Implementing and Evaluating Improved Branching Algorithms for Maximum Agreement Forests Using Split-or-Decompose Techniques

Bachelor Thesis

Semih Kan
Supervisor: Dr. David Mestel
github.com/SemihKanUM/Bachelor-Thesis

August 29, 2025

Contents

1 Abstract

Phylogenetic trees are commonly used to depict evolutionary relationships among species. When different trees are constructed for the same taxa, it is important to compare them to understand the differences. The Maximum Agreement Forest (MAF) problem provides a measure of tree dissimilarity and is known to be NP-hard. This thesis investigates a recent improvement in solving MAF using the split-or-decompose technique proposed by Mestel et al. (2024). I implement both a classical FPT algorithm (Chen's algorithm) and the improved variant for unrooted trees, evaluate their performance across various synthetic datasets, and analyze runtime, efficiency. The results show that split-or-decompose yields consistent improvements.

2 Introduction

Phylogenetic trees are a fundamental structure in evolutionary biology, representing hypothesized evolutionary relationships among a set of species or organisms. These species are typically referred to as *taxa* (singular: *taxon*). Each leaf of a phylogenetic tree corresponds to one such taxon, making the trees useful for comparing genetic or evolutionary divergence.

Discrepancies often arise between phylogenetic trees generated from different genetic data or methodologies. Quantifying and analyzing these differences is crucial for gaining a better understanding of evolutionary histories. One widely studied approach to compare phylogenetic trees is the Maximum Agreement Forest (MAF) problem. Given two trees over the same set of taxa, the goal is to partition the taxa into the minimum number of disjoint subsets such that the subtrees induced by each subset are identical in both trees (up to homeomorphism).

This problem is computationally challenging and has been shown to be NP-hard. As such, research efforts have focused on developing Fixed-Parameter Tractable (FPT) algorithms that are parameterized by the number of cuts needed to transform one tree into an agreement forest.

This thesis investigates the impact and performance of a recently proposed strategy, known as the split-or-decompose technique, for solving the MAF problem. Building upon prior algorithms developed by Chen, the new approach introduces a branching rule that efficiently resolves overlapping components by either splitting or decomposing them into simpler subproblems (1).

The remainder of this thesis is structured as follows: Section 2 provides preliminary definitions related to trees and forests. Section 3 reviews relevant literature. Section 4 outlines the methodology. Section 5 presents experimental results, while Section 6 discusses the implications. Section 7 concludes the study and suggests future directions.

3 Preliminaries

In this section, I introduce key concepts relevant to this work.

3.1 Unrooted Phylogenetic Trees

An **unrooted phylogenetic tree** is an undirected, acyclic graph where all internal nodes have degree three, and the leaves are bijectively labeled by a set of taxa X. These trees do not have a designated root, meaning they do not imply any directionality of evolution. They are used to represent the relatedness between species without assuming a specific ancestor.

Two leaves a and b in a tree T are said to form a **cherry** if they are adjacent to the same internal node. Cherries play an important role in simplifying trees and reducing them to smaller substructures during algorithmic processing.

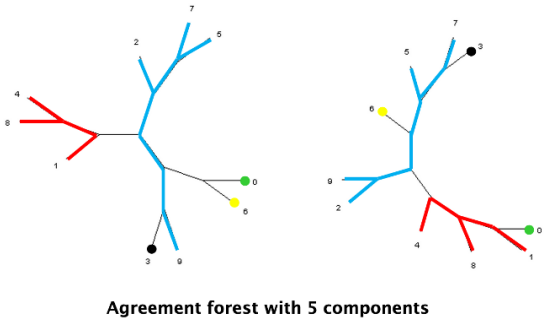

Figure 1: Two example unrooted trees where same-color branches represent homeomorphic components (adapted from materials provided by Steven Kelk[2]).

3.2 Agreement Forests

Let T and T' be two unrooted phylogenetic trees on the same set of taxa X. An **agreement forest** $F = B_1, B_2, \ldots, B_k$ is a partition of X such that:

- For each i, the subtrees $T|B_i$ and $T'|B_i$ are homeomorphic.
- For all $i \neq j$, the embeddings of B_i and B_j in both T and T' are vertex-disjoint.

In the context of our thesis on Maximum Agreement Forests, a homeomorphism between two subtrees means that, after simplifying them by sup-

pressing all degree-2 vertices (i.e., removing intermediate nodes that have exactly two connections and reconnecting their neighbors), the two subtrees have the same topology and leaf labeling.

The goal in the MAF problem is to find such a forest with the minimum number of components. The number of components in a minimum-size agreement forest is called the MAF distance, and corresponds to the fewest number of cuts needed to make both trees isomorphic across all components.

An important observation for understanding how the split-and-decompose approach works is that if two components of the forest overlap—meaning they share at least one edge in the original tree—then it is impossible for the current configuration to form a valid agreement forest.

4 Related Work

Problem connections and hardness. For two trees on the same taxon set, the size of a maximum agreement forest (MAF) characterizes classical rearrangement distances: in the rooted case, the rSPR distance equals (MAF size -1), and in the unrooted case, the TBR distance equals (MAF size -1) (8; 9). These problems, and the associated MAF computation, are NP-hard (9), which motivated a rich line of fixed-parameter and approximation algorithms.

Exact / FPT algorithms. A major thread has focused on parameterizing by the number of cuts k. For rooted binary trees, Whidden and Zeh introduced fast FPT methods with experimental validation (2), and Whidden, Beiko and Zeh established the now-classic fixed-parameter framework for MAF (3). In parallel, Chen, Li, Liu and Wang presented an improved parameterized algorithm for MAF with running time $O(3^k \cdot n)$ and additional approximation results (4). Most recently, Mestel, Chaplick, Kelk and Meuwese proposed the *split-or-decompose* branching strategy, achieving improved worst-case bounds for two-tree MAF (rooted and unrooted) and supplying a modern blueprint for practical branching (1). Our implementation follows this philosophy and instruments it to measure how often splits trigger and how much they help.

Approximation algorithms. On the approximation side, Rodrigues, Sagot and Wakabayashi gave early constant-factor approximations with computational experiments (5). For nonbinary trees, van Iersel, Kelk, Lekic and Stougie provided constant-factor approximations tailored to multifurcating inputs (6). More recently, Olver *et al.* obtained a combinatorial 2-approximation for two rooted binary trees via LP duality (7).

Implementations and practical solvers. Several software efforts make these ideas usable in practice. The `rSPR` package by Whidden implements rooted SPR distances and rooted agreement forests and has been widely used as a practical baseline (10). Kelk's public repository provides synthetic datasets and code used in recent kernelization and MAF studies (11).

Within exact (FPT) algorithms for two-tree MAF, the recent *Split-or-Decompose* framework of Mestel *et al.* refines the classic branching used by Whidden *et al.* and by Chen *et al.* by introducing a principled "split" step when components overlap and a "decompose" step otherwise, yielding improved worst-case bounds (1; 3; 4). In this work we implement that SoD scheme and compare it directly against a Chen-style baseline, *how much* they reduce search (recursion savings) as a function of problem difficulty k and instance features.

5 Methodology

This work implements and evaluates two fixed-parameter tractable (FPT) algorithms for solving the unrooted Maximum Agreement Forest (uMAF) problem. The first is the classical branching approach by Chen *et al.*, which is based on cherry picking and path splitting. The second is an enhanced variant that incorporates the split-or-decompose technique proposed by Mestel *et al.* (1), designed to improve efficiency by handling overlapping components more effectively.

5.1 Algorithm Descriptions

5.1.1 Chen's Algorithm

Chen's algorithm is a branching method that repeatedly searches for *cherries*—pairs of leaves that share a common parent in one of the trees. Cherries are crucial because they represent small, localized structures where disagreements between the two trees can be resolved with minimal cuts.

If a cherry is common to both trees, it can be collapsed into a single leaf without affecting the final solution. If it is present only in one tree, the algorithm branches into different scenarios: either cutting off one leaf of the cherry or strategically cutting edges along the path between the two leaves. This guarantees that all possible resolutions are explored, while steadily reducing the problem size.

5.1.2 Split-or-Decompose Technique

The split-or-decompose approach extends Chen's algorithm by introducing a new branching rule to address *overlapping components*. Two components in a forest are said to overlap if, when mapped back

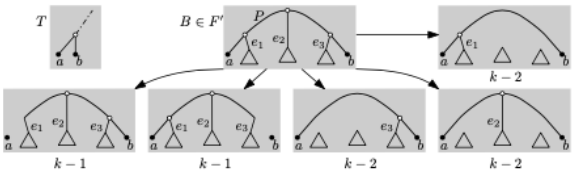

Figure 2: Chen's branching algorithm splitting the path between cherry nodes (adapted from (1)).

to the original tree, they share at least one common edge. Because an agreement forest requires components to be vertex-disjoint in both trees, such overlaps must be eliminated before a valid solution can be formed.

Branching Rule 1 from (1) is applied to split overlapping components by selectively cutting edges to separate their embeddings. Once overlaps are removed, the remaining disjoint components can be solved as independent subproblems. This decomposition often reduces the branching factor and improves overall runtime.

The theoretical details of both algorithms are explicitly described in Mestel *et al.* (1). In this work, we therefore focus primarily on the software implementation of these algorithms and the experimental evaluation of their performance.

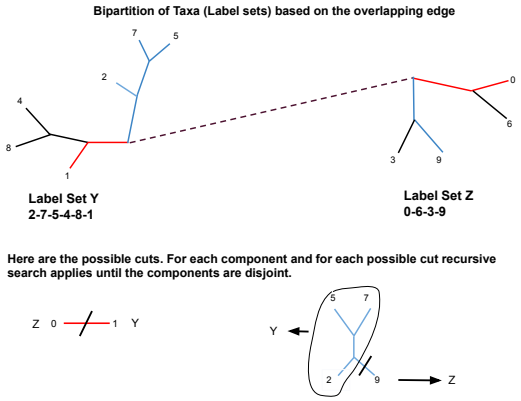

Figure 3: Illustration of a bipartition induced by an edge of the tree. Cutting the edge divides the taxa into two label sets Y and Z, shown here as $Y = \{2, 7, 5, 4, 8, 1\}$ and $Z = \{0, 6, 3, 9\}$. Such bipartitions form the basis for recursive branching, where possible cuts are considered until the resulting components are disjoint.

5.2 Implementation Details

Chen's algorithm is relatively straightforward and intuitive, which makes it a suitable baseline for this

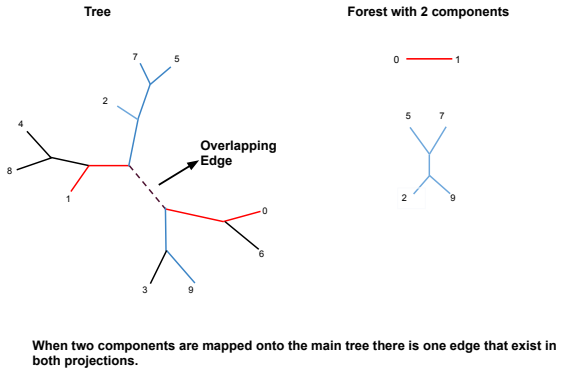

Figure 4: Example of overlapping components. When two components of the forest are mapped onto the main tree, they may share a common edge (highlighted here). This overlapping edge prevents the components from being disjoint, and thus triggers the split rule: one branch of the algorithm assigns the edge to the first component, and another branch assigns it to the second.

work. The Split-or-Decompose algorithm is built directly upon Chen's approach. It should be emphasized that this project focuses primarily on the Java implementation of these algorithms; therefore, this paper alone may not fully convey all technical details, and readers are encouraged to consult the main repository for the complete code. The central objective here is to ensure correct implementation of the methods and to evaluate their results. Both algorithms were implemented in Java and were verified to be accurate and robust.

5.2.1 Tidy-up Operations

Before each branching step, the algorithm performs a set of *tidy-up* operations to simplify the current instance and reduce unnecessary complexity. These preprocessing steps help ensure that the recursion proceeds on the smallest possible problem instance without changing the correctness of the result.

5.2.2 Implementation of Chen's Algorithm

The core idea of Chen's algorithm for the Maximum Agreement Forest problem is to iteratively simplify the trees and then branch on specific structures called *cherries*. The implementation proceeds as follows:

1. **Tidy-up operations:** Simplification steps are applied to the trees, including the removal of redundant nodes, suppression of degree-2 nodes, and collapsing of common cherries.

These operations reduce the problem size without affecting correctness.

2. **Cherry selection:** The algorithm identifies a cherry in the current tree. A cherry is defined as a pair of leaves that share a common parent.

3. **Branching:**

   - If the leaves of the cherry lie in different components of the forest, the algorithm branches by cutting off either leaf, creating two recursive calls. Each cut decreases the parameter k by one.
   - If the leaves lie in the same component, Chen's branching rule is applied: three recursive branches are created—(i) cutting off one leaf, (ii) cutting off the other leaf, or (iii) splitting the path between them. In the third case, one neighboring edge of the path is preserved while all other incident edges are removed.

4. **Base case:** If no cherries remain, the algorithm checks whether the forest forms a valid agreement forest for the current tree. If so, a solution is obtained.

This approach systematically explores all possibilities while using simplification steps to keep the search space manageable.

5.2.3 Validation of Chen's Algorithm

To validate the correctness of our implementation of Chen's algorithm, we first reconstructed the exact phylogenetic trees depicted in Figure 1, as provided by Steven Kelk. The algorithm was then tested for different values of k, starting from 0. For each value of k, possible agreement forests were generated, and after three valid agreement forests were identified, the search was terminated. The algorithm successfully reproduced the agreement forest shown in the reference figure, thereby confirming its correctness. Moreover, it identified additional valid agreement forests for the same value of k. In this test case, three valid agreement forests exist in total, including the one illustrated in the figure. Each forest is represented below as a set of components, where each component lists the leaf labels it contains:

[3, 6, 0, 1_4_8, 2_5_7_9]

[3, 2, 0, 1_4_8, 5_6_7_9]

[3, 5_7, 0, 1_4_8, 2_6_9]

These results demonstrate that the implementation not only matches known solutions but also exhaustively enumerates many other valid solutions for the specified k. The correctness of each of these agreement forests were checked manually including the one that was provided.

In addition to this specific test, we used a synthetic dataset from Steven Kelk's GitHub repository [11]. This dataset contains hundreds of phylogenetic tree pairs together with their corresponding k values (equal to the TBR distance for each instance, as provided in the accompanying Excel files). We applied this dataset to test our implementation of Chen's algorithm, and in all cases the results matched perfectly with the expected outputs. This confirms that the algorithm functions correctly not only on small hand-crafted examples but also on large synthetic benchmarks.

Instance number	True TBR known?	True TBR?	True TBR (if known), otherwise cherry bound	Chen's Result	name	taxa	random TBR moves applied
1	TRUE	5	5	5	TREEPAIR_50_5_50_01	50	5
2	TRUE	5	5	5	TREEPAIR_50_5_50_02	50	5
3	TRUE	4	4	4	TREEPAIR_50_5_50_03	50	5
4	TRUE	5	5	5	TREEPAIR_50_5_50_04	50	5
5	TRUE	4	4	4	TREEPAIR_50_5_50_05	50	5
6	TRUE	4	4	4	TREEPAIR_50_5_70_01	50	5
7	TRUE	5	5	5	TREEPAIR_50_5_70_02	50	5
8	TRUE	5	5	5	TREEPAIR_50_5_70_03	50	5
9	TRUE	5	5	5	TREEPAIR_50_5_70_04	50	5
10	TRUE	5	5	5	TREEPAIR_50_5_70_05	50	5
11	TRUE	5	5	5	TREEPAIR_50_5_90_01	50	5
12	TRUE	5	5	5	TREEPAIR_50_5_90_02	50	5
13	TRUE	5	5	5	TREEPAIR_50_5_90_03	50	5
14	TRUE	5	5	5	TREEPAIR_50_5_90_04	50	5
15	TRUE	5	5	5	TREEPAIR_50_5_90_05	50	5
16	TRUE	10	10	10	TREEPAIR_50_10_50_01	50	10
17	TRUE	9	9	9	TREEPAIR_50_10_50_02	50	10
18	TRUE	10	10	10	TREEPAIR_50_10_50_03	50	10
19	TRUE	10	10	10	TREEPAIR_50_10_50_04	50	10
20	TRUE	9	9	9	TREEPAIR_50_10_50_05	50	10
21	TRUE	9	9	9	TREEPAIR_50_10_70_01	50	10
22	TRUE	10	10	10	TREEPAIR_50_10_70_02	50	10
23	TRUE	10	10	10	TREEPAIR_50_10_70_03	50	10
24	TRUE	9	9	9	TREEPAIR_50_10_70_04	50	10
25	TRUE	9	9	9	TREEPAIR_50_10_70_05	50	10
26	TRUE	9	9	9	TREEPAIR_50_10_90_01	50	10
27	TRUE	10	10	10	TREEPAIR_50_10_90_02	50	10
28	TRUE	10	10	10	TREEPAIR_50_10_90_03	50	10
29	TRUE	8	8	8	TREEPAIR_50_10_90_04	50	10
30	TRUE	9	9	9	TREEPAIR_50_10_90_05	50	10
31	TRUE	13	13	13	TREEPAIR_50_15_50_01	50	15
32	TRUE	14	14	14	TREEPAIR_50_15_50_02	50	15

Figure 5: True TBR values and results of Chen's algorithm for the corresponding trees (TBR value equals to k value in the context of MAF.) There is 100 percent match without any exception or error.

5.2.4 Implementation of Split-or-Decompose

The split-or-decompose variant follows the same flow as Chen's algorithm (tidy-up → pick a structure → branch) but adds an extra step to deal with *overlapping components*. Informally, two components overlap if, when mapped back to the working tree, they share a common edge. When such a situation occurs, the conflict is resolved locally by recursively applying the split procedure. If no overlaps are present and the components are disjoint, the algorithm instead applies decomposition. In its current form, decomposition is a relatively simple strategy that attempts to accelerate the search by dividing the instance into independent subproblems.

How recursive splitting works (simple view).

1. **Detect an overlap.** After tidy-up operations, check whether any two forest components share a common edge in the working

tree (by comparing their embeddings on the same leaf set).

2. **Choose an edge.** Select one of the overlapping edges as a witness of the conflict.

3. **Partition leaves by the edge.** The chosen edge naturally divides the leaf set into two subsets, L_{left} and L_{right}.

4. **Split the overlapping component.** Apply recursive splitting to resolve the overlap. This involves cutting edges within one component while keeping the other fixed, producing smaller subcomponents. The process ensures that eventually all components become disjoint. If a solution exists, at least one of the resulting disjoint forests will yield it.

5. **Decompose if possible.** If no overlaps remain, the algorithm applies decomposition: each disjoint component is solved recursively as an independent subproblem. Components that are already homeomorphic to the corresponding restriction of the working tree are accepted immediately, while the others are processed further.

6. **Repeat.** Continue recursively until either a valid agreement forest is found or all possibilities are exhausted.

This keeps the search focused: we only split when an actual overlap blocks an agreement forest, and we recurse on smaller, independent pieces whenever the instance naturally breaks apart. For theoretical background, see the branching rule in (1), for more details of code implementations please check the appendix.

5.2.5 Validation of Split-or-Decompose

The algorithm was validated in the same pragmatic way as Chen's: it produced the correct TBR–k values for every instance in the synthetic dataset used previously to test Chen's algorithm. However, since the Split-or-Decompose method is built upon Chen's algorithm, simply reproducing the correct number of minimum cuts (k) is not sufficient to demonstrate its validity. It is also necessary to verify that the split phase provides a measurable improvement in efficiency.

Frequency distribution of overlaps. As shown in Figure 6 the frequency distribution of the number of overlaps encountered across all test instances, with k values ranging from 0 to 5 because the larger the k it is difficult to depict the histogram. The results confirm that overlaps are a recurring feature in the problem instances, and that their occurrence is not rare but rather common in practical cases.

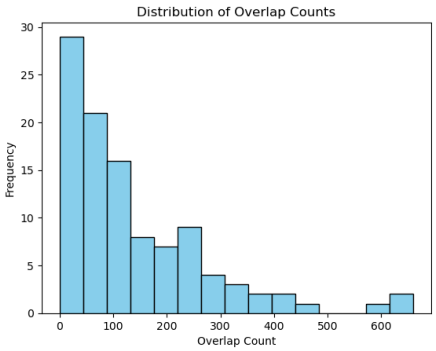

Figure 6: Frequency distribution of overlaps across instances ($k \in [0, 5]$).

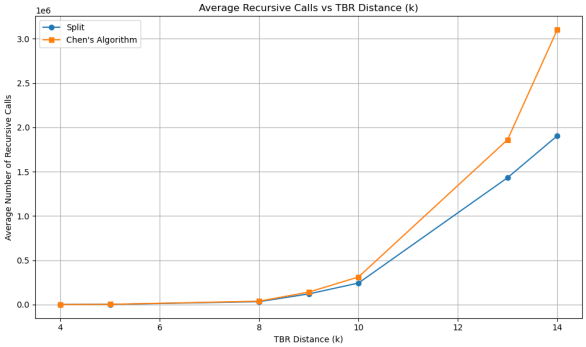

Figure 7: Split without decomposition compared to Chen's algorithm

Importantly, every time an overlap is detected, it is impossible to obtain a valid agreement forest without first making the components disjoint. This means that the split step is not only theoretically necessary but also practically indispensable.

Observing the effect of Split method after implementation. To assess whether the split method functions correctly and improves efficiency, we implemented a variant of the algorithm that applies only the split phase, without decomposition. We then measured the average number of recursive calls performed for varying TBR distances k. For each k value in the dataset, the algorithm was run across multiple test instances, and the average number of recursive calls was recorded for both the baseline Chen algorithm and the split-only variant.

The results, shown in Figure 7, plot the mean number of recursive calls against the TBR distance k. The curve for the split-only variant consistently lies below that of Chen's algorithm, demonstrating that the splitting strategy reduces recursion depth even without decomposition. Furthermore, the results matched the expected outputs exactly in all cases, confirming that the recursive splitting was implemented correctly.

To see the results better here are some instances and results in Table 1.

Table 1: Average number of recursive calls for Chen's Algorithm vs. Chen's Algorithm with only Split, for selected k values.

k	**Only Split**	**Pure Chen**
4	825	828
5	2192	2293
8	32855	37922
9	120131	140717
10	242814	311285
13	1431853	1860651
14	1904808	3101411

Observing the effect of Decomposition method after implementation. To isolate the effect of decomposition, we implemented a variant of the algorithm that applies only the decomposition step without invoking split. This allowed us to measure how decomposition alone impacts performance relative to Chen's algorithm.

For each TBR distance k in the dataset, both algorithms were run on multiple test instances, and the average number of recursive calls was recorded. The results are shown in Figure 8, where the x-axis represents k and the y-axis shows the mean number of recursive calls. The decomposition-only curve consistently lies below Chen's algorithm for most k values, demonstrating that early partitioning into independent subproblems can indeed reduce recursion depth.

However, the performance gain is modest compared to the split-only experiment. This is expected: decomposition does not resolve overlaps between components but merely separates the problem when the instance already divides naturally into disjoint leaf sets. As such, the current decomposition procedure is a basic and rather trivial approach. While it confirms the theoretical correctness and potential of decomposition, it leaves substantial room for future refinement and optimization. In particular, more sophisticated decomposition strategies may yield greater improvements by identifying non-trivial separations earlier in the recursion.

To see the results better here are some instances and results in Table 2.

Dataset check (synthetic instances). Using the synthetic dataset from Steven Kelk's GitHub repository [11], which contains hundreds of instances with provided k (equal to the TBR distance in the accompanying Excel files), split-or-decompose matched Chen's correctness on all tested cases. In addition, on instances with genuine overlaps it reduced the search effort (fewer recursive calls / less time), because splitting removed conflicts before branching.

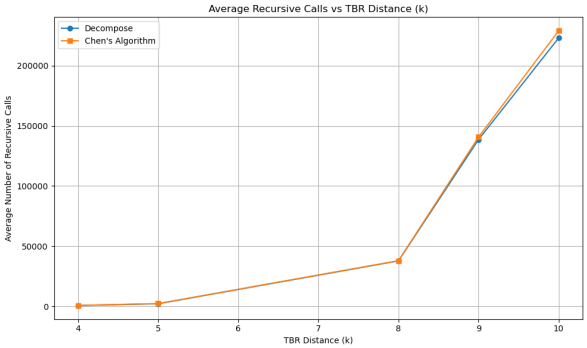

Figure 8: Decomposition without split compared to Chen's algorithm

Table 2: Average number of recursive calls for Chen's Algorithm vs. Chen's Algorithm with only Decomposition, for selected k values.

k	**Only Decomposition**	**Pure Chen**
4	689	781
5	2164	2321
8	37750	37922
9	138605	140716
10	223009	229006

5.3 Software Implementation

The algorithms were implemented in Java. The key components include:

- `PhylogeneticTree`: Represents an unrooted binary tree using adjacency lists.
- `Forest`: A collection of disjoint tree components.
- `SplitOrDecompose`: Implements the new introduced FPT branching strategy.
- `Parser`: It is used to build trees from text files, the trees are always in newick format and Parser class is used to build trees from the provided text file.
- `ChenAlgorithm`: Implements the recursive FPT branching strategy.
- `TreeUtils`: Utility functions for cherry detection, homeomorphism checks, and tree restriction.

6 Experimental Setup

6.1 Datasets

All experiments were conducted on synthetic phylogenetic trees generated using fixed taxa sets of size 50–350 leaves. The datasets were provided by Steven Kelk [11][3]. Trees were generated by starting from identical topologies and applying deliberate topological rearrangements that increase the TBR distance by one per edit, ensuring instances with known minimum k.

Trees have properties such as : Number of taxa (leaves) from 50 to 350, TBR- k values, skew (between 50 and 90, 50 is balanced, 90 is highly skewed).

6.2 Algorithms

We compare two implementations:

- **Chen's algorithm**: the baseline implementation.
- **Split-or-Decompose**: our variant which augments Chen's algorithm with split and decomposition phases.

Both were implemented in Java 17 and run on a standard desktop machine.

6.3 Evaluation Metrics

We measured:

- **Recursive calls**: the total number of recursive invocations for both algorithms Split-Or-Decompose and the Chen's algorithm, as a proxy for computational effort.
- **Split statistics**: *split call rate* (split calls per recursion) and *split commit rate* (successful splits per split call).
- **Decomposition statistics**: analogous call and commit rates.
- **Recursion savings**: the relative reduction in recursive calls compared to Chen's algorithm.

6.4 Experiments

We conducted the following experiments:

1. Baseline comparison of recursive calls between Chen's algorithm and Split-or-Decompose as k increases.
2. Analysis of split usage: how often splits are called and how often they function.

[3] https://github.com/skelk2001/kernelizing-agreement-forests

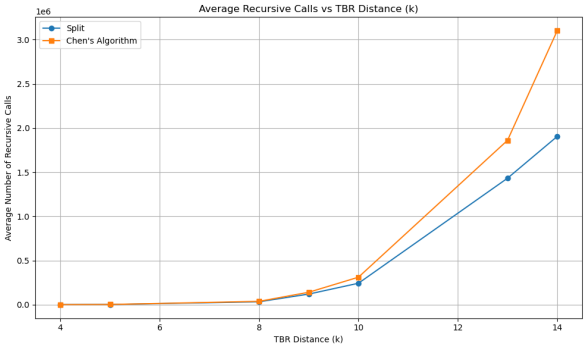

Figure 9: Improvement Over Chen for early k values

3. Correlation between split usage and recursion savings relative to Chen's algorithm.
4. Sensitivity to tree topology, comparing balanced vs. caterpillar trees.

7 Results

7.1 Experiment: Recursive Calls vs. k

Figure 9 compares the number of recursive calls made by Chen's algorithm and the Split-or-Decompose variant as the TBR distance k increases. For small values of k, the two algorithms perform similarly, with only minor differences in the number of recursive calls. This is expected, as when k is small, there are few conflicts between the input trees, and thus little opportunity for the split phase to be invoked.

However, as k increases, the curves diverge sharply. Chen's algorithm requires a rapidly growing number of recursive calls, while the Split-or-Decompose variant grows more slowly. The difference between the two algorithms therefore widens quickly as instances become more difficult.

This effect is quantified in Figure 10, which shows the relative savings of Split-or-Decompose compared to Chen's algorithm. The savings are modest for small k, but increase substantially as k increases. For larger values of k, Split-or-Decompose consistently avoids a significant fraction of recursive calls relative to Chen's algorithm.

Observation. The effectiveness of Split-or-Decompose grows with problem difficulty: while for small k both algorithms behave similarly, for larger k Split-or-Decompose requires substantially fewer recursive calls, confirming its advantage on hard instances.

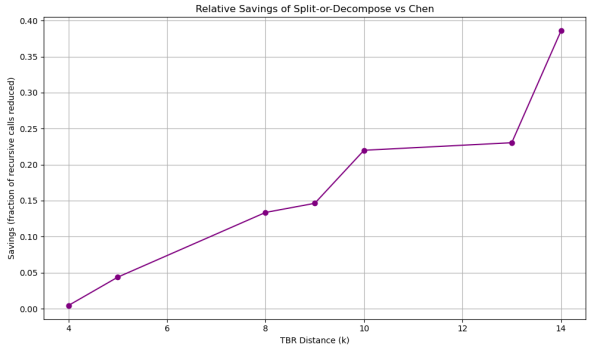

Figure 10: Improvement Over Chen for early k values

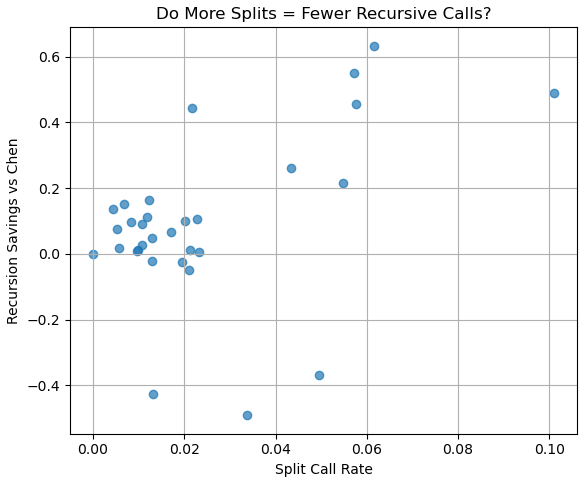

Figure 11: Effectiveness of Splits

7.2 Experiment: Effectiveness of Splits

To evaluate whether invoking the split phase more frequently actually improves performance, we measured the correlation between the *split call rate* (number of split calls per recursive call) and the *recursion savings* relative to Chen's algorithm.

Figure 11 shows a scatter plot of split call rate against recursion savings. Each point represents a test instance. A positive correlation is observed (Pearson correlation ≈ 0.47), indicating that instances where the split phase is triggered more often tend to achieve greater reductions in recursive calls compared to Chen's algorithm.

The relationship is not perfect: some points lie close to zero or even show negative savings, meaning that additional split calls did not always translate into an improvement. However, the overall trend demonstrates that higher split usage is generally associated with better performance.

Observation. While not every split contributes positively, on average there is a moderate positive correlation between split usage and recursion savings. This confirms that the split phase is beneficial in practice, especially on more complex instances.

7.3 Experiment: What Triggers the Split Phase?

We next investigated on which types of instances the split phase is most often invoked and whether this translates into performance improvements. For this purpose, we analyzed split call rate, split commit rate, and recursion savings relative to Chen's algorithm across different values of k.

The split call rate increases with k, rising from 1.3% at $k = 4$ to over 4% at $k = 10$. This suggests that more difficult instances create more opportunities for the split phase to be invoked. However, the split commit rate remains very low (below 0.3% for most values of k), indicating that only a small fraction of split attempts actually restructure the problem.

Despite this, the effect on performance is clear: recursion savings compared to Chen's algorithm grow with k, from about 7.7% at $k = 4$ to over 24% at $k = 10$. The figures confirm this trend. Figure 12 shows that split call rate increases with k, while Figure 13 illustrates that recursion savings are consistently larger on harder instances.

Observation. The split phase is more likely to be triggered on instances with larger k, where conflicts between trees are more complex. Although split commits are rare, the few successful cases yield substantial savings, making the split phase especially valuable for harder problems.

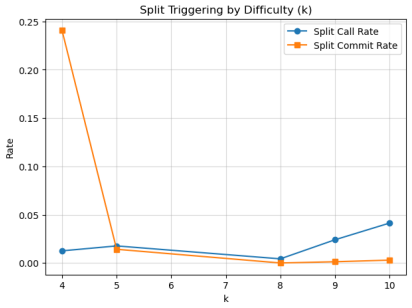

Figure 12: Split call and commit rates as a function of k. The call rate increases with k, but commit rates remain low.

7.4 Experiment: Split Triggering and Effectiveness

We investigated two aspects of the split phase: (i) how often the split phase is triggered during recursion, and (ii) when triggered, how often it actually

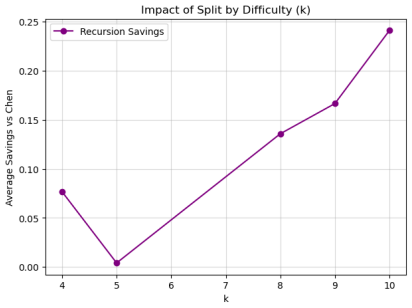

Figure 13: Recursion savings of Split-or-Decompose compared to Chen's algorithm as a function of k. Larger values of k yield consistently greater savings.

reduces the search space compared to Chen's algorithm.

Across $k = 5$ to 13, the split phase was triggered in only a small fraction of calls, with an average split call rate of 2.45%. When triggered, the split phase rarely committed, with an average split commit rate of just 3.03%. In other words, the split phase was considered relatively infrequently, and only a minority of these attempts led to an actual restructuring of the problem.

Nevertheless, the overall impact on performance was non-trivial. On average, Split-or-Decompose achieved a recursion savings of 9.36% compared to Chen's algorithm. This demonstrates that even though the split phase is invoked sparingly and only rarely commits, those few successful invocations are sufficient to noticeably reduce recursion effort.

Observation. Split is triggered infrequently and commits even more rarely, but when it does, the effect is significant enough to provide a clear reduction in recursion compared to Chen's algorithm.

7.5 Experiment: Decomposition Triggering and Effectiveness

The experiment split triggering and effectiveness was replicated for the decomposition phase. Specifically, I measured the *decomposition call rate* (decomposition calls per recursion) and the *decomposition commit rate* (committed decompositions per decomposition call) across values of k, and examined how these relate to the overall recursion savings relative to Chen's algorithm.

It is observed that the decomposition call rate varies with k, while the commit rate remains low overall. As with the split phase, this indicates that only a small fraction of attempted decompositions actually restructure the instance.

Observation. Although successful decompositions are rare, they contribute disproportionately to the reduction in recursion. Together with the split results, this suggests that the two phases are most beneficial on harder instances (larger k) where conflicts can be isolated and simplified.

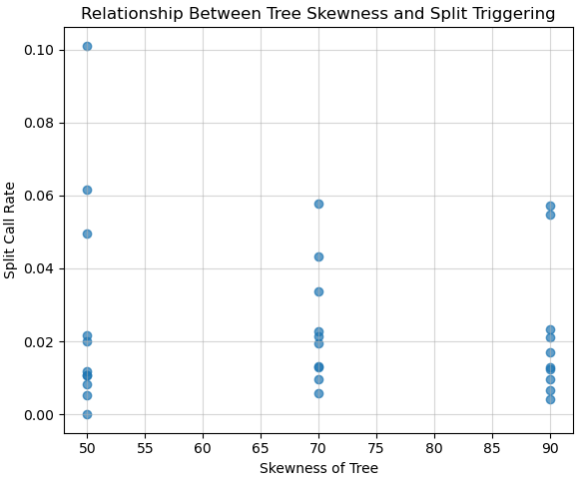

Figure 14: Relationship between Skewness and amount of Split Trigger

7.6 Experiment: Skewness and Split Triggering

We investigated whether tree imbalance, measured by skewness, affects the likelihood of triggering the split phase. Specifically, we computed the correlation between skewness and the split call rate (split calls per recursion).

The correlation was weakly negative ($r \approx -0.10$), indicating that tree skewness has almost no effect on how often the split phase is invoked. Figure 14 shows the scatter plot of skewness versus split call rate, confirming the absence of a strong relationship.

Observation. Tree skewness does not appear to drive split usage; instances with higher or lower skewness trigger the split phase at roughly similar rates.

7.7 Experiment: Skewness and Chen's Recursion

We also examined whether skewness impacts the number of recursive calls required by Chen's algorithm. The correlation between skewness and Chen's recursion counts was again very weakly negative ($r \approx -0.09$).

This suggests that the overall balance of a tree has little influence on Chen's algorithmic complexity see ; Figure 15. Instead, performance is domi-

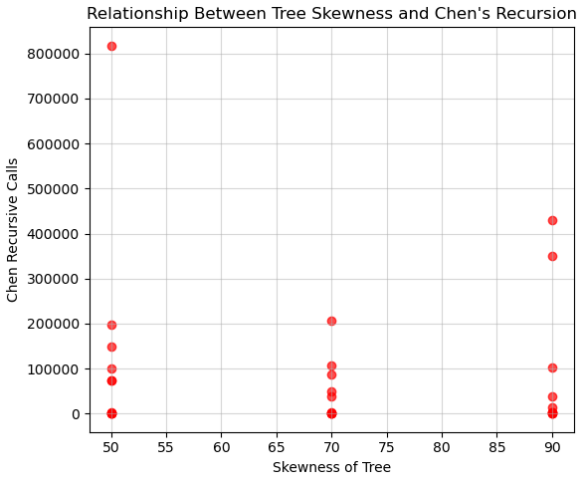

Figure 15: Relationship between Skewness and amount of Chen Recursive Call

nated by the number of cuts (k) and the degree of topological conflict between trees.

Observation. Skewness has no measurable impact on the recursion counts of Chen's algorithm, reinforcing that problem difficulty is primarily determined by k rather than tree shape.

7.8 Experiment: Effect of Taxa Count

To investigate whether the size of the input trees influences algorithm behavior, we extended our experiments to record the number of taxa (leaves) alongside the recursive call counts and split usage.

We computed the correlation between taxa count and three quantities: the number of split calls, the number of recursive calls in Split-or-Decompose, and the number of recursive calls in Chen's algorithm. The results were as follows:

- Taxa vs. split calls: $r \approx -0.003$
- Taxa vs. Split-or-Decompose recursions: $r \approx 0.18$
- Taxa vs. Chen recursions: $r \approx 0.15$

The first result shows that taxa count has almost no influence on whether the split phase is triggered: larger and smaller trees are equally likely to invoke splits. The second and third results indicate that recursion counts in both algorithms increase only weakly with taxa size. The effect is slightly stronger in the Split-or-Decompose variant, but remains small overall.

Observation. The number of taxa (tree size) is not a strong predictor of algorithmic behavior. Split usage is unaffected, and recursion counts increase only weakly with taxa count. This suggests that the primary driver of complexity is not tree size itself, but the level of conflict between trees as measured by k.

7.9 Experiment: Growth Extrapolation of Recursive Calls

To estimate how both algorithms scale for larger values of k, we fitted a quadratic model to the logarithm of the observed recursive call counts for both Split-or-Decompose and Chen's algorithm. Using this model, we extrapolated the number of recursive calls up to $k = 20$.

The predictions are shown in Figure 16, along with the observed data points for $k = 10, 13, 14$. While both algorithms exhibit exponential growth in the number of recursive calls as k increases, Chen's algorithm grows much faster.

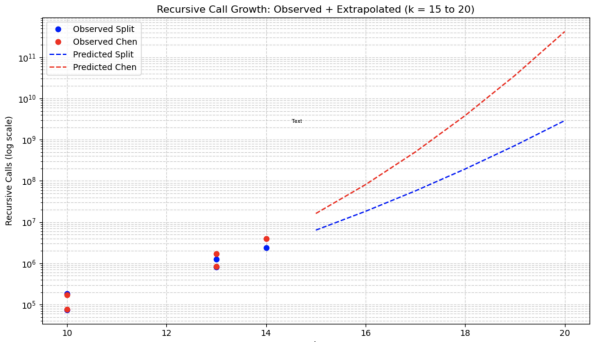

Figure 16: Observed and extrapolated recursive calls for Chen's algorithm and Split-or-Decompose. Both algorithms exhibit exponential growth, but Chen's algorithm grows much faster, with the gap widening substantially for larger k.

Observation. For $k = 15$, Chen requires only about 2.5 times as many recursive calls as Split-or-Decompose, but this factor increases to over 144 by $k = 20$. This widening gap highlights that the benefit of Split-or-Decompose is not only present at moderate values of k, but is expected to grow dramatically for larger and more difficult instances.

8 Discussion

The experimental evaluation demonstrates several important findings. First, the Split-or-Decompose algorithm consistently reduces recursion compared to Chen's algorithm, with the advantage growing substantially as the number of required cuts k increases. This confirms the intuition that split and decomposition are most valuable in harder instances,

where conflicts between trees are localized and can be separated efficiently.

Second, we observed that the split phase is rarely triggered and even more rarely commits. Nonetheless, the few successful splits account for a significant fraction of the savings, demonstrating that the quality of a split is more important than its frequency. This suggests that further improvements may be obtained by developing heuristics to better identify when a split will be effective.

Third, the analysis of skewness and taxa count indicates that tree shape and size have little impact on algorithm performance. Instead, the primary driver of complexity is the conflict structure between trees, captured by k. This finding aligns with the theoretical role of k as a parameter in fixed-parameter algorithms for maximum agreement forests.

Finally, extrapolation experiments suggest that the performance gap between Split-or-Decompose and Chen's algorithm will grow dramatically for larger k. While both algorithms exhibit exponential growth in recursive calls, the slower growth rate of Split-or-Decompose makes it a promising candidate for tackling more challenging instances in practice.

Summary. Overall, the experiments confirm that incorporating split and decomposition phases into Chen's algorithm provides a clear practical benefit, especially on difficult instances. At the same time, the rarity of successful splits highlights an opportunity for future work in designing more effective split selection strategies.

Limitations include the size of datasets and computational expensiveness of larger k values. **The current study is focused on unrooted trees only**.

9 Conclusion

This thesis implemented and evaluated the split-or-decompose strategy for solving the MAF problem. Compared to classical approaches, it consistently showed better performance in branching depth. These results confirm that handling overlaps explicitly through branching rules can lead to significant gains in fixed-parameter algorithms.

10 Future Work

Future studies could:

- Optimize the runtime for the algorithms
- Improve the decomposition step
- Extend the implementation to rooted phylogenetic trees
- Evaluate performance on biological datasets (e.g., real genome-based trees)

References

[1] David Mestel, Steven Chaplick, Steven Kelk, and Ruben Meuwese. *Split-or-decompose: Improved FPT branching algorithms for maximum agreement forests.* arXiv:2409.18634, 2024. `https://arxiv.org/abs/2409.18634`.

[2] Chris Whidden and Norbert Zeh. Fast FPT algorithms for computing rooted agreement forests: theory and experiments. In *Proc. SODA*, pages 1161–1176. SIAM, 2010.

[3] Chris Whidden, Robert G. Beiko, and Norbert Zeh. Fixed-parameter algorithms for maximum agreement forests. *SIAM Journal on Computing*, 42(4):1431–1466, 2013. doi:10.1137/110845045.

[4] Jianer Chen, Mingyu Li, Jianxin Liu, and Jiong Wang. An improved parameterized algorithm for the Maximum Agreement Forest problem. *Theoretical Computer Science*, 607:240–252, 2015. doi:10.1016/j.tcs.2015.09.017.

[5] Estela M. Rodrigues, Marie-France Sagot, and Yoshiko Wakabayashi. The Maximum Agreement Forest problem: Approximation algorithms and computational experiments. *Theoretical Computer Science*, 374(1–3):91–110, 2007. doi:10.1016/j.tcs.2006.12.011.

[6] Leo van Iersel, Steven Kelk, Nela Lekic, and Leen Stougie. Approximation algorithms for nonbinary agreement forests. *SIAM Journal on Discrete Mathematics*, 28(1):49–66, 2014. doi:10.1137/120903567.

[7] Neil Olver, Frans Schalekamp, Suzanne van der Ster, Leen Stougie, and Anke van Zuylen. A duality-based 2-approximation algorithm for maximum agreement forest. *Mathematical Programming*, 198(1):811–853, 2023. doi:10.1007/s10107-022-01790-y.

[8] Benjamin L. Allen and Mike Steel. Subtree transfer operations and their induced metrics on evolutionary trees. *Annals of Combinatorics*, 5(1):1–15, 2001. doi:10.1007/s00026-001-8006-8.

[9] Magnus Bordewich and Charles Semple. The computational complexity of the maximum agreement forest problem. *Journal of Graph Theory*, 54(2):143–156, 2006. (often cited as 2005 online first). doi:10.1002/jgt.20137.

[10] Chris Whidden. rSPR: computing rooted SPR distances and rooted agreement forests (software). GitHub repository, 2015. https://github.com/cwhidden/rspr. Accessed: 2025-08-20.

[11] Steven Kelk. Kernelizing Agreement Forests (dataset and code). GitHub repository, 2025. https://github.com/skelk2001/kernelizing-agreement-forests. Accessed: 2025-08-13.

A Appendix A: Algorithm Pseudocode

Also check the Java Implementation to understand better. The code is well structured and explained.

Algorithm 1 Chen's Algorithm (unrooted MAF)

Function `CHEN`(T, F', k)

```
// --- Reductions to fixed point ---
repeat
    changed ← false
    changed ← suppressDegree2(T) ∨ changed
    foreach component C in F' do
        changed ← suppressDegree2(C) ∨ changed
    changed ← removeSingletons(F', T) ∨ changed
    changed ← reduceCommonCherries(F', T) ∨ changed
until not changed
if k < 0 then
    return NO
if isAgreementForest(T, F') then
    return YES
cherries ← findCherries(T)
if cherries = ∅ then
    return NO
pick (a, b) from cherries
if a and b lie in different components of F' then
    return CHEN(T, cutOff(F', a), k − 1) ∨ CHEN(T, cutOff(F', b), k − 1)
else
    return CHEN(T, cutOff(F', a), k − 1) ∨ CHEN(T, cutOff(F', b), k − 1) ∨ SPLIT_PATH_CHEN(T, F', a, b, k)
```

Algorithm 2 Split path subroutine for Chen

Function `SPLIT_PATH_CHEN`(T, F', a, b, k)

```
let C be the component of F' containing both a and b
P ← unique path in C between a and b
S ← set of side edges incident to nodes on P that leave P
ans ← NO
foreach choice s ∈ S to preserve do
    F'' ← deep copy of F'
    C'' ← corresponding copy of C in F''
    cut in C'' every edge in S \ {s}
    cuts ← |S| − 1
    ans ← ans ∨ CHEN(T, F'', k − cuts)
return ans
```

Algorithm 3 Split-or-Decompose (driver)

Function `SOD`(T, F', k)

```
return SOD_INNER(T, F', k, allowDecompose=true)
```

Algorithm 4 Split-or-Decompose (inner)

Function `SOD_INNER`$(T, F', k,$ *allowDecompose*$)$

```
// --- Reductions to fixed point ---
repeat
    changed ← false
    changed ← suppressDegree2(T) ∨ changed
    foreach component C in F' do
        changed ← suppressDegree2(C) ∨ changed
    changed ← removeSingletons(F', T) ∨ changed
    changed ← reduceCommonCherries(F', T) ∨ changed
until not changed
if k < 0 then
    return NO
if F' = ∅ or isAgreementForest(T, F') then
    return YES
// --- Split phase if any overlap exists in T ---
if overlapExistsInT(T, F') then
    pick an overlapping original edge (u, v) in T
    (Y, Z) ← edgeBipartition(T, (u, v))
    // Y and Z disjoint, nonempty
    return RECURSIVE_SPLIT(C_Y, F' \ {C_Y}, Y, Z, k, T) ∨ RECURSIVE_SPLIT(C_Z, F' \ {C_Z}, Z, Y, k, T)
        // C_Y (resp. C_Z) is a component chosen to be split to respect Y|Z
// --- Decomposition if forest is disjoint in T and allowed ---
if allowDecompose then
    return DECOMPOSE(T, F', k)
return NO
```

Algorithm 5 Split phase with recursive splitting

Function `RECURSIVE_SPLIT`$(C, F_{\text{rest}}, Y, Z, k, T)$
- **if** $k = 0$ **then**
  - **return** NO
- $C \leftarrow$ `suppressDegree2`(C)
- **if** `isAgreementForest`$(T, F_{\text{rest}} \cup \{C\})$ **then**
  - **return** YES
- **if** $|E(C)| = 1$ **then**
  - split the single edge of C into (C_1, C_2) **return** `SOD_INNER`$(T, F_{\text{rest}} \cup \{C_1, C_2\}, k-1,$ *allowDecompose=false*$)$
- **foreach** *internal node* c *in* C **do**
  - compute for each neighbor w of c: $L(w) =$ leaves reachable from w without crossing c
  - $Y_{\text{only}} \leftarrow \{w \mid L(w) \subseteq Y\}$ $Z_{\text{only}} \leftarrow \{w \mid L(w) \subseteq Z\}$ $M \leftarrow \{w \mid L(w) \cap Y \neq \emptyset \wedge L(w) \cap Z \neq \emptyset\}$
  - **if** $Y_{\text{only}} \neq \emptyset$ ***and*** $Z_{\text{only}} \neq \emptyset$ ***and*** $M \neq \emptyset$ **then**
    - **foreach** $w \in Y_{\text{only}}$ **do**
      - split C on edge (c, w) into (C_1, C_2)
      - **if** `RECURSIVE_SPLIT`$(C_2, F_{\text{rest}} \cup \{C_1\}, Y, Z, k-1, T) = YES$ **then**
        - **return** YES
    - **foreach** $w \in Z_{\text{only}}$ **do**
      - split C on edge (c, w) into (C_1, C_2)
      - **if** `RECURSIVE_SPLIT`$(C_2, F_{\text{rest}} \cup \{C_1\}, Y, Z, k-1, T) = YES$ **then**
        - **return** YES
  - **else if** $Y_{\text{only}} \neq \emptyset$ ***and*** $Z_{\text{only}} \neq \emptyset$ ***and*** $M = \emptyset$ **then**
    - pick w from the smaller of Y_{only} or Z_{only} split C on edge (c, w) into (C_1, C_2)
    - **if** `SOD_INNER`$(T, F_{\text{rest}} \cup \{C_1, C_2\}, k-1,$ *allowDecompose=false*$) = YES$ **then**
      - **return** YES
- **return** NO

Algorithm 6 Decomposition phase for disjoint forest

Function `DECOMPOSE`(T, F', k)
- **if** $k < 0$ **then**
  - **return** NO
- **if** $F' = \emptyset$ **then**
  - **return** YES
- $m \leftarrow |F'|$ $a \leftarrow \min\{3, \lfloor k/m \rfloor\}$ `// initial per-component budget`
- spent $\leftarrow 0$ $U \leftarrow \emptyset$ `// unsolved components`
- **foreach** *component* C *in* F' **do**
  - $E \leftarrow$ `restrict`$(T, \text{leaves}(C))$ ok $\leftarrow$ false
  - **for** $j \leftarrow 0$ **to** a **do**
    - **if** `CHEN`$(E, \{C\}, j) = YES$ **then**
      - ok $\leftarrow$ true; spent $\leftarrow$ spent $+j$;
      - **break**
  - **if** ***not*** *ok* **then**
    - $U \leftarrow U \cup \{C\}$
- **if** $|U| = m$ **then**
  - **return** NO
- **if** $U = \emptyset$ **then**
  - **return** YES
- **return** `DECOMPOSE`$(T, U, k - spent)$

Algorithm 7 Common helper predicates

Function `isAgreementForest`(T, F')
- **foreach** *component* C *in* F' **do**
  - $E \leftarrow$ `restrict`$(T, \text{leaves}(C))$ **if** ***not*** `homeomorphic`(E, C) **then**
    - **return** false
- **return** true

Function `overlapExistsInT`(T, F')
- `// true iff two components' embeddings share an original edge of` T
- compute embeddings of all $C \in F'$ into T; **return** true if any original edge is used by ≥ 2 embeddings

Function `edgeBipartition`$(T, (u, v))$
- $Y \leftarrow$ leaves reachable from u without crossing v
- $Z \leftarrow$ leaves reachable from v without crossing u **return** (Y, Z)

B Appendix B: Dataset Samples

Sample synthetic trees used for evaluation:

- Example Small Tree in newick format: ((((6,5),(1,2)),((4,8),10)