSAs used for benchmarking were all generated under the GTR+Γ model (Figure reproduced from the Supplementary Material of [209]).

partitioning was repeated until a valid split was obtained. For testing MSAs, I did not filter duplicates, as RAxML-NG effectively handles alignments with duplicate sequences, during the testing log-likelihood evaluations. From the cleaned MSAs, I also discarded four-taxon alignments, as they are trivial to analyze. The resulting empirical/simulated MSAs comprise up to 10^6 sites, and up to 10^3 and 500 taxa in empirical and simulated MSAs, respectively. The scatter plots in Figure 5.4 illustrate the number of sites and taxa of the entire collection of empirical/simulated MSAs.

The cleaned MSAs cover the entire difficulty score spectrum as predicted by Pythia v2.0 [75]. Figure 5.5 shows the Pythia score distributions for the 9,062 empirical and 6,342 simulated MSAs, as well as for the MSAs used to benchmark RAxML-NG HV (see below). The Pythia score distributions are generally shifted toward lower Pythia scores,

suggesting easier MSAs. This effect is particularly pronounced for the empirical TreeBASE MSAs (Figure 5.5a), whose distribution is substantially more skewed toward lower values than the original TreeBASE difficulty score distribution (see Figure 3.3), which is more uniform. This shift can be attributed to the preprocessing I applied to all MSAs, which effectively improved the overall phylogenetic signal strength. To highlight this effect, I mention two characteristic examples of empirical DNA MSAs: dataset `11756_1`, whose Pythia score decreased from 0.62 to 0.13, after removing 36 duplicate sequences and 4 gap-only columns, and dataset `11269_0`, whose Pythia score decreased from 0.84 to 0.04, after removing 22 duplicate sequences. Further, MSAs with difficulty scores above 0.8 are rare. This reflects an inherent limitation of the definition of the Pythia score, as previously discussed in Section 2.8.

All datasets are treated as unpartitioned MSAs. This decision is straightforward and consistent with the data. All simulated MSAs were generated as single-partition datasets, and the vast majority of TreeBASE MSAs are, indeed, unpartitioned: ∼93% of DNA and ∼99% of AA MSAs are provided without any partitioning scheme. To conduct ML tree inferences, I used the GTR+Γ model for all empirical/simulated DNA MSAs (including those simulated under the JC model), and the LG+Γ model [116] for protein MSAs (empirical only). In the context of this study, restraining the analyses to unpartitioned MSAs and using standard substitution models provides a more controlled setting to assess overfitting.

To benchmark RAxML-NG HV, I sampled 535 long empirical MSAs (430 DNA and 105 AA) from the preprocessed collection, along with 241 long simulated DNA MSAs under the GTR+Γ model. Here, "long" refers to MSAs comprising: (a) more than 10,000 sites and an SoT ratio exceeding 10 for empirical MSAs; and (b) more than 5,000 sites and an SoT ratio exceeding 5 for simulated MSAs. The thresholds were relaxed for simulated MSAs, since relatively few datasets met the original criteria applied to empirical MSA sampling. I chose those thresholds to ensure that the HV version is evaluated under a statistically meaningful setting. That is, in datasets where the number of observations (sites) is large, both in absolute terms and relative to the number of taxa, I ensure that there is a sufficiently strong phylogenetic signal. As expected, the Pythia score distributions for these sampled, long MSAs, shifts toward lower values, indicating easier MSAs (see Figure 5.5b).

5.3.2 Commands and Computing Environment Details

I implemented the tree sampling algorithm for standard and ES RAxML-NG (described in Section 5.6.2), as well as the HV algorithm, in the `overfitting-early-stopping` branch of a forked RAxML-NG repository[1]. The same repository also contains the modified version of FastTree (Section 5.6.2) in a separate folder. The tree sampling modification of IQ-TREE (Section 5.6.2) is implemented in the `overfitting` branch of a distinct forked IQ-TREE repository[2].

RAxML-NG v1.2:

Tree sampling in RAxML-NG is enabled using the `--chkpt-method 2` option. In the experiments, I explicitly set the log-likelihood improvement ε-threshold (Section 2.6.5)

[1] `https://github.com/togkousa/raxml-ng/tree/overfitting-early-stopping`
[2] `https://github.com/togkousa/iqtree2/tree/overfitting`

to 0.1. The substitution models I used are GTR+Γ (`--model GTR+G` option) for DNA MSAs, and LG+Γ (`--model LG+G` option) for AA MSAs. Parallelization is implemented and tested for this version. Below is the command used to conduct one ML tree inference on a training MSA (e.g., `training.phy`) using 4 threads:

```
./raxml-ng-adaptive --adaptive off --threads auto{4} --msa training.phy
--model {model} --tree pars{1} --chkpt-method 2 --lh-epsilon 0.1
--prefix standard --seed 0
```

In the example command, the Adaptive version (see Chapter 3) is disabled using the `--adaptive off` option. For convenience, the output file prefix `standard` is used in the displayed command. RAxML-NG stores the intermediate improved topologies in the `standard.raxml.sprTrees` file, and the parsimony starting tree in `standard.raxml.startTree`. This starting tree is subsequently used as input for the remaining tools.

Note that RAxML-NG v1.2 is further used to evaluate intermediate trees sampled from all ML tools during the overfitting assessment, using the corresponding testing MSAs. As described in the main text, in this evaluation RAxML-NG conducts a full BLO and MPO. To slightly accelerate this evaluation, I applied an ε-threshold of 1.0. Below is an exemplary command to evaluate the stored intermediate improved topologies on a testing MSA (e.g., `testing.phy`), using 4 threads:

```
./raxml-ng-adaptive --evaluate --threads 4 --msa testing.phy
--model {model} --tree standard.raxml.sprTrees --lh-epsilon 1.0
```

RAxML-NG ES:

To invoke the KH-multiple testing stopping criterion (see Chapter 4), I use the command `--stopping-criterion KH-mult`. When using this command, the adaptive heuristic is automatically disabled. Further, in the ES version no ε-threshold is specified. Below is the command used to invoke RAxML-NG ES, using 4 threads and the output file prefix `es`:

```
./raxml-ng-adaptive --threads auto{4} --msa training.phy
--model {model} --tree standard.raxml.startTree
--chkpt-method 2 --prefix es --seed 0 --stopping-criterion KH-mult
```

RAxML-NG ES stores the topological trajectory in the `es.raxml.sprTrees` file.

Modified IQ-TREE:

One can enable the tree sampling in the modified IQ-TREE version using the `-wt` flag. Parallelization also works for this modified IQ-TREE version. Below is the command used to invoke our modified IQ-TREE version, using 4 threads and the prefix `iqtree` (for example, on a DNA dataset):

```
./iqtree2 -nt 4 -s training.phy --seqtype DNA -m GTR+G
-t standard.raxml.startTree --prefix iqtree -wt --seed 0
```

Modified IQ-TREE stores the sampled topologies in the `iqtree.treels` file.

Modified FastTree:

The tree sampling is enabled by default in the modified FastTree version. As I discuss in Section 5.6.1, FastTree uses a variant of the CAT approximation [150], which assigns each site to one of 20 precomputed, fixed rate categories by default. To further simplify the model approximations, I reduced this number to 4, using the `-cat 4` option (see Section 5.6.1). Therefore, I used the GTR+CAT4 model for DNA (`-gtr -nt -cat 4` option), and the LG+CAT4 model for AA MSAs (`-lg -cat 4` option). Below is the command used to invoke the modified FastTree version on a DNA dataset:

```
./FastTree -gtr -nt -cat 4 -intree1 standard.raxml.startTree
-log fasttree.log < training.phy > fasttree.bestTree
```

In the above command, the output log file is set to `fasttree.log`. The modified FastTree version uses the prefix of the logfile (here, `fasttree`) to name the sampled topologies file. In this example, it stores the intermediate topologies in the `fasttree.chkpTrees` file.

RAxML-NG HV:

Users can invoke the Holdout Validation algorithm in RAxML-NG via the `--holdout-es` flag. The algorithm randomly splits the input MSA into model-training and validation sites, based on the specified seed, and using a default splitting ratio of 0.8 (i.e., 80% of the sites) for the model-training MSA. One can define an alternative splitting ratio via `--split-ratio X`, where `X` is a floating point number between 0 and 1 that specifies the proportion of *model-training* sites. Further, the user can also specify the number of convergence iterations k (see Section 5.6.1) via `--conv-iters k`, with `k` being a positive integer value. The default value for convergence iterations is $k := 1$. The resulting model-training and validation MSAs can be stored by enabling the `--extra split-save-phylip` option. The MSAs are written to the `{prefix}.raxml.training.phy` and `{prefix}.raxml.testing.phy` files, respectively. Parallelization is implemented and tested for this version. An example RAxML-NG HV invocation, where I set $k := 5$ convergence iterations (HV-5; see Section 5.2.2), is:

```
./raxml-ng-adaptive --holdout-es --threads auto{4} --msa {model}
--model {model} --conv-iters 5 --prefix example
--extra split-save-phylip --seed 0
```

RAxML-NG HV stores the resulting model-training and validation MSA files in the `example.raxml.training.phy` and `example.raxml.testing.phy` files, respectively.

Computing Environment

I executed the overfitting evaluation pipeline on the bwForCluster Helix, located at the Heidelberg University Computing Centre. I utilized the `cpu-single` cluster partition, which allocates submitted jobs to AMD nodes. Each AMD node is equipped with two AMD Milan EPYC 7513 Processors, running at 2.60GHz, providing a total of 64 physical cores per node. The operating system is RedHat Linux. Further, I benchmarked RAxML-NG HV on one of our lab servers, equipped with two AMD EPYC 9684X (Genoa-X) processors, each with 96 physical cores and two threads per core, operating at a maximum frequency of 3.72 GHz. The operating system is Ubuntu 24.04.

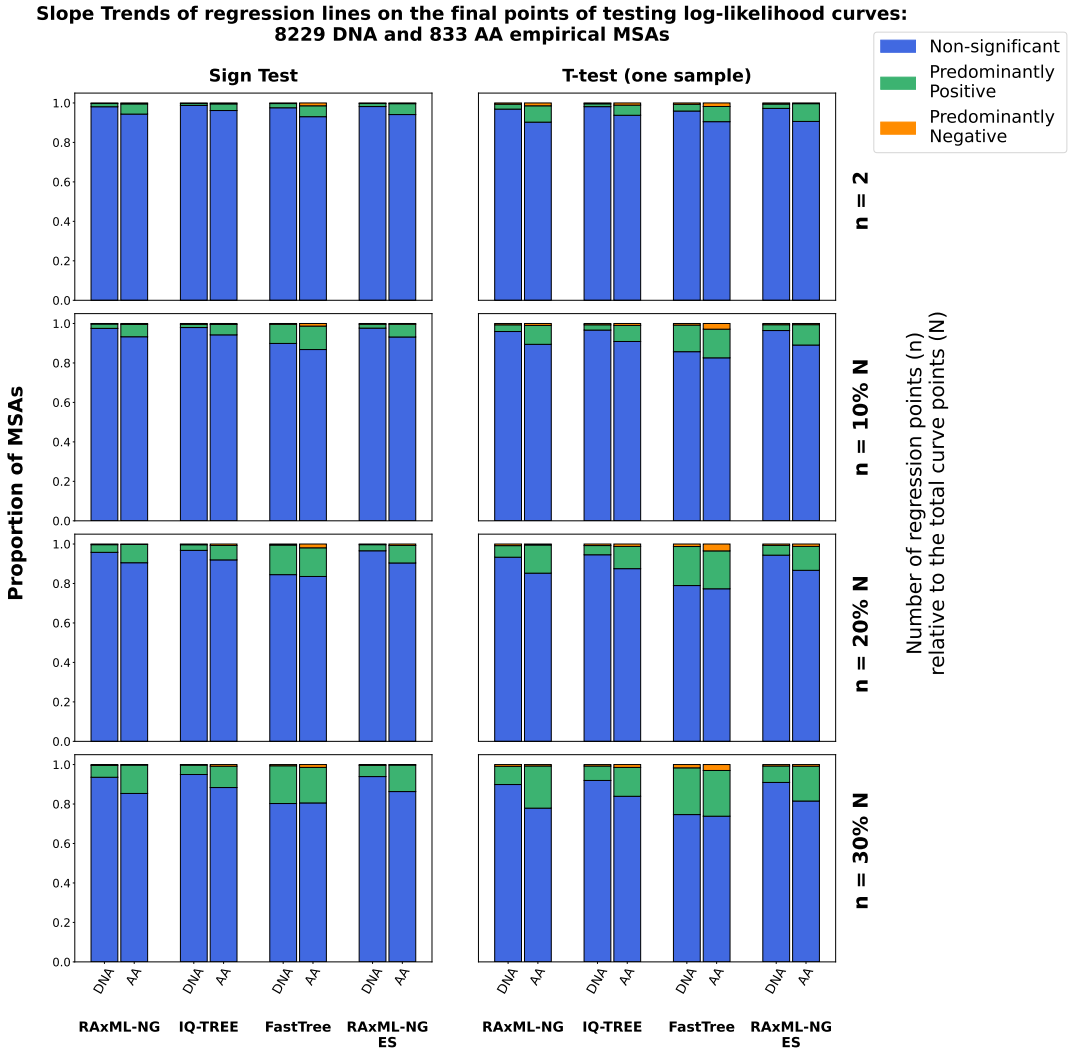


Figure 5.6: Slope trends of regression lines fitted to the final segments of testing log-likelihood curves for 9,062 empirical MSAs. The slope trends are evaluated using either the non-parametric sign test (left), or the one-sample T-test (right). Distinct columns within each subplot correspond to different ML tree inference tools: RAxML-NG, IQ-TREE, FastTree, and RAxML-NG ES. Subplot-rows correspond to different linear regression window sizes: the last 2 points, or the final 10%, 20%, or 30% of the points on each curve. The bar heights in the stacked bar charts indicate the proportion of MSAs exhibiting one of the three slope trends: Predominantly Positive, Predominantly Negative, or Non-significant, as classified via the statistical test used (Figure reproduced from the Supplementary Material of [209]).

5.4 Results

5.4.1 Results on the Statistical Overfitting Evaluation

Figure 5.6 summarizes the results of the overfitting evaluation pipeline across 9,062 empirical (8,229 DNA and 833 AA) MSAs. The left subfigure presents the results for the non-parametric sign test, and the right one for the one-sample t-test. Since no substantial difference is observed, in the following, I will only focus on the sign test results. As

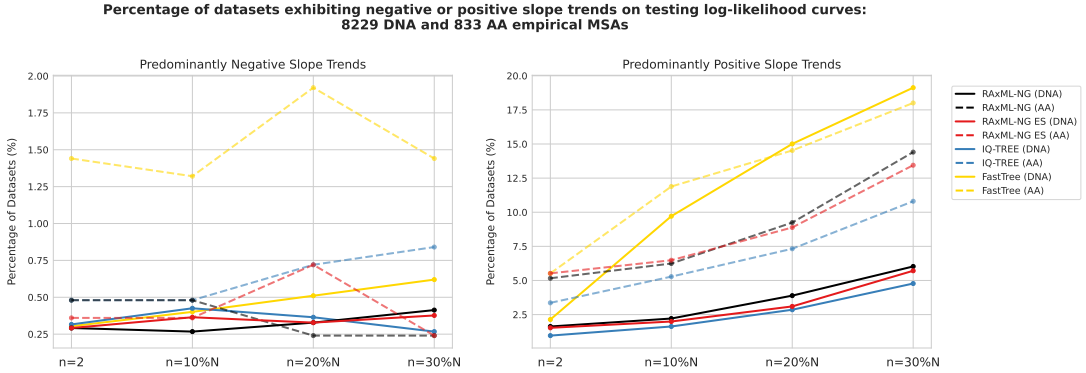


Figure 5.7: Percentage of empirical MSAs exhibiting Predominantly Negative (left) or Predominantly Positive (right) slope trends based on the sign-test results, across different linear regression window sizes: the final 2 points ($\mathsf{n} = 2$), the last 10% ($\mathsf{n} = 10\%N$), 20% ($\mathsf{n} = 20\%N$), or 30% ($\mathsf{n} = 30\%N$) of the points in each curve. The plot illustrates how the frequency of these slope trends varies across tools and regression window sizes, as those are observed in Figure 5.6 (left subfigure), for empirical MSAs (Figure reproduced from [209]).

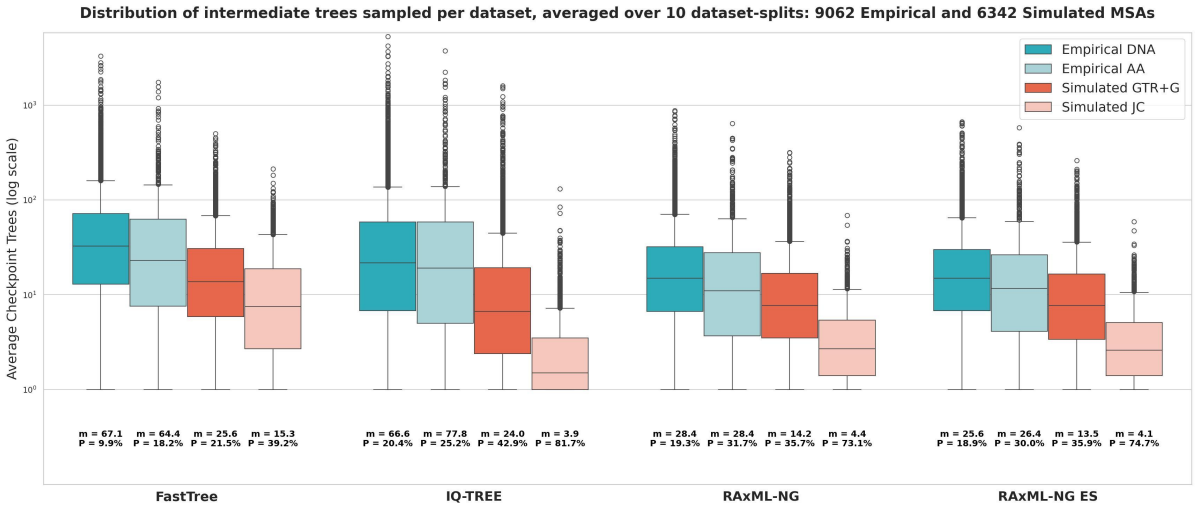


Figure 5.8: Distributions of the number of intermediate trees sampled by each tool for each dataset, averaged over the 10 dataset-splits, across 9,062 empirical and 6,342 simulated MSAs. For each boxplot, the mean value of the distribution (m), and the percentage (P) of MSAs with an average number of sampled topologies below 5 are reported. The y-axis is shown on a logarithmic scale (Figure reproduced from the Supplementary Material of [209]).

discussed below, the proportion of negative and positive slope trends as derived via the sign test (left subfigure of Figure 5.6) is visualized in Figure 5.7. Systematic overfitting, which is indicated by negative slope trends, is rarely observed.

RAxML-NG, IQ-TREE, and RAxML-NG ES show highly consistent behavior across different regression window sizes. On DNA MSAs, negative trends correspond to ∼0.3% of the MSAs, with only minor variations across tools and regression window sizes. For AA MSAs, the proportion remains similar (∼0.3%) in most cases, except for IQ-TREE, where it increases to ∼0.7%–0.8% for $\mathsf{n} = 20\%N$ and $\mathsf{n} = 30\%N$. RAxML-NG ES also

shows a slight increase to ∼0.7% for $\mathsf{n} = 20\%N$. It subsequently drops to baseline levels for $\mathsf{n} = 30\%N$.

The above tools exhibit similar patterns for positive trends as well. Standard RAxML-NG and ES show positive trends on ∼2% of DNA and ∼5% of AA MSAs when only considering the final two points in the regression. As expected, the proportion of positive trends increases (roughly linearly) with regression window size: to around 2.5% and 6% (DNA and AA MSAs) for $\mathsf{n} = 10\%N$, to 3.5% and 9% for $\mathsf{n} = 20\%N$, and to 6% and 14% for $\mathsf{n} = 30\%N$. IQ-TREE consistently yields lower positive trend rates: 1% and 3.3% (DNA and AA MSAs) for $\mathsf{n} = 2$, 1.6% and 5.3% for $\mathsf{n} = 10\%N$, 3% and 7.3% for $\mathsf{n} = 20\%N$, and 5% and 11% for $\mathsf{n} = 30\%N$. Consequently, the proportion of non-significant trends ranges from ∼94% and ∼86% (DNA and AA MSAs, respectively) for $\mathsf{n} = 30\%N$, to ∼98% and ∼95% for $\mathsf{n} = 2$. As one may expect, the proportion of non-significant trends increases when focusing on the very final parts of the curve that are closer to convergence (due to convergence on noisy plateaux), resulting in a corresponding decrease in positive trends.

Note that the similarity between standard RAxML-NG and ES is expected. Early Stopping in RAxML-NG operates at the level of SPR rounds (see Section 4.2) by omitting potentially redundant/unnecessary SPR rounds towards the end of the search and substantially reducing runtime. However, most SPR moves are accepted during the initial rounds. Therefore, in both RAxML-NG versions, a comparable number of intermediate topologies (i.e., curve points) is being sampled. Figure 5.8 illustrates the distribution of the number of intermediate trees sampled by each tool and for each dataset, averaged over the 10 MSAs-splits, across all empirical and simulated MSAs. The average number of sampled topologies by RAxML-NG ES is only modestly reduced compared to the standard version.

Returning to Figures 5.7–5.8, FastTree exhibits similar (compared to the other tools) negative trends for DNA MSAs that range between 0.3% and 0.6%. For AA MSAs, however, the overfitting trends are substantially higher, with the baseline levels at approximately 1.4%, and increasing up to 1.9% for $\mathsf{n} = 20\%N$ (see Figure 5.8). These substantially higher negative trends for AA MSAs are difficult to interpret. They could be attributed to the plethora of approximations[1] FastTree applies to calculate the log-likelihood scores of visited topologies, during the tree search. These can potentially misguide the search towards topologies that, when evaluated thoroughly on unseen sites, result in progressively decreasing testing log-likelihoods, for every intermediate accepted topology. Thorough tools, on the other hand, employ more accurate log-likelihood evaluations, yielding lower overfitting rates for AA MSAs. With respect to continuous parameter optimizations, RAxML-NG ES belongs to the group of thorough tools, since it invokes the same optimization routines as RAxML-NG, without applying any shortcuts (see Section 4.2). Nevertheless, despite being inflated, the overfitting rates of FastTree are still small.

Regarding positive trends, FastTree exhibits relatively high rates for $\mathsf{n} = 30\%N$: 19.1% for DNA and 18% for AA MSAs. These rates gradually decline for narrower final segments, reaching approximately 15% for $\mathsf{n} = 20\%N$, and 10%–12% for $\mathsf{n} = 10\%N$. This pattern suggests that slope trends remain strongly positive in the later stages of the tree search, and indicates that FastTree could potentially benefit from additional topological optimization steps. When focusing on the final two points on the curve, though, FastTree attains levels that are comparable to the other tools, with 2.1% for DNA and 5.5% for AA

[1] I describe those approximations in Section 2.6.7.

Figure 5.9: Slope trends derived from the sign test, versus taxa and sites in 9,062 empirical MSAs, illustrated for the specific case where $\mathsf{n} = 20\%N$. Both horizontal and vertical axes have a logarithmic scale (Figure reproduced from the Supplementary Material of [209]).

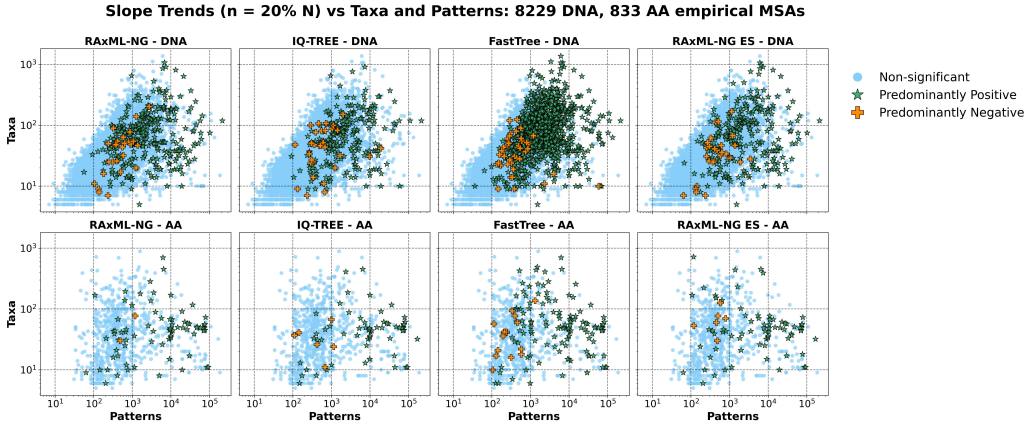


Figure 5.10: Slope trends derived from the sign test, versus taxa and patterns in 9,062 empirical MSAs, illustrated for the specific case where $\mathsf{n} = 20\%N$. Both horizontal and vertical axes have a logarithmic scale (Figure reproduced from the Supplementary Material of [209]).

MSAs. Hence, the proportion of non-significant trends in FastTree ranges from ∼80% on both DNA and AA MSAs at $\mathsf{n} = 30\%N$, to 97.6% and 93% for $\mathsf{n} = 2$ on DNA and AA MSAs, respectively. The observed convergence indicated by the decline of positive trends for narrower regression segments, which is consistent among tools, does not imply that the quality of the inferred trees is equivalent. For instance, one way to assess the quality of inferred trees would be to compare them via the AU test, as I already did in the benchmarks of Chapters 3 and 4. Such comparisons constitute a static assessment of the model-fit of the inferred ML trees. In contrast, the current study investigates dynamic overfitting trends, by independently analyzing the testing curves of each tool and evaluating whether model degradation is observed due to excessive optimization.

For the specific case of $\mathsf{n} = 20\%N$, I investigated potential correlations between overfitting trends and MSA characteristics, such as the number of taxa, sites, patterns, and Pythia scores. The results are presented in Figures 5.9–5.11. A modest correlation suggests that overfitting trends are more prevalent for MSAs with relatively short

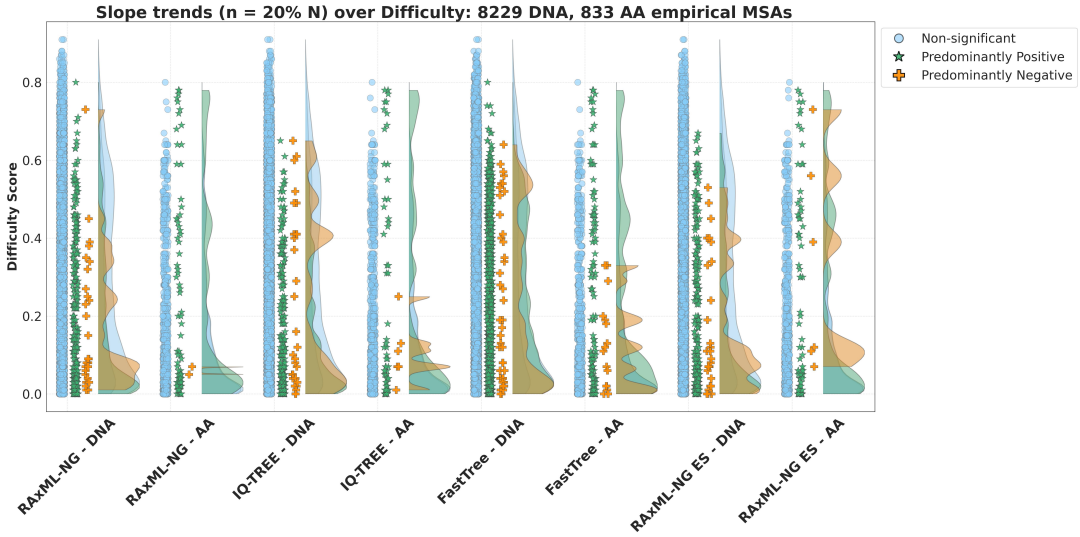


Figure 5.11: Slope trends derived from the sign test, over difficulty scores for 9,062 empirical MSAs, illustrated for the specific case where $\mathsf{n} = 20\%N$ (Figure reproduced from the Supplementary Material of [209]).

sequences and low site pattern counts (fewer than ∼10,000 sites and ∼5,000 patterns), that is, as expected, with weaker phylogenetic signal (Figures 5.9–5.10). This upper sequence-length limit is particularly consistent for RAxML-NG (both standard and ES versions), that is, no negative trends on MSAs exceeding those site/pattern thresholds are observed. In contrast, for IQ-TREE and FastTree, a small number of DNA MSAs exhibiting overfitting trends were detected as outliers, despite their high sites-over-taxa and patterns-over-taxa ratios. One notable outlier, which was observed in FastTree, was the DNA MSA `13654_5`. This dataset exhibited negative trends despite comprising 10 taxa, 747,575 sites, and 58,956 patterns that render its phylogenetic signal extremely strong (Pythia score 0.0). A direction of future work would be to focus on MSAs that exhibit systematic overfitting and identify MSA-specific characteristics that predispose particular tools to overfit. Comparative analyses may also be informative, for example by detecting MSA features that trigger overfitting in RAxML-NG versus IQ-TREE. Based on the current results, the two tools do not overfit on the same MSAs. For $\mathsf{n} = 20\%N$, RAxML-NG shows negative trends on 29 MSAs and IQ-TREE on 36, yet the overlap comprises only 4 MSAs. Three of these MSAs show strong phylogenetic signal (Pythia score below 0.1), and the fourth is still relatively easy to analyze (Pythia score 0.29). Overall, no clear correlation between higher difficulty scores and overfitting trends was observed (Figure 5.11).

Regarding simulated DNA MSAs, Figures 5.12–5.13 show the results of the testing log-likelihood and topological accuracy curves, respectively, when evaluated using the same framework (linear regression followed by the sign test). In both Figures, the results from the 5,020 GTR+Γ-generated datasets are presented in juxtaposition to those from the 1,322 JC-based simulated MSAs. As a reminder, on both simulated MSA types I used the GTR+Γ model for the ML tree inferences and evaluations (GTR+CAT for FastTree; see Section 5.6.1) to also assess the effect of substitution model over-parametrization on JC-based MSAs. For the testing log-likelihood slope trends (Figure 5.12), overfitting is essentially absent, on both the GTR+Γ- and JC-generated datasets, across all tools (exact percentages are not reported, since they are negligible). JC-based MSAs display

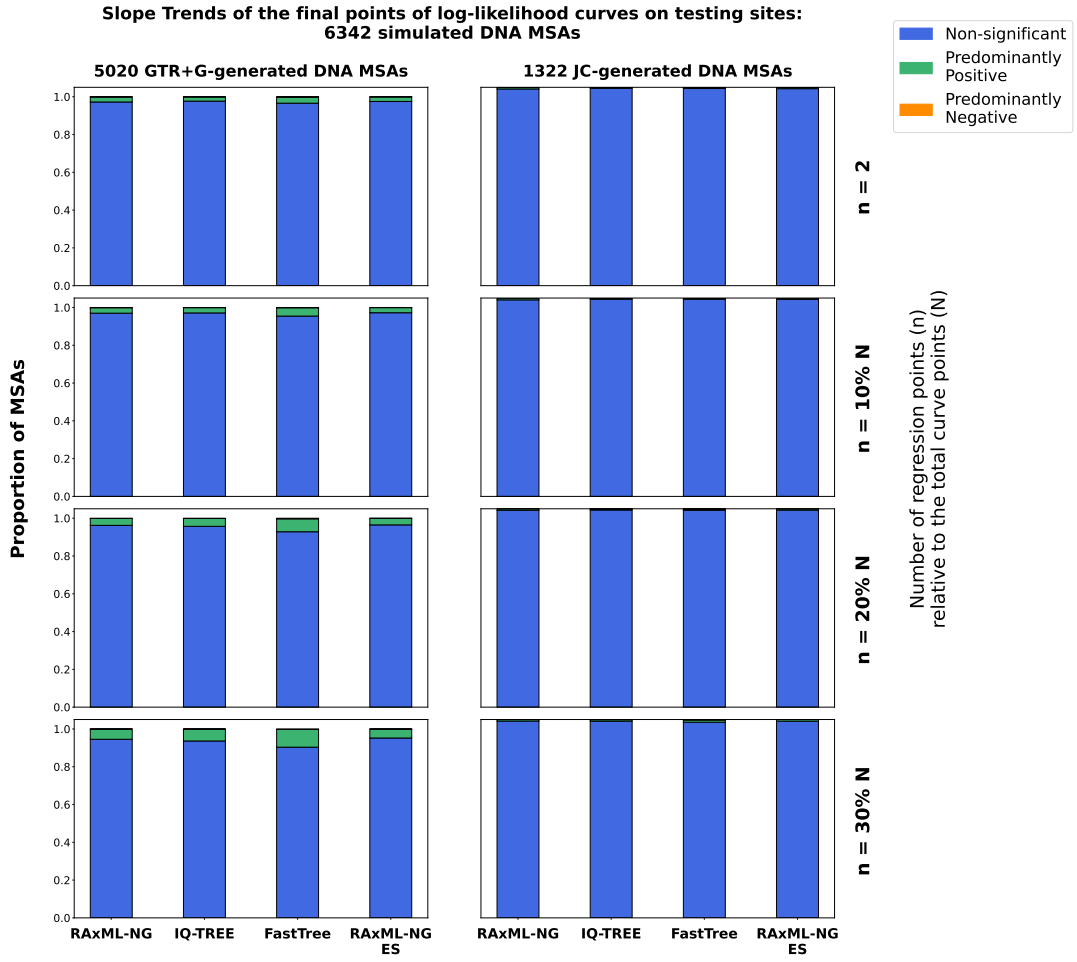


Figure 5.12: Slope trends of regression lines fitted to the final segments of the testing log-likelihood curves, derived from 6,342 simulated DNA MSAs, comprising 5,020 GTR+Γ-generated (left), and 1,322 JC-generated MSAs. Distinct columns within each subplot correspond to a different ML tree inference tool: RAxML-NG, IQ-TREE, FastTree, and RAxML-NG ES. Subplot-rows correspond to different linear regression window sizes: the last 2 points, or the final 10%, 20%, or 30% of points in each curve. The bar heights in the stacked bar charts indicate the proportion of MSAs exhibiting one of the three slope trends: Predominantly Positive, Predominantly Negative, or Non-significant, as inferred via the non-parametric sign-test (Figure reproduced from the Supplementary Material of [209]).

lower proportions of positive trends, primarily because these datasets tend to converge faster (see Figure 5.8; also discussed below). Specifically, JC-based MSAs exhibit non-significant trends in ∼99% of the cases, while the remaining 1% are attributed to positive trends.

In topological accuracy curves, the slope trends are consistent among GTR+Γ- and JC-generated MSAs. The overall trend is non-significant on more than 99% of the MSAs for RAxML-NG (standard and ES versions) and IQ-TREE, across all regression segment configurations. In FastTree, the proportion of non-significant trends is slightly lower, ranging between 98% and 99%, with the respective reduction being primarily due to negative trends. Specifically, while negative trends are observed for approximately 0.1%

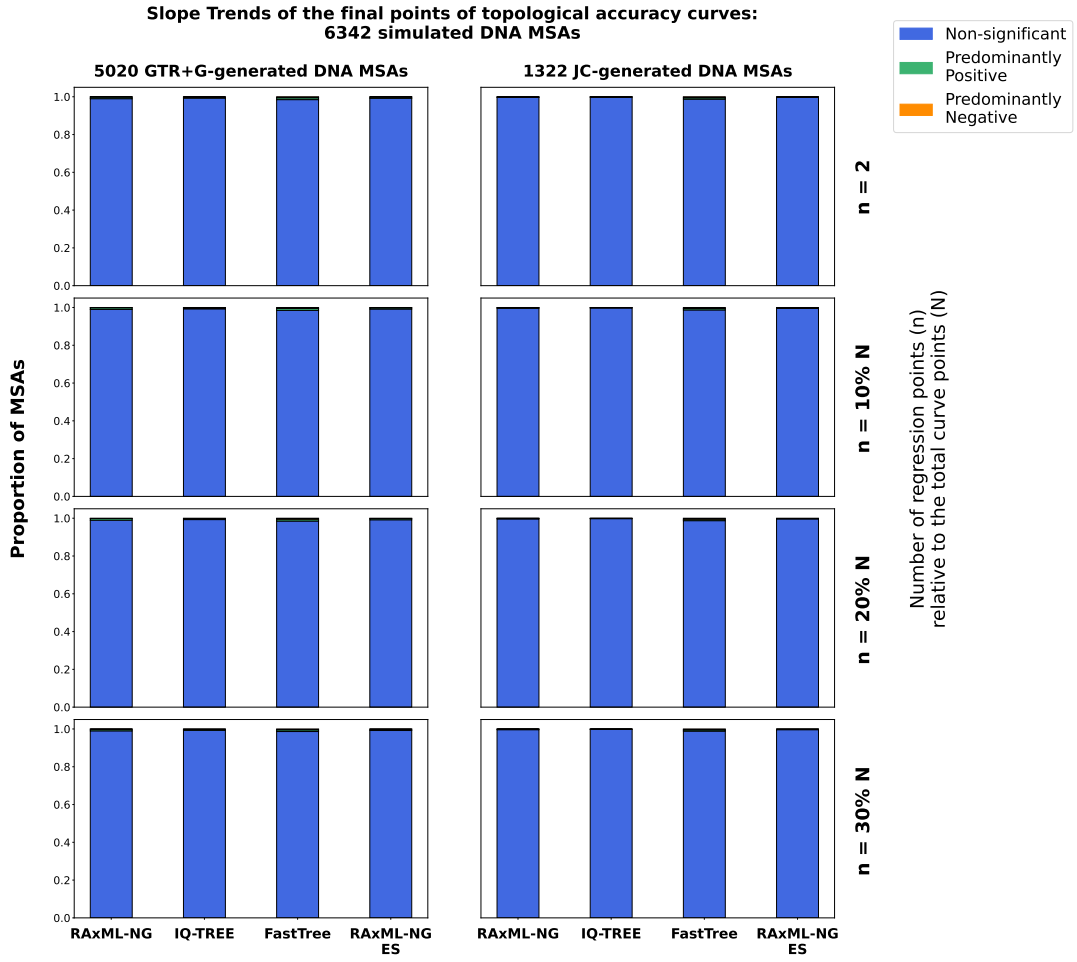


Figure 5.13: Slope trends of regression lines fitted to the final segments of the topological accuracy curves, derived from 6,342 simulated DNA MSAs, comprising 5,020 GTR+Γ-generated (left) and 1,322 JC-generated MSAs (right). Distinct columns within each subplot correspond to a different ML tree inference tool: RAxML-NG, IQ-TREE, FastTree, and RAxML-NG ES. Subplot-rows correspond to different linear regression window sizes: the last 2 points, or the final 10%, 20%, or 30% of the points in each curve. The bar heights in the stacked bar charts indicate the proportion of MSAs exhibiting one of the three slope trends: Predominantly Positive, Predominantly Negative, or Non-significant, as inferred via the non-parametric sign-test (Figure reproduced from the Supplementary Material of [209]).

of the cases in RAxML-NG and IQ-TREE, FastTree consistently exhibits negative trends for 0.5% of cases, across all regression segment window sizes. Notably, the excess of free substitution model parameters induced by using the GTR+Γ model on JC-based MSAs does not increase overfitting trends.

Overall, the increased ratio of non-significant trends on simulated datasets can be explained by the fact that ML tree inferences generally converge more rapidly on such MSAs [214]. The number of sampled topologies during ML inferences is substantially lower for simulated MSAs than for empirical ones, with the effect being particularly pronounced on JC-based MSAs (see Figure 5.8). For instance, RAxML-NG visits on average twice as many intermediate topologies on empirical MSAs compared to GTR+Γ-

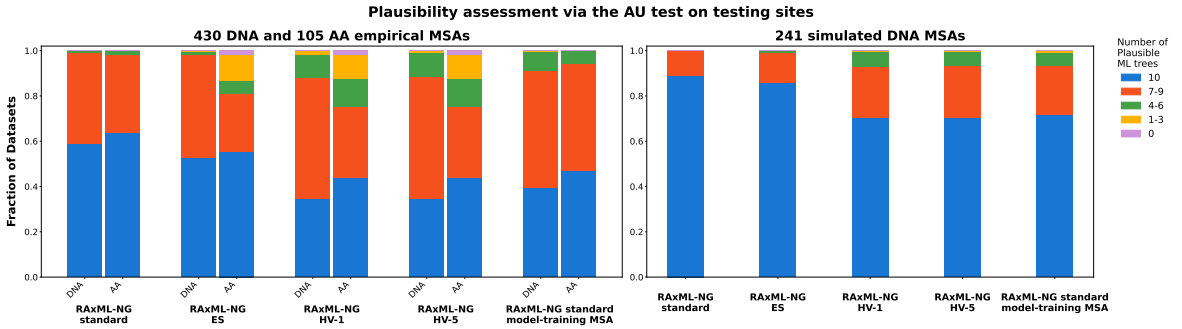


Figure 5.14: Plausibility test results on ML trees inferred via RAxML-NG versions on the training MSAs within each MSA-split, on empirical (left) and simulated (right) MSAs. I conduct this evaluation on the testing MSA of the corresponding MSA-split, by applying the AU test, as implemented in CONSEL, to the five ML trees inferred by: (a) RAxML-NG standard, (b) RAxML-NG ES, (c) HV-1, (d) HV-5, and (e) RAxML-NG standard on the model-training MSA of the HV versions. ML trees with a p-value ≥ 0.05 are considered as being plausible. I aggregate the plausible ML trees across splits, and for each method and dataset, record the number (out of 10) of plausible ML trees (Figure reproduced from the Supplementary Material of [209]).

simulated MSAs, and seven times as many compared to JC-simulated MSAs. There exist cases where the parsimony starting tree is identical to the final ML tree, that is, no topological moves are applied at all. In such cases, only one intermediate topology is sampled, and I conventionally set the corresponding slope to 0, biasing the slope data in favor of the non-significant outcome in the sign test. In this context, a non-significant trend reflects easy convergence.

5.4.2 Results on RAxML-NG HV benchmarking

Figure 5.14 illustrates the plausibility assessment results on 535 empirical (left) and 241 simulated (right) long MSAs. The figure shows the fraction of MSAs, for which each tool version inferred x (out of 10) plausible ML trees, where x represents specific counts or intervals. Overall, the HV versions exhibit the worst performance among all versions.

On empirical MSAs (left subfigure), standard RAxML-NG infers 10 (out of 10) plausible ML trees for ∼59% of DNA MSAs, ES for ∼53%, the HV versions for ∼35%, and RAxML-NG on the model-training MSA for ∼40%. The corresponding values for AA MSAs are 64% for standard RAxML-NG, 55% for ES, 44% for the HV versions, and 47% for RAxML-NG (model-training MSA), respectively. The average number of plausible ML trees inferred per DNA dataset is ∼9.3 for standard RAxML-NG and ES, 8.5 for the HV versions, and 8.7 for RAxML-NG on model-training MSAs; for AA MSAs, the corresponding averages are 9.4 for standard RAxML-NG, 8.2 for ES, 7.8 for HV versions, and 8.9 for RAxML-NG on model-training MSAs. Further, standard RAxML-NG runs (both on training and model-training MSAs) always infer at least one plausible tree (100%) on both DNA and AA MSAs. The remaining versions (ES and HV) fail to infer any plausible tree on 1 DNA and 2 AA MSAs, respectively. The failing DNA MSA is the same MSA for all three versions; the HV versions fail on the same two AA MSAs, one of which is also shared with ES. As expected, the performance patterns are similar across empirical and simulated datasets (right subfigure), although the differences for the latter are less

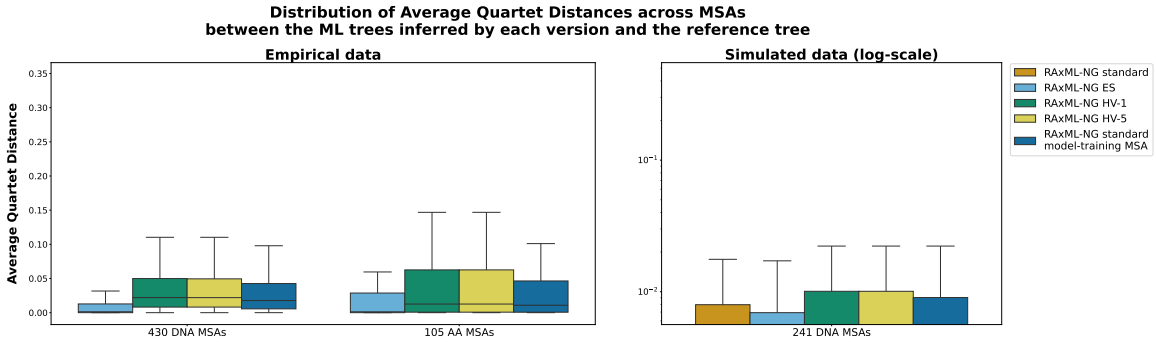


Figure 5.15: Average Quartet Distance distributions between the ML trees inferred by each RAxML-NG version, and the reference tree topology, for empirical (left subfigure) and simulated (right subfigure) MSAs. For empirical MSAs, the reference tree is the ML tree inferred by standard RAxML-NG on the training MSA of the corresponding MSA-split. Consequently, the distribution of standard RAxML-NG (on training MSAs) is absent on the empirical MSAs (left subfigure), as it reduces to zero. For simulated MSAs it is the true tree topology used for sequence simulation. (Figure reproduced from the Supplementary Material of [209]).

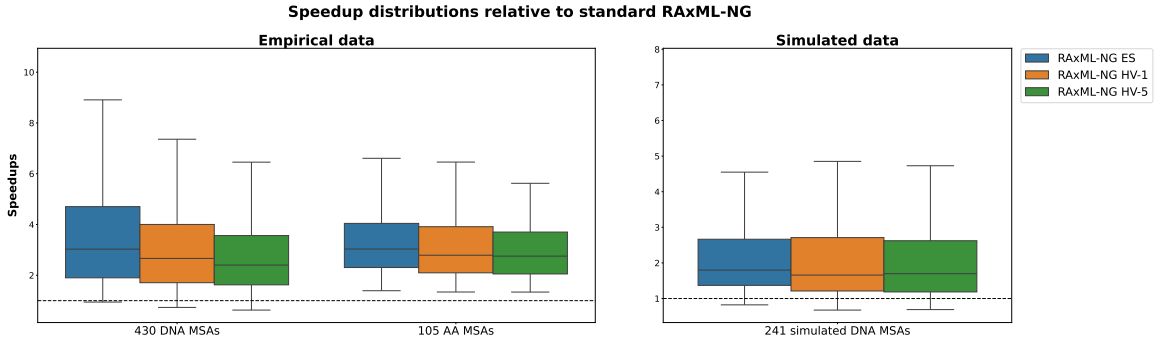


Figure 5.16: Speedup distributions of RAxML-NG ES and HV versions, relative to standard RAxML-NG inferences conducted on the training MSAs, for empirical (left subfigure) and simulated (right subfigure) MSAs. The dashed lines at the bottom of each subplot correspond to speedups of 1× (Figure reproduced from the Supplementary Material of [209]).

pronounced, since simulated MSAs generally converge faster [214].

Figure 5.15 illustrates the average quartet distance distributions between the ML trees inferred by each version and the corresponding reference tree, on empirical as well as simulated DNA MSAs. For empirical MSAs, the reference tree is the ML tree inferred by standard RAxML-NG, on the training MSA of the corresponding MSA-split. Hence, the distribution of standard RAxML-NG (training MSAs) on empirical MSAs reduces to zero. For simulated MSAs, the reference tree is the true tree used for sequence simulation. The pairwise quartet distances between each ML tree inferred by each version, and the reference tree are averaged over the 10 ML trees that each tool version infers per MSA (10 splits per MSA).

On empirical DNA MSAs (Figure 5.15, left subfigure), the mean values of the average quartet distance distributions are: 0.014 for ES, 0.039 for both HV versions, and 0.033 for standard RAxML-NG using the model-training sites. The corresponding mean values for empirical AA MSAs are: 0.043 for ES, 0.058 for both HV versions, and 0.041 for

standard RAxML-NG on model-training MSAs. Notably, although standard RAxML-NG (on model-training MSAs) outperforms ES on protein datasets in terms of mean values, the corresponding medians do favor ES. The median quartet distance for ES is 0.001, compared to 0.011 for RAxML-NG (model-training MSAs). This difference can also be observed by considering the shapes of the two distributions in Figure 5.15 (left subfigure), since the ES distribution is skewed toward lower values. Thus, ES more frequently approximates the reference tree topology than RAxML-NG (model-training MSAs) on empirical (as well as on simulated) MSAs, albeit it also yields more pronounced outliers.

On simulated MSAs (Figure 5.15, right subfigure), all distributions exhibit a median of 0, indicating that the average quartet distance is 0 for the majority of the datasets across all methods. The mean values of the average quartet distance distributions are 0.014 for standard RAxML-NG and ES, and 0.017 for all other versions. Although HV versions and RAxML-NG (on model-training MSAs) exhibit identical mean values, the distribution of the latter appears to also be slightly skewed towards zero values on the log-scale. This indicates that thorough RAxML-NG inferences that are constrained to the model-training sites only, almost outperform HV with respect to topological accuracy, even on such long simulated MSAs. Most importantly, the results, from both plausibility and topological accuracy comparisons, show that HV consistently underperforms relative to all other versions.

Finally, Figure 5.16 illustrates the speedup distributions of the ES and HV versions, relative to standard RAxML-NG execution times on the training MSA. I compute speedups by comparing the accumulated runtimes of each version for conducting 10 ML tree inferences. For empirical DNA MSAs, the average speedups are 3× for HV-1, 2.7× for HV-5, and 3.5× for ES; on AA MSAs, the corresponding averages are 3.1× for HV versions and 3.3× for ES, respectively. On simulated DNA MSAs, HV versions show an average speedup of 2×, and ES a speedup of 2.2×. Hence, the HV versions do not yield any runtime improvement over ES on the tested datasets.

These results provide a straightforward answer to the two questions posed in Section 5.2.2. First, HV does not offer any advantage w.r.t. preventing overfitting. This aligns with the findings from Section 5.4.1 that overfitting is negligible in ML-based phylogenetic inference. Using a deep learning-inspired validation set approach thus appears inappropriate. Notably, standard RAxML-NG restrained to the model-training sites performs better than HV, despite using the same data fraction but without validation-based termination. Second, optimizing on all available sites yields better overall results, and runtime acceleration is more efficiently achieved via ES, which applies stopping rules that do not discard any data. Overall, these findings strongly suggest that holdout validation does not constitute a useful strategy in ML based phylogenetics, not even so on apparently sufficiently long MSAs.

5.5 Discussion

The first part of the analysis focused on whether topological overfitting constitutes the statistically expected behavior of ML tree inference tools. I used the sign test to evaluate the statistical significance of the global trend on the final points of the testing curves. I evaluated RAxML-NG and IQ-TREE, which employ thorough heuristics, alongside with FastTree, a more superficial tool, and RAxML-NG ES, a substantially accelerated, Early

Tool	Window	Pos	Neg	Zero
RAxML-NG	$\mathsf{n} = 2$	22,519 / 2.48	17,827 / 1.97	50,274 / 5.55
	$\mathsf{n} = 10\%N$	26,029 / 2.87	20,042 / 2.21	44,549 / 4.92
	$\mathsf{n} = 20\%N$	30,230 / 3.34	21,052 / 2.32	39,338 / 4.34
	$\mathsf{n} = 30\%N$	33,720 / 3.72	21,173 / 2.34	35,727 / 3.94
RAxML-NG ES	$\mathsf{n} = 2$	21,648 / 2.39	17,640 / 1.95	51,332 / 5.66
	$\mathsf{n} = 10\%N$	24,697 / 2.72	19,429 / 2.14	46,494 / 5.13
	$\mathsf{n} = 20\%N$	28,725 / 3.17	20,959 / 2.31	40,936 / 4.52
	$\mathsf{n} = 30\%N$	32,550 / 3.59	21,200 / 2.34	36,870 / 4.07
IQ-TREE	$\mathsf{n} = 2$	16,597 / 1.83	14,500 / 1.60	59,523 / 6.57
	$\mathsf{n} = 10\%N$	24,096 / 2.66	19,828 / 2.19	46,696 / 5.15
	$\mathsf{n} = 20\%N$	27,671 / 3.05	20,804 / 2.29	42,145 / 4.65
	$\mathsf{n} = 30\%N$	30,230 / 3.33	20,359 / 2.25	40,031 / 4.42
FastTree	$\mathsf{n} = 2$	20,641 / 2.28	16,353 / 1.80	53,626 / 5.92
	$\mathsf{n} = 10\%N$	32,104 / 3.54	19,248 / 2.12	39,268 / 4.33
	$\mathsf{n} = 20\%N$	37,675 / 4.16	20,131 / 2.22	32,814 / 3.62
	$\mathsf{n} = 30\%N$	40,627 / 4.48	20,376 / 2.25	29,617 / 3.27

Table 5.1: Accumulated number of positive ("Pos"), negative ("Neg"), and zero ("Zero") slope signs across 90,620 empirical dataset-splits in total, derived from linear regressions on the final segments of the testing log-likelihood curves, for 9,062 empirical MSAs (10 splits per MSA). The table further reports the average number of slopes being positive, negative, and zero per dataset. The regression windows are: $\mathsf{n} = 2$, $\mathsf{n} = 10\%N$, $\mathsf{n} = 20\%N$, and $\mathsf{n} = 30\%N$. Each cell (in "Pos", "Neg" and "Zero" columns) reports two values in the format X / Y: X is the accumulated number of regression lines exhibiting the specified slope sign, and Y is the corresponding average per dataset (Table adapted from the Supplementary Material of [209]).

Stopping-enhanced version of RAxML-NG (see Chapter 4). The findings show that, for empirical MSAs, overfitting, as indicated by negative slope trends, is only detected for less than 1% of the MSAs, across all tools and MSA types (DNA or AA). The only exception is FastTree on AA MSAs, where overfitting is observed on approximately 1.5% of the datasets. This might be attributed to the extensive approximations that FastTree applies for log-likelihood calculations (see Section 2.6.7). FastTree also exhibits an increased ratio of positive slope trends, suggesting premature termination of the optimization process before reaching a plateau. In such cases, further optimization might be beneficial.

Across all tools, the majority of empirical MSAs (above 80%) show non-significant slope trends. This pattern is even more pronounced for simulated MSAs, where non-significant trends dominate both the testing log-likelihood and the topological accuracy curves. Interestingly, the excess of free substitution model parameters, introduced by analyzing the JC-generated simulated MSAs under the GTR+Γ model, does not induce topological overfitting. Instead, the tree inferences merely tend to converge more rapidly in such cases, potentially because the parsimony starting trees are more compatible with the JC model.

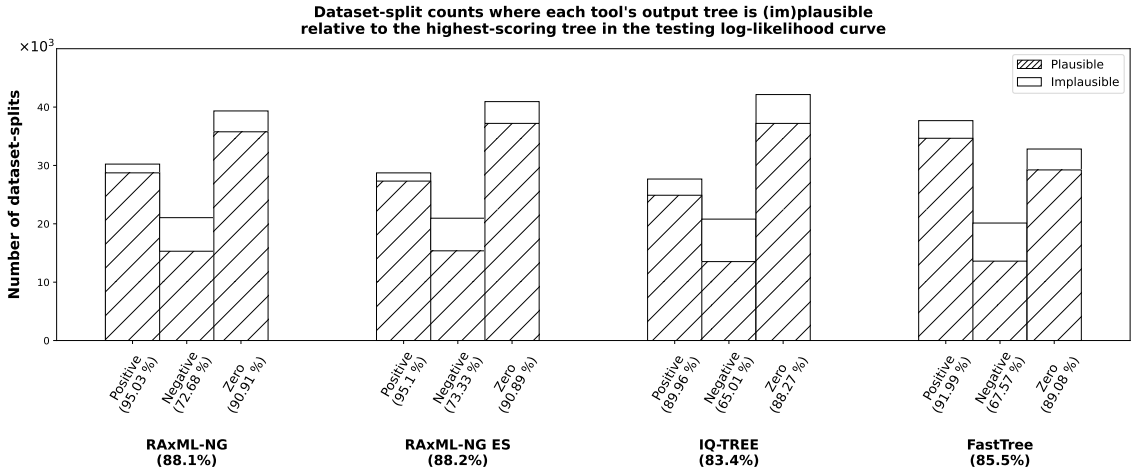


Figure 5.17: Barplot showing the accumulated number of positive, negative, and zero slope signs, across a total of 90,620 empirical dataset-splits. The slopes are derived from linear regressions on the final segments of the testing log-likelihood curves, for 9,062 empirical MSAs (10 splits per MSA). Bar heights represent the accumulated slope sign counts for $\mathsf{n} = 20\%N$. Results are shown separately for each inference tool. Within each bar, a distinct filling pattern indicates the percentage of cases where the final tree in the curve is plausible, compared to the highest-scoring tree (w.r.t. its log-likelihood on the testing MSA) on the same curve, as assessed via the AU test on the testing MSA. If the final tree is the highest-scoring one on the curve, it is directly marked as being plausible. I further report the percentages of plausible final trees in parentheses along the x-axis (for each slope sign category), and next to each tool label (aggregated across slope signs) (Figure reproduced from [209]).

Table 5.1 summarizes the accumulated number of positive, negative, and zero slope values, observed across all 90,620 dataset-splits of all 9,062 empirical MSAs (10 splits per MSA). The table also reports the average number of each slope type per dataset. Our results show that, across 10 inferences per MSA, an average number of 4–6 inferences result in a zero slope on the final segment of the curve. The only exception is FastTree that shows a majority of positive slopes for $\mathsf{n} = 20\%N$ and $\mathsf{n} = 30\%N$. This aligns with the earlier observation that FastTree might benefit from additional optimization. Further, positive slopes represent the second most frequent slope type, both in terms of accumulated counts, and average per-dataset values. On average, 0.5 to 1.5 more inferences per MSA show positive, rather than negative slope values.

There may be instances, however, where the regression-based approach on the final parts of the testing curves fails to capture the actual model-fit degradation (overfitting), which manifests itself via a decrease in the testing log-likelihood score. One such scenario arises when the optimal point occurs too early on the testing curve, and is therefore excluded from the regression window (e.g., split 7 in Figure 5.3). Indeed, the sign test-based approach identifies strong tendencies specifically in the final points of the testing curves for each MSA. I deliberately did not compare the optimal point on the curve with the final one. This is because, due to the "bumpy" nature of the (testing) log-likelihood curves, the optimal point is of reduced value. Nevertheless, in Figure 5.17, I do present a two-point comparison between the corresponding optimal and the final points. The Figure

comprises barplots that correspond to the accumulated number of positive, negative, and zero slope values, as observed across all 90,620 dataset-splits on empirical MSAs (10 for each MSA, across 9,062 MSAs), for $\mathsf{n} = 20\%N$. In addition to these slope type counts, the Figure also highlights the proportion of cases where the final ML tree is plausible (based on the AU test on the testing MSA) compared to the highest-scoring tree residing on the same testing log-likelihood curve. If the final tree is the highest-scoring one on the curve, I immediately consider it as being plausible. For RAxML-NG (both standard and ES), in 88% of dataset-splits the final tree is statistically indistinguishable from the optimal tree on the testing curve. In other words, under this plausibility test, overfitting is virtually absent in 88% of the cases. Moreover, in ∼73% of dataset-splits with negative regression slopes (for $\mathsf{n} = 20\%N$), the final and the optimal trees are statistically indistinguishable. For dataset-splits with positive regression slopes, the final tree is statistically equivalent to the optimal tree in 95% of cases, implying that in the remaining 5% the final tree is significantly worse.

To explain instances where the final ML tree is non-plausible, despite of positive or zero slope signs, we consider two main factors: (a) strong oscillations along the curve ("bumpy" nature of the log-likelihood curves), and (b) scenarios where the highest-scoring tree occurs early on the curve and is excluded from the regression window (see Figure 5.3). These effects are, indeed, not captured by the sign test, and require additional trials as well as data points to be accurately assessed. This slight discrepancy, however, is expected when applying multiple statistical evaluation methods. On the other hand, the comparative approach depicted in Figure 5.17 has a specific limitation relative to the slope analysis using the sign test: because the optimal tree is always at least as good as the final tree, the two-point comparison can only yield two outcomes: (i) a significant difference, indicating overfitting, or (ii) a non-significant difference, indicating a likelihood-equivalence plateau. It is therefore unable to detect a perpetually increasing model-fit towards the final stages of optimization (i.e., positive slopes), which however is equally important when examining the dynamics of the testing curves, and thus tends to exaggerate overfitting trends.

At this point, I recap and synthesize my findings. The sign-test analysis (Sections 5.2.1 and 5.4.1) showed that ML tools do not exhibit a strong tendency to overfit. Even when using thorough inference tools (RAxML-NG, IQ-TREE), topological overfitting is not statistically expected. At the same time, the two-point comparison (Figure 5.17) showed that in 12% of cross-validation splits (when using RAxML-NG), the model-fit degradation observed in the final tree is significant. This second result warns practitioners that, even if overfitting is not statistically expected in most cases, a downstream analysis of the inferred ML tree constitutes an important step of phylogenetic analysis. A fully resolved bifurcating tree can be over-parameterized, and some of its branches might not be supported by the data, as discussed in in [14]. J. Felsenstein formalized this idea via the so-called *null-branch test* [53]. The null-branch test is a likelihood-ratio test (LRT) whose null hypothesis states that the length of a fixed inner branch is zero. If the null hypothesis is not rejected, the branch is collapsed and yields a polytomy. Building on this idea, the approximate likelihood-ratio test (aLRT; [5, 69]) derives branch support values from likelihood comparisons and reports them as confidence-like measures on inner branches. An alternative approach is the standard bootstrap method [51], which assesses the stability of inner branches under random resampling of alignment sites with replacement, and thus addresses a related, but conceptually distinct support notion. Several other methods have also been proposed to detect/eliminate overfitted branches [125, 172, 181, 183,

233], while an alternative strategy introduces regularization via a penalized log-likelihood score [104]. Overall, while the impact of topological overfitting is not expected to be large, practitioners should be aware of it and apply appropriate post-inference assessments.

Finally, the RAxML-NG HV benchmark (Section 5.4.2) highlights the limitations of holdout-based Early Stopping in ML tree inference. First, HV consistently underperformed in terms of both, statistical plausibility (based on the AU test), and topological accuracy, even when analyzing long MSAs. This is consistent with the previous observations, that is, that topological overfitting does not constitute an issue in ML tree inference. Second, the HV versions did not yield any run time speedups over the ES version. Therefore, randomly excluding a portion of the MSA during the inference, decreases the available phylogenetic signal and degrades the tree inference quality [204]. I therefore recommend practitioners to conduct ML inference using the full MSA, either via RAxML-NG for maximum accuracy, or the ES version as a faster, yet reliable alternative.

5.6 Methods

5.6.1 Tools

For the overfitting evaluation, I executed RAxML-NG and IQ-TREE under a thorough setting. As discussed in Section 2.6.5 and Chapter 4, RAxML-NG uses a fixed log-likelihood improvement threshold (ε) to determine convergence of the tree inference process and subsequently terminate. I set an ε-threshold of 0.1 (`--lh-epsilon 0.1` option) in standard RAxML-NG executions, thereby protracting RAxML-NG's tree search until the log-likelihood improvements between consecutive SPR rounds fall consistently below the specified value.

Moreover, as discussed in Chapter 4, an ε-value of 0.1 is considered as a rather thorough setting in RAxML-NG. This value was the default in v1.1, yet was relaxed to 10 in v1.2, owing to the results of a systematic study [74], which showed that the original value was unnecessarily strict. Similarly, IQ-TREE uses a fixed ε-threshold of 0.1 to accept improvements in the current best-scoring topology; this parameter is not user-configurable. IQ-TREE only terminates after 100 consecutive NNI rounds fail to yield improvements, rendering its heuristic inherently exhaustive. Regarding continuous parameters, IQ-TREE performs full BLO on each *candidate tree* (see Section 2.6.6), while thorough MPOs are conducted periodically, similar to RAxML-NG. Importantly, in all executions of RAxML-NG (including the ES and HV versions) and IQ-TREE, I accounted for RHAS (see Section 2.5.3) using four discrete gamma categories, thus applying the GTR+Γ model for DNA, and the LG+Γ model for AA MSAs. Overall, both RAxML-NG and IQ-TREE employ thorough heuristics, combining strict ε-thresholds, exhaustive topological searches, and repeated optimizations of continuous parameters.

Investigating systematic overfitting under thorough heuristics can be informative with respect to whether the prolonged and compute-intensive optimization is justified, given the risk of inferring models that capture both signal and noise. To provide a contrasting perspective, I included FastTree in the comparison, which is a more superficial alternative. A brief overview of FastTree's heuristic is provided in Section 2.6.7. Importantly, FastTree does not account for gamma-distributed rate heterogeneity during tree search. Instead, it employs a variant of the CAT approximation [150], which assigns each site to a fixed, precomputed rate category (from up to 20 categories). I refer to this as a variant because FastTree's implementation differs from the original CAT approximation

described in [190]. Although FastTree offers an option to optimize gamma rates (via the `-gamma` option), this optimization is only applied to the final output tree and not during the tree search. I disabled this option. I additionally reduced the number of fixed rate categories to 4 (`-cat 4` option). This modification was intentional, as reducing the number of precomputed rate categories further simplifies the substitution model and accentuates the contrast between FastTree's superficial optimization strategy and the more thorough ML heuristics evaluated in this study. Therefore, I used the GTR+CAT4 model for DNA, and the LG+CAT4 model for AA MSAs. Overall, FastTree serves as a fast, yet superficial alternative, which I opposed to thorough heuristics, in order to directly compare overfitting trends under alternative ML tree inference strategies.

The final tool included in the statistical overfitting assessment is RAxML-NG ES. The ES version deploys a simplified heuristic (sRAxML-NG) in combination with the KH-multiple testing stopping criterion, and is described in detail in Section 4.2. RAxML-NG ES provides a balance between thorough versus superficial heuristics, employing a cautiously simplified topological optimization heuristic, while retaining the standard continuous parameter optimization methods.

Finally, RAxML-NG HV is an iterative, holdout-based Early Stopping algorithm, analogous to holdout methods in deep learning. After parsing the input MSA, HV randomly splits the alignment into model-training and validation sites based on a user-defined ratio. The algorithm conducts tree inference on the model-training sites, and monitors convergence with the validation sites. The tree search heuristic employed by HV (on the model-training sites) is almost identical to the sRAxML-NG heuristic, described in detail in Section 4.2.1 and schematically depicted in Figure 4.1. Hence, the parameter configuration of FAST/SLOW SPR rounds is identical to that of the sRAxML-NG heuristic.

Specifically, HV first optimizes branch lengths and model parameters on the parsimony starting tree. Next, it conducts a series of FAST SPR rounds ($r_{min} := 1$ and $r_{max} := 10$, see Section 4.2.1), evaluating each resulting topology on the validation MSA. HV interrupts this FAST-SPR sequence either when the validation log-likelihood decreases, or when improvement is below an ε-threshold of 0.1. In either case, HV retains the topology with the highest validation log-likelihood. The algorithm then conducts an MPO and continues with a sequence of SLOW SPR rounds. Here, it terminates either when the validation log-likelihood decreases, or does not improve, for at most k consecutive rounds. I say at most k, because if no SPR move is performed within an SPR round, this indicates that the topology is already SPR-optimal on the model-training MSA, and repeating the SPR round is redundant. In such a case, HV interrupts the SLOW-SPR sequence, even before reaching k rounds. After the SLOW SPRs, HV retrieves and prints the topology with the highest validation log-likelihood to file.

5.6.2 Tree Sampling

The initial strategy was to apply the testing sites—and thereby devise testing curves—on checkpoint trees printed by each tool under standard execution settings. However, differences in heuristics and in the frequencies of recording such checkpoints resulted in incomparable testing curves. To resolve this, I reduced the tree sampling process to the most atomic level by storing *all* intermediate, accepted topologies which improve upon the currently best log-likelihood score, as computed (i.e., approximated) by each tool. This sampling strategy ensures consistency across tools and improves curve resolution for

overfitting assessment. However, the ML tools used in this study do not provide built-in options to record all accepted topologies in the manner described. Therefore, I modified the source code of each tool to implement the above tree sampling strategy.

Implementing this fine-grained approach in RAxML-NG and FastTree was straightforward, whereas it proved to be more challenging for IQ-TREE. Both, RAxML-NG, and FastTree operate on a single tree during the tree inference, upon which they apply topological moves using greedy, hill-climbing heuristics. Within an SPR round in RAxML-NG, the algorithm prunes each subtree from the current tree and evaluates its possible re-insertions into neighboring branches, up to a certain radius from the pruning edge. The move that yields the highest score improvement, if such a move exists, is accepted; at this point I sample the updated topology, and the algorithm proceeds to the next subtree. The same sampling strategy is adopted in RAxML-NG ES, since the two versions invoke the same SPR round function, differing only w.r.t. to the input parameters. FastTree follows an analogous greedy approach (yet applies additional shortcuts), and, hence, I sample every accepted topology on FastTree's tree search trajectory.

The IQ-TREE heuristic differs substantially from the former tools, and hence the sampling strategy needs to be different. I describe its tree search heuristic in Section 2.6.6. The key difference is that IQ-TREE maintains a candidate tree set (CTS) during the tree search, that is, a fixed-size population of locally (sub-)optimal trees. At the beginning of every round, the algorithm randomly draws a tree from the CTS and applies random NNI moves to it (perturbation), to escape from the local optimum. The perturbed tree serves as input for the next hill-climbing NNI round. After each NNI round, both, the CTS, and the best-found ML tree are updated accordingly. The process is repeated and the algorithm terminates only after the best-found topology has remained unchanged for a predefined number of consecutive rounds (100 by default).

Given the additional complexity of the IQ-TREE heuristic, it is challenging to reduce the tree sampling process to its most atomic level. Specifically, in cases where the log-likelihood of the resulting topology after the application of multiple, sequential, concordant NNI moves (see Section 2.6.6) improves upon the score of the currently best-found ML tree, it is unclear via which move this so far best log-likelihood was surpassed. This is because, the resulting topology is only evaluated once, after all concordant NNI moves have been performed. Sampling only the final topology, which is guaranteed to improve the log-likelihood, may result in undersampling. Consequently, the sampling strategy I adopted for IQ-TREE involves sampling *all* intermediate topologies encountered during the sequential application of concordant NNI moves, when this set of moves yields an improved ML tree. This strategy might introduce a minor training curve oscillation. While the resulting training curves may resemble a stochastic optimization process, the generic trend remains increasing, capturing even marginal score improvements resulting from topological moves.

Finally, the current study investigates overfitting trends for each tool independently. That is, the devised testing curves are evaluated separately for each tool. The consistent tree sampling strategy among tools allows for meaningful cross-comparisons, so as to examine the effect of the specific heuristics employed by each method. The tool-dependent approximations of log-likelihood score calculations for each visited topology directly affect the search trajectories. For instance, RAxML-NG and IQ-TREE rely upon more accurate likelihood calculations than FastTree, thereby guiding the search towards topologies that do not systematically degrade model-fit (on unseen sites), whereas FastTree exhibits such a degradation more frequently. However, the testing curves devised by RAxML-NG and

FastTree exhibit greater similarity than those from IQ-TREE, rendering them directly comparable. This is because the devised testing curves comprise an incremental sequence of SPR-neighboring topologies, where each accepted topology is an SPR-neighbor of its predecessor, yielding a smooth tree space trajectory. This is not the case for IQ-TREE, and the apparent discontinuities in IQ-TREE testing curves may complicate direct comparisons between IQ-TREE and the other tools.

5.7 Data and Software Availability

I implement both RAxML-NG HV, and the tree sampling algorithm, in a forked RAxML-NG v1.2 repository. These modifications are available as open source code under GNU GPL at `https://github.com/togkousa/raxml-ng/tree/overfitting-early-stopping`. This repository further includes the modified version of FastTree v2.1 that implements the sampling algorithm. FastTree is added as a separate folder of the same repository and is available as open source code under GNU GPL. Finally, the sampling algorithm has been integrated into IQ-TREE v2.3 and is available as open source code under GNU GPL at `https://github.com/togkousa/iqtree2/tree/overfitting`. Invocations for all tools are detailed in Section 5.3.2. All datasets and results generated in this study are publicly available at `https://cme.h-its.org/exelixis/material/overfitting_data.tar.gz`.

6. Towards RAxML-NG v2.0

6.1 Introduction

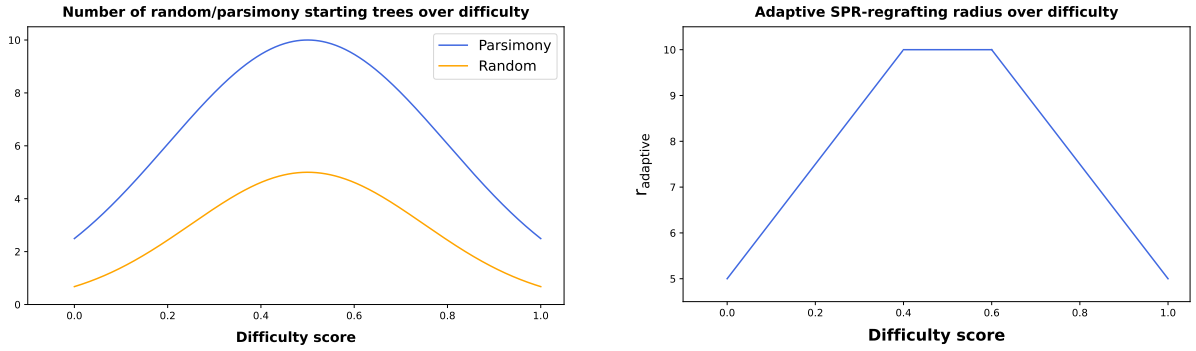
In this Chapter, I provide a full description of the default heuristic that RAxML-NG v2.0 deploys. As already mentioned in the Introduction (see Section 1.2), this new default heuristic synthesizes ideas from the Adaptive (see Chapter 3) and the Early Stopping (ES; see Chapter 4) versions. I call it the *Hegelian heuristic*, as it represents a trade-off between speed and accuracy. In addition to the Hegelian heuristic, RAxML-NG v2.0 implements an alternative, fast heuristic, that enables rapid inference of very large phylogenies. This fast heuristic is essentially already described in Chapter 4, and it correspond to the simplified heuristic (sRAxML-NG) combined with the KH-multiple testing stopping criterion.

Therefore, this Chapter describes the novel RAxML-NG v2.0 heuristics. I further describe a cost-free, algorithmic optimization that I implemented for the SPR rounds in RAxML-NG v1.2[1], which yielded an average 1.7× speedup (and up to 5× on a dataset comprising 7,664 taxa) over RAxML-NG v1.1. Finally, I present preliminary benchmark results on the 222 empirical DNA MSAs used in Chapter 4 (Section 4.4.1). These are preliminary benchmarks, because a comprehensive benchmark of RAxML-NG v2.0 is planned for the corresponding upcoming paper, which we (as a lab) intend to submit for peer-review in 2026. In this preliminary evaluation, the following tools/heuristics are compared:

1. RAxML-NG v1.1;
2. the default Hegelian heuristic of RAxML-NG v2.0;
3. the default heuristic of IQ-TREE 3 [227];
4. the fast heuristic of RAxML-NG v2.0;
5. the VeryFastTree tool [148], which is a parallel implementation of FastTree 2 [148];
6. the fast tree search mode of IQ-TREE 3 [227].

Although all heuristics/tools are jointly compared, it is important to distinguish between two groups: the thorough methods—RAxML-NG v1.1, the default Hegelian heuristic of RAxML-NG v2.0, and the default heuristic of IQ-TREE—, and the fast alternatives—the fast RAxML-NG v2.0 heuristic, VeryFastTree, and IQ-TREE's fast mode. Faster alternatives cannot reasonably compete with the thorough methods in terms of accuracy, just as the thorough methods do not match the faster ones in terms of speed. The joint comparison is nevertheless informative, as it reveals the extent to which the fast alternatives trade accuracy for speed, and which method attains the best trade-off. The benchmark indicates that the fast RAxML-NG search strategy is the most effective accelerated heuristic, and outperforms the other fast alternatives, both in terms of accuracy, and speed.

[1] See the RAxML-NG v1.2 GitHub release [112].



(a) (b)

Figure 6.1: (a) Number of random and parsimony starting trees that are used by the Hegelian RAxML-NG heuristic to conduct independent tree inferences, as a function of the difficulty score. **(b) Adaptive SPR-regrafting radius** parameter over the difficulty score in RAxML-NG v2.0.

6.2 Personal Contributions to RAxML-NG v2.0

In this section, I initially describe all ML tree inference heuristics that can be invoked in RAxML-NG v2.0. I then present the algorithmic optimization I introduced for the SPR rounds, which consists in applying incremental CLV updates (see Section 2.6.5). In fact, I substituted a full CLV recomputation by an incremental CLV update at a certain point of the SPR round; namely, at the beginning of every SPR round iteration, before pruning the next subtree and regrafting it onto candidate reinsertion edges (see Section 2.6.4). This simple modification yielded an average 1.7× acceleration of RAxML-NG, which scales with the number of taxa, without altering the results.

6.2.1 RAxML-NG v2.0 Heuristics

RAxML-NG v2.0 implements five ML tree inference heuristics that the user can invoke: (i) the default Hegelian heuristic; (ii) the fast RAxML-NG heuristic; (iii) an adaptive-fast heuristic alternative (described below); (iv) the standard RAxML-NG v1.2 heuristic; and (v) the NNI round heuristic. In the following, I discuss these heuristics separately. Since some of the heuristics have already been described in earlier Chapters, I refer the reader to the corresponding Sections where appropriate.

The Default Hegelian Heuristic

Figure 6.1a shows the number of random and parsimony starting trees that RAxML-NG v2.0 initiates by default, as a function of the input MSA's difficulty score, predicted by the Pythia v2.0 tool [75]. For both random and parsimony starting trees, the function is a bell-shaped curve with a peak at a difficulty score of 0.5. The number of parsimony starting trees reaches a peak of 10 at this difficulty level, whereas the number of random starting trees peaks at 5. This is due to the observations made in Chapter 4 (see the Discussion in Section 4.5), which question the overall utility of random starting trees. In general, the contribution of random starting trees to obtaining the best-known tree is negligible, suggesting that their number should be reduced. However, extensive experiments with ML heuristics while tuning the default RAxML-NG v2.0 heuristic revealed that,

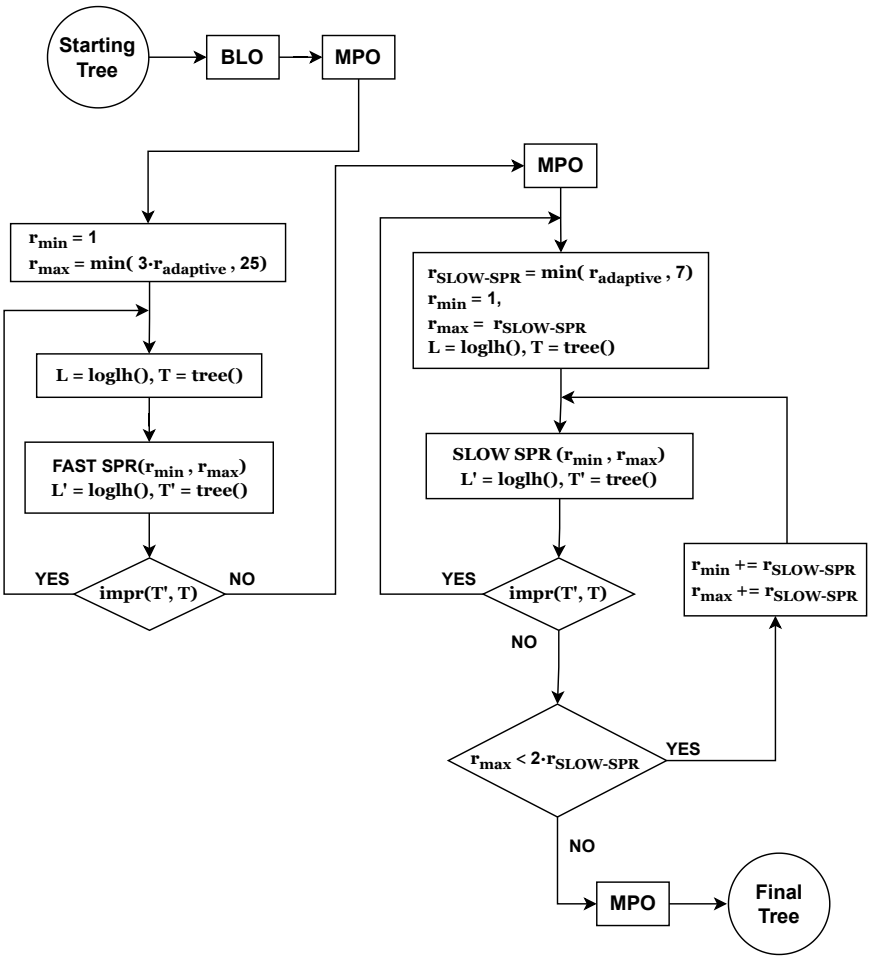


Figure 6.2: A schematic representation of the default RAxML-NG v2.0 tree inference heuristic.

in some rare cases, inferences starting on random starting trees may yield significantly better trees (based on the AU test) than searches starting on parsimony trees. Hence, random starting trees cannot be entirely discarded. Empirical results indicate that a peak of five random starting trees at a difficulty score of 0.5 is sufficient to consistently infer plausible ML trees, compared to earlier versions of RAxML-NG.

Figure 6.1b shows the adaptive SPR-regrafting radius parameter as a function of the difficulty score, hereafter denoted as r_{adaptive}. This parameter starts from a value of 5 for a difficulty score of 0.0, and increases linearly to a value of 10 at a difficulty of 0.4. It then remains constant up to a difficulty of 0.6, and then it decreases linearly again to a value of 5 at a difficulty score of 1.0. The r_{adaptive} parameter determines the thoroughness of each independent tree inference as a function of the difficulty score. Furthermore, it is multiplied by different factors in the two distinct phases of the default RAxML-NG heuristic (as well as in the adaptive-fast heuristic), to yield the minimum and maximum SPR-regrafting radii (see below).

In the following, I assume that the reader is familiar with the concepts from Chapter 2, and especially with Section 2.6, which provides a detailed description of the RAxML-NG v1.x heuristic. In Section 2.6 the reader can find relevant information on, for example,

the distinction between FAST and SLOW SPR rounds, or the continuous parameter optimization routines in RAxML-NG, namely the Branch-Length Optimization (BLO) and Model Parameter Optimization (MPO) rounds. These details are omitted here.

The default Hegelian heuristic of RAxML-NG v2.0 for each independent ML tree inference is illustrated in Figure 6.2. The functions L = `loglh()` and T = `tree()`, shown in the workflow blocks before and after SPR rounds, return the current best log-likelihood score L and the corresponding current best tree T, respectively. After the initial BLO and MPO rounds (which use the same settings as in RAxML-NG v1.2), the algorithm executes two main phases:

1. **The FAST SPR phase**: The algorithm conducts a sequence of FAST SPR rounds (see Section 2.6.4) with minimum and maximum SPR regrafting radii set to $r_{min} := 1$ and $r_{max} := \min(3 \cdot r_{adaptive}, 25)$, respectively. The value of $r_{adaptive}$ is determined by the function shown in Figure 6.1b. The sequence of FAST SPR rounds continues as long as the log-likelihood improvement between the tree T prior to a FAST SPR round (with log-likelihood L) and the tree T′ after the SPR round (with log-likelihood L′) is significant. The decision function `impr`(T, T′), illustrated by the rhombus-shaped decision block in the workflow, determines whether the improvement is significant. This evaluation is performed either by using a fixed log-likelihood ε-threshold, or statistically by employing the KH-based stopping criterion (described in Chapter 4). By default, the Hegelian heuristic uses an ε-threshold of $\varepsilon := 10$, which is the same as in RAxML-NG v1.2. Alternatively, the user may enable the KH-based stopping criteria via the `--stop-rule KH` option (for the simple KH test) or the `--stop-rule KH-mult` option (for the KH test with multiple-testing correction).

2. **The SLOW SPR phase**: After this sequence of FAST SPR rounds, the algorithm performs an MPO round and enters the SLOW SPR phase, which is divided into two stages. During the first stage (Stage I), the SPR round is invoked with $r_{min} := 1$ and $r_{max} := \min(r_{adaptive}, 7)$, respectively. The tree T (with log-likelihood L) prior to Stage I is stored. After the SLOW SPR round, the resulting tree T′ (with log-likelihood L′) is also stored. The two trees T and T′ are then compared using the function `impr`(T, T′), in an analogous way as in the FAST SPR rounds (i.e., either using the default ε-threshold of 10, or a user-enabled statistical test). If the improvement is significant, the algorithm returns to the state preceding Stage I, otherwise it proceeds to Stage II. In Stage II, the SLOW SPR round is invoked with $r_{min} := r_{max} + 1$ and $r_{max} := 2 \cdot r_{max}$, respectively. Only the resulting tree after the SPR round is stored as T′ (with log-likelihood L′), while the reference tree T (with log-likelihood L) remains the tree prior to Stage I, that is, the tree before the invocation of the SPR round with radii $r_{min} := 1$ and $r_{max} := \min(r_{adaptive}, 7)$, respectively. The two trees T and T′ are then compared again. If the log-likelihood improvement is significant, the algorithm returns to the state preceding Stage I. Otherwise, the algorithm exits the sequence of SLOW SPR rounds, performs an MPO round, and prints the best-found ML tree to file. In other words, the sequence of SLOW SPR rounds only terminates when two consecutive SPR rounds (the first with $r_{min} := 1$ and $r_{max} := \min(r_{adaptive}, 7)$, and the second with $r_{min} := r_{max} + 1$ and $r_{max} := 2 \cdot r_{max}$, respectively) yield non-significant log-likelihood improvements, with respect to the tree T prior to the first SPR round.

In both FAST and SLOW SPR rounds, the default heuristic maintains a list of the $X := 20$ best-scoring topologies encountered during the search. It re-evaluates all trees in this list via full BLOs after the FAST/SLOW SPR round (see Section 2.6.5). This parameter setting is identical to that used in RAxML-NG v1.2. Moreover, in both FAST and SLOW SPR rounds, the subtree cutoff feature is enabled, with an initial threshold value of $\varepsilon^0_{cutoff} = -L_0 \ / \ 100$ (see Section 2.6.5). This initial threshold is higher than in RAxML-NG v1.2, where the corresponding value was set to $-L_0 \ / \ 1,000$. Practically, this results in a larger number of SPR topologies being evaluated during the initial invocation(s) of both the FAST and the SLOW SPR rounds. RAxML-NG v2.0 resets the subtree cutoff threshold to its initial value after each MPO round, that is, before entering the FAST and SLOW SPR phases.

Note that, in addition to the KH-based stopping criteria described in Chapter 4 and in the respective publication [211], the user can also invoke two additional stopping criteria. For the Early Stopping project, I also experimented with statistical criteria that attempt to quantify the MSA sampling noise. These stopping criteria are the Sampling Noise Normal (SN-Normal; `--stop-rule sn-normal`) and the Sampling Noise RELL (SN-RELL; `--stop-rule sn-rell`) criterion. I do not provide a description of those criteria neither in this thesis nor in the respective publication, as their performance was substantially inferior to that of the KH-based stopping criteria. Nevertheless, I provide a full description of the methods, along with the (negative) benchmark results in my GitHub repository: `https://github.com/togkousa/raxml-ng/tree/stopping-criteria`.

The Fast Heuristic

RAxML-NG v2.0 also implements a very fast search heuristic alternative, that is particularly useful for the rapid inference of very large phylogenies. This very fast mode essentially combines the Simplified RAxML-NG heuristic (sRAxML-NG; see Chapter 4) with the KH multiple-testing stopping criterion. A schematic representation of the sRAxML-NG heuristic is shown in Figure 4.1. RAxML-NG v2.0 executes this very fast heuristic by only conducting a single ML tree inference, initiated on a parsimony starting tree.

The user can invoke this fast heuristic via the `--fast` option. Alternatively, the sRAxML-NG heuristic can be enabled by using the `--opt-topology simpl` option, optionally in combination with the `--stop-rule KH-mult` option, which enables the KH multiple-testing criterion. In effect, the `--fast` option serves as a shortcut for this command combination. The sRAxML-NG heuristic can also be invoked using the `--opt-topology simpl` option alone, without enabling the KH multiple-testing stopping rule. In this case, sRAxML-NG uses a fixed log-likelihood threshold of $\varepsilon := 10$ to determine tree search convergence.

Importantly, the `--fast` option also fixes the number of starting trees to 1 parsimony tree and 0 random trees. By contrast, when using the `--opt-topology simpl` command, the number of starting trees follows the default RAxML-NG v1.2 configuration, namely 10 parsimony and 10 random starting trees. To obtain an execution equivalent to the `--fast` mode when using `--opt-topology simpl`, the user must additionally set the number of parsimony starting trees to 1 via the `--tree pars{1}` option.

The Adaptive-Fast Heuristic

While developing and tuning the default tree inference heuristic in RAxML-NG v2.0, I designed the so-called *Adaptive-Fast* heuristic. This heuristic is illustrated in Figure 6.3.

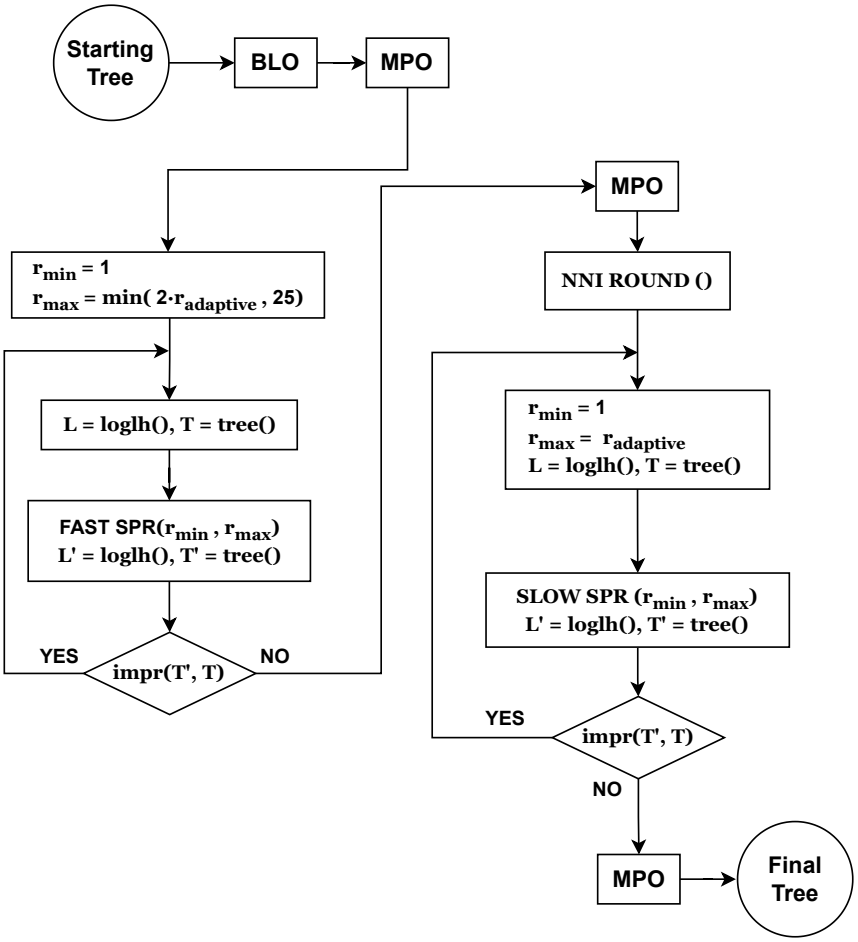


Figure 6.3: A schematic representation of the Adaptive-Fast tree inference heuristic in RAxML-NG v2.0.

In terms of performance, the Adaptive-Fast heuristic is faster than the Hegelian heuristic but substantially slower than the very fast heuristic. At the same time, its accuracy—measured as the proportion of plausible ML trees inferred according to the AU test—is situated between the Hegelian heuristic and the very fast mode. Since the Hegelian heuristic outperformed the Adaptive-Fast heuristic in terms of accuracy, and the primary goal was to provide the most accurate possible default heuristic, the Hegelian heuristic was selected as the default. Nevertheless, RAxML-NG v2.0 still offers the Adaptive-Fast heuristic, which can be invoked via the `--opt-topology adafast` option.

The Adaptive-Fast heuristic uses the functions illustrated in Figure 6.1a to determine the number of starting trees as a function of the input MSA difficulty score. However, it only uses parsimony starting trees. This means that it aggregates the numbers derived from both bell-shaped curves in Figure 6.1a and uses their sum as the total number of parsimony starting trees.

For each starting tree and after performing the initial BLO and MPO rounds, the Adaptive-Fast heuristic is again divided into two SPR phases. In the **FAST SPR phase**, the minimum and maximum SPR-regrafting radii are set to $r_{min} := 1$ and $r_{max} := \min(2 \cdot r_{adaptive}, 25)$, respectively. The value of $r_{adaptive}$ is determined by the function shown in

Figure 6.1b. The sequence of FAST SPR rounds continues as long as the log-likelihood improvement between the tree T prior to a FAST SPR round (with log-likelihood L) and the tree T′ after the SPR round (with log-likelihood L′) remains significant. The decision function `impr`(T, T′) determines whether the improvement is significant, and uses the KH-multiple testing stopping rule as a default criterion. The user may modify this behavior by selecting, for example, the KH-simple criterion via `--stop-rule KH`, or by entirely disabling the statistical stopping criteria via the `--stop-rule off` option; in the latter case, an ε-threshold of $\varepsilon := 10$ is used to assess convergence. If the improvement is insignificant, the algorithm exits the FAST SPR phase.

Next, the algorithm performs an MPO round, followed by an NNI round. The NNI round is described in detail in Section 2.6.4. Subsequently, the Adaptive-Fast heuristic enters the **SLOW SPR phase**. In this phase, it sets $r_{min} := 1$ and $r_{max} := r_{adaptive}$, again as determined by the function shown in Figure 6.1b. Analogously to FAST SPR rounds, the sequence of SLOW SPR rounds continues as long as the log-likelihood improvement between the tree T prior to a SLOW SPR round (with log-likelihood L) and the tree T′ after the SPR round (with log-likelihood L′) is significant (see above). When the sequence of SLOW SPR rounds terminates, the algorithm conducts a final MPO and returns the best-found ML tree.

In both FAST and SLOW SPR rounds, the Adaptive-Fast heuristic maintains a list of the $X := 20$ best-scoring topologies, similar to the Hegelian heuristic. Moreover, in both FAST and SLOW SPR rounds, the subtree cutoff is enabled, with an initial value of $\varepsilon^0_{cutoff} = -L_0 \,/\, 1,000$ (see Section 2.6.5), matching the setting used in RAxML-NG v1.2. This initial threshold is reset at the beginning of both FAST and SLOW SPR phases. Finally, the NNI tolerance parameter is set to $\varepsilon_{NNI} := 1.0$ (see Section 2.6.4); this value is increased compared to the Adaptive RAxML-NG heuristic, where the corresponding ε_{NNI} parameter is 0.1.

The benchmark results presented in Section 6.4 do not include the Adaptive-Fast heuristic. Since this heuristic was not selected as the default, it is more informative to focus the comparison on the advantages of the default Hegelian heuristic over earlier RAxML-NG versions and other ML tools, as well as on the performance of the very fast heuristic. As already mentioned, the benchmark results presented here are preliminary, and a more comprehensive evaluation is planned for the upcoming RAxML-NG v2.0 paper.

The Standard RAxML-NG v1.2 Heuristic

RAxML-NG v2.0 allows to invoke the standard heuristic of RAxML-NG v1.2 via the `--opt-topology classic` option. This heuristic is described in detail in Section 2.6.5. The only difference compared to v1.2 is that, in RAxML-NG v2.0, the number of random and parsimony starting trees is determined by the bell-shaped functions shown in Figure 6.1. Hence, a difficulty prediction on the input MSA is performed by Pythia in the beginning.

The NNI Round

The final ML tree inference heuristic offered by RAxML-NG v2.0 is the NNI round. This heuristic can be invoked using the `--opt-topology nni-round` option. By default, the numbers of random and parsimony starting trees to be generated are determined by the bell-shaped functions shown in Figure 6.1a. Hence, a difficulty prediction of the input

MSA is performed by Pythia in the beginning. For each starting tree, the algorithm first performs an MPO round and subsequently invokes the NNI round, as described in detail in Section 2.6.4, using an NNI tolerance of $\varepsilon_{NNI} := 1.0$. Once a local NNI optimum is reached, the algorithm performs a full BLO round and returns the resulting NNI-optimal tree.

This heuristic is particularly useful when the user has already inferred an optimal or near-optimal tree (for example, stored in a file called `user.tree`) using any tree inference method, and intends to compute SH-aLRT branch support values [69]. RAxML-NG v2.0 implements this branch support metric via the `--sh` option. Since the SH-aLRT test assumes an NNI-optimal input tree, the user must first refine the topology using the NNI option, to ensure NNI-optimality. The SH-aLRT test can then be applied to the resulting tree. A complete command for this workflow is:

```
./raxml-ng-2 --msa {msa} --model {model} --tree user.tree
--opt-topology nni-round --sh
```

6.2.2 Optimization in the SPR Round

In an SPR round, when RAxML-NG evaluates a visited SPR-topology (i.e., a topology resulting from an SPR move), it does not perform a full CLV recomputation. Instead, it employs incremental CLV updates, that are described in detail in Section 2.6.5. In brief, for a given SPR-topology, not all CLVs need to be recomputed in order to evaluate its log-likelihood. For example, in an SPR move where a subtree P is pruned from branch b_1 and re-inserted into a different branch b_2, the two trees before and after the SPR move have at least one (and typically not only one) common subtree, namely the subtree P itself. For all such common subtrees, the respective CLVs, which descend from the subtree's root towards their taxa, remain unaffected by the move, and do therefore not require an update. RAxML-NG can efficiently identify the subset of CLVs that must be recomputed via the incremental CLV updating mechanism.

As described in Section 2.6.4, when RAxML-NG invokes the SPR round routine, the algorithm first performs a full tree traversal starting from the current (virtual) root node, using Left-Child Moves (LCMs), Right-Child Moves (RCMs), and a single Root-Back Move (RBM). During this traversal, it stores all the inner nodes of the tree topology. For each node-pointer u in the stored list, the algorithm:

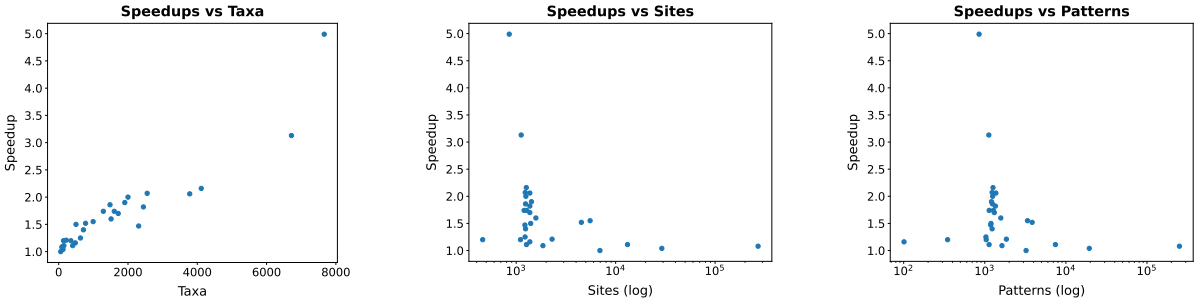
1. Re-roots the tree to node u and updates all CLVs with respect to the new root.
2. Prunes the corresponding subtree with root u (the one that extends in the direction of LCM and RCM traversal moves, see Section 2.6.5) and evaluates its possible reinsertions into neighboring branches.
3. The move that yields the highest likelihood improvement, if such a move exists, is accepted.
4. The algorithm proceeds to the next node-pointer in the list and returns to Step 1, until the list is empty.

While experimenting with `coraxlib` [47], I observed that the CLV updates after the re-rooting step (Step 1 above) were not incremental. Instead, RAxML-NG computed all tree CLVs from scratch in Step 1, while at the same time, the CLV computations performed in Steps 2–3 were all incremental. I therefore replaced the full CLV recomputation in Step 1 with an incremental CLV update and conducted benchmark experiments to assess the runtime gains achieved by this modification.

As a benchmark dataset, I used a collection of empirical datasets provided by my

Dataset	Taxa	Sites	Patterns	Speedup
59.phy	59	6,951	3,230	1.00
94.phy	94	270,580	251,152	1.08
101.phy	101	1,857	1,629	1.09
125.phy	125	29,149	19,436	1.04
140.phy	140	1,104	1,041	1.20
150.phy	150	1,269	1,130	1.11
218.phy	218	2,294	1,846	1.21
354.phy	354	460	348	1.20
404.phy	404	13,158	7,429	1.11
479.phy	479	1,371	1,011	1.16
500.phy	500	1,398	1,193	1.50
628.phy	628	1,228	1,033	1.25
714.phy	714	1,241	1,231	1.40
775.phy	775	4,519	3,838	1.52
994.phy	994	5,533	3,363	1.55
1288.phy	1,288	1,200	1,132	1.74
1481.phy	1,481	1,241	1,241	1.86
1512.phy	1,512	1,577	1,576	1.60
1604.phy	1,604	1,276	1,275	1.74
1718.phy	1,718	1,371	1,310	1.70
1908.phy	1,908	1,424	1,209	1.90
2000.phy	2,000	1,251	1,251	2.00
2308.phy	2,308	1,224	1,184	1.47
2445.phy	2,445	1,371	1,351	1.82
2554.phy	2,554	1,232	1,232	2.07
3782.phy	3,782	1,371	1,370	2.06
4114.phy	4,114	1,263	1,263	2.16
6718.phy	6,718	1,122	1,122	3.13
7664.phy	7,664	851	851	4.99

Table 6.1: Per-dataset speedups attained through incremental CLV updates after re-rooting in the SPR round. For each dataset, the number of taxa, sites, and patterns is displayed.



(a) (b) (c)

Figure 6.4: Scatter plots showing the speedups attained through incremental CLV updates after re-rooting in the SPR round, as a function of (a) taxa, (b) sites, and (c) patterns. In subplots (b) and (c), the x-axes are shown on a logarithmic scale.

supervisor, Alexandros Stamatakis [189]. This dataset has been used to benchmark RAxML in one of the classic RAxML publications [197]. For each dataset, I conducted two tree inferences using RAxML-NG v1.1, using the same random starting tree. The first inference used incremental CLV updates after the re-rooting step in the SPR round (Step 1), while the second one performed full CLV recomputations. I used 4 threads for both tree inferences. As test platform I used an Intel(R) Xeon(R) Platinum 8260 48 double-core processor running at 2.4 GHz.

The runtime improvements are shown in Figure 6.4. Speedups scale linearly with the number of MSA taxa, whereas no clear trend is observed with respect to the number of MSA sites or patterns. The speedup values are also reported numerically in Table 6.1. As shown in the Table, replacing the full CLV recomputation with incremental CLV updates after re-rooting the tree in the SPR round, yields a speedup of up to 5× for the dataset with 7,664 taxa. This optimization was integrated into RAxML-NG v1.2 (see the respective version release on GitHub [112]).

6.3 Benchmarking RAxML-NG v2.0

The following experiments were conducted by fellow lab member Dimitri Höhler. He used PhyloSmew [89], a pipeline he developed to benchmark phylogenetic inference tools. In the next Section I present preliminary benchmark results for 222 empirical DNA MSAs sampled from TreeBASE [147], which were also used to benchmark the ES version of RAxML-NG [211] (see Chapter 4). The PhyloSmew pipeline failed for one dataset, which was therefore excluded from the analysis, yielding a total of 221 empirical DNA MSAs. For each input MSA, the following tools were executed:

1. RAxML-NG v1.1;
2. RAxML-NG v2.0 (default);
3. IQ-TREE 3 [227] (default);
4. the fast heuristic of RAxML-NG v2.0 (`--fast` option);
5. VeryFastTree [148], the parallel implementation of FastTree 2;
6. the fast tree search mode of IQ-TREE 3 [227], which conducts a substantially accelerated version of IQ-TREE's NNI heuristic.

Each tool/version conducts a different number of independent tree searches. Specifically, RAxML-NG v1.1, by default, conducts 20 independent tree searches, with 10 random and

10 parsimony starting trees, whereas the default RAxML-NG v2.0 heuristic determines the number of tree searches from the input MSA's Pythia score, using the functions shown in Figure 6.1a. The fast mode of RAxML-NG only conducts a single tree inference using a parsimony starting tree. Further, the default heuristic of IQ-TREE 3 begins by generating 99 parsimony and one BioNJ tree [61], which populate the initial candidate tree set (see Section 2.6.6), whereas the fast IQ-TREE mode initializes the candidate set with only one parsimony tree and one BioNJ tree. Finally, VeryFastTree performs a single tree inference on a starting tree generated via a heuristic variant of the Neighbor-Joining algorithm [151]. The number of tree searches was left unchanged for each tool/version.

The PhyloSmew pipeline exclusively focuses on the best ML trees (i.e., the output trees) inferred by each tool/version. After completing the tree inferences and collecting the best ML tree from each tool, PhyloSmew re-evaluates all six trees using RAxML-NG v2.0. The tree with the highest re-evaluated log-likelihood serves as the reference tree for the dataset. PhyloSmew statistically compares the six trees using the AU test [178], as implemented in the CONSEL tool [179]. Here, a ML tree is considered to be plausible if at least one of the following conditions holds: (a) the AU test p-value exceeds 0.05; (b) its log-likelihood score after re-evaluation is less than 0.5 units worse than that of the reference tree; or (c) it is topologically identical to the reference tree. Conditions (b) and (c) are important to avoid numerical instabilities in CONSEL, especially when the log-likelihood scores of two trees under comparison are very similar.

I then aggregate the number of datasets for which each tool/version yielded a plausible ML tree, and use this number as the accuracy metric for the evaluation. Additionally, I compare the runtimes of all tools relative to RAxML-NG v1.1, which conducts the most thorough tree search and is therefore expected to be the slowest. Importantly, Dimitri Höhler executed each tool/version twice; once using 4 threads, and once using 8 threads. All tools inferred identical trees under both configurations; the only difference was their relative runtime compared to RAxML-NG v1.1, which also reflects how well each tool scales when more threads are being used. This benchmark was conducted on one of our lab servers, equipped with two AMD EPYC 9684X (Genoa-X) processors, each with 96 physical cores and two threads per core, operating at a maximum frequency of 3.72 GHz. The operating system is Ubuntu 24.04.

6.4 Results

Figure 6.5 illustrates the proportion of plausible trees for each tool/version, across 221 empirical DNA MSAs from TreeBASE. In this Figure, the tools/versions are grouped into two categories: (a) the *thorough tools*, which are RAXML-NG v1.1, the default RAxML-NG v2.0 search heuristic, and the default IQ-TREE 3 search heuristic; and (b) the *faster alternatives*, that is, the fast RAxML-NG v2.0, the fast IQ-TREE 3, and VeryFastTree. Both RAxML-NG v1.1 and the default RAxML-NG v2.0 heuristic inferred plausible ML trees for all datasets (100%). The performance of the default IQ-TREE 3 heuristic is slightly worse, yielding plausible trees for 96.4% of the datasets (213 out of 221 MSAs). Interestingly, the fast mode of RAxML-NG substantially outperforms both fast IQ-TREE 3 and VeryFastTree, while achieving an inference accuracy comparable to that of default IQ-TREE 3. Specifically, fast RAxML-NG v2.0 inferred plausible trees for 211 out of 221 MSAs (95.5%), whereas fast IQ-TREE 3 and VeryFastTree yielded plausible trees for only 95 (43%) and 20 (9%) out of 221 MSAs, respectively.

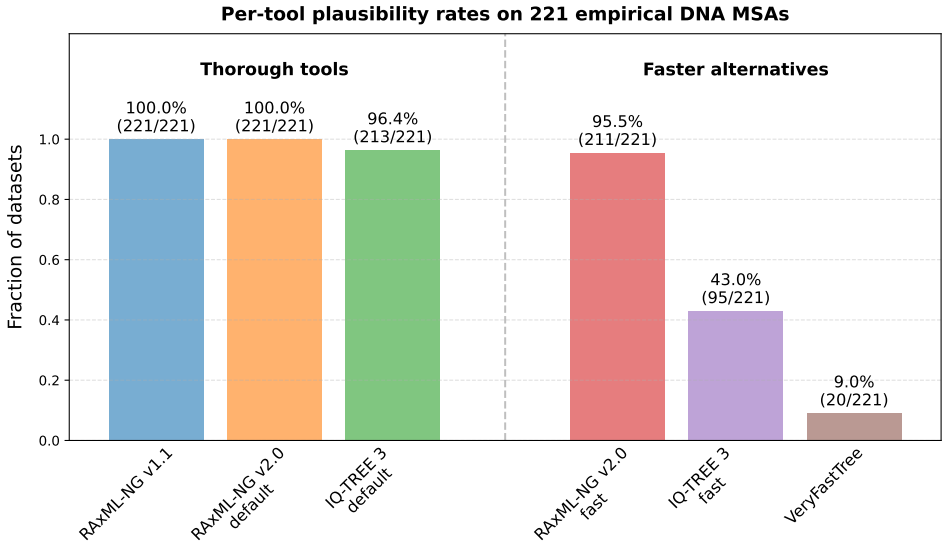


Figure 6.5: Plausibility assessment results on 221 empirical DNA MSAs. The tools/versions are grouped into two categories: the thorough tools (RAxML-NG v1.1, RAxML-NG v2.0 default, and IQ-TREE 3 default), and the faster alternatives (fast RAxML-NG v2.0, fast IQ-TREE 3, and VeryFastTree). The numbers above the bars indicate both the percentage and the absolute number of datasets on which the respective tool inferred a plausible tree.

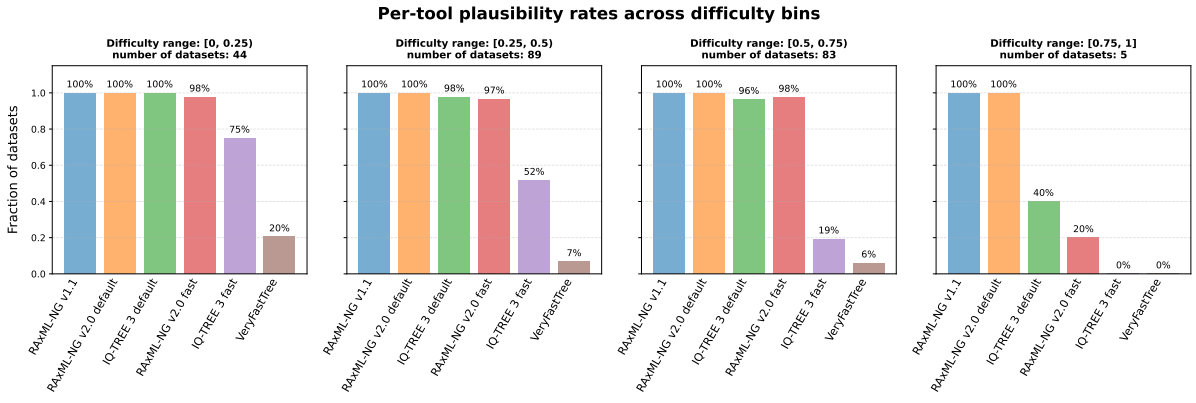


Figure 6.6: Plausibility assessment results on 221 empirical DNA MSAs, across four MSA difficulty bins. Each subplot corresponds to a distinct MSA difficulty range. At the top of the bars, the percentage of datasets on which the respective tool inferred a plausible tree is reported.

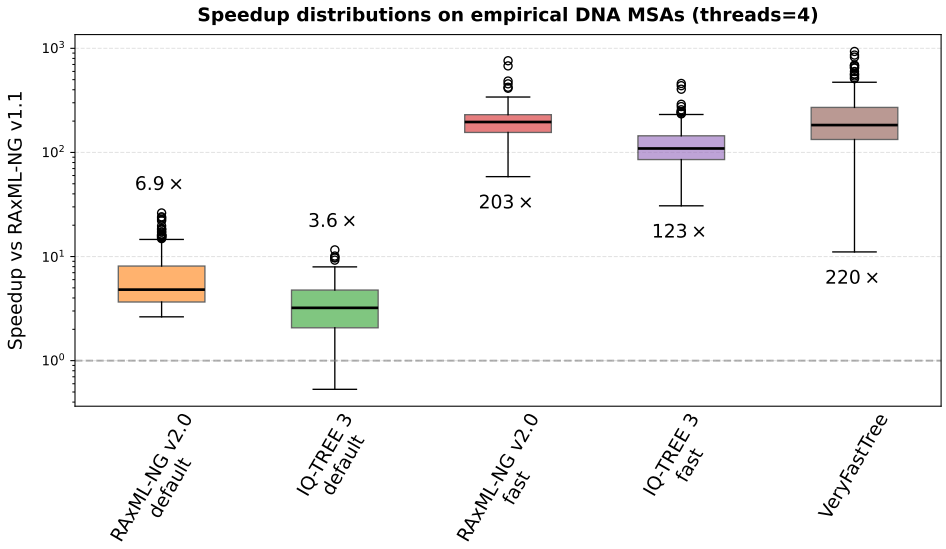


Figure 6.7: Speedup distributions of RAxML-NG v2.0 (default and fast modes), IQ-TREE 3 (default and fast modes), and VeryFastTree relative to RAxML-NG v1.1, on 221 empirical DNA MSAs, when using 4 threads. The average speedup for each tool is also reported above or below the respective boxplot, depending on the available space. The vertical axis is shown on a logarithmic scale.

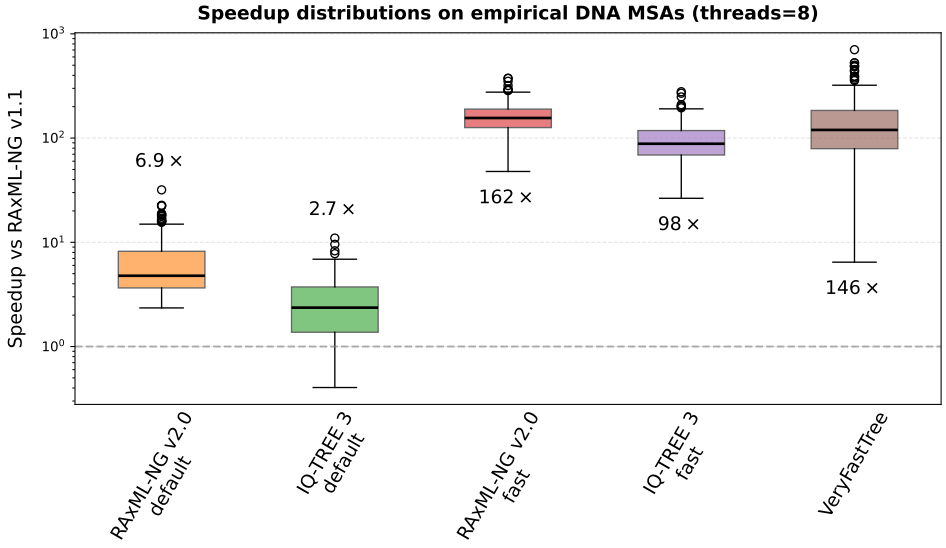


Figure 6.8: Speedup distributions of RAxML-NG v2.0 (default and fast modes), IQ-TREE 3 (default and fast modes), and VeryFastTree relative to RAxML-NG v1.1, on 221 empirical DNA MSAs, when using 8 threads. The average speedup for each tool is also reported above or below the respective boxplot, depending on the available space. The vertical axis is shown on a logarithmic scale.

Figure 6.6 presents the same plausibility results as Figure 6.5, but here the results are stratified into four distinct MSA difficulty bins. Fast RAxML-NG v2.0 exhibits strong performance on MSAs with difficulty scores of up to 0.75, achieving accuracy rates of up to 98%. However, its accuracy drops to 20% on more challenging MSAs. A similar trend is observed for default IQ-TREE 3, whose accuracy decreases to 40% on these difficult datasets. On the other hand, only five MSAs fall within this highest difficulty range ([0.75, 1.0]). As already mentioned in Section 2.8, datasets with difficulty scores higher than 0.8 are rather rare, and this is due to an inherent—yet minor—drawback of the current definition of the Pythia score (also described in the Supplementary Material of [210]). Regardless, the phylogenetic signal in those MSAs is extremely weak, and even the plausible trees (as indicated by the AU score) cannot be interpreted as being robust solutions, since their (e.g., bootstrap) support from the data is limited.

Finally, Figures 6.7 and 6.8 illustrate the speedup distributions of RAxML-NG v2.0 (default and fast modes), IQ-TREE 3 (default and fast modes), and VeryFastTree relative to RAxML-NG v1.1. Figure 6.7 presents the speedups when all tools are executed using 4 threads, whereas Figure 6.8 shows the corresponding results for 8 threads. In both cases, the default RAxML-NG v2.0 is on average ∼7× faster than RAxML-NG v1.1, and consistently outperforms default IQ-TREE 3, whose speedup is on average 3.6× for 4 threads, and 2.7× for 8 threads. Both the default and the fast RAxML-NG v2.0 modes scale better on more threads than the remaining tools. Specifically, with 4 threads, fast RAxML-NG v2.0 achieves an average speedup of 203×, compared to 123× for fast IQ-TREE 3, and 220× for VeryFastTree, respectively. When using 8 threads, the average speedups are 162× for fast RAxML-NG v2.0, 98× for fast IQ-TREE 3, and 146× for VeryFastTree, respectively. Overall, the runtime of fast RAxML-NG v2.0 is comparable to that of VeryFastTree (and is even lower when using 8 threads), and it consistently outperforms the fast IQ-TREE 3 heuristic. At the same time, its accuracy is directly comparable to that of thorough methods and substantially better than that of other fast alternatives, indicating that fast RAxML-NG v2.0 achieves a more favorable speed vs. accuracy trade-off.

6.5 Discussion

I presented my personal contributions to the second major version of RAxML-NG, which is the main tool that our lab actively develops and maintains. I designed the default RAxML-NG v2.0 heuristic by synthesizing ideas from both the Adaptive and the Early Stopping (ES) versions, and call it the Hegelian heuristic, as it provides a fair balance between speed and accuracy.

RAxML-NG v2.0 initiates the phylogenetic analysis by invoking Pythia, which rapidly predicts the difficulty score of the input MSA. The Pythia score is a reliable quantification of the phylogenetic signal that the input MSA contains. Based on the predicted Pythia score, RAxML-NG v2.0 determines the number of independent tree searches to conduct, as well as the thoroughness within each tree search. The search heuristic that RAxML-NG v2.0 deploys within each tree search was re-designed, and substantially differs from that of RAxML-NG v1.1. In particular, the AUTODETECT SPR rounds of RAxML-NG v1.1 (see Section 2.6.5) were discarded. The new heuristic more closely resembles the sRAxML-NG algorithm, which was originally developed as the underlying heuristic for evaluating the ES criteria. It comprises two sequences of SPR rounds—a

FAST phase followed by a SLOW phase—where the maximum SPR regrafting radius is determined as a function of the Pythia score, and directly reflects the intended search thoroughness. In addition, RAxML-NG v2.0 provides a fast heuristic alternative for the rapid, yet still sufficiently accurate, estimation of large phylogenies. The fast heuristic is the combination of the sRAxML-NG heuristic with the KH-multiple testing stopping criterion (see Chapter 4).

I also presented preliminary benchmark results on 221 empirical DNA MSAs from TreeBASE, conducted by fellow lab member D. Höhler. In this evaluation, both the default and fast RAxML-NG v2.0 modes were compared against thorough methods (RAxML-NG v1.1 and default IQ-TREE 3), as well as faster alternatives (fast IQ-TREE 3 and VeryFastTree). Default RAxML-NG v2.0 matches the accuracy of RAxML-NG v1.1 (long regarded as the gold-standard among ML phylogenetic inference tools in terms of accuracy), while being on average 7× faster. It also consistently outperforms default IQ-TREE 3 in terms of accuracy, while achieving an approximately twofold speedup. The fast mode of RAxML-NG v2.0 attains accuracy comparable to that of default IQ-TREE 3, and is therefore substantially more accurate than the other fast alternatives (fast IQ-TREE 3 and VeryFastTree). At the same time, it surpasses these methods regarding runtimes when using 8 threads. Overall, fast RAxML-NG v2.0 is on the order of ∼200× faster than RAxML-NG v1.1.

6.6 Data and Software Availability

A beta version of RAxML-NG v2.0 has already been released on the official RAxML-NG GitHub repository and is available under GNU GPL at: `https://github.com/amkozlov/raxml-ng/releases/tag/2.0-beta3`. The official version release is planned for the upcoming RAxML-NG v2.0 paper. The DNA datasets used for the preliminary benchmarking correspond to the already published datasets from the Early Stopping study [211], and are available for download from the Dryad Digital Repository: `https://doi.org/10.5061/dryad.8gtht76zz`.

7. Parallel Gentrius

This Chapter is based on the following peer-reviewed conference paper:

- **Anastasis Togkousidis**, Olga Chernomor, and Alexandros Stamatakis. "Parallel Inference of Phylogenetic Stands with Gentrius." *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (pp. 139-148). IEEE.

Contributions: Anastasis Togkousidis designed and implemented the parallelization of the Gentrius algorithm, and also conducted the benchmarking experiments. Olga Chernomor was the main developer of the sequential version of Gentrius. Alexandros Stamatakis supervised the project and helped with the writing of the paper.

In Section 2.11, I introduced the concept of *stands* and explained how they arise in phylogenetic tree space. In short, stands emerge due to missing data in multi-locus MSAs. They are sets of trees that are compatible with the corresponding incomplete per-locus trees. I also linked stands with the concept of *terraces*, that is, sets of trees that have analytically identical optimality scores. Indeed, under many common scoring criteria, the trees from a stand exhibit identical scores and therefore constitute a terrace. For example, when evaluating the trees on a stand using the log-likelihood score under the supermatrix framework with unlinked branch lengths (see Section 2.5.4), all stand trees yield equal log-likelihood scores [166]. Consequently, identifying stands and determining their sizes is of crucial importance for a robust phylogenetic inference.

Chernomor *et al.* developed Gentrius [28], a branch-and-bound algorithm that enumerates all stand trees given a set of unrooted incomplete per-locus trees (see Section 7.1). Despite its efficiency, the pattern and proportion of missing data in multi-locus datasets can still induce extremely long execution times. This motivates the development of a parallel version of Gentrius, to accelerate the enumeration of trees on a stand by utilizing all available CPUs. The Chapter begins with an introduction and brief overview of the Gentrius algorithm. I then describe the parallel version of the Gentrius algorithm, which utilizes a thread-pooling mechanism so as to maintain threads that finish early in busy-wait mode, such that they can contribute to solving long-running tasks. Finally, I present a comprehensive benchmark evaluation of the parallel implementation.

7.1 Introduction

Multi-locus datasets are typically represented as partitioned MSAs (see Section 2.5.4), where per-gene MSAs are assembled/concatenated into a large supermatrix that is typically divided into disjoint partitions. Partitioned MSAs often exhibit patches of missing

data. That is, for certain taxa and loci, the alignment may exclusively contain gap characters. This is because species may have no data present at a specific locus, either due to sampling issues, or because the target locus is simply absent from the genome of the respective species. Consider any such partitioned MSA comprising n taxa and p partitions, and let Y denote the full set of taxa. Given such multi-locus dataset, one may, for example, infer incomplete per-locus ML trees using RAxML-NG. A stand is defined as the set of all complete species trees on Y that are *compatible* with the incomplete per-locus (sub-)trees (see Sections 2.1.2 and 2.11).

Alternatively, one may start from a known (complete) species tree on Y, for instance derived from prior knowledge. Let T be such a binary tree on Y. Data availability in multi-locus datasets is typically denoted via a presence-absence matrix (PAM). Its elements are 1's and 0's, indicating the presence or absence of data for each species and locus. Let $Y_i \subseteq Y$ be the subset of available taxa for locus i, as indicated by the PAM. Hence, $Y = \bigcup_{i=1}^{p} Y_i$, where p is the number of partitions/loci. For each locus i, let T_i be the induced subtree of T on Y_i, that is, $T_i = T|Y_i$ (see Section 2.11). This again yields a collection of incomplete per-locus subtrees, which can be used to enumerate stand trees. In this second scenario, the stand is guaranteed to comprise at least one tree, namely T, which displays the induced per-locus subtrees for every $Y_i \subseteq Y$.

Missing data are rather common in multi-locus empirical datasets. The RAxML Grove v0.7 database [39, 90] contains 7,295 empirical, partitioned multi-gene datasets, 4,959 (68%) of which have a non-zero proportion of missing data, and 1,390 (19%) a missing data proportion exceeding 30%. Identifying stands/terraces in such datasets constitutes an important step, both when searching tree space, and when post-analyzing the tree inference results. The presence of terraces indicates that a single inferred tree only represents one among many equally good solutions. The first systematic attempt for detecting stands was the implementation of the SUPERB algorithm [30] in a python tool called *terraphy* [236]. Thereafter, Biczok *et al.* [15] developed two efficient and independent `C++` implementations of the same algorithm. The main limitation is that the original SUPERB algorithm is defined on rooted trees. In practice, though, almost all "classic" phylogenetic inference tools and methods return unrooted trees. In order to consistently root these unrooted trees, tools implementing the SUPERB algorithm require the input dataset to contain at least one so-called *comprehensive taxon*, that is, a taxon that has data for *all* partitions of the MSA.

The Gentrius algorithm [28], developed by O. Chernomor, constitutes a novel approach to enumerate trees on a stand from a set of unrooted, incomplete trees. Gentrius is a branch-and-bound type algorithm, and improves upon two aspects over the previous methods; it directly operates on unrooted trees and does not rely on the presence of a comprehensive taxon to conduct the enumeration. Gentrius is implemented in IQ-TREE [131] in `C++` and is executed sequentially. However, the computational complexity of building stands for unrooted trees is intractable [21], and the number of stand trees can grow exponentially with the number of taxa [19]. Both aspects, that depend on the input tree and PAM, contribute to excessive running times even when attempting to obtain a lower bound on stand size.

To this end, I developed a parallel shared memory version of Gentrius that deploys a thread-pooling approach. That is, active threads continuously create and push new tasks into a queue, while inactive threads—those that have finished their jobs—remain in waiting mode until a new task becomes available in the queue. Essentially, each thread enumerates a fraction of the overall trees on the stand, and the entire stand size

is determined by summing over those individual calculations. The experimental results (see Section 7.5) yield linear parallel speedups up to 16 cores, on both simulated and empirical data.

7.2 The Gentrius Algorithm

7.2.1 Background

Gentrius is a deterministic, branch-and-bound algorithm that generates all trees on a stand, given a set of incomplete, unrooted subtrees. One of these subtrees serves as the initial tree, which is called the *agile tree*. The algorithm uses the principle of *stepwise taxon insertion*, meaning that taxa which are not present in the initial agile tree are inserted sequentially into it, until it comprises the union of taxa contained in the set of subtrees. The incomplete subtrees are also called *constraint trees*, since they restrict taxon insertion positions because they need to comply with the compatibility condition. There might be multiple edges where the non-present taxon can be inserted into the agile tree. For the enumeration of all stand trees, Gentrius tests all possible edges by successively inserting, removing, and re-inserting (on a distinct branch) taxa into the agile tree (see Figure 7.1). The standard output of the Gentrius algorithm is the number of trees on the stand and their topologies in Newick tree format.

A second Gentrius input option is to provide a complete species tree inferred, for instance, through some phylogenetic method, together with a PAM. Gentrius will then extract the set of *induced subtrees* from this species tree. Each induced subtree is obtained by removing the taxa with zero entries for the corresponding locus in the PAM. This set of induced trees now constitutes the initial set of constraint trees described above, and serves as a starting point for the algorithm.

A new taxon can only be inserted into *admissible branches*. A branch is called admissible, if, after the taxon insertion, the agile tree extended thereby is *pairwise compatible* with every constraint tree. One can effectively identify the set of admissible branches for each taxon by using the so-called *double-edge mapping* approach, which is described in detail in the Supplementary Material of [28]. Briefly, for each agile-constraint tree pair, Gentrius constructs their common subtree and maps the branches of the agile tree and constraint tree onto the branches of the common subtree. These mappings follow precise rules to comply with the compatibility criteria[1]. Both mappings are *surjective*, meaning that for every branch in the common subtree, there is at least one branch in the agile tree (and the constraint tree as well) that is being mapped onto it. Multiple branches, though, from the agile/constraint tree can be mapped onto a single branch in the common subtree. For taxon insertion, Gentrius maps its incident branch from the constraint tree that contains this taxon, onto a branch $\hat{b}$ in the common subtree. Next, the set of admissible branches is determined via the inverse mapping of $\hat{b}$ onto the agile tree. In case a taxon occurs in more than one constraint trees, the intersection of the corresponding individual branch sets is taken. After each taxon insertion or removal, the double-edge mappings are updated.

I define a *state* of the algorithm to be the current agile tree, together with the set of constraint trees, the common subtrees, and the corresponding mappings at a given point

[1] I do not describe the mapping rules here, since they are very technical and not relevant for the parallelization of Gentrius.

Algorithm 1 generateStandTrees (state, list_of_taxa, step)

```
1: taxon ← getNextTaxon (state, list_of_taxa, step)
2: branches ← getAllowedBranches (state, taxon)
3: for branch in branches do
4:     new_state ← extendTaxon (state, taxon, branch)
5:     generateStandTrees (new_state, list_taxa, step + 1)
6:     if no more taxa then
7:         standTrees ← standTrees + 1
8:     end if
9:     state ← removeTaxon (new_state, taxon)
10: end for
```

in time. Moreover, an *intermediate state* is a state where the current agile tree is still incomplete (i.e., some taxa have not been inserted yet).

7.2.2 The Algorithm

The core of the Gentrius algorithm is the recursive `generateStandTrees()` function in Algorithm 1. The functions, by order of appearance, are:

- `generateStandTrees()`: The main recursive function. Its arguments are the algorithm state, the list of taxa to be inserted, and an integer `step` which indicates the recursion depth.
- `getNextTaxon()`: Takes as input the algorithm state, the list of taxa to be inserted, and the integer `step`. It returns the next taxon to be inserted into the agile tree, based on a dynamic taxon insertion heuristic (see below).
- `getAllowedBranches()`: Takes as input the algorithm state and the taxon to be inserted. Returns the set of admissible branches based on the double-edge mappings.
- `extendTaxon()`: Takes as input the algorithm state, the taxon to be inserted and the admissible branch. It inserts the taxon into the agile tree, updates the double-edge mappings and returns the new algorithm state.
- `removeTaxon()`: Takes as input the algorithm state and the taxon to be deleted. It removes the taxon from the agile tree, updates the double-edge mappings, and returns the updated (previous) state.

An example of the Gentrius workflow is provided in Figure 7.1a. Under this simple scenario, Gentrius inserts taxa a and b, which are initially missing from the incomplete agile tree, into all possible combinations of admissible branches. It begins by inserting taxon a, and then proceeds to taxon b. In the example of Figure 7.1a, the algorithm counts four stand trees in total.

The simple example illustrated in Figure 7.1b shows that it is impossible to know the set of admissible branches for all taxa *a priori*. Chernomor *et al.* argue that the choice of the initial agile tree and the taxon insertion order directly affect the efficiency of Gentrius [28]. The efficiency is determined by the number of all generated intermediate states that Gentrius visits, as well as the the number of *dead ends*. A dead end is defined as an intermediate state where at least one of the remaining taxa cannot be inserted into the agile tree without violating the compatibility condition. In this case, the algorithm removes the last inserted taxon from the agile tree and re-inserts it into a distinct admissible branch, if such a branch exists. Thus, an unfavorable initial tree choice and the taxon insertion order can induce visiting numerous irrelevant intermediate

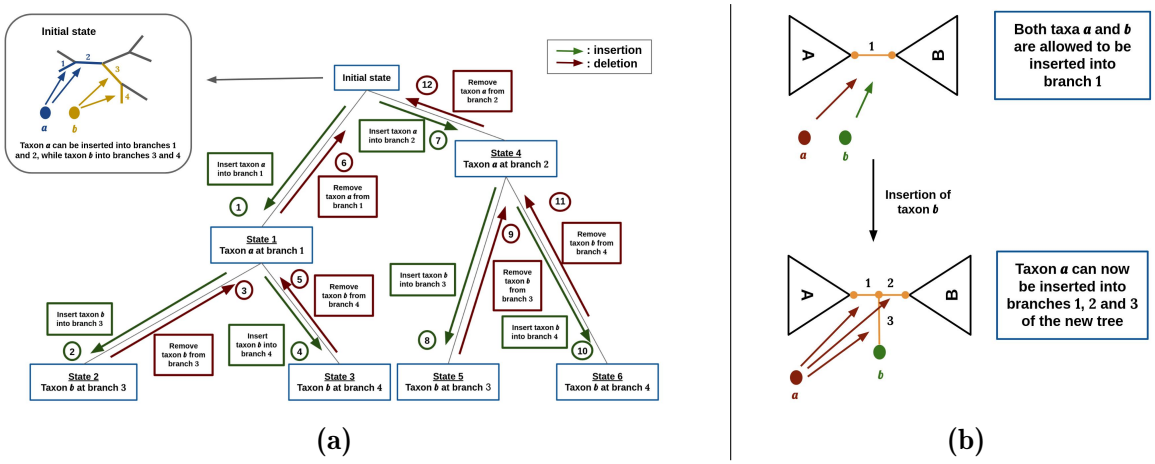


Figure 7.1: Schematic overview of the Gentrius workflow. (a) A simple example where only two taxa, a and b, are missing from the initial tree. The admissible branches for taxon a are $\{1, 2\}$, and for taxon b are $\{3, 4\}$, respectively. There is no overlap between the two admissible branch sets. Gentrius initially inserts taxon a, and then proceeds to taxon b. Arrows 1–12 illustrate the sequence of alternating states, generated by the recursive function of Algorithm 1. **(b) A simple example showing that one cannot know the set of admissible branches for all taxa *a priori*.** Here, taxa a and b can be both inserted into branch 1 of the initial tree. The two sets of admissible branches overlap. Following the insertion of taxon b, taxon a can now be inserted into branches $\{1, 2, 3\}$ of the new tree, without violating the compatibility condition (Figure reproduced from [208]).

states that lead to dead ends. While the algorithm will nonetheless correctly generate all stand trees, the execution time may be substantially longer.

To accelerate the enumeration of stand trees, Gentrius employs two heuristics. The first one chooses a "good" initial tree. Gentrius selects the constraint tree that shares the largest number of taxa with all remaining constraint trees. The second heuristic is called *dynamic taxon insertion*. At each step, Gentrius selects the taxon with the smallest number of admissible branches to be inserted next. To illustrate the validity of both heuristics, I tested them on dataset `emp-data-42370`. The number of stand trees on this dataset is 2,448,225. When both heuristics were applied, Gentrius visited 547,786 intermediate states and 0 dead ends. This translates into 14 seconds of sequential execution time[1]. When deactivating the initial tree selection heuristic and starting from a random constraint tree, Gentrius visits 6,829,128 intermediate states and 0 dead ends in 50 seconds (3.5× slowdown). When deactivating the dynamic taxon insertion heuristic and randomly shuffling the taxon order, Gentrius visits 30,124,986 intermediate states and 1,547,640 dead ends in 174 seconds (12× slowdown).

In the worst-case scenario, the number of trees on a stand can grow exponentially over the number of taxa [19]. Moreover, generating even a single tree from the stand is also computationally intractable [21]. To prevent excessive runtimes, Gentrius employs three stopping rules. The algorithm terminates when:

(1) it counts more than N stand trees;
(2) more than M intermediate states have been visited; or
(3) the execution time exceeds a time limit of T hours.

The default values for these stopping rules are: $N := 10^6$ stand trees, $M := 10^7$ interme-

[1]On an 11th Gen Intel(R) Core(TM) i7-1165G7 processor @ 2.8 GHz.

diate states and $T := 168$ hours. Hence, the algorithm terminates when either, all stand trees have been enumerated, or when one of the stopping conditions is satisfied.

7.3 Parallelization

In this section I explicate the parallelization scheme for Gentrius, which is based on a thread pooling approach. Thereby, I attain load balance among threads and a substantial parallel speedup for Gentrius.

7.3.1 Underlying Idea

Figure 7.1a illustrates the tree-like structure of the Gentrius branch-and-bound workflow. Intuitively, one can divide the entire process into two threads, by assigning the left subtree of the workflow (rooted at State 1) to one thread, and the right subtree (rooted at State 4) to a second thread. The final set of stand trees will then be the union of the disjoint trees counted by each thread independently.

In principle, the parallelization strategy generalizes this idea to N_t threads. All threads begin operating concurrently, and each thread independently parses the entire input set of constraint trees. This redundant input parsing and storage by all threads is necessary, as each thread requires some degree of flexibility to insert and remove taxa *on its own* agile tree, and update the double-edge mappings accordingly. Since Gentrius is a deterministic algorithm and the input is fixed, the first execution stages are identical across all threads. They all select the same initial agile tree and begin by adding taxa in the exact same order to it.

It is possible that, for some datasets—especially those with relatively small amount of missing data—the first few taxa can only be inserted into a single branch. This is because the dynamic taxon insertion heuristic selects the next taxon based on the smallest number of admissible branches. Eventually, however, Gentrius reaches a state in which the next taxon admits insertion into multiple branches. I refer to this state as the *state of initial split*, where the first division of the algorithm into independent subprocesses (concurrent threads) occurs. The parallel algorithm distributes the set of admissible branches for a given taxon as uniformly as possible among threads. For example, if there are five admissible branches and four threads are being used, Gentrius will assign two branches to one thread and one branch to each of the remaining threads, respectively. Thereafter, each thread proceeds by adding the remaining taxa independently. In other words, after this initial parallel workload split, each thread operates on a distinct subset of branches of the branch-and-bound algorithm. In cases where the very first taxon already needs to be inserted into multiple branches, the initial split takes place at the root of the workflow (branch-and-bound) tree. This idea is outlined in Figure 7.2a.

A substantially distinct splitting scheme example is provided in Figure 7.2b, where the state of the initial split has two admissible branches for inserting the next taxon. If parallel Gentrius is executed with three threads, however, the thread number exceeds the cardinality of this set. In this case, Gentrius assigns the right subtree of the workflow to one thread. The two remaining threads both operate on the left subtree and their task separation occurs at a later stage, when they encounter the first taxon with two or more admissible branches.

The *path* that connects two states on the branch-and-bound graph is the sequence of

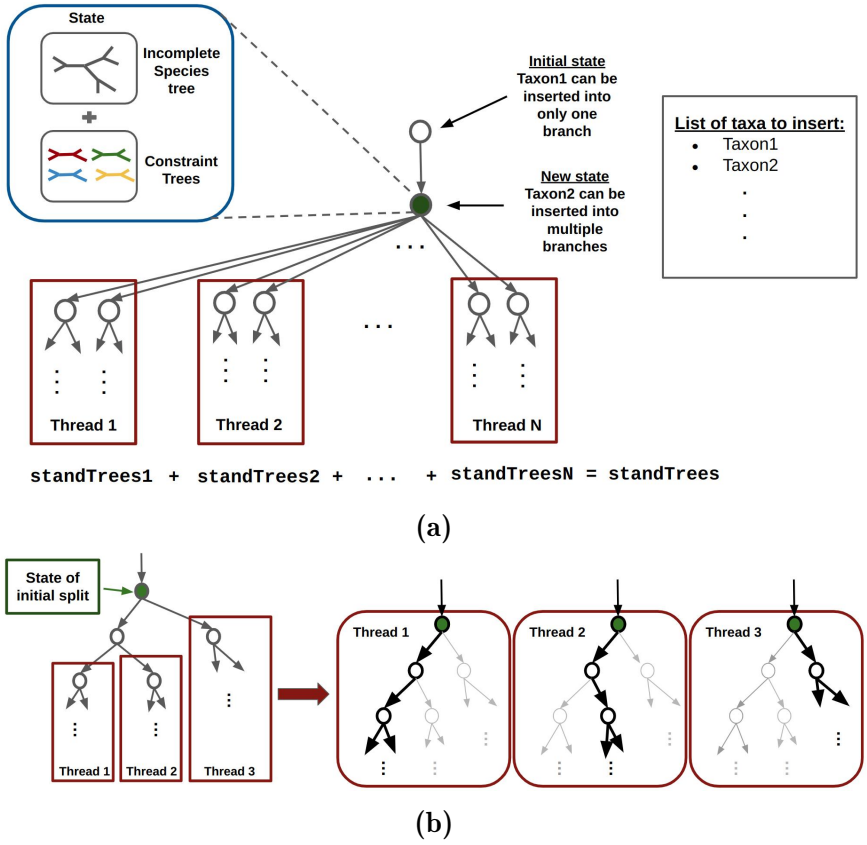


Figure 7.2: Parallelization scheme examples. **(a)** Division of the total process into multiple parallel threads. The green state is the state of the initial split. Parallel execution is initiated when the algorithm encounters the first taxon to be inserted into multiple admissible branches. **(b)** An alternative splitting scheme, where the number of admissible branches in the initial split state (green state) exceeds the number of threads (Figure reproduced from [208]).

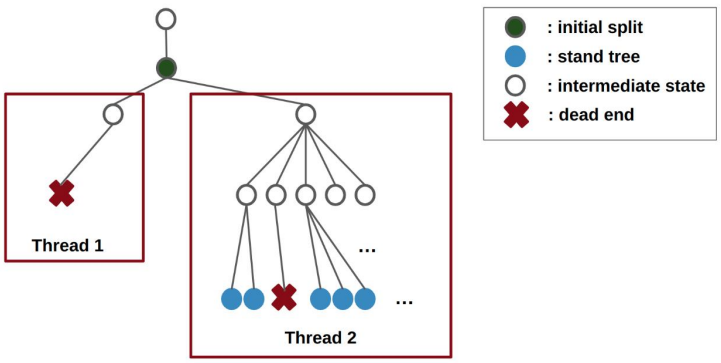


Figure 7.3: An example of an unbalanced work distribution among threads. Thread 1 chooses the path that leads to the left subtree that only consists of a single dead end. The right subtree comprising a substantially larger amount of work is assigned to Thread 2. This imbalance occurs because one cannot predict the structure of the workflow tree *a priori* (Figure reproduced from [208]).

taxon insertions or removals[1] that, when applied to the agile tree of a thread being in one state, it becomes topologically identical to the agile tree of a second thread in a different state. In Figure 7.1a, for example, if two threads are in States 2 and 4 respectively, for the first thread to reach the state of the second, taxa b and a need to first be removed from its agile tree (from branches 3 and 1 respectively) and, then, taxon a must be re-inserted into branch 2. Since double-edge mappings are automatically updated after each taxon insertion/removal, synchronization between the states of the two threads is achieved. This idea is crucial for the second part of the parallelization scheme.

As outlined in Section 7.2.2 through Figure 7.1b, the set of allowed branches for taxon insertion is highly dependent on the insertion position of the previous taxon. The main problem that arises from this lack of knowledge (i.e., simply not knowing the structure of the workflow graph in advance) is, that the initial division of the workload can be unbalanced. For instance, it is likely that parallel Gentrius will assign a part of the workflow tree with a high number of stand trees, intermediate states, and dead ends to one thread, and a separate path with substantially less work to another. Figure 7.3 illustrates such an extreme example.

To alleviate this load imbalance, I devise a thread-pooling parallelization scheme. A thread-pool maintains active threads that have completed their task early, instead of terminating them. This concept is also known as work-stealing. In analogy to the initial split case, the division of the workload among threads requires them to be in the exact same state, and is accomplished by assigning a distinct subset of the next taxon's admissible branches to each thread. The work-stealing mechanism relies on the same idea.

The first step for the threads that finished early to attain the state of the working threads is to remove all taxa from their agile tree up to the state of the initial split (state I_0), which is the last common as well as consistent state. Thereafter, these threads (that finished early) switch into busy-wait mode and become *inactive*. On the other hand, working threads (i.e., *active* threads) constantly create and push tasks onto a shared *task queue*, so as to enable inactive threads to retrieve a task and resume execution. A *task* is a data structure that consists of the following components:

1. a path from state I_0 to a desired intermediate state, including the set of taxa to be added, their exact insertion order, and positions
2. the remaining taxa, the very next taxon to be inserted, and a precomputed subset of admissible branches

More specifically, having passed state I_0, working threads are allowed to create and push tasks into a task queue. Each time a working thread moves between intermediate states by adding or removing taxa, it keeps track of the path that connects I_0 with its current state I_c. For creating a task, it calculates the set of admissible branches for the next taxon and divides it in half. The first half is submitted into the task queue, together with the path that connects states I_0 and I_c. The working thread proceeds by executing the second half. An inactive busy-waiting thread in the pool detects the new task and switches into working mode. It sequentially inserts all taxa into the predefined branches on its own agile tree, to reach state I_c. The next taxon and the corresponding subset of admissible branches are already specified in the dequeued task. The new thread skips line 2 in Algorithm 1 (Section 7.2.2) and directly executes the recursive function. Figure 7.4 provides a summary of the entire parallelization strategy.

[1]The taxon insertion order is determined by the dynamic taxon insertion heuristic, while the removal order is the reverse insertion order.

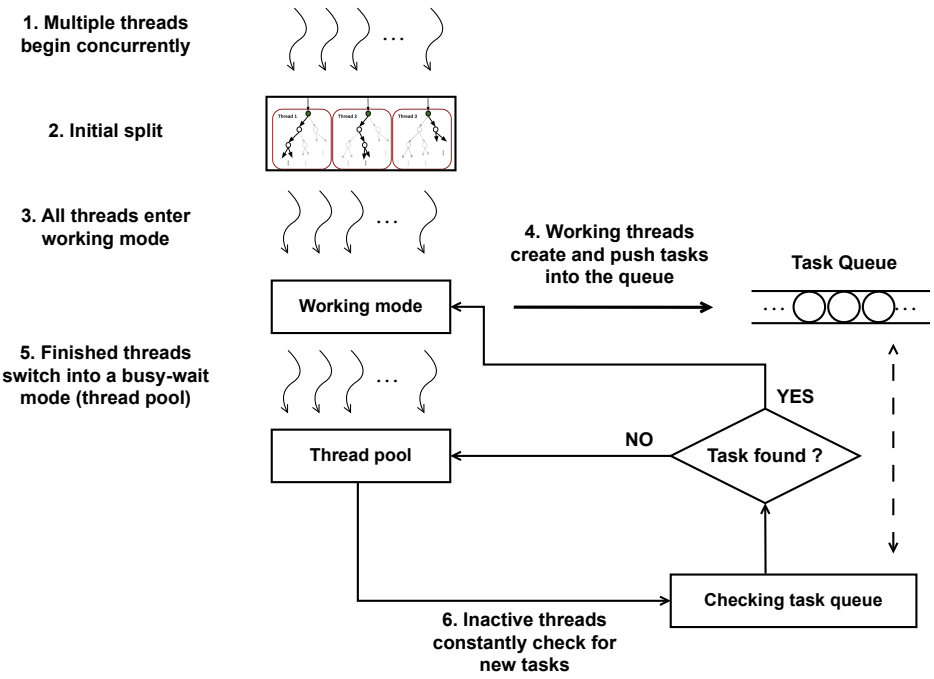


Figure 7.4: Schematic overview of the parallelization scheme. 1–3. All threads begin concurrently. At some point during execution, they reach the initial split state, where each thread descends into a distinct subtree of the branch-and-bound workflow tree. From this point onward, threads enter working mode. **4.** Working threads compute the set of admissible branches for the next taxon to be inserted, and divide this set in half. The working thread executes the first half and submits the second half as a task into the task queue. **5–6.** Finished threads switch into a busy-wait mode, and constantly check the task queue for new tasks. When a task is found, the thread moves to the state I_c specified by the dequeued task and switches back into working mode (Step 3). It then descends into all subtrees defined by the admissible branches of the dequeued task. For the child states of I_c (and recursively further down the workflow tree), the thread again divides the corresponding admissible branches in half and creates new tasks, which are pushed into the queue (Figure adapted from [208]).

The following implementation details need to be considered. The parallelization scheme requires each thread to exclusively work on its own copy of the agile tree. When threads assume independent tasks, they concurrently insert the exact same taxa into different branches of the agile tree, to generate all possible combinations of comprehensive species trees. This would be infeasible on an agile tree that is shared among multiple threads, since concurrent insertion of the same taxon into distinct branches would cause an assertion to fail in the code due to inconsistency. Moreover, regarding the additional time required for one thread to reach the state of another, Gentrius processes hundreds of thousands of states per second[1]. For datasets comprising up to a few thousand taxa, this translates into a few milliseconds.

Finally, I have implemented some restrictions to the generation and submission of tasks to avoid task overload. There is an upper limit to the number of tasks that the task queue can hold concurrently. I define this limit as $N_t + 1$, if $N_t < 8$, and $N_t/2$

[1]on an 11th Gen Intel(R) Core(TM) i7-1165G7 processor @ 2.8 GHz.

otherwise (where N_t is the number of threads). The second restriction is that threads whose current state is deep down in the workflow tree, that is, threads that have less than three remaining taxa to insert, are not allowed to submit a new task, because counting the stand trees beneath their current state is fast, and hence will not benefit from additional parallelization (work stealing). I empirically obtained the values for both thresholds, based on preliminary experiments.

7.3.2 Implementation

The implementation combines the OpenMP API with the `C++` Thread library, for two reasons. First, OpenMP provides convenient parallelization features at an abstract level, that is, easy creation/destruction of threads and fundamental synchronization primitives, such as locks and barriers. The Thread library, on the other hand, provides advanced synchronization primitives, such as conditional variables, which are of crucial importance in the proposed design.

Parallel Gentrius creates/destroys threads using the OpenMP API. Moreover, threads going into busy-wait mode, wait for a condition to be satisfied to assume a new task. To implement this functionality and facilitate inter-thread communication, I used a combination of the `std::condition_variable` and `std::mutex` classes from the `C++` Thread library. This synchronization primitive blocks a thread until a shared variable is modified by another thread (the condition). In this case, the shared variable is the task-queue. Each time a working thread submits a new task to the queue, it thereby notifies all inactive threads, and one of them will subsequently dequeue the task. To guarantee that the queue is accessed or modified by a single thread at a time, I use OpenMP locks.

Here, mixing the OpenMP API with the Thread Library is not problematic, since the creation/destruction of threads is exclusively controlled by OpenMP, while the use of the Thread Library is limited to thread synchronization via conditional variables. Furthermore, parallel Gentrius does not introduce any additional dependencies, because it is implemented as a submodule within IQ-TREE [28, 131], which in turn is written in `C/C++` and requires OpenMP as a prerequisite to compile.

In the original sequential version of Gentrius, the number of stand trees, intermediate states, and dead ends is stored in global variables. Each time the algorithm generates a new state by inserting a new taxon, it updates the respective global variable and checks if one of the stopping rules is satisfied. To maintain this functionality in the parallel version, I deploy shared-memory atomic integer variables via the `std::atomic` template. I use them to protect the counters for stand trees, intermediate states, and dead ends.

Note that a single thread approximately visits hundreds of thousands of states per second (see Section 7.3.1), depending on the number of constraint trees and their corresponding topologies. This translates into hundreds of thousands of shared-memory writes per second, or one write every few micro-seconds. In modern multiprocessing system architectures, the cost of atomic primitives is estimated to require up to a few thousand CPU cycles [92]. This translates into a time requirement to update these counters in the order of hundreds to thousands of nanoseconds. As the number of threads increases, the lock congestion probability when updating these counters likewise increases. To avoid lock congestion, I modified the counter updates. Each thread is allowed to update the global stand trees counter only when it has counted 2^{10} stand trees. For the global intermediate states counter, the restriction is 2^{13} states, while for the global dead ends counter the increment interval is, again, set to 2^{10}. Those settings have been empirically determined.

I observed an average parallel speedup improvement of 2–5% among datasets when 16 threads were used. For example, in dataset `emp-data-3802` the speedup improved by 4%. Each time a thread updates a global variable, it also checks whether the stopping rule for the corresponding variable is satisfied and, if so, notifies all threads to terminate. Evidently, this might lead parallel Gentrius to occasionally exceed the limits specified by the stopping rules, as compared to the sequential version, where the termination is instant. This does not constitute a problem, however, due to the capacity of Gentrius to process hundreds of thousand states per second. In practice, the stopping rules are exceeded by a few thousand stand trees/intermediate states or a few milliseconds of execution time, while the actual specified boundaries are in the order of millions stand trees/intermediate states and hours of execution time.

The required data structures and functionalities for enumerating stand trees are stored in an object of type `Terrace` class, comprising the agile tree, the constraint trees and the double-edged mappings. Functionalities for taxon insertion/removal and mapping updates are also implemented in the `Terrace` class. The trees are implemented based on the respective IQ-TREE template. At startup, each thread initializes its own private `Terrace` object and operates on it consistently. Hence, each thread maintains its own copies of the agile tree, constraint trees, and double-edge mappings within its own `Terrace` object, and performs taxon insertions and removals independently of the other threads. Finally, a task is a structure of arrays containing the information about the taxa that need to be inserted.

7.4 Experimental Setup

The following benchmark evaluates the scalability of the parallel version of the Gentrius algorithm. This is done by computing speedups of the parallel relative to the sequential version, using varying numbers of cores/threads, namely $N_t \in \{1, 2, 4, 8, 12, 16\}$. Although the test platform provides 48 physical cores (see Section 7.4.3), preliminary observations indicated that, given the size of the available datasets (see Section 7.4.2), scalability gains tend to become on average sublinear beyond 16 threads. This behavior is primarily due to synchronization overhead, as well as the limited amount of work that can be done in parallel on the given datasets. Therefore, the systematic evaluation was restricted to at most 16 threads.

As shown in Sections 7.5.1 and 7.5.2, parallel Gentrius generally exhibits near-linear scalability up to 16 threads when compared to the sequential version. For a more complete overview, I additionally report speedup measurements using up to 48 cores for two anecdotal "large" datasets in Section 7.5.4, that is, datasets with sequential execution time exceeding 3 hours.

Certain datasets complicate the performance assessment, since the Gentrius stopping rules may distort the results. In some cases, and depending on the dataset, one encounters slowdowns, speedup plateaus, sub-linear, or even super-linear speedups. I explain why this is the case via simple examples in Section 7.4.1. Further, in conjunction with the scalability results presented in Sections 7.5.1 and 7.5.2, I present a short analysis on how stopping rules (1) and (2) (see Section 7.2.2) affect parallel efficiency.

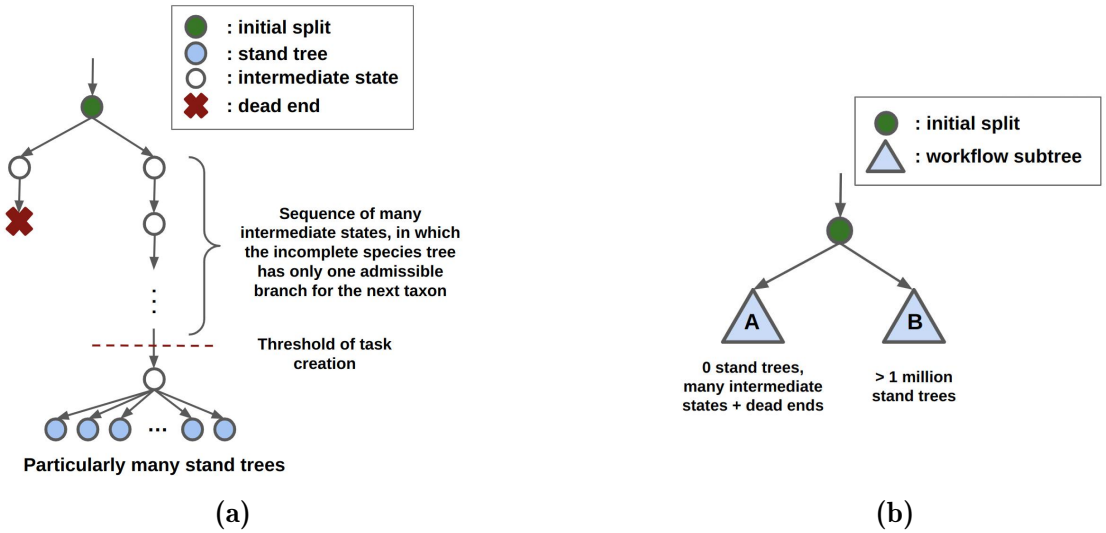


Figure 7.5: Examples of unbalanced workflow structures. **(a)** First example of an unbalanced workflow structure which induces a speedup plateau. **(b)** Second example of an unbalanced workflow structure, which may induce either a super-linear speedup or a speedup plateau (Figure reproduced from [208]).

7.4.1 Variance in Speedups

In general, small datasets yield slowdowns in parallel Gentrius. Here, a dataset is considered to be "small" when the sequential execution time is below one second. Such small datasets typically comprise up to a few thousand stand trees. In these cases, the observed slowdown is unsurprising and is caused by the thread creation and destruction overhead, as well as inter-thread communication and task distribution, which are time consuming in relation to the small sequential runtime.

Apart from these slowdowns, I observed speedup plateaux as well as sub-linear and super-linear speedups in parallel Gentrius. The main cause for this behavior are the stopping rules and the unbalanced structure of the Gentrius branch-and-bound tree. A simple example for such an unbalanced branch-and-bound tree is depicted in Figure 7.5a. In this simple example, one encounters a speedup plateau, as the unbalanced workflow structure forbids the working thread to create and push new tasks to the queue. Note that, for threads to create new tasks, the parallelization scheme requires them to be in a state with two or more admissible branches for the next taxon, while two additional conditions need to hold: (a) the number of tasks in the queue should not exceed a pre-specified upper limit, and (b) the thread creating the task should be in a state with more than two remaining taxa to be inserted into its agile tree (see Section 7.3.1). In Figure 7.5a, if one assumes an execution with two threads, the first thread chooses the left path on the initial split state, faces a dead-end and directly switches into busy-wait mode. On the other hand, the second thread chooses the right path and, after a sequence of intermediate states where each intermediate state only comprises one admissible branch, the second thread reaches the deepest levels of the workflow tree with less than three remaining taxa, where task creation for work-stealing is disallowed. Hence, the second thread is unable to push new tasks into the queue, neither during the sequence of intermediate states, nor after attaining the task creation threshold. Irrespective of the number of threads being used, the parallel runtime will be almost equal to the sequential runtime,

imposing a speedup plateau. I have observed such speedup plateaus of around 3× and 5× in simulated datasets `sim-data-1511`, `sim-data-1792`, and `sim-data-1795`. The sequential runtime for those datasets is below 10 seconds.

A second example of an unbalanced workflow structure is shown in Figure 7.5b. In sequential execution, after the initial split state, the algorithm first descends into workflow subtree A. Since this part of the workflow graph contains zero stand trees, it is possible that the algorithm will exceed the maximum intermediate state limit—stopping rule (2) in Section 7.2.2—and terminate. When executing parallel Gentrius with two threads, the second thread concurrently proceeds to the workflow subtree B, and it may count more than a million stand trees and terminate—due to stopping rule (1) in Section 7.2.2. I observed this branch-and-bound workflow structure in dataset `sim-data-5001` under the default stopping rule parameter settings. Under sequential execution, the algorithm terminates after 113 seconds, having counted 10 million intermediate states but zero stand trees. However, when executed in parallel using two threads, the algorithm counts 1 million stand trees within 5 seconds, resulting in a 22.6× speedup. Notably, even when the threshold is raised to 100 million intermediate states, sequential execution still fails to enumerate any stand tree, indicating that the workload in the one side of the branch-and-bound workflow tree is particularly heavy, and that the sequential version gets trapped there. Under this setting, sequential Gentrius terminates after reaching the 100 million intermediate trees threshold in 1,104 seconds. The corresponding parallel execution with two threads, which terminates in 5 seconds (see above), attains a 220× speedup.

Conversely, the example in Figure 7.5b can also be applied to the scenario in which stopping rule (1) is triggered during sequential execution (see Section 7.2.2), assuming that sequential Gentrius first descends into workflow subtree B and terminates, after counting 1 million stand trees. Under parallel execution, however, the speedup depends on the number of threads executing tasks from the workflow subtree B, where more than 1 million stand trees are detected, compared to the number of threads descending into workflow subtree A. In such cases, the user might encounter speedups or plateaux. Overall, the observed variance in speedups is a result of the concurrent parallel descent of distinct threads into different branches of the branch-and-bound workflow, in conjunction with the effects of stopping rules (1) and (2).

Finally, the third stopping rule sets an execution time threshold. I discuss one dataset that exhibited apparent unexpected behavior while running the main experiments on empirical data, that is `emp-data-5873`. The stopping rule was set to 5h of execution time. The sequential execution terminated (due to the stopping condition) after 5h (18,000 seconds), and having counted 387,985,999 stand trees. In contrast, parallel Gentrius managed to enumerate the entire stand, comprising 485,240,625 trees, without triggering any stopping rule; the execution with two threads required 11,333 seconds. In this context, claiming that the speedup for two threads is $18,000/11,333 = 1.58\times$ underestimates the actual speedup. Because the direct comparison of execution times in such cases is inaccurate, I introduce the *adapted speedup*, as a more appropriate metric. I thus define the adapted speedup for N_t threads (ASP_{N_t}) as:

$$ASP_{N_t} = \frac{ST_{N_t}/T_{N_t}}{ST_1/T_1} = \frac{ST_{N_t}}{ST_1} \cdot SP_{N_t} \tag{7.1}$$

where ST_k, T_k denote the number of enumerated stand trees and the execution time when k threads are being used, and SP_{N_t} is the standard speedup metric for N_t threads, calculated by simply dividing the corresponding execution times. Table 7.1 shows these

Dataset	Number of Threads				
	2	4	8	12	16
emp-data-5873	1.9	3.8	6.9	8.8	12.2
emp-data-43820	2.4	3.9	8.7	9.7	11.2
emp-data-55114	2.3	4.5	8.1	9.6	9.1
sim-data-4666	1.9	3.8	7.0	8.1	11.1
sim-data-4843	1.6	3.0	6.9	8.8	10.1

Table 7.1: Adapted speedups of five datasets that reached the time limit in sequential execution. The adapted speedups are calculated based on Equation (7.1) (Table reproduced from [208]).

adapted speedups for five datasets that reached the time limit threshold—stopping rule (3)—in sequential execution.

7.4.2 Datasets and Filtering

To evaluate the performance of parallel Gentrius, I ran experiments on both simulated (Section 7.5.1) and empirical (Section 7.5.2) datasets, on up to 16 threads (16 physical cores). For simulated data, I used all instances from the original Gentrius manuscript [28]. The set comprises 4,997 instances with varying sizes, different patterns as well as proportions of missing data; more specifically, taxon numbers ranged between 50 and 300, locus numbers between 5 and 30, and the proportion of missing data ranged between 30% and 50%.

To avoid the difficulties described in Section 7.4.1 and establish a fair base comparison, I filtered the datasets to extract instances that do not trigger any stopping rules. This was achieved by initially running parallel Gentrius on all datasets with 16 threads and keeping only those datasets for which Gentrius successfully calculated the entire stand. For these datasets, I then re-ran the analysis using $N_t = \{12, 8, 4, 2, 1\}$ threads, making sure the time limit was not exceeded as the number of threads decreased. For the aforementioned reasons (see Section 7.4.1), small datasets were also excluded, resulting in 443 final datasets with sequential execution times ranging from 1 second up to 4 hours.

Finally, I used the RAxML Grove database v0.7 [90] as empirical dataset source. I randomly sampled 3,089 datasets for my experiments (out of 4,959 empirical, partitioned datasets with missing data in total). Taxon numbers ranged between 7 and 37,849, locus numbers between 2 and 5,219, and the proportion of missing data ranged between 10% and 90%. I applied the exact same pipeline as for simulated data to filter datasets and post-process results. This yielded 162 final datasets with sequential execution times ranging between 1 second up to 3 hours.

7.4.3 Commands and Computing Environment Details

Parallel Gentrius is implemented in the `terragen` branch of the IQ-TREE 2 GitHub repository[1]. All empirical and simulated datasets used in the experiments comprise a complete species tree along with a PAM file, which indicates the presence/absence of data for the respective taxon at a given locus. For the experiments, I set the following stopping

[1]https://github.com/iqtree/iqtree2/tree/terragen/terrace

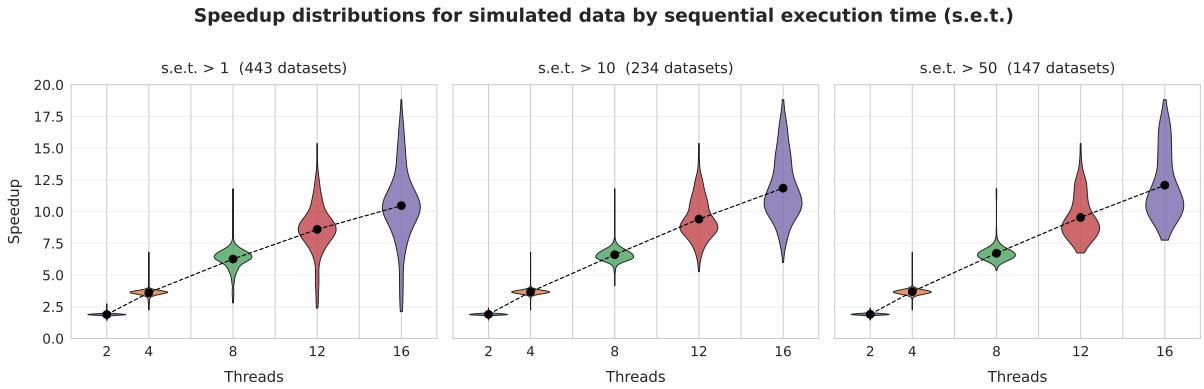


Figure 7.6: Per-thread speedup distributions on simulated data. The dashed line connects the mean speedup values for each distribution, corresponding to different numbers of threads being used. Speedup rates were calculated for: **(left)** 443 datasets with the sequential execution time (s.e.t.) exceeding 1 second, **(middle)** 234 datasets with s.e.t. exceeding 10 seconds, and **(right)** 147 datasets with s.e.t. exceeding 50 seconds (Figure adapted from [208]).

rule parameters: $N := 10^9$ stand trees (`-g_stop_t 1000000000` option), $M := 10^9$ intermediate states (`-g_stop_i 1000000000` option), and $T := 5$ hours (`-g_stop_h 5` option) for all executions (both sequential and multi-threaded). The reason why the values for rules (1) and (2) are equal is because, although it seems plausible to expect more intermediate states, in the vast majority of instances parallel Gentrius ends up counting more stand trees, due to the dynamic taxon insertion heuristic. As test platform I used an Intel(R) Xeon(R) Platinum 8260 48 double-core processor running at 2.4 GHz.

An example command to execute parallel Gentrius using four threads, for a dataset with the species tree stored in `tree.newick`, the PAM in `iqt.pam`, and using the above stopping-rule configuration, is:

```
./iqtree2 -nt 4 --gentrius tree.newick -pr_ab_matrix iqt.pam
-g_stop_t 1000000000 -g_stop_i 1000000000 -g_stop_h 5
```

7.5 Results

7.5.1 Results on Simulated Data

Per-thread speedup distributions on simulated data are shown in Figure 7.6. To assess the effect of small datasets, I progressively increased the threshold for dataset inclusion from 1 second to 10 and 50 seconds of sequential execution time (s.e.t.), respectively. Overall, parallel Gentrius exhibits near-linear average speedups with respect to the number of threads/physical cores being used.

Specifically, for the 443 datasets with the s.e.t. exceeding 1 second, the average speedup is 1.9× for 2 threads, 3.6× for 4 threads, 6.3× for 8 threads, 8.6× for 12 threads, and 10.6× for 16 threads, respectively. The per-thread speedups are analogous for the 234 datasets with the s.e.t. exceeding 10 seconds, and for the 147 datasets with the s.e.t. exceeding 50 seconds. In the former case, the average speedup attained with 16 threads is 11.9×, while in the latter case it reaches 12.1×.

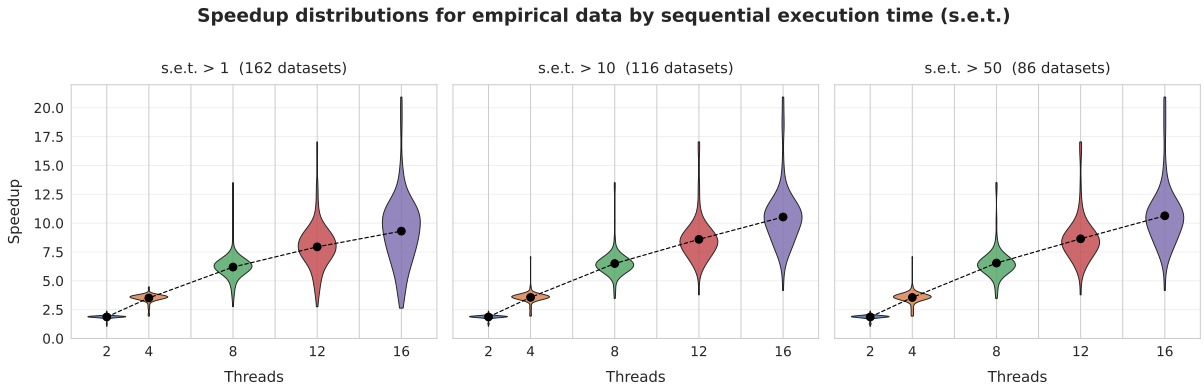


Figure 7.7: Per-thread speedup distributions on empirical data. The dashed line connects the mean speedup values for each distribution, corresponding to different numbers of threads being used. Speedup rates were calculated for: **(left)** 162 datasets with the sequential execution time (s.e.t.) exceeding 1 second, **(middle)** 116 datasets with s.e.t. exceeding 10 seconds, and **(right)** 86 datasets with s.e.t. exceeding 50 seconds (Figure adapted from [208]).

7.5.2 Results on Empirical Data

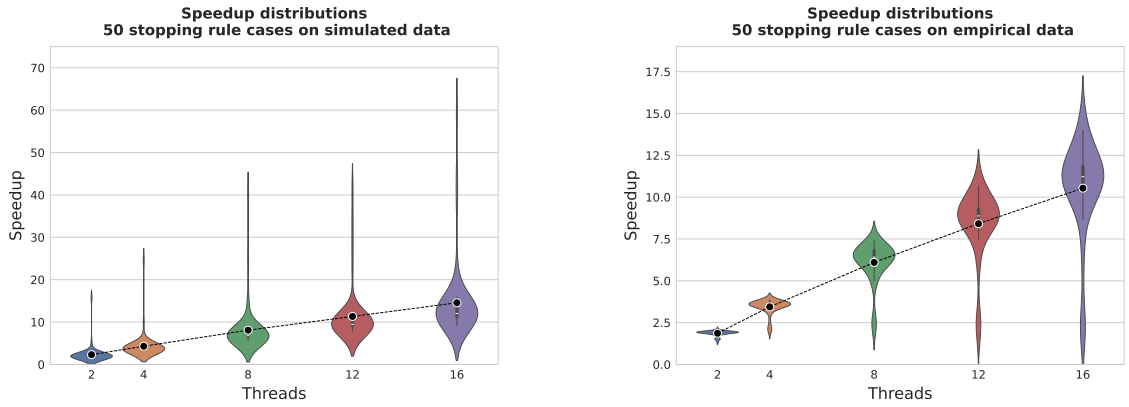
Per-thread speedup distributions on empirical data are shown in Figure 7.7. In analogy to simulated data, here I also progressively increased the threshold for dataset inclusion from 1 second to 10 and 50 seconds of s.e.t., respectively. Parallel Gentrius exhibits near-linear average speedups with respect to the number of threads/cores being used, especially on datasets with the s.e.t. exceeding 50 seconds.

Specifically, for the 162 datasets with s.e.t. exceeding 1 second, the average speedup is 1.9× for 2 threads, 3.5× for 4 threads, 6.2× for 8 threads, 8.0× for 12 threads, and 9.3× for 16 threads, respectively. The per-thread speedups are analogous for the 116 datasets with the s.e.t. exceeding 10 seconds, and for the 86 datasets with the s.e.t. exceeding 50 seconds, attaining an average speedup of 10.6× when using 16 threads in both cases.

7.5.3 Impact of Stopping Rules (1) and (2) on Speedups

Next, I analyze the impact of stopping rules (1) and (2) on parallel efficiency in more detail. To this end, I conducted a short analysis of datasets that encounter those stopping rules. I characterize these analyses as "short", since the threshold values were reduced to 10 million stand trees and intermediate states respectively, so as to minimize the runtime requirements of these experiments. I collected 50 datasets of each type (empirical/simulated) and measured the speedups by dividing the sequential by the parallel execution times, although, as I argued in Section 7.4.1, this comparison might sometimes be misleading.

Figure 7.8 summarizes the results of this analysis. The distribution of speedups on both simulated and empirical data is substantially distorted, particularly so in the case of simulated data where a super-linear speedup is observed, for only a few datasets. An example of such a dataset is `sr_sim-data-44`. The tree-like workflow structure of this specific dataset turns out to be highly unbalanced, yielding a misleading impression that it must comprise a plethora of dead ends and no stand trees (as in the example of Figure 7.5b), when being analyzed with $N_t = \{1, 2\}$ threads. In contrast, when more threads



(a) **(b)**

Figure 7.8: Per-thread speedup distribution on datasets that trigger stopping rules (1) or (2). Analysis on: **(a)** 50 simulated datasets, and **(b)** 50 empirical datasets (Figure adapted from [208]).

Dataset	Number of Threads		
	16	32	48
emp-data-60587	12.0	20.37	26.18
sim-data-4677	13.43	23.01	29.46

Table 7.2: Speedups for two large datasets, when more than 16 threads are used. The sequential execution times for the two datasets are 11,199.6 (`emp-data-60587`) and 17,163.4 (`sim-data-4677`) seconds, respectively (Table reproduced from [208]).

are used, $N_t = \{4, 8, 12, 16\}$, the algorithm terminates having counted 10 million stand trees, yielding 5×, 25×, 41× and 59× speedups for 4,8,12, and 16 threads, respectively. This striking variance among results is an outcome of the unbalanced branch-and-bound workflow structure combined with how threads chose their paths along the workflow. The more threads are being used, the more efficient and faster the exploration of stand trees becomes for this specific dataset.

7.5.4 Scalability on more than 16 threads

To anecdotally test scalability when more than 16 threads are used, I selected two datasets, one empirical and one simulated, with comparatively long sequential execution times. The two datasets are `emp-data-60587` and `sim-data-4677`, with sequential execution times of 11,199.6 and 17,163.4 seconds, respectively. The calculated speedups for executions using $N_t = \{16, 32, 48\}$ threads are shown in Table 7.2.

7.6 Discussion

I described, implemented, evaluated, and made available an efficient parallel open-source version of Gentrius. Fast threads that finish their own tasks early, can subsequently contribute to accelerating slower threads in completing their tasks via a thread pooling mechanism that implements work stealing. The parallel version of Gentrius now allows

for seamless computation of stands on large datasets using off-the-shelf multi-core desktop and server systems.

Apart from the fact that parallel Gentrius provides a faster solution for counting trees on a stand, in some cases, the multi-threaded execution identifies trees on a stand that the sequential execution fails to detect in reasonable time. This is due to the parallel descent into the distinct branches of the branch-and-bound algorithm in conjunction with the implemented stopping rules. Thus, multi-threaded execution allows for a potentially faster and more efficient exploration of the workflow graph.

In summary, identifying stands constitutes a crucial step in species tree inference from multiple loci and allows to investigate the inherent uncertainty due to missing data. Hence, efficient bioinformatic tools such as parallelized Gentrius implementation can be very helpful for pipelines and methods assessing the robustness of the phylogenetic analysis to be conducted, alongside with the difficulty score prediction of the given MSA via the Pythia tool [73].

7.7 Data and Software Availability

Parallel Gentrius is implemented in the `terragen` branch of the IQ-TREE 2 GitHub repository: `https://github.com/iqtree/iqtree2/tree/terragen/terrace`. All data used for benchmarking are available for download at `https://cme.h-its.org/exelixis/material/gentrius-parallel.tar.gz`.

8. Conclusion and Future Work

8.1 Conclusion

The core of this thesis focused on developing, implementing, and benchmarking heuristics for maximum likelihood (ML)-based phylogenetic tree inference. I integrated my methods into RAxML-NG, the main tool which our lab actively develops and maintains.

I began by presenting the Adaptive version of RAxML-NG (Chapter 3), a tool which modifies the thoroughness of the tree search as a function of the MSA difficulty score, predicted by the Pythia tool. The Pythia score provides a reliable quantification of the phylogenetic signal of the input MSA. Accordingly, on easy MSAs, Adaptive RAxML-NG deploys a faster heuristic, because the tree space typically only exhibits a single log-likelihood peak. On such a likelihood landscape, even more superficial search strategies rapidly converge to this unique optimum. In contrast, on difficult-to-analyze MSAs, the likelihood-score tree space exhibits a rugged surface with multiple, yet statistically equivalent, local optima. Analyzing such data is hopeless, and therefore Adaptive RAxML-NG quickly infers only a few, out of the many, locally optimal ML trees. In contrast, it deploys a more thorough search strategy on MSAs with intermediate difficulty. On such instances, although the tree space still exhibits many distinct peaks, a few among those are statistically better. Therefore, more thorough heuristics have been shown to perform better than superficial ones on those datasets. The experimental results showed that the idea of adapting the thoroughness of the search strategy based on the MSA difficulty score is effective: Adaptive RAxML-NG inferred statistically equivalent trees to RAxML-NG v1.1 in $\sim$ 99.7% of the cases, while attaining speedups of up to 16$\times$ on easy and difficult MSAs.

Next, I presented Early Stopping (ES) criteria for ML tree inference (Chapter 4). They offer a practical way to evade over-optimization in ML tools, while at the same time preserving statistical plausibility of the output trees. The underlying idea was that input MSAs are noisy, and pushing the optimization to its limit increases the risk of ML tools to overfit. But even if overfitting does not occur, previous empirical observations have shown that RAxML-NG often remains on long log-likelihood plateaus for an extended amount of time before termination, due to its exhaustive search strategy. This indicated that this excessive search effort may be omitted. I integrated the Kishino-Hasegawa (KH) test into RAxML-NG, as a reliable statistical test to determine convergence and terminate RAxML-NG. The KH test was applied in combination with a simplified tree search heuristic (sRAxML-NG), which proved to be an important by-product of this study. In short, the KH test is used to assess whether the log-likelihood difference between the trees prior to and after an SPR round (in sRAxML-NG) is significant; if the two trees are statistically equivalent, the search is terminated. I further refined the approach by incorporating a multiple-testing correction into the KH test. The sRAxML-NG heuristic,

in conjunction with the KH-based stopping criteria, inferred plausible ML trees in up to ∼98% of the cases compared to standard RAxML-NG v1.2, while yielding an average per-search speedup of 5×. Importantly, sRAxML-NG in combination with the KH-multiple testing stopping criterion achieves a favorable balance between speed and accuracy, and was therefore integrated into the second major version of RAxML-NG as a fast, yet reliable alternative.

In Chapter 5, I investigated whether widely used ML tree inference tools are prone to overfitting. I put particular emphasis on the tree topology parameter, which arguably constitutes the biologically most important parameter. I evaluated overfitting via two complementary approaches. In the first, I statistically evaluated overfitting trends in ML tools via a 10-fold Monte-Carlo cross-validation framework. The main question was whether the statistically expected behavior of ML tools, and particularly the thorough ones, is to overfit and, if so, to which extent. The second approach consisted of benchmarking a Holdout-Validation (HV) version of RAxML-NG, against the standard version. In analogy to overfitting mitigation strategies in deep learning, HV randomly splits the input MSA into model-training and validation sites; tree inference is then only conducted on the training data, while the validation data are used to detect overfitting, with the algorithm returning the tree that maximizes the validation likelihood. Both experiments indicated that topological overfitting does not constitute a major concern for ML phylogenetic inference tools.

Combining my experience on developing ML heuristics and the structure of the likelihood tree space, I designed the default Hegelian heuristic in RAxML-NG v2.0, which I present in Chapter 6. RAxML-NG v2.0 is on average 7× faster than RAxML-NG v1.1, while it infers statistically equivalent, or even better ML trees. It also outperforms IQ-TREE 3 with respect to accuracy and runtime. At the same time, the fast mode of RAxML-NG v2.0 (i.e., the sRAxML-NG heuristic combined with KH-multiple testing stopping criterion) achieves comparable accuracy to that of IQ-TREE, while it outruns (w.r.t. execution time) *all* other fast alternatives (VeryFastTree and fast IQ-TREE). Via this fast mode, RAxML-NG users can rapidly infer highly accurate phylogenies on large datasets (e.g., viral MSAs), attaining an average speedup of 200× compared to RAxML-NG v1.1.

Finally, in Chapter 7, I presented the parallelization scheme that I implemented for the Gentrius algorithm, a branch-and-bound method for enumerating all trees on a stand. Parallel Gentrius employs a thread-pooling mechanism that keeps early-finishing threads in a busy-wait state, allowing them to steal workload from active threads and thereby achieve "good" load balancing. Through this approach, parallel Gentrius attains linear scalability on up to 16 cores.

Overall, this thesis can be characterized as a diligent introspection into the likelihood tree space and the algorithmic strategies required to explore it efficiently. Collectively, the methods developed here contribute to making ML phylogenetic inference both faster and more reliable, thereby supporting scalable and robust evolutionary analyses.

8.2 Future Work

With respect to the RAxML-NG v2.0 heuristic, the objectives of this thesis have largely been achieved. RAxML-NG will continue to evolve, yet concrete directions for further heuristic adjustments remain unclear at present.

Regarding the relationship between the structure of the likelihood tree space and the MSA difficulty score, a recurring idea is the so-called *Plausible Tree Set (PTS)* problem. Rather than inferring a single ML tree, ML tools may instead aim to reconstruct a set of statistically equivalent trees. The PTS size (i.e., number of trees it contains) is expected to vary with the Pythia difficulty score: on easy MSAs, independent tree searches are more likely to converge to the same topology, whereas on difficult MSAs, they yield topologically contradicting, yet still plausible, trees. Developing methods for the rapid inference of a representative PTS will provide a practical ML-based alternative to Bayesian approaches, and enable the reconstruction of summary trees that better reflect the inherent topological uncertainty induced by difficult datasets. Fellow lab member Johannes Hengstler is working towards this goal.

Another possible extension is the application of reinforcement learning techniques for phylogenetic tree inference. Specifically, deep neural networks (DNNs) trained via Q-learning could be used to guide the tree search process, for example by learning which topological moves are more likely to yield substantial likelihood improvements. Early studies in the literature [7, 8] already provide proof-of-concept results. Beyond move selection at the SPR level, DNNs could also operate at the workflow level, learning to select the next optimization routine (i.e., BLO, MPO, or an SPR round) along with the appropriate parameter configurations. Such a method will reduce the reliance on static and manually tuned heuristics, and allow for a more dynamic exploration of the tree space. Fellow lab member Dimitri Höhler is currently investigating this direction.

Finally, I would be interested to evaluate the HV version on genome-wide alignments, such as those used to benchmark the first major version of RAxML-NG (see Supplementary Material in [111]). These datasets, which comprise millions of sites, could provide deeper insight into whether overfitting is negligible on extreme input data dimensions. Additionally, it would be worthwhile to examine whether the HV approach yields runtime benefits on such long datasets, given that inference is conducted on only 80% of the sites. However, this is rather unlikely because each evaluation on the validation MSA requires a full BLO and MPO from scratch, as well as a CLV recomputation on the validation MSA. This substantially increases runtime and explains why the HV versions were slower than the ES versions. Nevertheless, testing HV on large phylogenomic alignments would clarify whether the method offers advantages, when extreme amounts of data are available.

Bibliography

[1] F. Abascal, D. Posada, and R. Zardoya. MtArt: a new model of amino acid replacement for Arthropoda. *Molecular Biology and Evolution*, 24(1):1–5, 2007.

[2] J. Adachi and M. Hasegawa. Model of amino acid substitution in proteins encoded by mitochondrial DNA. *Journal of Molecular Evolution*, 42:459–468, 1996.

[3] J. Adachi, P. J. Waddell, W. Martin, and M. Hasegawa. Plastid genome phylogeny and a model of amino acid substitution for proteins encoded by chloroplast DNA. *Journal of Molecular Evolution*, 50:348–358, 2000.

[4] H. Akaike. A new look at the statistical model identification. *IEEE Transactions on Automatic Control*, 19(6):716–723, 1974.

[5] M. Anisimova and O. Gascuel. Approximate likelihood-ratio test for branches: a fast, accurate, and powerful alternative. *Systematic Biology*, 55(4):539–552, 2006.

[6] E. Avni, R. Cohen, and S. Snir. Weighted quartets phylogenetics. *Systematic Biology*, 64(2):233–242, 2015.

[7] D. Azouri, S. Abadi, Y. Mansour, I. Mayrose, and T. Pupko. Harnessing machine learning to guide phylogenetic-tree search algorithms. *Nature Communications*, 12(1):1983, 2021.

[8] D. Azouri, O. Granit, M. Alburquerque, Y. Mansour, T. Pupko, and I. Mayrose. The tree reconstruction game: phylogenetic reconstruction using reinforcement learning. *Molecular Biology and Evolution*, 41(6):msae105, June 2024.

[9] H.-J. Bandelt and A. W. Dress. A canonical decomposition theory for metrics on a finite set. *Advances in Mathematics*, 92(1):47–105, 1992.

[10] H. Baños, E. Susko, and A. J. Roger. Is over-parameterization a problem for profile mixture models? *Systematic Biology*, 73(1):53–75, 2024.

[11] G. Batzolis. *Towards learning the geometry of data: from diffusion models to Riemannian geometry*. PhD thesis. Apollo - University of Cambridge Repository, 2025. DOI: 10.17863/CAM.120391. URL: https://www.repository.cam.ac.uk/handle/1810/387713.

[12] M. Belkin, D. Hsu, S. Ma, and S. Mandal. Reconciling modern machine-learning practice and the classical bias–variance trade-off. *Proceedings of the National Academy of Sciences*, 116(32):15849–15854, 2019.

[13] D. R. Bentley, S. Balasubramanian, H. P. Swerdlow, G. P. Smith, J. Milton, C. G. Brown, K. P. Hall, D. J. Evers, C. L. Barnes, H. R. Bignell, et al. Accurate whole human genome sequencing using reversible terminator chemistry. *Nature*, 456(7218):53–59, 2008.

[14] V. Berry, O. Gascuel, and G. Caraux. Choosing the tree which actually best explains the data: another look at the bootstrap in phylogenetic reconstruction. *Computational Statistics & Data Analysis*, 32(3):273–283, 2000.

[15] R. Biczok, P. Bozsoky, P. Eisenmann, J. Ernst, T. Ribizel, F. Scholz, A. Trefzer, F. Weber, M. Hamann, and A. Stamatakis. Two C++ libraries for counting trees on a phylogenetic terrace. *Bioinformatics*, 34(19):3399–3401, 2018.

[16] L. J. Billera, S. P. Holmes, and K. Vogtmann. Geometry of the space of phylogenetic trees. *Advances in Applied Mathematics*, 27(4):733–767, 2001.

[17] C. M. Bishop and N. M. Nasrabadi. *Pattern recognition and machine learning*. Vol. 4. Springer, 2006.

[18] S. P. Blomberg, T. Garland Jr, and A. R. Ives. Testing for phylogenetic signal in comparative data: behavioral traits are more labile. *Evolution*, 57(4):717–745, 2003.

[19] S. Böcker. Exponentially many supertrees. *Applied mathematics letters*, 15(7):861–865, 2002.

[20] J. P. Bollback. Bayesian model adequacy and choice in phylogenetics. *Molecular Biology and Evolution*, 19(7):1171–1180, July 2002.

[21] M. Bordewich, C. Semple, and J. Talbot. Counting consistent phylogenetic trees is #P-complete. *Advances in Applied Mathematics*, 33(2):416–430, 2004.

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

[23] P. Breitling, A. Stamatakis, O. Chernomor, B. Bettisworth, and L. Reszczynski. Empirical analysis of phylogenetic quasi-terraces. *bioRxiv*,810309, 2019.

[24] R. P. Brent. *Algorithms for minimization without derivatives*. Courier Corporation, 2013.

[25] G. S. Brodal, R. Fagerberg, and C. N. Pedersen. Computing the quartet distance between evolutionary trees in time O (n log n). *Algorithmica*, 38(2):377–395, 2004.

[26] D. Bryant, J. Tsang, P. E. Kearney, and M. Li. Computing the quartet distance between evolutionary trees. *Symposium on Discrete Algorithms: Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*. Vol. 9. 11. 2000, pp. 285–286.

[27] P. Buneman. The recovery of trees from measures of dissimilarity. *Mathematics in the Archaeological and Historical Sciences*, 1971.

[28] O. Chernomor, C. Elgert, and A. von Haeseler. Gentrius: generating trees compatible with a set of unrooted subtrees and its application to phylogenetic terraces. *Molecular Biology and Evolution*, 41(11):msae219, Oct. 2024.

[29] J. Clarke, H.-C. Wu, L. Jayasinghe, A. Patel, S. Reid, and H. Bayley. Continuous base identification for single-molecule nanopore DNA sequencing. *Nature Nanotechnology*, 4(4):265–270, 2009.

[30] M. Constantinescu and D. Sankoff. An efficient algorithm for supertrees. *Journal of Classification*, 12(1):101–112, 1995.

[31] A. Cornish-Bowden. Nomenclature for incompletely specified bases in nucleic acid sequences: rcommendations 1984. *Nucleic Acids Research*, 13(9):3021–3030, May 1985.

[32] F. Crick, L. Barnett, S. Brenner, and R. J. Watts-Tobin. General Nature of the Genetic Code for Proteins. *Nature*, 192(4809):1227–1232, 1961.

[33] C. C. Dang, Q. S. Le, O. Gascuel, and V. S. Le. FLU, an amino acid substitution model for influenza proteins. *BMC Evolutionary Biology*, 10(1):1–11, 2010.

[34] D. Darriba, D. Posada, A. M. Kozlov, A. Stamatakis, B. Morel, and T. Flouri. ModelTest-NG: a new and scalable tool for the selection of DNA and protein evolutionary models. *Molecular Biology and Evolution*, 37(1):291–294, 2020.

[35] C. Darwin. *On the Origin of Species by means of natural selection: or the preservation of favored races in the struggle for life*. London: John Murray, 1859.

[36] W. H. Day, D. S. Johnson, and D. Sankoff. The computational complexity of inferring rooted phylogenies by parsimony. *Mathematical Biosciences*, 81(1):33–42, 1986.

[37] M. O. Dayhoff. A model of evolutionary change in proteins. *Atlas of Protein Sequence and Structure*, 5:89–99, 1972.

[38] R. Desper and O. Gascuel. Fast and accurate phylogeny reconstruction algorithms based on the minimum-evolution principle. *Journal of Computational Biology*, 9(5):687–705, 2002.

[39] Dimitri Höhler. RAxMLGrove v0.7. Available at: `https://github.com/angtft/RAxMLGrove/releases/tag/v0.7`. Accessed: 2026-01-14.

[40] M. W. Dimmic, J. S. Rest, D. P. Mindell, and R. A. Goldstein. rtREV: an amino acid substitution matrix for inference of retrovirus and reverse transcriptase phylogeny. *Journal of Molecular Evolution*, 55(1):65, 2002.

[41] T. Dobzhansky. Nothing in biology makes sense except in the light of evolution. *The American Biology Teacher*, 75(2):87–91, 2013.

[42] A. W. Dress, C. Flamm, G. Fritzsch, S. Grünewald, M. Kruspe, S. J. Prohaska, and P. F. Stadler. Noisy: identification of problematic columns in multiple sequence alignments. *Algorithms for Molecular Biology*, 3:1–10, 2008.

[43] R. C. Edgar. MUSCLE: multiple sequence alignment with high accuracy and high throughput. *Nucleic Acids Research*, 32(5):1792–1797, 2004.

[44] P. Edman. Method for determination of the amino acid sequence in peptides. *Acta Chemica Scandinavica*, 4:283–293, 1950.

[45] J. Eid, A. Fehr, J. Gray, K. Luong, J. Lyle, G. Otto, P. Peluso, D. Rank, P. Baybayan, B. Bettman, et al. Real-time DNA sequencing from single polymerase molecules. *Science*, 323(5910):133–138, 2009.

[46] G. F. Estabrook, F. McMorris, and C. A. Meacham. Comparison of undirected phylogenetic trees based on subtrees of four evolutionary units. *Systematic Zoology*, 34(2):193–200, 1985.

[47] Exelixis-Lab. coraxlib (COre RAXml LIBrary). Available at: `https://codeberg.org/Exelixis-Lab/coraxlib`. Accessed: 2025-10-20.

[48] I. T. Farah, M. M. Islam, K. T. Zinat, A. H. Rahman, and M. S. Bayzid. Phylogenomic terraces: presence and implication in species tree estimation from gene trees. *bioRxiv*, 2020.

[49] J. Felsenstein. PHYLIP (phylogeny inference package) version 3.6. Distributed by the author. Seattle (WA): Department of Genome Sciences, University of Washington. 2005.

[50] J. Felsenstein. Cases in which parsimony or compatibility methods will be positively misleading. *Systematic Biology*, 27(4):401–410, Dec. 1978.

[51] J. Felsenstein. Confidence limits on phylogenies: an approach using the bootstrap. *Evolution*, 39(4):783–791, 1985.

[52] J. Felsenstein. Evolutionary trees from DNA sequences: a maximum likelihood approach. *Journal of Molecular Evolution*, 17(6):368–376, 1981.

[53] J. Felsenstein. *Inferring phylogenies*. Sinauer Associates, 2004. ISBN: 978-0-87893-177-4.

[54] J. Felsenstein. PHYLIP-phylogeny inference package (version 3.2). *Cladistics*, 5:164–166, 1989.

[55] J. Felsenstein. Statistical inference of phylogenies. *Royal Statistical Society. Journal. Series A: General*, 146(3):246–262, Dec. 2018.

[56] W. M. Fitch. Toward defining the course of evolution: minimum change for a specific tree topology. *Systematic Biology*, 20(4):406–416, 1971.

[57] R. Fletcher. *Practical methods of optimization*. John Wiley & Sons, 2013.

[58] T. Flouri, F. Izquierdo-Carrasco, D. Darriba, A. Aberer, L.-T. Nguyen, B. Minh, A. Von Haeseler, and A. Stamatakis. The phylogenetic likelihood library. *Systematic Biology*, 64(2):356–362, Oct. 2014.

[59] R. E. Franklin and R. G. Gosling. Molecular configuration in sodium thymonucleate. *Nature*, 171(4356):740–741, 1953.

[60] J. H. Friedman. Greedy function approximation: a gradient boosting machine. *Annals of Statistics*,1189–1232, 2001.

[61] O. Gascuel. BIONJ: an improved version of the NJ algorithm based on a simple model of sequence data. *Molecular Biology and Evolution*, 14(7):685–695, 1997.

[62] S. Geman, E. Bienenstock, and R. Doursat. Neural networks and the bias/variance dilemma. *Neural computation*, 4(1):1–58, 1992.

[63] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys (CSUR)*, 23(1):5–48, 1991.

[64] N. Goldman, J. P. Anderson, and A. G. Rodrigo. Likelihood-based tests of topologies in phylogenetics. *Systematic Biology*, 49(4):652–670, 2000.

[65] P. A. Goloboff, S. A. Catalano, and A. Torres. Parsimony analysis of phylogenomic datasets (II): evaluation of PAUP*, MEGA and MPBoot. *Cladistics*, 38(1):126–146, 2022.

[66] A. D. Gordon. Consensus supertrees: the synthesis of rooted trees containing overlapping sets of labeled leaves. *Journal of Classification*, 3(2):335–348, 1986.

[67] W. K. Gregory. *The orders of mammals*. Vol. 27. Columbia University., 1910.

[68] S. Guindon, J.-F. Dufayard, V. Lefort, M. Anisimova, W. Hordijk, and O. Gascuel. New algorithms and methods to estimate maximum-likelihood phylogenies: assessing the performance of PhyML 3.0. *Systematic Biology*, 59(3):307–321, 2010.

[69] S. Guindon, J.-F. Dufayard, V. Lefort, M. Anisimova, W. Hordijk, and O. Gascuel. New algorithms and methods to estimate maximum-likelihood phylogenies: assessing the performance of PhyML 3.0. *Systematic Biology*, 59(3):307–321, 2010.

[70] S. Guindon and O. Gascuel. A simple, fast, and accurate algorithm to estimate large phylogenies by maximum likelihood. *Systematic Biology*, 52(5):696–704, Oct. 2003.

[71] S. K. Gupta, J. D. Kececioglu, and A. A. Schäffer. Improving the practical space and time efficiency of the shortest-paths approach to sum-of-pairs multiple sequence alignment. *Journal of Computational Biology*, 2(3):459–472, 1995.

[72] D. Gusfield. Algorithms on stings, trees, and sequences: computer science and computational biology. *Acm Sigact News*, 28(4):41–60, 1997.

[73] J. Haag, D. Höhler, B. Bettisworth, and A. Stamatakis. From easy to hopeless—predicting the difficulty of phylogenetic analyses. *Molecular Biology and Evolution*, 39(12):msac254, 2022.

[74] J. Haag, L. Hübner, A. M. Kozlov, and A. Stamatakis. The free lunch is not over yet—systematic exploration of numerical thresholds in maximum likelihood phylogenetic inference. *Bioinformatics Advances*, 3(1):vbad124, 2023.

[75] J. Haag and A. Stamatakis. Pythia 2.0: New Data, New Prediction Model, New Features. *bioRxiv*,2025–03, 2025.

[76] J. A. M. Haag. *Novel applications of machine learning and data science methods in evolutionary genomics*. PhD thesis. Karlsruhe Institute of Technology, 2025.

[77] M. Habib, K. Roy, S. Hasan, A. H. Rahman, and M. S. Bayzid. Terraces in species tree inference from gene trees. *BMC Ecology and Evolution*, 24(1):135, 2024.

[78] E. Haeckel. *Generelle Morphologie der Organismen: Allgemeine Grundzüge der organischen Formen-Wissenschaft, mechanisch begründet durch die von Charles Darwin reformierte Descendenz-Theorie. Band 1: Allgemeine Anatomie. Band 2: Allgemeine Entwicklungsgeschichte*. de Gruyter, 1866.

[79] J. A. Hartigan. Minimum mutation fits to a given tree. *Biometrics*,53–65, 1973.

[80] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. 2nd. New York: Springer, 2009. ISBN: 978-0-387-84857-0.

[81] L. Häuser, G. Jäger, and A. Stamatakis. A systematic exploration of current limitations of cognate-based phylogenetic inference. *Open Research Europe*, 5(258): 2025.

[82] L. S. Heath and N. Ramakrishnan. *Problem solving handbook in computational biology and bioinformatics.* 1st. Springer-Verlag GmbH, 2010. ISBN: 9780387097602.

[83] S. Henikoff and J. G. Henikoff. Amino acid substitution matrices from protein blocks. *Proceedings of the National Academy of Sciences*, 89(22):10915–10919, 1992.

[84] N. J. Higham. The accuracy of floating point summation. *SIAM Journal on Scientific Computing*, 14(4):783–799, 1993.

[85] D. M. Hillis and J. P. Huelsenbeck. Signal, noise, and reliability in molecular phylogenetic analyses. *Journal of Heredity*, 83(3):189–195, June 1992.

[86] D. M. Hillis and J. P. Huelsenbeck. Signal, noise, and reliability in molecular phylogenetic analyses. *Journal of Heredity*, 83(3):189–195, June 1992.

[87] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv:1207.0580*, 2012.

[88] A. E. Hoerl and R. W. Kennard. Ridge regression: biased estimation for nonorthogonal problems. *Technometrics*, 12(1):55–67, 1970.

[89] D. Höhler, J. Haag, A. M. Kozlov, B. Morel, and A. Stamatakis. Performance assessment of phylogenetic inference tools using PhyloSmew. *Bioinformatics Advances*,vbaf300, 2025.

[90] D. Höhler, W. Pfeiffer, V. Ioannidis, H. Stockinger, and A. Stamatakis. RAxML Grove: an empirical phylogenetic tree database. *Bioinformatics*, 38(6):1741–1742, Dec. 2021.

[91] D. Höhler, W. Pfeiffer, V. Ioannidis, H. Stockinger, and A. Stamatakis. RAxML Grove: an empirical phylogenetic tree database. *Bioinformatics*, 38(6):1741–1742, Dec. 2021.

[92] F. Hoseini, A. Atalar, and P. Tsigas. Modeling the performance of atomic primitives on modern architectures. *Proceedings of the 48th International Conference on Parallel Processing*. 2019, pp. 1–11.

[93] J. P. Huelsenbeck and D. M. Hillis. Success of phylogenetic methods in the four-taxon case. *Systematic Biology*, 42(3):247–264, 1993.

[94] D. H. Huson, R. Rupp, and C. Scornavacca. *Phylogenetic networks: concepts, algorithms and applications*. Cambridge University Press, 2010.

[95] E. D. Jarvis, S. Mirarab, A. J. Aberer, B. Li, P. Houde, C. Li, S. Y. Ho, B. C. Faircloth, B. Nabholz, J. T. Howard, et al. Whole-genome analyses resolve early branches in the tree of life of modern birds. *Science*, 346(6215):1320–1331, 2014.

[96] D. T. Jones, W. R. Taylor, and J. M. Thornton. The rapid generation of mutation data matrices from protein sequences. *Bioinformatics*, 8(3):275–282, 1992.

[97] T. H. Jukes, C. R. Cantor, et al. Evolution of protein molecules. *Mammalian Protein Metabolism*, 3(21):132, 1969.

[98] Julia Haag. PyPythia: Phylogenetic Difficulty Prediction Library. Available at: `https://github.com/tschuelia/PyPythia`. Accessed: 2025-11-28.

[99] Julia Haag. Pythia C Library. Available at: `https://github.com/tschuelia/CPythia`. Accessed: 2025-11-28.

[100] W. Just. Computational complexity of multiple sequence alignment with SP-score. *Journal of Computational Biology*, 8(6):615–623, 2001.

[101] K. Katoh and D. M. Standley. MAFFT multiple sequence alignment software version 7: improvements in performance and usability. *Molecular Biology and Evolution*, 30(4):772–780, 2013.

[102] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu. LightGBM: a highly efficient gradient boosting decision tree. *Advances in Neural Information Processing Systems*, 30: 2017.

[103] F. Keck, F. Rimet, A. Bouchez, and A. Franc. phylosignal: an R package to measure, test, and explore the phylogenetic signal. *Ecology and Evolution*, 6(9):2774–2780, 2016.

[104] J. Kim and M. J. Sanderson. Penalized likelihood phylogenetic inference: bridging the parsimony-likelihood gap. *Systematic Biology*, 57(5):665–674, 2008.

[105] D. P. Kingma and J. Ba. Adam: a method for stochastic optimization. 2017. arXiv: `1412.6980 [cs.LG]`. URL: `https://arxiv.org/abs/1412.6980`.

[106] H. Kishino and M. Hasegawa. Evaluation of the maximum likelihood estimate of the evolutionary tree topologies from DNA sequence data, and the branching order in Hominoidea. *Journal of Molecular Evolution*, 29:170–179, 1989.

[107] H. Kishino, T. Miyata, and M. Hasegawa. Maximum likelihood inference of protein phylogeny and the origin of chloroplasts. *Journal of Molecular Evolution*, 31:151–160, 1990.

[108] K. Kobert, A. Stamatakis, and T. Flouri. Efficient Detection of Repeating Sites to Accelerate Phylogenetic Likelihood Calculations. *Systematic Biology*, 66(2):205–217, Aug. 2016.

[109] C. Kosiol and N. Goldman. Different versions of the Dayhoff rate matrix. *Molecular Biology and Evolution*, 22(2):193–199, 2005.

[110] A. Kozlov. *Models, optimizations, and tools for large-scale phylogenetic inference, handling sequence uncertainty, and taxonomic validation.* PhD thesis. Karlsruhe Institute of Technology, 2018.

[111] A. M. Kozlov, D. Darriba, T. Flouri, B. Morel, and A. Stamatakis. RAxML-NG: a fast, scalable and user-friendly tool for maximum likelihood phylogenetic inference. *Bioinformatics*, 35(21):4453–4455, May 2019.

[112] O. M. Kozlov. RAxML-NG v1.2.0 Github Release. Available at: `https://github.com/amkozlov/raxml-ng/releases/tag/1.2.0`. Accessed: 2026-01-28.

[113] M. K. Kuhner and J. Felsenstein. A simulation comparison of phylogeny algorithms under equal and unequal evolutionary rates. *Molecular Biology and Evolution*, 11(3):459–468, 1994.

[114] J.-B. d. M. de Lamarck. *Philosophie zoologique*. Vol. 1. F. Savy, 1873.

[115] R. Lanfear, B. Calcott, S. Y. Ho, and S. Guindon. PartitionFinder: combined selection of partitioning schemes and substitution models for phylogenetic analyses. *Molecular Biology and Evolution*, 29(6):1695–1701, 2012.

[116] S. Q. Le and O. Gascuel. An improved general amino acid replacement matrix. *Molecular Biology and Evolution*, 25(7):1307–1320, 2008.

[117] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 2002.

[118] V. Lefort, R. Desper, and O. Gascuel. FastME 2.0: a comprehensive, accurate, and fast distance-based phylogeny inference program. *Molecular Biology and Evolution*, 32(10):2798–2800, 2015.

[119] V. Lefort, J.-E. Longueville, and O. Gascuel. SMS: smart model selection in PhyML. *Molecular Biology and Evolution*, 34(9):2422–2424, 2017.

[120] F. Lemoine, J.-B. Domelevo Entfellner, E. Wilkinson, D. Correia, M. Dávila Felipe, T. De Oliveira, and O. Gascuel. Renewing Felsenstein's phylogenetic bootstrap in the era of big data. *Nature*, 556(7702):452–456, 2018.

[121] W. Li, A. Koshkarov, and N. Tahiri. Comparison of phylogenetic trees defined on different but mutually overlapping sets of taxa: A review. *Ecology and Evolution*, 14(8):e70054, 2024.

[122] C. Liu, X. Zhou, Y. Li, C. T. Hittinger, R. Pan, J. Huang, X.-x. Chen, A. Rokas, Y. Chen, and X.-X. Shen. The influence of the number of tree searches on maximum likelihood inference in phylogenomics. *Systematic Biology*, 73(5):807–822, 2024.

[123] K. Liu, C. R. Linder, and T. Warnow. RAxML and FastTree: comparing two methods for large-scale maximum likelihood phylogeny estimation. *PloS ONE*, 6(11):e27731, 2011.

[124] W. P. Maddison. Gene trees in species trees. *Systematic Biology*, 46(3):523–536, Sept. 1997.

[125] U. Mai and S. Mirarab. TreeShrink: fast and accurate detection of outlier long branches in collections of phylogenetic trees. *BMC Genomics*, 19(Suppl 5):272, 2018.

[126] E. R. Mardis. The impact of next-generation sequencing technology on genetics. *Trends in Genetics*, 24(3):133–141, 2008.

[127] M. Margulies, M. Egholm, W. E. Altman, S. Attiya, J. S. Bader, L. A. Bemben, J. Berka, M. S. Braverman, Y.-J. Chen, Z. Chen, et al. Genome sequencing in microfabricated high-density picolitre reactors. *Nature*, 437(7057):376–380, 2005.

[128] E. Markowski and E. Susko. Performance of Topology Tests under Extreme Selection Bias. *Molecular Biology and Evolution*, 41(1):msad280, Dec. 2023.

[129] M. F. Mickevich. Taxonomic congruence. *Systematic Zoology*,143–158, 1978.

[130] B. Q. Minh, R. Lanfear, T. Wong, N. Ly-Trong, J. Trifinopoulos, D. Schrempf, and H. A. Schmidt. IQ-TREE version 3.0.0 - Tutorials and Manual: Phylogenomic software by maximum likelihood. `https://iqtree.github.io/doc/iqtree-doc.pdf`. Accessed: 2025-11-04.

[131] B. Q. Minh, H. A. Schmidt, O. Chernomor, D. Schrempf, M. D. Woodhams, A. von Haeseler, and R. Lanfear. IQ-TREE 2: new models and efficient methods for phylogenetic inference in the genomic era. *Molecular Biology and Evolution*, 37(5):1530–1534, Feb. 2020.

[132] G. Montavon, G. Orr, and K.-R. Müller. *Neural networks: tricks of the trade*. Vol. 7700. Springer Berlin, Heidelberg, 2012. ISBN: 978-3-642-35288-1. DOI: `https://doi.org/10.1007/978-3-642-35289-8`.

[133] B. Morel, P. Barbera, L. Czech, B. Bettisworth, L. Hübner, S. Lutteropp, D. Serdari, E.-G. Kostaki, I. Mamais, A. M. Kozlov, et al. Phylogenetic analysis of SARS-CoV-2 data is difficult. *Molecular Biology and Evolution*, 38(5):1777–1791, 2021.

[134] B. Morel, T. Flouri, and A. Stamatakis. A novel heuristic for data distribution in massively parallel phylogenetic inference using site repeats. *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. 2017, pp. 81–88. DOI: 10.1109/HPCC-SmartCity-DSS.2017.11.

[135] B. Morel, A. M. Kozlov, A. Stamatakis, and G. J. Szöllősi. GeneRax: a tool for species-tree-aware maximum likelihood-based gene family tree inference under gene duplication, transfer, and loss. *Molecular Biology and Evolution*, 37(9):2763–2774, June 2020.

[136] B. Morel, P. Schade, S. Lutteropp, T. A. Williams, G. J. Szöllősi, and A. Stamatakis. SpeciesRax: a tool for maximum likelihood species tree inference from gene family trees under duplication, transfer, and loss. *Molecular Biology and Evolution*, 39(2):msab365, Jan. 2022.

[137] D. A. Morrison. Increasing the efficiency of searches for the maximum likelihood tree in a phylogenetic analysis of up to 150 nucleotide sequences. *Systematic Biology*, 56(6):988–1010, 2007.

[138] I. Moutsopoulos, L. Maischak, E. Lauzikaite, S. A. Vasquez Urbina, E. C. Williams, H.-G. Drost, and I. I. Mohorianu. noisyR: enhancing biological signal in sequencing datasets by characterizing random technical noise. *Nucleic Acids Research*, 49(14):e83–e83, June 2021.

[139] T. Müller and M. Vingron. Modeling amino acid replacement. *Journal of Computational Biology*, 7(6):761–776, 2000.

[140] T. Münkemüller, S. Lavergne, B. Bzeznik, S. Dray, T. Jombart, K. Schiffers, and W. Thuiller. How to measure and test phylogenetic signal. *Methods in Ecology and Evolution*, 3(4):743–756, 2012.

[141] S. Naser-Khdour, B. Q. Minh, W. Zhang, E. A. Stone, and R. Lanfear. The prevalence and impact of model violations in phylogenetic analysis. *Genome Biology and Evolution*, 11(12):3341–3352, Sept. 2019.

[142] National Center for Biotechnology Information (NCBI). The genetic codes. Available at: https://www.ncbi.nlm.nih.gov/Taxonomy/Utils/wprintgc.cgi. Accessed: 2025-12-12.

[143] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, 1970.

[144] L.-T. Nguyen, H. A. Schmidt, A. von Haeseler, and B. Q. Minh. IQ-TREE: a fast and effective stochastic algorithm for estimating maximum-likelihood phylogenies. *Molecular Biology and Evolution*, 32(1):268–274, Nov. 2014.

[145] D. C. Nickle, L. Heath, M. A. Jensen, P. B. Gilbert, J. I. Mullins, and S. L. Kosakovsky Pond. HIV-specific probabilistic models of protein evolution. *PloS ONE*, 2(6):e503, 2007.

[146] R. R. Picard and R. D. Cook. Cross-validation of regression models. *Journal of the American Statistical Association*, 79(387):575–583, 1984.

[147] W. H. Piel, L. Chan, M. J. Dominus, J. Ruan, R. A. Vos, and V. Tannen. TreeBASE v. 2: a database of phylogenetic knowledge. *e-BioSphere 2009*, 2009.

[148] C. Piñeiro, J. M. Abuín, and J. C. Pichel. Very Fast Tree: speeding up the estimation of phylogenies for large alignments through parallelization and vectorization strategies. *Bioinformatics*, 36(17):4658–4659, June 2020.

[149] D. Posada and K. A. Crandall. MODELTEST: testing the model of DNA substitution. *Bioinformatics*, 14(9):817–818, 1998.

[150] M. N. Price. FastTree 2.2: Approximately-Maximum-Likelihood Trees for Large Alignments (README, "CAT/Gamma20 Likelihoods" section). Available at: `https://morgannprice.github.io/fasttree`. Accessed: 2025-11-03.

[151] M. N. Price, P. S. Dehal, and A. P. Arkin. FastTree 2 – approximately maximum-likelihood trees for large alignments. *PLoS ONE*, 5(3):1–10, Mar. 2010.

[152] M. Rabiee, E. Sayyari, and S. Mirarab. Multi-allele species reconstruction using ASTRAL. *Molecular Phylogenetics and Evolution*, 130:286–296, 2019.

[153] M. A. Ragan. Trees and networks before and after Darwin. *Biology Direct*, 4(1):43, 2009.

[154] L. J. Revell, L. J. Harmon, and D. C. Collar. Phylogenetic signal, evolutionary process, and rate. *Systematic Biology*, 57(4):591–601, Aug. 2008.

[155] D. Robinson and L. Foulds. Comparison of phylogenetic trees. *Mathematical Biosciences*, 53(1):131–147, 1981.

[156] D. F. Robinson and L. R. Foulds. Comparison of weighted labelled trees. *Combinatorial Mathematics VI: Proceedings of the Sixth Australian Conference on Combinatorial Mathematics, Armidale, Australia, August 1978*. Springer. 2006, pp. 119–126.

[157] S. Roch. A short proof that phylogenetic tree reconstruction by maximum likelihood is hard. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 3(1):92–94, 2006.

[158] A. Rokas and S. B. Carroll. Bushes in the tree of life. *PLoS Biology*, 4(11):e352, 2006.

[159] M. S. Rosenberg and S. Kumar. Incomplete taxon sampling is not a problem for phylogenetic inference. *Proceedings of the National Academy of Sciences*, 98(19):10751–10756, 2001.

[160] O. Rota-Stabelli, Z. Yang, and M. J. Telford. MtZoa: a general mitochondrial amino acid substitutions model for animal evolutionary studies. *Molecular Phylogenetics and Evolution*, 52(1):268–272, 2009.

[161] A. RoyChoudhury. Consistency of the maximum likelihood estimator of evolutionary tree. *arXiv preprint arXiv:1405.0760*, 2014.

[162] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[163] A. Rzhetsky and M. Nei. Theoretical foundation of the minimum-evolution method of phylogenetic inference. *Molecular Biology and Evolution*, 10(5):1073–1095, 1993.

[164] N. Saitou and M. Nei. The neighbor-joining method: a new method for reconstructing phylogenetic trees. *Molecular Biology and Evolution*, 4(4):406–425, 1987.

[165] A. Sand, M. K. Holt, J. Johansen, G. S. Brodal, T. Mailund, and C. N. Pedersen. tqDist: a library for computing the quartet and triplet distances between binary or general trees. *Bioinformatics*, 30(14):2079–2080, 2014.

[166] M. J. Sanderson, M. M. McMahon, A. Stamatakis, D. J. Zwickl, and M. Steel. Impacts of terraces on phylogenetic inference. *Systematic biology*, 64(5):709–726, 2015.

[167] M. J. Sanderson, M. M. McMahon, and M. Steel. Terraces in phylogenetic tree space. *Science*, 333(6041):448–450, 2011.

[168] F. Sanger. The arrangement of amino acids in proteins. *Advances in Protein Chemistry*. Vol. 7. Elsevier, 1952, pp. 1–67.

[169] F. Sanger, S. Nicklen, and A. R. Coulson. DNA sequencing with chain-terminating inhibitors. *Proceedings of the National Academy of Sciences*, 74(12):5463–5467, 1977.

[170] F. Sanger and E. Thompson. The amino-acid sequence in the glycyl chain of insulin. 1. The identification of lower peptides from partial hydrolysates. *Biochemical Journal*, 53(3):353, 1953.

[171] D. Sankoff. Minimal mutation trees of sequences. *SIAM Journal on Applied Mathematics*, 28(1):35–42, 1975.

[172] E. Sayyari and S. Mirarab. Testing for polytomies in phylogenetic species trees using quartet frequencies. *Genes*, 9(3):132, 2018.

[173] G. Schwarz. Estimating the dimension of a model. *The Annals of Statistics*,461–464, 1978.

[174] C. Semple, M. Steel, et al. *Phylogenetics*. Vol. 24. Oxford University Press, 2003.

[175] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 1948.

[176] X.-X. Shen, Y. Li, C. T. Hittinger, X.-x. Chen, and A. Rokas. An investigation of irreproducibility in maximum likelihood phylogenetic inference. *Nature Communications*, 11(1):6096, 2020.

[177] H. Shimodaira. An application of multiple comparison techniques to model selection. *Annals of the Institute of Statistical Mathematics*, 50(1):1–13, 1998.

[178] H. Shimodaira. An approximately unbiased test of phylogenetic tree selection. *Systematic Biology*, 51(3):492–508, May 2002.

[179] H. Shimodaira and M. Hasegawa. CONSEL: for assessing the confidence of phylogenetic tree selection. *Bioinformatics*, 17(12):1246–1247, 2001.

[180] H. Shimodaira and M. Hasegawa. Multiple comparisons of log-likelihoods with applications to phylogenetic inference. *Molecular Biology and Evolution*, 16(8):1114, 1999.

[181] H. Shimodaira and Y. Terada. Selective inference for testing trees and edges in phylogenetics. *Frontiers in Ecology and Evolution*, 7:174, 2019.

[182] F. Sievers, A. Wilm, D. Dineen, T. J. Gibson, K. Karplus, W. Li, R. Lopez, H. McWilliam, M. Remmert, J. Söding, et al. Fast, scalable generation of high-quality protein multiple sequence alignments using Clustal Omega. *Molecular Systems Biology*, 7(1):539, 2011.

[183] M. P. Simmons and J. Kessenich. Divergence and support among slightly suboptimal likelihood gene trees. *Cladistics*, 36(3):322–340, 2020.

[184] M. P. Simmons, H. Ochoterena, and J. V. Freudenstein. Amino acid vs. nucleotide characters: challenging preconceived notions. *Molecular Phylogenetics and Evolution*, 24(1):78–90, 2002.

[185] G. G. Simpson. *Tempo and mode in evolution*. 15. Columbia University Press, 1944.

[186] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[187] K. St. John. Review paper: the shape of phylogenetic treespace. *Systematic Biology*, 66(1):e83–e94, June 2016.

[188] A. Stamatakis, T. Ludwig, and H. Meier. RAxML-III: a fast program for maximum likelihood-based inference of large phylogenetic trees. *Bioinformatics*, 21(4):456–463, Dec. 2004.

[189] A. Stamatakis. Datasets. `https://github.com/stamatak/test-Datasets`. Accessed: 2026-02-02.

[190] A. Stamatakis. Phylogenetic models of rate heterogeneity: a high performance computing perspective. *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*. IEEE. 2006, 8–pp.

[191] A. Stamatakis. Phylogenetic search algorithms for maximum likelihood. *Algorithms in Computational Molecular Biology: Techniques, Approaches and Applications*,547–577, 2011.

[192] A. Stamatakis. Phylogenetic search algorithms for maximum likelihood. *Algorithms in computational biology: techniques, approaches and applications*. Ed. by M. Elloumi and A. Y. Zomaya. John Wiley & Sons, 2011, pp. 547–577.

[193] A. Stamatakis. RAxML version 8: a tool for phylogenetic analysis and post-analysis of large phylogenies. *Bioinformatics*, 30(9):1312–1313, 2014.

[194] A. Stamatakis. Standard RAxML version 8.2.12. `https://github.com/stamatak/standard-RAxML`. Accessed: 2025-12-15.

[195] A. Stamatakis. The RAxML v8.2.X Manual. `https://cme.h-its.org/exelixis/resource/download/NewManual.pdf`. Accessed: 2025-12-15.

[196] A. Stamatakis and N. Alachiotis. Time and memory efficient likelihood-based tree searches on phylogenomic alignments with missing data. *Bioinformatics*, 26(12):i132–i139, 2010.

[197] A. Stamatakis, F. Blagojevic, D. S. Nikolopoulos, and C. D. Antonopoulos. Exploring new search algorithms and hardware for phylogenetics: RAxML meets the IBM cell. *The Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, 48(3):271–286, 2007.

[198] M. A. Steel and D. Penny. Distributions of tree comparison metrics—some new results. *Systematic Biology*, 42(2):126–141, 1993.

[199] M. A. Steel and D. Penny. Distributions of tree comparison metrics—some new results. *Systematic Biology*, 42(2):126–141, June 1993.

[200] C. Stelz, L. Huebner, and A. Stamatakis. Bit-reproducible phylogenetic tree inference under varying core-counts via reproducible parallel reduction operators. *bioRxiv*,2025–06, 2025.

[201] K. Strimmer and A. Rambaut. Inferring confidence sets of possibly misspecified gene trees. *Proceedings of the Royal Society of London. Series B: Biological Sciences*, 269(1487):137–142, 2002.

[202] J. Sullivan and P. Joyce. Model selection in phylogenetics. *Annu. Rev. Ecol. Evol. Syst.* 36(1):445–466, 2005.

[203] D. Swofford. PAUP*. Phylogenetic analysis using parsimony. 2003.

[204] G. Tan, M. Muffato, C. Ledergerber, J. Herrero, N. Goldman, M. Gil, and C. Dessimoz. Current methods for automated filtering of multiple sequence alignments frequently worsen single-gene phylogenetic inference. *Systematic Biology*, 64(5):778–791, 2015.

[205] S. Tavaré. Some probabilistic and statistical problems in the analysis of DNA sequences. *Lect Math Life Sci (Am Math Soc)*, 17:57–86, 1986.

[206] R. Tillyard. A new classification of the order Perlaria. *The Canadian Entomologist*, 53(2):35–43, 1921.

[207] J. E. Toews. *Hegelianism: the path toward dialectical humanism, 1805-1841*. Cambridge University Press, 1980.

[208] A. Togkousidis, O. Chernomor, and A. Stamatakis. Parallel inference of phylogenetic stands with gentrius. *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 2023, pp. 139–148.

[209] A. Togkousidis, O. Gascuel, and A. Stamatakis. A systematic investigation of overfitting in maximum likelihood phylogenetic inference. *bioRxiv*, 2025.

[210] A. Togkousidis, O. M. Kozlov, J. Haag, D. Höhler, and A. Stamatakis. Adaptive RAxML-NG: accelerating phylogenetic inference under maximum likelihood using dataset difficulty. *Molecular Biology and Evolution*, 40(10):msad227, 2023.

[211] A. Togkousidis, A. Stamatakis, and O. Gascuel. Accelerating maximum likelihood phylogenetic inference via early stopping to evade (over-) optimization. *Systematic Biology*,syaf043, 2025.

[212] J. P. Townsend, Z. Su, and Y. I. Tekle. Phylogenetic signal and noise: predicting the power of a data set to resolve phylogeny. *Systematic Biology*, 61(5):835, 2012.

[213] N. Ly-Trong, S. Naser-Khdour, R. Lanfear, and B. Q. Minh. AliSim: A fast and versatile phylogenetic sequence simulator for the genomic era. *Molecular Biology and Evolution*, 39(5): May 2022.

[214] J. Trost, J. Haag, D. Höhler, L. Jacob, A. Stamatakis, and B. Boussau. Simulations of sequence evolution: how (un)realistic they are and why. *Molecular Biology and Evolution*, 41(1):msad277, 2024.

[215] P. Vachaspati and T. Warnow. SIESTA: enhancing searches for optimal supertrees and species trees. *BMC Genomics*, 19(Suppl 5):252, 2018.

[216] L. Van Dorp, M. Acman, D. Richard, L. P. Shaw, C. E. Ford, L. Ormond, C. J. Owen, J. Pang, C. C. Tan, F. A. Boshier, et al. Emergence of genomic diversity and recurrent mutations in SARS-CoV-2. *Infection, Genetics and Evolution*, 83:104351, 2020.

[217] S. Veerassamy, A. Smith, and E. R. Tillier. A transition probability model for amino acid substitutions from blocks. *Journal of Computational Biology*, 10(6):997–1010, 2003.

[218] D. M. de Vienne, F. Supek, and T. Gabaldon. Overtraining often results in topologically incorrect species trees with maximum likelihood methods. *bioRxiv*,140780, 2017.

[219] L. S. Vinh and A. Von Haeseler. IQPNNI: moving fast through tree space and stopping in time. *Molecular Biology and Evolution*, 21(8):1565–1571, 2004.

[220] A. Wald. Note on the consistency of the maximum likelihood estimate. *The Annals of Mathematical Statistics*, 20(4):595–601, 1949.

[221] J. D. Watson and F. H. Crick. Molecular structure of nucleic acids: a structure for deoxyribose nucleic acid. *Nature*, 171(4356):737–738, 1953.

[222] K. A. Wetterstrand. DNA sequencing costs: data from the NHGRI genome sequencing program (GSP). `https://www.genome.gov/sequencingcostsdata`. Accessed: 2025-11-10.

[223] S. Whelan and N. Goldman. A general empirical model of protein evolution derived from multiple protein families using a maximum-likelihood approach. *Molecular Biology and Evolution*, 18(5):691–699, 2001.

[224] S. Whelan and N. Goldman. A general empirical model of protein evolution derived from multiple protein families using a maximum-likelihood approach. *Molecular Biology and Evolution*, 18(5):691–699, 2001.

[225] T. A. Williams, C. J. Cox, P. G. Foster, G. J. Szöllősi, and T. M. Embley. Phylogenomics provides robust support for a two-domains tree of life. *Nature Ecology & Evolution*, 4(1):138–147, 2020.

[226] C. R. Woese, O. Kandler, and M. L. Wheelis. Towards a natural System of organisms: proposal for the domains Archaea, Bacteria, and Eucarya. *Proceedings of the National Academy of Sciences*, 87(12):4576–4579, 1990.

[227] T. K. Wong, N. Ly-Trong, H. Ren, H. Baños, A. J. Roger, E. Susko, C. Bielow, N. De Maio, N. Goldman, M. W. Hahn, et al. IQ-TREE 3: Phylogenomic inference software using complex evolutionary models, 2025.

[228] Z. Yang. *Molecular evolution: a statistical approach*. OUP Oxford, 2014. ISBN: 9780191023309. URL: `https://books.google.gr/books?id=T-LoAwAAQBAJ`.

[229] Z. Yang. Maximum likelihood phylogenetic estimation from DNA sequences with variable rates over sites: approximate methods. *Journal of Molecular Evolution*, 39(3):306–314, 1994.

[230] Z. Yang, N. Goldman, and A. Friday. Maximum likelihood trees from DNA sequences: a peculiar statistical estimation problem. *Systematic Biology*, 44(3):384–399, 1995.

[231] Z. Yang, R. Nielsen, and M. Hasegawa. Models of amino acid substitution and applications to mitochondrial protein evolution. *Molecular Biology and Evolution*, 15(12):1600–1611, 1998.

[232] C. Zhang, C. Scornavacca, E. K. Molloy, and S. Mirarab. ASTRAL-Pro: quartet-based species-tree inference despite paralogy. *Molecular Biology and Evolution*, 37(11):3292–3307, 2020.

[233] C. Zhang, V. Dinh, and F. A. Matsen IV. Nonbifurcating phylogenetic tree inference via the adaptive LASSO. *Journal of the American Statistical Association*, 116(534):858–873, 2021.

[234] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals. Understanding deep learning requires rethinking generalization. *arXiv:1611.03530*, 2016.

[235] X. Zhou, X.-X. Shen, C. T. Hittinger, and A. Rokas. Evaluating fast maximum likelihood-based phylogenetic programs using empirical phylogenomic data sets. *Molecular Biology and Evolution*, 35(2):486–503, 2018.

[236] D. J. Zwickl. Terraphy tool. `https://github.com/zwickl/terraphy`. Accessed: 2026-01-14.

Usage of Generative Models

In accordance with the statement of the Deutsche Forschungsgemeinschaft[1] on the 21st of September 2023, regarding the usage of generative models for text and image generation, I declare that:

1. Parts of the code accompanying this thesis were developed with the assistance of ChatGPT (versions 4 and 5) and DeepSeek, for: (a) writing Snakemake pipelines for benchmarking, and (b) assistance in generating Python code for figure creation. All generated code has been verified manually by the author. Further, no code generation or other generative model has been used for interface, data structure, or algorithm design.

2. I used ChatGPT (versions 3 and 4) for grammar checking of all publications mentioned in this thesis, as well as all text in this thesis. All proposed edits have been manually evaluated and accepted by the author.

3. No image in this thesis was generated using generative models.

[1] https://www.dfg.de/de/aktuelles/neuigkeiten-themen/info-wissenschaft/2023/info-wissenschaft-23-72