Compilation-Based Explainability of Tree Ensembles

Dissertation

zur Erlangung des Grades eines

D o k t o r s d e r I n g e n i e u r w i s s e n s c h a f t e n

der Technischen Universität Dortmund
an der Fakultät für Informatik

von

Alnis Murtovi

Dortmund
2025

Tag der mündlichen Prüfung:
04.04.2025

Dekan:
Prof. Dr. Jens Teubner

Gutachter:
Prof. Dr. Bernhard Steffen
Prof. Dr. Nils Jansen

Acknowledgements

This dissertation has been shaped by the support of many wonderful people I've had the privilege of working with along the way. Their contributions have been instrumental, and I'm truly grateful for their lasting impact.

First, I would like to express my sincere appreciation to my PhD advisor, Bernhard Steffen, for his constant support, mentorship, and encouragement throughout this journey. Over the years, I have greatly benefited from your expertise and guidance. You were always open to discussions at any time and on any topic, and your insights have profoundly shaped not only this thesis but also my development as a researcher. Although some of our discussions were challenging, they reflected your dedication to helping me succeed, and for that I am truly grateful.

I would also like to extend my heartfelt thanks to my second examiner, Nils Jansen, for kindly agreeing to evaluate this dissertation on short notice. I am also grateful to Jakob Rehof for chairing my defense and to Ben Hermann for serving alongside him on the doctoral committee. Your time and effort are greatly appreciated.

During my PhD, I had the privilege of interning at Lawrence Livermore National Laboratory. I am deeply grateful to Giorgis Georgakoudis, Marc Jasper, and Konstantinos Parasyris for their mentorship, collaboration, and inspiring discussions. I also had the invaluable opportunity to intern at Meta, where I thank Horace He for his guidance, support, and for fostering an environment that encouraged meaningful contributions. Both experiences were instrumental in shaping my growth as a researcher and broadened my technical and professional perspective.

I have spent nearly nine years at the Chair of Programming Systems, starting already during my bachelor studies, and it has been a truly formative part of my academic journey. Over the years, I had the pleasure of working with many wonderful colleagues and made lasting friendships along the way. Rather than listing names and risking leaving someone out, I'll simply say thank you to everyone I shared this time with. I'll fondly remember not just the work, but also the countless conversations, laughs, and lunches at the university cafeteria.

Finally, I am deeply grateful to my parents for their unconditional love, encouragement, and patience. Your support has been the foundation of everything I've achieved, and I especially thank you for standing by me during the more stressful moments.

Abstract

Machine learning, particularly deep learning, has achieved remarkable success in areas such as image recognition, natural language processing, and speech recognition. However, for structured or tabular data, tree ensemble approaches, such as random forests and gradient boosted trees, often outperform deep learning-based approaches. Although they perform strongly, tree ensembles are in general considered to be black boxes, because the complexity of combining multiple decision trees makes it difficult to trace the reasoning behind individual predictions. Recent advancements in Explainable Artificial Intelligence have presented heuristic post-hoc explanation methods like LIME and SHAP. Nevertheless, these methods often rely on a model's input-output behavior and therefore only approximate how it internally arrives at its predictions. As a result, there is an urgent need for efficient and precise approaches to explaining ensembles, especially in domains where safety or fairness is critical.

To tackle the interpretability gap, this thesis explores compilation-based techniques that transform an entire tree ensemble into a single, semantically equivalent structure such as a directed acyclic graph. This single graph representation reveals the ensemble's underlying logic, making it amenable to formal analysis. Once compiled, the model's internal logic becomes more transparent, allowing efficient generation of formal explanations, as well as support for verification tasks such as pre- and postcondition checks and model equivalence checking. One significant advantage is that, after the one-time cost of building the unified representation, subsequent explanations can be generated efficiently. This makes the proposed solutions particularly well-suited for real-time or interactive settings where many explanations are requested in sequence.

The main focus of this thesis is efficiency and scalability. Existing compilation-based approaches can be very expensive on large ensembles. To address this, the thesis introduces novel compilation algorithms and optimizations, significantly reducing transformation time and memory usage while maintaining exact equivalence with the original tree ensemble. The experimental results show over an order of magnitude speedup in model compilation and multiple orders of magnitude in explanation generation compared to state-of-the-art solver-based approaches.

Furthermore, the thesis introduces a user-friendly, web-based tool (*Forest GUMP*) that allows non-experts to train, visualize, verify, and explain tree ensembles interactively. Overall, these contributions advance the field of explainable AI by delivering efficient, formally grounded, and practical solutions for tree ensemble interpretability and explainability.

Attached Publications

Parts of this dissertation have already been published in collaboration with other researchers. They are listed here with accompanying comments on my participation and will be cited distinctly throughout the main body of this thesis to clearly differentiate them from other referenced literature.

I Alnis Murtovi, Alexander Bainczyk, and Bernhard Steffen. **Forest GUMP: A Tool for Explanation**. In: *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part II*. vol. 13244. Lecture Notes in Computer Science. Springer, 2022, pp. 314–331. DOI: 10.1007/978-3-030-99527-0_17

Below cited as [1]$_{\text{AP}}$.

The presented concepts were discussed among all authors. I was the main author of this publication. The paper is accompanied by a software artifact that was jointly developed by Alexander Bainczyk and me.

II Alnis Murtovi, Alexander Bainczyk, Gerrit Nolte, Maximilian Schlüter, and Bernhard Steffen. **Forest GUMP: a tool for verification and explanation**. In: *Int. J. Softw. Tools Technol. Transf.* 25.3 (2023), pp. 287–299. DOI: 10.1007/S10009-023-00702-5

Below cited as [2]$_{\text{AP}}$.

The presented concepts were discussed among all authors. I was the main author of this publication. The paper is accompanied by a software artifact that was jointly developed by Alexander Bainczyk and me.

III Alnis Murtovi, Maximilian Schlüter, and Bernhard Steffen. **Computing Inflated Explanations for Boosted Trees: A Compilation-Based Approach**. In: *The Combined Power of Research, Education, and Dissemination - Essays Dedicated to Tiziana Margaria on the Occasion of Her 60th Birthday*. Vol. 15240. Lecture Notes in Computer Science. Springer, 2025, pp. 183–201. DOI: 10.1007/978-3-031-73887-6_14

Below cited as [3]$_{\text{AP}}$.

The presented concepts were discussed among all authors. I was the main author of this publication and responsible for its implementation and evaluation, except for the introduction (Section 1).

IV Alnis Murtovi, Maximilian Schlüter, and Bernhard Steffen. **Voting-Based Shortcuts through Random Forests for Obtaining Explainable Models**. In: *Real Time and Such - Essays Dedicated to Wang Yi to Celebrate His Scientific Career*. Vol. 15230. Lecture Notes in Computer Science. Springer, 2025, pp. 135–153. DOI: 10.1007/978-3-031-73751-0_11

Below cited as [4]$_{\text{AP}}$.

The presented concepts were discussed among all authors. I was the main author of Sections 5, 6, and 7 and responsible for the implementation and evaluation. I co-authored the remaining sections.

V Alnis Murtovi, Maximilian Schlüter, and Bernhard Steffen. **An Efficient Compilation-Based Approach to Explaining Random Forests Through Decision Trees**. In: *Proceedings of the 17th International Conference on Agents and Artificial Intelligence, ICAART 2025 - Volume 2, Porto, Portugal, February 23-25, 2025*. SCITEPRESS, 2025, pp. 484–495. DOI: 10.5220/0013188600003890

Below cited as [5]$_{\text{AP}}$.

The presented concepts were discussed among all authors. I was the main author of this publication and responsible for its implementation and evaluation.

VI Alnis Murtovi, Giorgis Georgakoudis, Konstantinos Parasyris, Chunhua Liao, Ignacio Laguna, and Bernhard Steffen. **Enhancing Performance Through Control-Flow Unmerging and Loop Unrolling on GPUs**. In: *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2024, Edinburgh, United Kingdom, March 2-6, 2024*. IEEE, 2024, pp. 106–118. DOI: 10.1109/CGO57630.2024.10444819

Below cited as [6]$_{\text{AP}}$.

The presented concepts were discussed among all authors. I was the main author of this publication and responsible for its implementation and evaluation.

Contents

1 Introduction 1
- 1.1 My Contributions . . . 4
- 1.2 Context of Attached Publications . . . 6
- 1.3 Organization of this Dissertation . . . 8

2 Background 9
- 2.1 Classification Problems . . . 9
- 2.2 Tree Ensembles . . . 9
  - 2.2.1 Decision Trees . . . 9
  - 2.2.2 Random Forests . . . 10
  - 2.2.3 Gradient Boosted Trees . . . 10
- 2.3 Logic-based Explanations . . . 12
- 2.4 Decision Diagrams . . . 13
  - 2.4.1 Binary Decision Diagrams . . . 13
  - 2.4.2 Algebraic Decision Diagrams . . . 14
  - 2.4.3 Predicate-based Decision Diagrams . . . 15

3 Transforming Tree Ensembles to Algebraic Decision Diagrams 17
- 3.1 Transforming Random Forests into Algebraic Decision Diagrams . . . 17
  - 3.1.1 Infeasible Path Elimination . . . 19
  - 3.1.2 Class Characterizations . . . 20
  - 3.1.3 End-to-End Implementation . . . 22
- 3.2 Early Stopping for Random Forests . . . 22
  - 3.2.1 Stochastic Problem Description . . . 24
  - 3.2.2 Semantics-Preserving Early Stopping . . . 25
  - 3.2.3 Non-Semantics-Preserving Early Stopping . . . 26
  - 3.2.4 Abstract Early Stopping . . . 29
- 3.3 Evaluation: Transforming Random Forests into ADDs . . . 32
  - 3.3.1 Experimentation Results . . . 32
- 3.4 Transforming Gradient Boosted Trees into Algebraic Decision Diagrams . . . 36
  - 3.4.1 Binary Classification . . . 36
  - 3.4.2 Multiclass Classification . . . 39
- 3.5 Early Stopping for Gradient Boosted Trees . . . 40
  - 3.5.1 Early Stopping for Binary Classification . . . 41
  - 3.5.2 Abstract Early Stopping for Binary Classification . . . 42
  - 3.5.3 Early Stopping for Multiclass Classification . . . 43
- 3.6 Evaluation: Transforming Gradient Boosted Trees into ADDs . . . 44

4 Transforming Tree Ensembles into Decision Trees 47
- 4.1 Transforming Random Forests to Decision Trees . . . 48
  - 4.1.1 Baseline Transformation Method . . . 48
  - 4.1.2 Optimizations . . . 50
  - 4.1.3 Using a DAG . . . 50

4.1.4 Infeasible Path Elimination . . . 51
4.1.5 Early Stopping . . . 51
4.1.6 Abstract Early Stopping Heuristic . . . 52
4.1.7 Tree Order Heuristic . . . 52
4.1.8 Tree Simplification . . . 53
4.1.9 Optimized Approach . . . 55
4.2 Evaluation: Transforming Random Forests into Decision Trees . . . 56
4.2.1 Experimentation Results . . . 56
4.3 Transforming Gradient Boosted Trees into Decision Trees . . . 59
4.3.1 Binary Classification . . . 59
4.3.2 Binary Classification with Early Stopping . . . 61
4.3.3 Multiclass Classification . . . 62
4.3.4 Score Decision Trees . . . 63
4.3.5 Majority Vote Decision Trees . . . 64
4.3.6 Multiclass Early Stopping . . . 65
4.3.7 Path-Sensitive Multiclass Early Stopping . . . 67
4.4 Evaluation: Transforming Gradient Boosted Trees into Decision Trees . . . 69
4.4.1 Experimental Setup . . . 70
4.4.2 Experimentation Results . . . 70

5 Analyzing Tree Ensembles **75**
5.1 Explaining Tree Ensemble Decisions . . . 75
5.1.1 Generating Abductive Explanations . . . 75
5.1.2 Generating Inflated Explanations . . . 76
5.2 Evaluation: Generating Abductive and Inflated Explanations for Tree Ensembles 77
5.2.1 Generating Explanations for Random Forests . . . 78
5.2.2 Generating Explanations for Gradient Boosted Trees . . . 84
5.3 Verification of Tree Ensembles . . . 88
5.3.1 Formulating Verification Queries . . . 88
5.3.2 Verifying Queries via Compilation-Based Approaches . . . 89
5.3.3 Verifying Queries via Compilation-Based Approaches for Multiple Queries 89
5.3.4 Verifying Equivalence of Tree Ensembles . . . 90

6 Forest GUMP: A Tool For Explainability **93**
6.1 Overview of Forest GUMP . . . 93
6.2 Forest GUMP in Action . . . 94
6.2.1 Learning a Random Forest . . . 95
6.2.2 Model Explanation . . . 96
6.2.3 Class Characterization . . . 96
6.2.4 Outcome Explanation Problem . . . 97
6.2.5 Verification . . . 98

7 Related Work **99**
7.1 Heuristic Explainability Approaches . . . 99
7.2 Logic-based Explainability Approaches . . . 99
7.3 Compilation-based Explainability Approaches . . . 100
7.4 Tree Ensemble Verification . . . 101
7.5 Visualization Techniques . . . 101

8 Conclusion and Future Work 103
8.1 Future Work 104

References 109

Online References 119

A Abductive Explanation Sizes 121
A.1 Random Forests 121
A.2 Gradient Boosted Trees 121

B Class Characterization Sizes 125
B.1 Random Forests 125
B.2 Gradient Boosted Trees 126

C Transforming Gradient Boosted Trees into Algebraic Decision Diagrams 129

Chapter 1

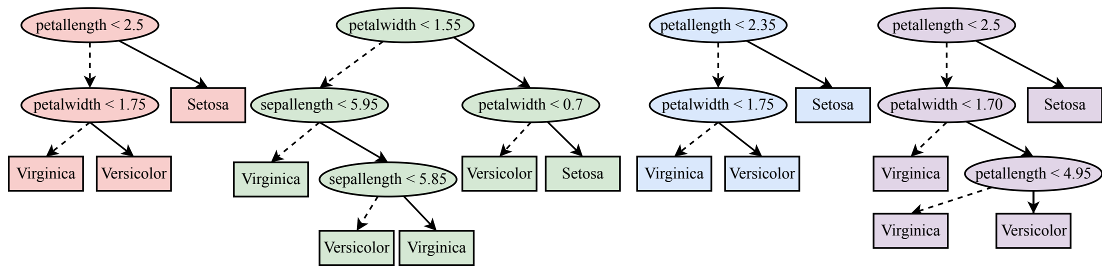
Introduction

Machine learning has seen significant growth in recent years, with deep learning models [7] drawing much of the attention due to their successes in tasks like image recognition [8], natural language processing [9], and speech recognition [10]. Yet, for structured or tabular data, tree ensemble methods like random forests [11] and gradient boosted trees [12] frequently outperform other approaches [13, 14, 15]. This trend is reflected on Kaggle [16], a leading platform for data science competitions, where gradient boosting libraries like XGBoost [17] frequently power many of the top-ranked solutions [18]. By aggregating the predictions of multiple decision trees, tree ensembles generally offer robust predictive performance [19]. However, despite each individual decision tree's interpretability, a forest or boosted ensemble of trees is typically perceived as a black-box model, raising concerns about transparency and trustworthiness in safety-critical domains [20, 21, 22].

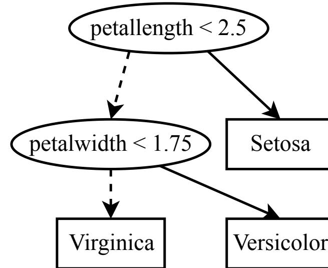
In machine learning, a black-box model refers to a model or system where the inputs and outputs are observable but the internal mechanisms that lead to the outputs remain opaque [21, 22]. Although these models achieve high accuracy and handle complex problems effectively, their lack of interpretability raises concerns, especially in critical applications where accountability and fairness are important [23, 24]. In healthcare, for example, a black-box model might predict a patient's likelihood of developing a disease with high accuracy, but without understanding the reasons behind the prediction, doctors cannot fully trust the model or use its insights to guide treatment [25, 26]. Similarly, in finance, a black-box model used to assess creditworthiness might deny a loan application without revealing the reasons behind the decision, potentially leading to unfair outcomes or regulatory violations [27, 28]. These examples highlight the importance of interpretability to ensure that such models are deployed responsibly and ethically [29, 30].

To address interpretability challenges, a range of Explainable Artificial Intelligence (XAI) techniques has emerged [22, 23, 31]. These approaches aim to make machine learning models more transparent without compromising predictive power [32]. For example, feature importance methods highlight the variables most influential on a model's prediction [11], while visualization tools offer intuitive ways to see how a model processes inputs [33]. Post-hoc explanation methods such as Local Interpretable Model-Agnostic Explanations (LIME) [20] and SHapley Additive exPlanations (SHAP) [34] provide simplified, human-readable explanations for individual predictions, enabling better understanding among end-users. Such tools are especially useful in domains like healthcare and finance, where understanding a model's decision-making process is essential for building trust and ensuring ethical use [26, 27].

Despite their utility, post-hoc explanation methods such as LIME and SHAP are inherently heuristic. These methods rely on analyzing the input-output relationships of a model rather than reasoning about its internal mechanisms, such as its structure or learned parameters. As a result, the explanations they provide can only be approximate. To overcome these limitations, more formal explainability approaches have been proposed. Two notable post-hoc formal methods are abductive [35, 36, 37] and inflated explanations [38]. Abductive explanations identify minimal subsets of feature assignments sufficient to produce a specific prediction. In the context of random forests, for instance, these explanations pinpoint the smallest set of input features that, once



(a) A random forest learned on the Iris dataset.



(b) A DAG that is semantically equivalent to the random forest in Figure 1.1a.

Figure 1.1: A random forest and a semantically equivalent DAG obtained by applying the ADD-based approach [42] to the random forest.

fixed, guarantee the output for a particular instance. Typically, algorithms for generating abductive explanations rely on formal reasoning tools such as Boolean Satisfiability (SAT) [38, 39], Satisfiability Modulo Theories (SMT) [40], or Maximum Satisfiability (MaxSAT) [41] solvers, where the model's decision logic is encoded into logical constraints. Inflated explanations [38] extend abductive explanations by specifying an interval for each feature that cannot be increased without changing the prediction, which can provide additional insights into the model's decision-making process.

This dissertation explores the potential of compilation-based explainability approaches, which aim to translate a machine learning model's decision process into an interpretable representation, often using logical formulas or another compact structure. Such representations can enable more efficient computation of explanations and provide deeper insights into the model's decision-making.

One interesting compilation-based method is presented in [42], where a random forest is transformed into a single, semantically equivalent Algebraic Decision Diagram (ADD), a generalization of Binary Decision Diagrams (BDDs) whose leaves may store arbitrary values rather than simple Booleans. In other words, for every input instance, the ADD returns the exact same prediction as the original random forest. To illustrate, consider the small random forest of four decision trees shown in Figure 1.1a, trained on the Iris dataset. Typically, one would need to trace the root-to-leaf path of each tree to see how the forest arrives at its predictions, which can turn into a complicated process even for moderately sized ensembles. When the random forest makes a prediction, each tree votes for a class, and the class with the most votes is returned.

Figure 1.1b shows a directed acyclic graph (DAG), obtained by applying the ADD-based approach [42], that is semantically equivalent to the entire random forest. In this particular case, the graph actually coincides with the first decision tree, which means that for all inputs, the aggregated votes of the four trees match the outcomes of the first tree alone. Therefore, each leaf in the DAG corresponds to the final class label that the random forest would predict, allowing one to follow a single path to see precisely how the model classifies an instance. Having a single, unified

structure significantly reduces the complexity of interpreting the model compared to traversing separate decision paths.

Although such a compact visualization is appealing for small random forests, real-world models often result in much larger DAGs, making them difficult to inspect directly. Nevertheless, once a semantically equivalent DAG or ADD exists, one can still compute explanations relatively easily by tracing paths within that structure. The challenge, however, lies in the construction of these representations, which can be extremely expensive. In the worst case, compiling a random forest of n trees $T_1, \ldots, T_n$ can take $O(|T_k|^n)$ time, where $|T_k|$ denotes the number of nodes in the largest tree. In fact, the experimental results in this thesis show that compilation alone can be excessively time-consuming: for instance, on one particularly large random forest, the existing approach [42] required over two hours, whereas our new method needed only 20 seconds. Motivated by these scalability challenges, the central focus of this dissertation is to develop more efficient and widely applicable compilation techniques. By significantly reducing the time required to transform tree ensembles into a single graph-based representation, our approach can handle larger models and enable efficient explanation generation, as well as a range of additional analyses.

For a more complete understanding of this approach, we now compare it with solver-based (SMT/SAT) methods for generating explanations. SMT/SAT-based methods typically construct and solve logical formulas on demand: to generate an explanation, the model's decision rules must be encoded into a SAT or SMT instance, which is then handed off to a solver. While this can work well for single queries, it becomes increasingly costly when multiple explanations need to be derived, because each request requires a fresh solver invocation over a potentially large logical encoding.

In contrast, compilation-based approaches incur a one-time transformation cost: the entire tree ensemble is converted into a single, semantically equivalent structure (for example, a DAG or ADD). Although this transformation can be expensive, once it is completed, generating multiple explanations or performing different forms of analysis becomes extremely efficient. This makes compilation-based techniques especially advantageous in scenarios where numerous explanations are likely to be requested, such as interactive systems.

Interestingly, our experiments show that even for a single explanation request, the overhead of constructing a DAG may be outweighed by its benefits. In some cases, we can compile the tree ensemble and produce a single explanation faster than a state-of-the-art SMT/SAT-based approach can compute an explanation. As a result, compilation-based methods offer a balanced trade-off between upfront cost and subsequent performance, effectively scaling to a variety of real-world use cases where interpretability on demand is critical.

These advantages can be leveraged through novel techniques that transform entire tree ensembles into single, semantically equivalent graphs. In this dissertation, we introduce a set of compilation-based methods that significantly reduce the cost of generating explanations and enable additional forms of analysis. By focusing on both random forests and gradient boosted trees, we illustrate how these techniques maintain predictive accuracy while offering interpretability on demand. The approaches that we will introduce in this dissertation transform a tree ensemble, e.g., a random forest or gradient boosted trees, into a single semantically equivalent graph. This transformation has several advantages:

- **Applicability of single-tree algorithms:** The resulting semantically equivalent graph enables the application of algorithms traditionally designed for individual decision trees. As a result, generating abductive or inflated explanations for complex models (like random forests or gradient boosted trees) becomes straightforward, avoiding the need for specialized modifications or custom methods.

- **Efficient explanation generation:** The resulting semantically equivalent graph enables extremely efficient generation of abductive and inflated explanations. In our evaluation,

we show that, once the transformation has been performed, one can generate explanations several orders of magnitude faster compared to state-of-the-art solutions.

To demonstrate the effectiveness of these contributions, we now highlight their concrete impact in terms of compilation speedups, reduced model sizes, and faster generation of explanations. Specifically, we achieve the following performance improvements:

- **Speeding up ADD-based compilation for random forests:** By incorporating our proposed optimizations into the method from [42], which transforms random forests into ADDs, we achieve a $3.62\times$ speedup.
- **Introducing a new depth-first search (DFS)-based transformation:** We present a novel approach based on DFS that generates a DAG instead of an ADD. On average, this DFS-based method is $20.49\times$ faster and produces a representation 2.63% smaller than the optimized ADD-based approach [42].
- **Advancing compilation for gradient boosted trees:** We propose a new method for transforming gradient boosted trees, improving upon our earlier ADD-based approach $[3]_{\text{AP}}$ by $51.98\times$ in speed and producing DAGs that are 8.22% smaller.
- **Accelerating explanation generation for random forests:** In comparison to the state-of-the-art SAT-based solution [38], for random forests our approach generates abductive explanations $7314\times$ faster and inflated explanations $687\times$ faster.
- **Delivering efficient explanations for gradient boosted trees:** Our method outperforms the MaxSAT-based solution [41] by $21140\times$ in abductive explanation speed and uniquely supports inflated explanations[1]. Although inflated explanations are harder to compute, we still generate them $2544\times$ faster than [41] can compute abductive explanations.

1.1 My Contributions

In this dissertation, I present four key contributions that advance the field of explainability for tree ensemble models. My contributions focus on improving compilation-based methods to make explainability more efficient and scalable, while also enabling comprehensive analyses and intuitive visualization of the compiled representations. To provide an overview of my contributions and their relationships, Figure 1.2 illustrates the key components and the flow of the proposed methods. Specifically, this dissertation delivers the following contributions:

Contribution 1: Novel Techniques to Compile Tree Ensembles into a Single Graph Representation

The first contribution of this dissertation is a set of novel compilation techniques that transform tree ensembles into a single, semantically equivalent graph representation. While previous work [42] has demonstrated how to convert random forests into ADDs, I advance this research in two key ways:

1. I adapt the ADD-based approach to support gradient boosted tree models, making it applicable to a wider range of ensembles.

[1] To our knowledge, no other approach supports inflated explanations for boosted trees, so we compare our method's inflated-explanation time to the abductive-explanation time of [41] for reference.

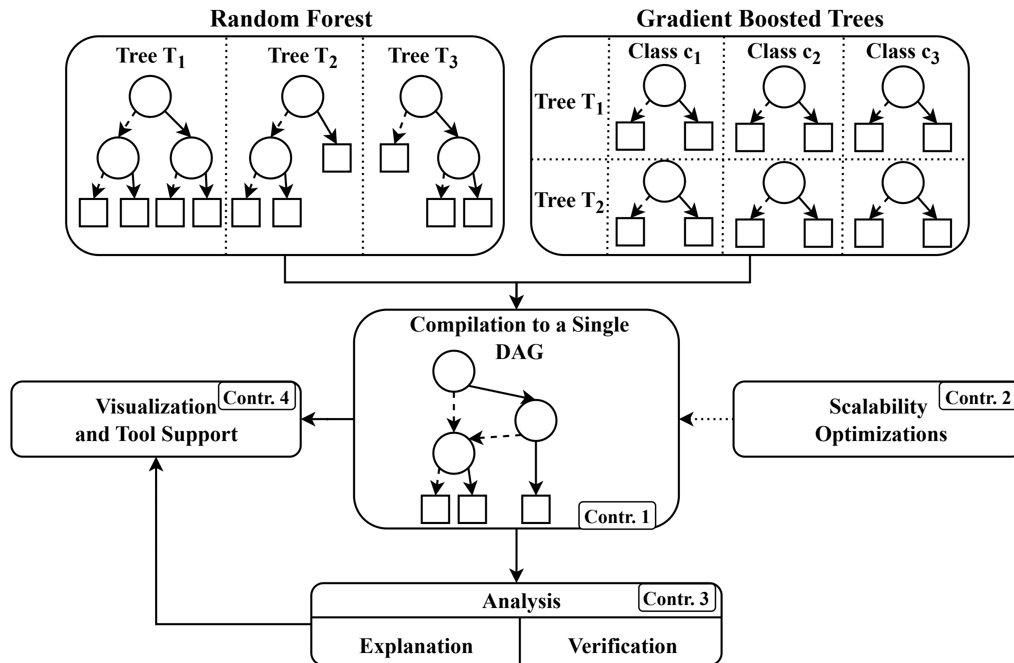


Figure 1.2: Overview of my contributions.

2. I propose an alternative compilation strategy that generates a single DAG representation based on DFS applicable to both random forests and gradient boosted trees. This DFS-based method significantly reduces the transformation time compared to the ADD-based approaches, enabling the compilation and analysis of larger random forests and gradient boosted trees that would otherwise be infeasible.

This unified structure enables algorithms originally designed for individual decision trees or DAGs, such as explanation generation and verification, to be applied efficiently to large ensembles. In contrast to operating on the original forest/ensemble, working on a single DAG substantially reduces the overhead of repeated computations, enabling the efficient generation of explanations and other forms of analysis.

Contribution 2: Methods to Improve the Scalability of the Compilation Techniques

The second contribution addresses a key challenge of compilation-based explainability: scalability [43]. Applying these methods to large models is often impractical due to high computational costs. To overcome this bottleneck, I introduce several optimization techniques that accelerate the transformation process and lower peak memory requirements. These methods can be incorporated into both the ADD-based strategies [42] and the novel DFS-based approach. By combining these optimizations with various heuristic techniques, the compilation process is significantly accelerated. Thus, my work contributes to making explainability methods for tree ensemble models more practical and applicable in real-world scenarios.

Contribution 3: Methods to Analyze the Compiled Representations

My third contribution is a suite of analysis techniques that directly utilize the compiled representations. In $[3]_{AP}$, I demonstrate how to derive abductive and inflated explanations from these unified structures, revealing the factors influencing the model's decisions. Furthermore,

in [2]$_{\text{AP}}$, I show how this single DAG or ADD representation can be leveraged for tasks such as pre-/postcondition verification and equivalence checking of different random forests. These techniques provide insights into model behavior while enabling robustness analysis through verifying pre-/postconditions.

Contribution 4: Web-Based Tool for Exploring Compiled Ensemble Models

Finally, I introduce a set of visualization techniques and a web-based platform designed to make these compiled representations more intuitive for non-domain experts. Traditional tree ensembles, like random forests and gradient boosted trees, are frequently employed by users without extensive backgrounds in machine learning. To address this audience, I developed an interactive platform that provides a user-friendly interface to (i) train random forest models, (ii) perform all intermediate compilation steps based on [42], (iii) generate explanations, (iv) verify pre-/postconditions, (v) check model equivalences, and (vi) export the resulting compiled models in various programming languages (e.g., Java, Python, C++). This platform simplifies access to robust and interpretable ensemble models by providing both explainability and transparency through user-friendly visualizations and automated tools.

In sum, these contributions collectively advance the state-of-the-art in explainable machine learning by improving the efficiency, scalability, and interpretability of tree ensemble models. By introducing novel compilation strategies, practical optimization techniques, advanced analyses, and user-friendly visualization tools, this dissertation enables more transparent deployment of ensemble models in practical applications.

1.2 Context of Attached Publications

This dissertation forms part of a cumulative thesis, consisting of previously published articles co-authored with fellow researchers. While the publications are listed on page v and vi, with details of my individual contributions provided in accordance with the doctoral degree requirements of TU Dortmund, this section offers a broader explanation of how these works relate to the overarching theme of this dissertation and its research objectives.

Forest GUMP: A Tool for Explanation ([1]$_{\text{AP}}$)

This paper introduces Forest GUMP, a tool for explainability and verification of random forests. [1]$_{\text{AP}}$ leverages the ideas from [42] to compile random forests into ADDs. Through an interactive interface, users can (i) train random forest models, (ii) inspect intermediate steps of the ADD-based compilation process, and (iii) generate explanations. Additionally, the system supports exporting the compiled representations to various programming languages (e.g., C++, Java, Python). Overall, [1]$_{\text{AP}}$ demonstrates the feasibility of turning theory into practice by allowing non-expert users to analyze and interpret complex models.

Forest GUMP: a tool for verification and explanation ([2]$_{\text{AP}}$)

This paper is an extended version of [1]$_{\text{AP}}$ that introduces verification and equivalence checking features in Forest GUMP. We show how, once the random forest has been transformed into an ADD, one can perform pre-/postcondition verification, and check the equivalence of two random forests. We extend the tool Forest GUMP by adding these features and show by means of examples how these features can be used.

Computing Inflated Explanations for Boosted Trees: A Compilation-Based Approach ([3]$_{AP}$)

This paper extends the techniques introduced in [42] to compile boosted trees into a single semantically equivalent ADD. We also show how the ADD can then be used to generate abductive and inflated explanations for the predictions of the boosted trees. We evaluated our approach on boosted trees learned on real-world data sets and showed that our approach is able to generate explanations several orders of magnitude faster than the state-of-the-art.

Voting-Based Shortcuts through Random Forests for Obtaining Explainable Models ([4]$_{AP}$)

Focusing on scalability, [4]$_{AP}$ presents a series of optimizations aimed at reducing the intermediate representation size and the associated computational overhead in the ADD compilation process. These optimizations are divided into semantic-preserving and non-semantic-preserving categories. Semantic-preserving techniques preserve exact equivalence with the original random forest but can only be applied after half of the trees have been compiled. In contrast, non-semantic-preserving techniques may slightly alter the forest's behavior but can be used at any point in the compilation process, and as a result trimming the intermediate representation earlier in the process. For both categories, [4]$_{AP}$ evaluates the effect on prediction accuracy and compilation time, concluding that these optimizations significantly accelerate compilation while causing negligible reductions in model accuracy.

An Efficient Compilation-based Approach to Explaining Random Forests Through Decision Trees ([5]$_{AP}$)

[5]$_{AP}$ proposes a novel approach to compile a random forest directly into a single semantically equivalent DAG based on DFS. Unlike prior work that is based on ADDs, this method performs a depth-first traversal of the forest, following the root of the subsequent tree once a leaf node is reached, while creating a new decision tree at the same time. Additionally, we present an approach based on abstract interpretation that enables the application of early stopping before half of the trees have been compiled while still preserving semantics. Our experiments show that this approach reduces transformation time by more than an order of magnitude compared to previous methods [42], confirming its suitability for large-scale random forests.

Enhancing Performance Through Control-Flow Unmerging and Loop Unrolling on GPUs ([6]$_{AP}$)

While not directly related to AI explainability, [6]$_{AP}$ was my first work involving extensive experimental evaluations and compiler optimizations. Here, we proposed a GPU compiler optimization that combines control-flow unmerging with loop unrolling to achieve speedups by enabling the application of subsequent optimizations such as constant folding and read elimination. Implementing this approach in LLVM and evaluating it on CUDA benchmarks resulted in performance gains of up to 81%. Although [6]$_{AP}$ primarily addresses general-purpose computing, the lessons learned proved invaluable for enhancing the scalability of the compilation-based explainability tools developed in this dissertation. On a high level, the approaches introduced in Chapter 4 align closely with the transformations discussed in this paper, demonstrating their usefulness in diverse compilation scenarios.

Additional Unpublished Contributions

While this dissertation builds upon previously published research, it also introduces entirely new techniques that have not appeared in prior work. In particular, Section 4.3 extends the approach from $[5]_{\text{AP}}$ to handle gradient boosted trees, including novel early stopping methods specifically tailored to this type of ensemble. This novel approach allows one to transform gradient boosted trees into a single semantically equivalent DAG several orders of magnitude faster than the ADD-based approach presented in $[3]_{\text{AP}}$. Additionally, Section 4.1.8 presents an additional optimization to $[5]_{\text{AP}}$ that reduces both the transformation time and the size of the resulting DAG.

1.3 Organization of this Dissertation

This dissertation is organized as follows: Chapter 2 introduces the necessary background. Chapter 3 reviews the existing method from [42] for compiling random forests into Algebraic Decision Diagrams (ADDs) and presents its adaptation for gradient boosted trees. Chapter 4 describes a DFS-based procedure for generating a DAG representation of both random forests and gradient boosted trees, avoiding the use of ADDs. Chapter 5 examines methods for deriving abductive and inflated explanations from the compiled representations and demonstrates pre-/postcondition-based verification and equivalence checking. Chapter 6 introduces *Forest GUMP*, a web application for the explainability and verification of random forests. Chapter 7 provides an overview of related work in the field, and Chapter 8 concludes the thesis by suggesting directions for further research.

Chapter 2

Background

In this chapter, we provide the necessary background information for the subsequent chapters. We start by introducing the basic concepts of classification problems and tree ensembles. We then discuss logic-based explanations, focusing on abductive explanations and inflated explanations. Finally, we introduce Decision Diagrams, specifically Algebraic Decision Diagrams (ADDs), which are extensively used throughout this thesis.

2.1 Classification Problems

In this section, we introduce the relevant concepts similar to previous work in this area [41]. For some natural number $n \in \mathbb{N}$, let $[n] := \{1, \ldots, n\}$ be the set of all natural numbers less than or equal to n.

This thesis is concerned with classification problems where $\mathcal{F} = \{1, \ldots, m\}$ denotes the set of features and $\mathcal{C} = \{1, \ldots, K\}$ the set of classes. Each feature $j \in \mathcal{F}$ is characterized by a domain D_j, which for the purposes of this thesis is assumed to be a numerical domain, e.g., a subset of $\mathbb{R}$[1]. The feature space is defined as $\mathbb{F} := D_1 \times D_2 \times \cdots \times D_m$. A *sample* or an *instance* is a specific point $\vec{v} = (v_1, \ldots, v_m) \in \mathbb{F}$ in the feature space. We use $\vec{x} = (x_1, \ldots, x_m) \in \mathbb{F}$, for an arbitrary point in the feature space. The value of a feature $j \in \mathcal{F}$ is denoted as x_j where $x_j \in D_j$. In general, when referring to the value of a feature $j \in \mathcal{F}$, we will use a variable x_j, with x_j taking values from D_j. A classifier implements a total classification function $\tau \colon \mathbb{F} \to \mathcal{C}$.

2.2 Tree Ensembles

Tree ensembles are machine learning models that combine multiple decision trees to form a single, more powerful classifier. By combining multiple decision trees, tree ensembles avoid issues commonly observed in individual decision trees, such as overfitting, and in general achieve higher predictive accuracy. Two widely used types of tree ensembles are *random forests* [11] and *gradient boosted trees* [12]. In this section, we introduce these methods, and explain their underlying mechanisms.

2.2.1 Decision Trees

A decision tree [44] is a machine learning model that makes predictions through a sequence of feature-based decisions for classification or regression tasks. Decision trees consist of two types of nodes: internal nodes, which represent a decision based on a feature, and leaf nodes which are associated with the model's decision. In this thesis, the decisions of an internal node are of the form $x_j < t$ where x_j is a feature and $t \in D_j$ is a threshold value. Internal nodes have exactly two successor nodes where based on the result of the decision, either the true- or false branch is

[1] The ideas and methods introduced in this thesis also extend naturally to categorical domains.

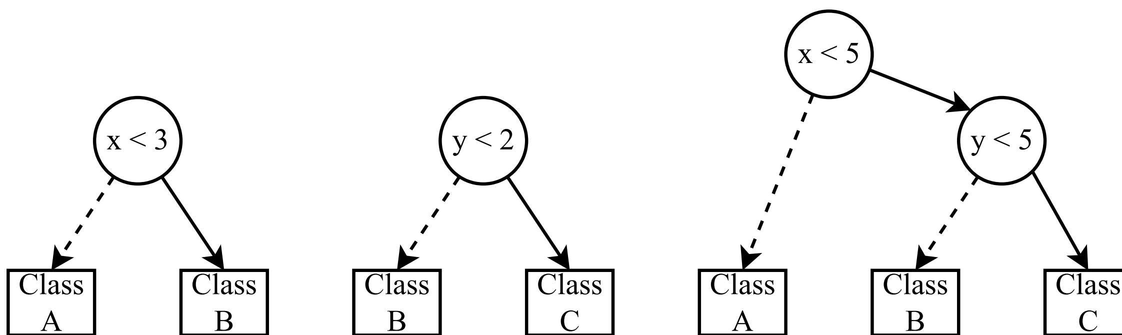


Figure 2.1: A random forest with 3 decision trees. Dotted edges represent false branches, while solid edges represent true branches.

followed. For classification tasks, each leaf is associated with a class $c \in \mathcal{C}$ which represents the prediction of the tree for instances that reach them.

Decision trees have several strengths, such as their simplicity, interpretability, and ease of visualization, which is why they are generally regarded as white-box models [22]. However, they are prone to overfitting the training data, resulting in high variance and reduced generalization performance on unseen data.

2.2.2 Random Forests

Random forests [11] are a tree ensemble method that uses *bootstrap aggregating* (bagging) to form a classifier from several decision trees. In this approach, each decision tree $T_1, \ldots, T_n$ is trained on a bootstrap sample of the training data. A bootstrap sample is obtained by sampling a number of instances with replacement from the original training data. By combining the predictions of multiple trees, random forests reduce the variance of the model and improve the predictive power.

Given an input instance $\vec{x}$, each decision tree T_i in the random forest produces a prediction $T_i(\vec{x}) \in \mathcal{C}$, where $i = 1, \ldots, n$. The final prediction $\hat{y}$ is then obtained by aggregating these individual predictions through majority voting:

$$\hat{y} = \arg\max_{c \in \mathcal{C}} \sum_{i=1}^{n} \mathbb{I}(T_i(\vec{x}) = c),$$

where $\mathbb{I}(\cdot)$ is the indicator function that returns 1 if the condition is true and 0 otherwise.

Example 1. Figure 2.1 shows a random forest consisting of three trees used to predict *Class A*, *Class B*, or *Class C*. Dotted edges represent false branches, while solid edges represent true branches. For the input $\vec{v} = (x = 4, y = 6)$, the votes of the three trees would be *Class A*, *Class B*, and *Class B*. Since *Class B* receives the majority of the votes, this would be the final prediction of the random forest.

2.2.3 Gradient Boosted Trees

Gradient boosted trees [12] are another popular tree ensemble method that combines multiple trees to form a single classifier, often referred to as a gradient boosted tree model. In contrast to random forests, where the decision trees are learned independently, gradient boosted trees are learned iteratively aiming to correct the errors of the previously learned trees.

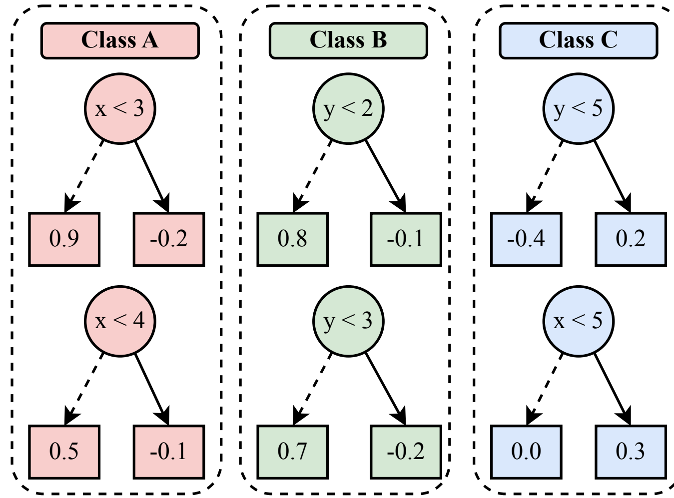


Figure 2.2: A gradient boosted tree model with two trees per class.

Multiclass Classification: For a classification task with $K > 2$ classes, the model learns $n \in \mathbb{N}_{>0}$ trees for each class $j \in [K]$. We denote the i-th tree for class j by T_i^j, where $i \in [n]$. Each decision tree is associated with a specific class j, and the leaves of the trees are associated with a real-valued score.

For an input $\vec{x} \in \mathbb{F}$, the total weight (score) assigned to class j is:

$$w_j(\vec{x}) \;:=\; \sum_{i=1}^{n} T_i^j(\vec{x}).$$

The model's prediction is the class with the highest total weight:

$$\hat{y} \;=\; \arg\max_{j \in [K]} w_j(\vec{x}).$$

Example 2. Figure 2.2 shows a gradient boosted tree model with two trees per class. For the input $\vec{v} = (x = 6,\, y = 2.5)$, the trees for *Class A* would return $w_1(\vec{v}) = 0.9 + 0.5 = 1.4$, the trees for *Class B* would return $w_2(\vec{v}) = 0.8 + (-0.2) = 0.6$, and the trees for *Class C* would return $w_3(\vec{v}) = 0.2 + 0.0 = 0.2$. Since the sum of weights for *Class A* is the highest, this is the final prediction of the gradient boosted tree model.

Binary Classification: For binary classification with two classes, i.e., a positive and a negative class, a common approach is to learn a single sequence of n trees, where each tree $T_i(\vec{x})$ outputs a real-valued score. The overall score for an input $\vec{x}$ is:

$$\text{score}(\vec{x}) \;=\; \sum_{i=1}^{n} T_i(\vec{x}),$$

A common approach is to apply the logistic (sigmoid) function,

$$\sigma\big(\text{score}(\vec{x})\big) = \frac{1}{1 + e^{-\text{score}(\vec{x})}}$$

to obtain a probability in $(0, 1)$. The final classification is obtained by thresholding this probability at 0.5, or equivalently, by predicting the positive class if $\text{score}(\vec{x}) > 0$ and the negative

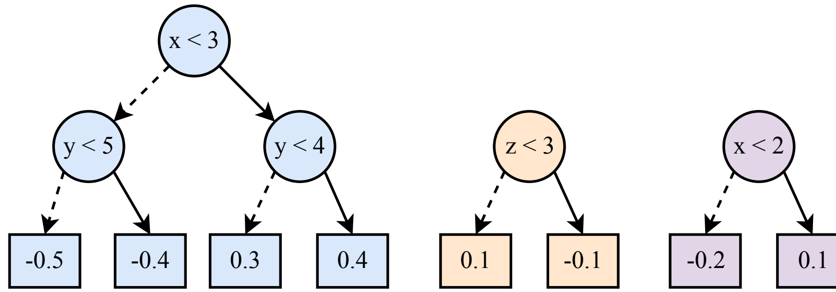


Figure 2.3: A gradient boosted tree model for binary classification consisting of three trees.

class otherwise.

Example 3. Figure 2.3 presents a gradient boosted tree model for binary classification with three trees. When given the input $\vec{v} = (x = 2.5,\ y = 3,\ z = 4)$, each tree outputs a real value, and these values are summed to produce a single score:

$$\text{score}(\vec{v}) \ = \ 0.4 \ + \ 0.1 \ + \ (-0.2) \ = \ 0.3.$$

Next, we apply the sigmoid function to convert this score into a probability:

$$\sigma(0.3) \ \approx \ 0.57.$$

Since 0.57 exceeds the threshold of 0.5, the model predicts the positive class for this instance.

2.3 Logic-based Explanations

In this thesis, we focus on explanation methods that must satisfy specific logical requirements. The explanations considered here come with a rigorous formal definition, enabling mathematical reasoning and precise derivations.

One well-known type of formal explanation is the Abductive Explanation (AXp) [35][2]. An AXp for a sample $\vec{v} \in \mathbb{F}$ is an *inclusion-minimal* set of feature-value pairs sufficient to preserve the model's prediction for $\vec{v}$.

Definition 1 (Abductive Explanation [35])**.** For an instance $\vec{v} \in \mathbb{F}$, an *abductive explanation* (AXp) is an inclusion-minimal subset $\mathcal{X} \subseteq \mathcal{F}$ such that:

$$\forall \vec{x} \in \mathbb{F}. \left(\bigwedge_{j \in \mathcal{X}} (x_j = v_j) \right) \implies \tau(\vec{x}) = \tau(\vec{v}) \ .$$

Abductive explanations are not unique, i.e., given a fixed sample $\vec{v}$ there can exist multiple $\mathcal{X}$ satisfying the above equation.

Example 4. Given the random forest in Figure 2.1 and the input $\vec{v} = (x = 4, y = 6)$, we already determined that the random forest's prediction is *Class B* in Example 1. The only abductive explanation that exists is $\mathcal{X} = \{x, y\}$ because we can change the random forest's prediction to another class by removing either x or y. If we remove x from $\mathcal{X}$, *Class A* might win, as x can take on any value and thus receive one vote each from the first and third tree, e.g., if $x = 6$. Similarly, if we remove y from $\mathcal{X}$, *Class C* might win because for $y = 1$ it would receive one vote each from the second and the third tree.

[2]These are also known as *PI explanations* [36] and *sufficient reasons* [37].

Building on abductive explanations, the concept of inflated explanations [38] was introduced to specify not only which features are sufficient but also which ranges of values they may take while preserving the prediction. This extension provides deeper insight into the model's decision process.

Definition 2 (Inflated Explanation [38])**.** For an instance $\vec{v} \in \mathbb{F}$ with AXp $\mathcal{X} \subseteq \mathcal{F}$, an *inflated (abductive) explanation* (iAXp) is a tuple $(\mathcal{X}, \mathbb{X})$, where $\mathbb{X}$ contains for each feature $j \in \mathcal{X}$ a set $\mathbb{E}_j \subseteq D_j$ with $v_j \in \mathbb{E}_j$, satisfying:

$$\forall \vec{x} \in \mathbb{F}. \left(\bigwedge_{j \in \mathcal{X}} (x_j \in \mathbb{E}_j) \right) \implies \tau(\vec{x}) = \tau(\vec{v}) \tag{2.1}$$

When j is a numerical feature, $\mathbb{E}_j$ must be an interval. Additionally, each $\mathbb{E}_j$ must be inclusion-maximal in the sense that any trivial extension of $\mathbb{E}_j$ within the allowed bounds $\mathbb{E}_j \subseteq D_j$ violates Equation (2.1).

Example 5. Given the random forest in Figure 2.1 and the input $v = (x = 4, y = 6)$, one inflated explanation is $(\mathcal{X} = \{x, y\}, \mathbb{X} = (\mathbb{E}_x = (-\infty, 5), \mathbb{E}_y = [5, \infty)))$. These intervals ensure that the predicate $x < 5$ is true, and the predicates $y < 5$ and $y < 2$ are false, which ensures that *Class B* receives one vote each from the second and third tree. The truth value of $x < 3$ does not matter in this case since *Class B* already receives two votes from trees 2 and 3, which is enough to win the majority vote. This is why the lower bound of x is $-\infty$.

2.4 Decision Diagrams

In this section, we introduce Binary Decision Diagrams (BDDs) and ADDs. We explain their core properties, describe their role in efficiently representing complex functions, and motivate why they are important for the remainder of this thesis.

2.4.1 Binary Decision Diagrams

Binary Decision Diagrams [45, 46, 47] are a widely used data structure for the compact representation, and efficient manipulation of Boolean functions, i.e., functions of the form $\{0,1\}^n \to \{0,1\}$. A BDD is a directed acyclic graph (DAG) consisting of internal nodes and leaf nodes. An internal node is associated with a Boolean variable x_i and has two outgoing edges representing the two possible values of x_i. The leaf node represents either $true$ or $false$ (or 1 and 0). In this thesis, the term BDD is used synonymously with Reduced Ordered Binary Decision Diagrams (ROBDDs). ROBDDs have the following properties:

- **Reduced:** Isomorphic subgraphs are merged, and nodes whose children are isomorphic are deleted as well.
- **Ordered:** There exists a fixed order in which the variables $x_1, \ldots, x_n$ are tested along any path from the root to a leaf node. On any path, each variable is tested at most once.

By satisfying these properties, ROBDDs provide a canonical representation of a Boolean function for a fixed variable order, i.e., for each Boolean function there exists a unique ROBDD representing that function. This enables e.g., to check the equivalence of two BDDs efficiently [48].

Example 6. Figure 2.4 shows three BDDs that all represent the Boolean function $x_1 \vee x_2 \vee x_3$. The leftmost BDD is neither ordered nor reduced. It is not ordered because on one path x_2 appears

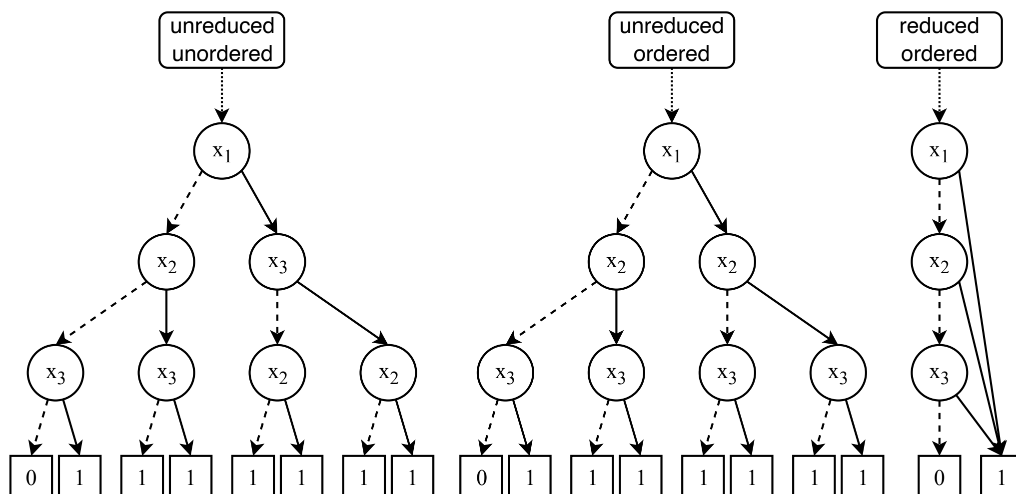


Figure 2.4: Three BDDs representing the same function.

before x_3, while on another path x_3 appears before x_2. It is also not reduced, since multiple isomorphic subgraphs (e.g., all leaves labeled 1) are not merged.

By swapping x_3 with x_2 on the right side of the leftmost BDD, one obtains the middle BDD, which is now an ordered BDD. Finally, the rightmost BDD is both ordered and reduced, making it the canonical representation of $x_1 \vee x_2 \vee x_3$ for the variable ordering $x_1 < x_2 < x_3$.

To manipulate BDDs, e.g., to create the conjunction $f \wedge g$ or disjunction $f \vee g$ of two BDDs, one can use binary aggregation. Given two BDDs f and g, the time complexity of binary aggregation is $O(|f| \cdot |g|)$.

Definition 3 (Binary Aggregation [47])**.** Given a binary operation $\bullet \in O$, it is possible to lift this operation on the BDD level such that for BDDs $f, g\colon \{0,1\}^n \to \{0,1\}$ and all inputs $\vec{x} \in \{0,1\}^n$ the following holds

$$(f \bullet_A g)(\vec{x}) := f(\vec{x}) \bullet g(\vec{x})$$

Just like binary operations, monadic operations, such as negation $\neg f$, can also be lifted to BDDs.

Definition 4 (Monadic Aggregation [47])**.** Given a unary operation $\bullet \in O$, it is possible to lift this operation on the BDD level such that for a BDD $f\colon \{0,1\}^n \to \{0,1\}$ and all inputs $\vec{x} \in \{0,1\}^n$ the following holds

$$\big(\bullet_A(f)\big)(\vec{x}) := \bullet\big(f(\vec{x})\big)$$

While BDDs can represent Boolean functions in a compact way, in the worst case the size of a BDD is still exponential in the number of variables [47].

2.4.2 Algebraic Decision Diagrams

Algebraic Decision Diagrams (ADDs) [49] generalize BDDs [47] from the boolean to any algebraic co-domain. ADDs represent functions $\{0,1\}^n \to A$ where A is the carrier set of an algebraic structure (A, O) with operations O. ADDs have the same properties as BDDs, i.e., they are reduced and ordered, and just as for BDDs, one can use binary and monadic aggregation to construct new ADDs.

Example 7. Figure 2.5 shows an example of three ADDs. The first ADD represents the function $2x_1 + 2x_2$, whereas the second ADD represents the function x_3. The third ADD represents their product, i.e., $(2x_1 + 2x_2) \cdot x_3$, which can represent it more compactly than a decision tree.

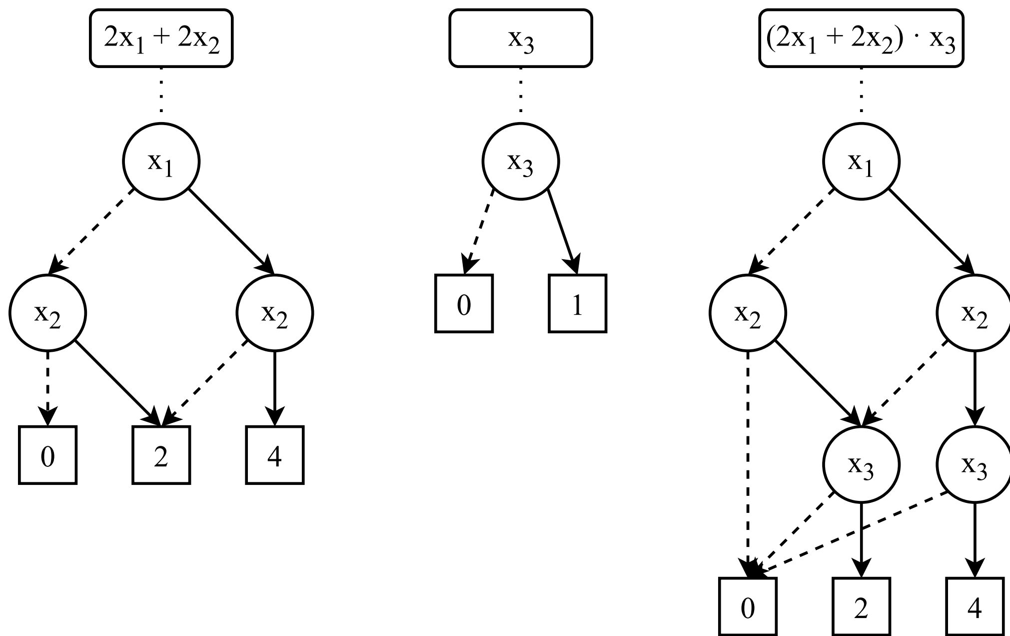


Figure 2.5: An example of two ADDs representing the functions $2x_1 + 2x_2$ and x_3, and their product $(2x_1 + 2x_2) \cdot x_3$.

2.4.3 Predicate-based Decision Diagrams

Traditionally, ADDs operate on Boolean variables, in contrast to the decision trees of a tree ensemble which operate on predicates, such as $x < 5$. While Boolean variables are independent of each other, the truth value of one predicate can influence that of another predicate. For example, if we know that $x < 5$ is true, we can infer that $x < 6$ is also true. We can extend BDDs and ADDs to operate on predicates as described in [42] by keeping track of a mapping between Boolean variables and predicates, e.g., $x < 5$ would be mapped to x_1 while $x < 6$ would be mapped to x_2. This mapping alone is not sufficient because it is possible to create ADDs that contain infeasible paths. We will discuss in detail approaches that can deal with infeasible paths in Section 3.1.1.

Chapter 3

Transforming Tree Ensembles into Algebraic Decision Diagrams

This chapter introduces techniques for transforming random forests and gradient boosted trees into Algebraic Decision Diagrams (ADDs). In Section 3.1, we summarize the work of [42] on converting random forests into semantically equivalent ADDs. Section 3.2 then presents optimizations that accelerate this transformation [4, 5]$_{\text{AP}}$, and Section 3.3 evaluates both the original approach and our optimizations.

Next, we present our method for transforming gradient boosted trees into ADDs [3]$_{\text{AP}}$ which extends the ADD-based technique [42]. Section 3.5 introduces optimizations similar to those in Section 3.2, adapted specifically for gradient boosted trees, and we evaluate their performance in Section 3.6.

3.1 Transforming Random Forests into Algebraic Decision Diagrams

In this section, the work of Gossen and Steffen [42] is summarized, as it forms the foundation for the subsequent approach transforming gradient boosted trees into ADDs. In addition to summarizing their approach, we also provide in-depth examples on how their approach works, and we discuss their infeasible path elimination in more detail.

Let us consider a random forest R composed of $T_1, \ldots, T_n$ decision trees. As previously discussed, the evaluation of a random forest follows the majority vote principle. According to [42], this voting procedure can be encoded in an ADD defined over the vector space $\mathbb{N}^K$, where K is the number of classes. First, each individual decision tree is transformed into an ADD by replacing each leaf node with a unit vector $e \in \mathbb{N}^K$, where e has a 1 in the position corresponding to the predicted class and 0 elsewhere. Formally, if a tree leaf predicts class $c \in [K]$, it is replaced with the vector e satisfying $e_c = 1$ and $e_j = 0$ for $j \neq c$. Once each decision tree T_i has been transformed into an ADD $\mathcal{A}_i$, they can be aggregated into a single ADD, the *Voting ADD*, by lifting vector addition to the ADDs:

$$\mathcal{A}^V = \mathcal{A}_1 +_A \cdots +_A \mathcal{A}_n$$

Essentially, given an input $\mathcal{A}^V$ returns a voting vector e that represents the number of votes each class receives from the decision trees $T_1, \ldots, T_n$. Finally, we can apply the $\arg\max$ function[1] to obtain the *Majority Vote ADD* $\mathcal{A}^{MV}$

$$\mathcal{A}^{MV} = \arg\max(\mathcal{A}^V)$$

[1] When two classes receive the same number of votes, the argmax function returns the class with the lower index.

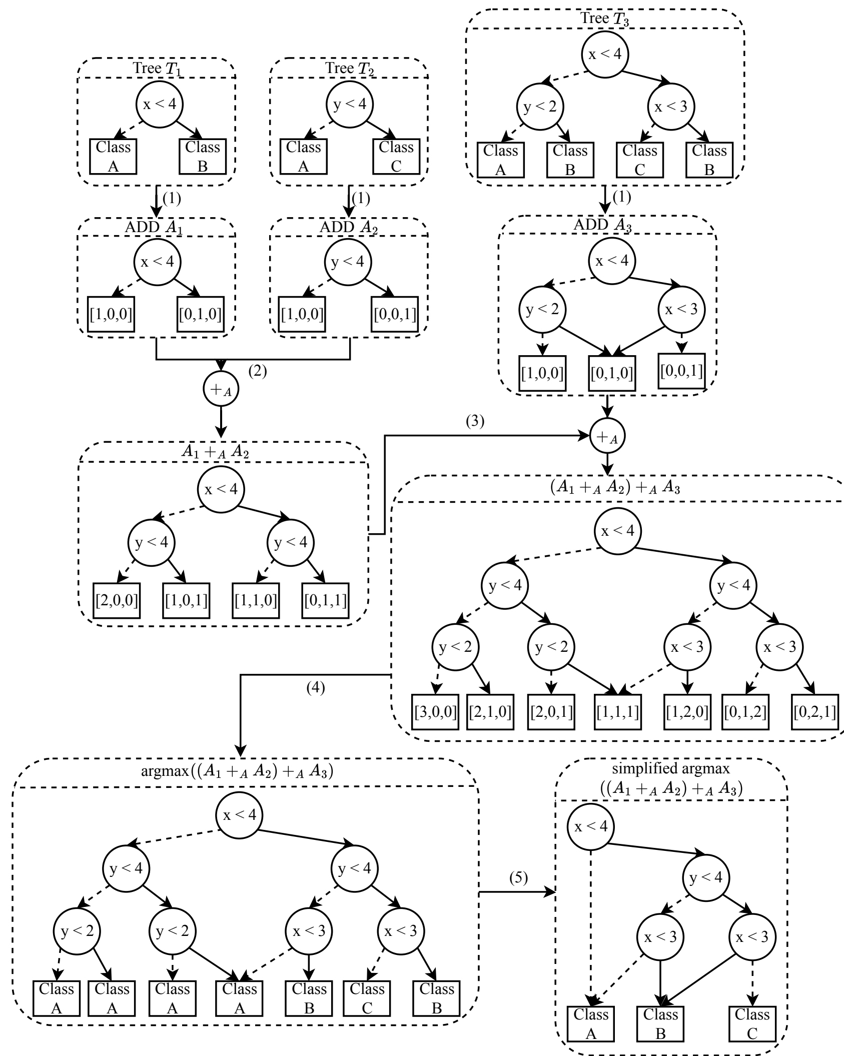


Figure 3.1: An overview of the transformation of a random forest into an ADD.

The Majority Vote ADD $\mathcal{A}^{MV}$ encodes the semantics of the random forest R:

$$R(x) = c_i \iff \mathcal{A}^{MV}(x) = i \quad \text{for all } x \in \mathbb{F}$$

Example 8. Figure 3.1 illustrates this procedure for a random forest with three decision trees T_1, T_2, and T_3[2]. In the first step, the decision trees T_1, T_2, and T_3 are transformed into ADDs $\mathcal{A}_1$, $\mathcal{A}_2$, and $\mathcal{A}_3$ where the classes *Class A*, *Class B*, and *Class C* are represented by the vectors $[1, 0, 0]$, $[0, 1, 0]$, and $[0, 0, 1]$ respectively. In the second step, the ADDs $\mathcal{A}_1$ and $\mathcal{A}_2$ are added and the resulting ADD would return the sum of votes each class receives from the first two trees, e.g., if $x < 4$ and $y < 4$ the ADD returns the voting vector $[0, 1, 1]$ indicating that *Class B* and *Class C* both receive a vote. Next, this ADD is added with $\mathcal{A}_3$, resulting in the Voting ADD that returns the number of votes from the three trees. Now, that all ADDs have been aggregated, it is possible to apply the argmax function, which replaces each vector with the winning class. Finally, the $\arg\max$ function is applied to obtain the *Majority Vote ADD*, and standard ADD-reduction rules are then used to simplify the ADD.

When we take a closer look at the ADD $(\mathcal{A}_1 +_A \mathcal{A}_2) + \mathcal{A}_3$ in Figure 3.1, we can see that the

[2]Dotted edges represent false branches, while solid edges represent true branches.

Algorithm 1 Infeasible Path Elimination for ADDs.

```
1: procedure ElimInfeasPaths(ADD 𝒜)
2:   pc ← newHyperrectangle(all features ↦ (−∞, +∞))
3:   return ElimInfeasPathsRecursive(𝒜, pc)
4: end procedure
5: procedure ElimInfeasPathsRecursive(ADD 𝒜, Hyperrectangle pc)
6:   if 𝒜.isLeaf() then
7:     return 𝒜
8:   end if
9:   feat ← 𝒜.feature
10:  thr ← 𝒜.threshold
11:  lower ← pc[feat].lower
12:  upper ← pc[feat].upper
13:  // Case 1: comparison is always false if lower >= threshold
14:  if lower ≥ thr then
15:    return ElimInfeasPathsRecursive(𝒜.falseChild(), pc)
16:  else if upper ≤ thr then
17:    // Case 2: comparison is always true if upper <= threshold
18:    return ElimInfeasPathsRecursive(𝒜.trueChild(), pc)
19:  else
20:    // Case 3: both branches are feasible; explore both
21:    pc[feat].upper ← thr
22:    trueSubtree ← ElimInfeasPathsRecursive(𝒜.trueChild(), pc)
23:    pc[feat].upper ← upper
24:    pc[feat].lower ← thr
25:    falseSubtree ← ElimInfeasPathsRecursive(𝒜.falseChild(), pc)
26:    pc[feat].lower ← lower
27:    return new ADD(feat, thr, trueSubtree, falseSubtree)
28:  end if
29: end procedure
```

path $\neg(y < 4) \wedge (y < 2)$ is not feasible because if $y \geq 4$, then $y < 2$ cannot be true. The next subsection deals with eliminating such infeasible paths.

3.1.1 Infeasible Path Elimination

In this section, we discuss the infeasible path elimination introduced in [42] in more detail as it is crucial for understanding the efficiency of the approach. By removing paths that cannot be taken due to the predicates encountered along them, this technique removes redundancy from the ADD and often reduces its size, accelerating the transformation. It is also crucial as it enables efficient analysis of random forests, as we will later see in Chapter 5.

To keep track of whether a path is feasible or infeasible, we use a hyperrectangle representation of the path condition, pc. Each feature x is associated with an interval $[l, u]$ that restricts the possible values of x. Initially, for every feature we allow all real numbers $(-\infty, \infty)$.

Algorithm 1 shows how the infeasible path elimination can be implemented for ADDs[3]. The algorithm takes an ADD $\mathcal{A}$ and a path condition pc as input and returns a new ADD that does not contain infeasible paths. The algorithm traverses the ADD in a depth-first manner. When a leaf

[3] Note that ADDs are automatically reduced and reordered.

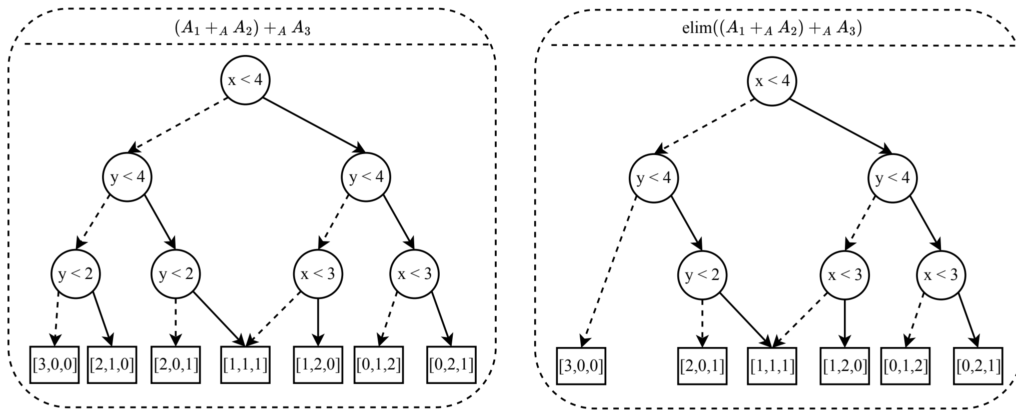


Figure 3.2: The ADD of Figure 3.1 and the same ADD after infeasible path elimination.

node is reached, no pruning is needed, and the algorithm simply returns the leaf node.

For an internal node with predicate *feature* $<$ *threshold*, we check its satisfiability. Let *lower* and *upper* be the current hyperrectangle bounds for *feature*.

- **Case 1:** If *lower* $\geq$ *threshold*, then the condition can never be satisfied, so we discard the "true" branch and traverse only the "false" branch.
- **Case 2:** If *upper* $\leq$ *threshold*, then the condition is always satisfied, so we discard the "false" branch and traverse only the "true" branch.
- **Case 3:** Otherwise, both branches are feasible, and we must traverse both branches. We temporarily modify *upper* and *lower* in *pc* before each recursive call, and then restore it again afterwards.

Essentially, the algorithm follows every path in the ADD that is feasible and reconstructs the ADD with the infeasible paths removed. So, the time it takes to eliminate infeasible paths depends on the number of feasible paths in the ADD.

There are different ways in which the infeasible path elimination can be applied. One can either apply it at the end of the transformation, i.e., after the majority vote ADD has been created. Alternatively, one can apply it after each aggregation. In many cases, it is beneficial to apply the infeasible path elimination after each aggregation, as it can reduce the size of the ADD and thus accelerate the whole transformation process.

Example 9. Figure 3.2 shows the ADD $(\mathcal{A}_1 +_A \mathcal{A}_2) + \mathcal{A}_3$ of Figure 3.1, and the same ADD after the infeasible path has been eliminated by applying Algorithm 1. Essentially, the false edge of $y < 4$ can be rerouted to $[3, 0, 0]$ since $y < 2$ is false if $y < 4$ is false. This results in the elimination of two nodes, the node labeled $y < 2$ and the leaf node $[2, 1, 0]$ as those are not reachable anymore.

In some cases, the infeasible path elimination can also increase the size of the ADD, though the average length of the paths is reduced. Consider the ADD in Figure 3.3 where the infeasible path elimination is applied. The path $x < 5 \land y < 8 \land x \geq 6$ is infeasible since $x < 5$ implies $x < 6$. The ADD after the infeasible path elimination is shown in Figure 3.4 which contains one additional node.

3.1.2 Class Characterizations

The behavior of the majority vote ADD is semantically equivalent to that of the initial random forest, but if one is interested only in a specific class, there is a more efficient representation

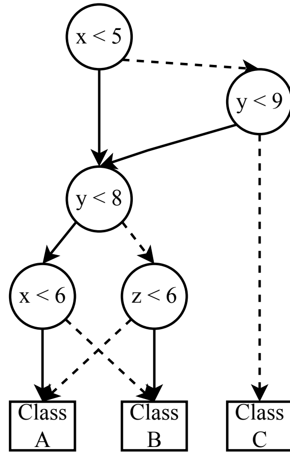


Figure 3.3: ADD before the infeasible path elimination.

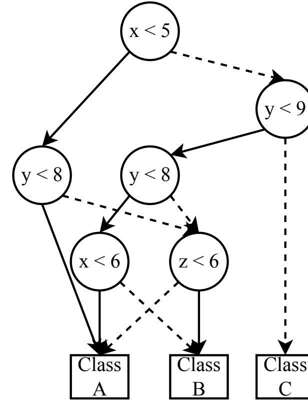


Figure 3.4: ADD after the infeasible path elimination.

of the random forest's behavior for that class. In [42] the concept of the *class characterization* is introduced. Given the majority vote ADD and a class c of interest, we can easily create a BDD B^c, the class characterization BDD for class c, that returns 1 when the prediction of the random forest is c and 0 otherwise. We obtain B^c by lifting the following monadic operation to the majority vote ADD:

$$\delta_c(c') = \begin{cases} 1 & \text{if } c = c', \\ 0 & \text{otherwise.} \end{cases}$$

The main advantage of the class characterization is that it provides a more compact representation of the random forest's behavior for a specific class.

Example 10. Figure 3.5 shows a majority vote ADD and the class characterizations for the three classes *Class A*, *Class B*, and *Class C*. Each class characterization is smaller than the majority vote ADD since there are predicates that are only necessary to differentiate between two classes. For *Class A* the predicate $z < 8$ is redundant since it is only used to differentiate between *Class B* and *Class C*. The same holds for *Class B* and predicate $z < 9$ and, *Class C* and predicate $y < 6$.

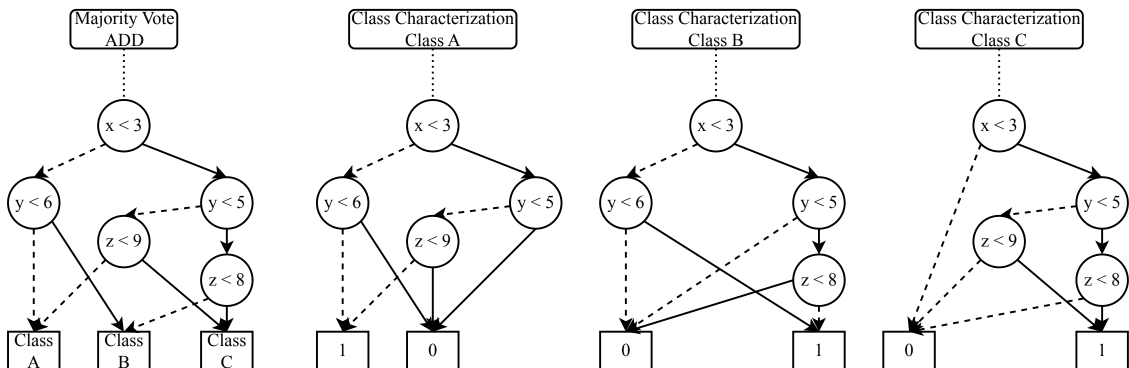


Figure 3.5: A majority vote ADD and the class characterization for the three classes.

Algorithm 2 RF2ADD: Transforming a random forest into an algebraic decision diagram.

```
1:  procedure RF2ADD(Random Forest T_1, ..., T_n)
2:      // Step 1: Convert each tree into an ADD
3:      for i ← 1 to n do
4:          A_i ← DT2ADD(T_i)
5:      end for
6:
7:      // Step 2: Aggregate the individual ADDs into a Voting ADD
8:      A^V ← A_1
9:      for i ← 2 to n do
10:         // Aggregate ADDs by adding the voting vectors from A_i.
11:         A^V ← A^V +_A A_i
12:         // Eliminate infeasible paths (cf. Algorithm 1).
13:         A^V ← ElimInfeasPaths(A^V)
14:     end for
15:
16:     // Step 3: Apply majority vote to obtain the final classification ADD
17:     A^MV ← ARGMAX(A^V)
18:
19:     // Step 4: Generate class characterizations if per-class analysis is desired
20:     for i ← 1 to K do
21:         B^i ← δ_i(A^MV)
22:     end for
23:     return (A^MV, B^1, ..., B^K)
24: end procedure
```

3.1.3 End-to-End Implementation

Algorithm 2 combines all the steps discussed so far into a single procedure, taking as input a random forest $T_1, \ldots, T_n$, and iteratively transforms it into an ADD. The algorithm consists of four main steps:

1. Each decision tree T_i is transformed into an ADD $\mathcal{A}_i$.
2. The individual ADDs are aggregated into a single ADD, the Voting ADD $\mathcal{A}^V$. After each aggregation, ElimInfeasPaths removes infeasible paths.
3. The $\arg\max$ function is applied to obtain the majority vote ADD $\mathcal{A}^{MV}$.
4. For each class $i \in [K]$, the class characterization is constructed.

In the end, the algorithm returns the majority vote ADD $\mathcal{A}^{MV}$ and the class characterizations $B^1, \ldots, B^K$.

3.2 Early Stopping for Random Forests

One limitation of compilation-based approaches, such as the one presented in the previous section, is their scalability. As illustrated in Figure 3.1, the intermediate ADD $\mathcal{A}^V$ can be significantly larger than the final ADD $\mathcal{A}^{MV}$. This section presents optimizations aimed at controlling the size of the intermediate ADD, reducing both transformation time and peak memory consumption.

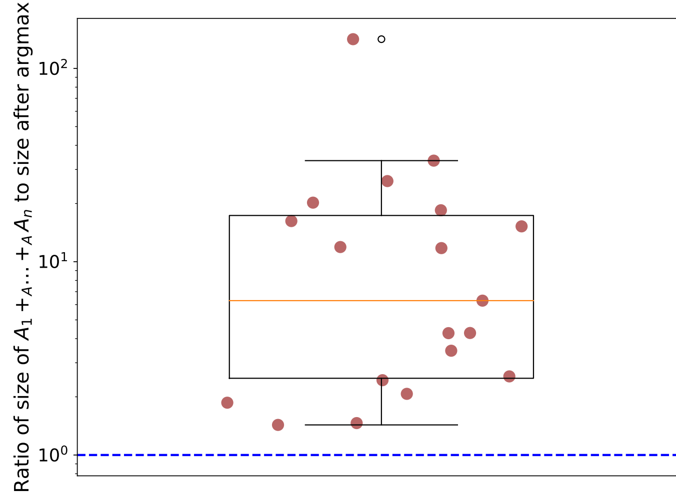


Figure 3.6: Ratio of the size of $\mathcal{A}^V$ to the size of $\mathcal{A}^{MV}$ for 19 random forests [4]$_{\text{AP}}$.

Figure 3.6 motivates such optimizations. It shows the ratio $\frac{|\mathcal{A}^V|}{|\mathcal{A}^{MV}|}$ for 19 random forests, which we will examine in Section 3.3. In one instance, $\mathcal{A}^V$ is 141 times larger than $\mathcal{A}^{MV}$. For 9 of these random forests, the intermediate ADD is at least 10 times larger than the final ADD. Thus, there is substantial opportunity for optimization.

The optimizations presented in this section are based on the observation that for a class to win the majority vote, it is sufficient for the class to receive more than half of the votes. We leverage this observation in the following way. Given the ADDs $\mathcal{A}_1, \ldots, \mathcal{A}_n$ representing the decision trees $T_1, \ldots, T_n$ of a random forest, and let

$$P_k = \mathcal{A}_1 +_A \ldots +_A \mathcal{A}_k$$

be the partial aggregation of the first k ADDs. The core idea is to detect scenarios in which the winner can no longer be overtaken or a runner-up can no longer close the gap, enabling some votes to be skipped. In other words, once a class is guaranteed to win or guaranteed to lose, further counting of its votes is unnecessary. To implement this, the voting vector domain is extended to include the symbols $\top$ and $\bot$ in addition to the natural numbers $\mathbb{N}$, where $\top$ indicates a guaranteed winner and $\bot$ indicates a guaranteed loser. This extension reduces the size of intermediate ADDs by removing unneeded predicates.

Example 11. Figure 3.7 demonstrates the approach of replacing the most probable class entry with $\top$ and setting all other entries to $\bot$. Let $\mathcal{A}_1 \ldots, \mathcal{A}_{25}$ denote the decision trees of a random forest each represented as an ADD. The leftmost ADD in the figure represents the partial voting ADD $\mathcal{A}_1 +_A \ldots +_A \mathcal{A}_{17}$, with the remaining 8 trees yet to be aggregated.

In this leftmost ADD, the predicate $y < 6$ exists solely to distinguish between the voting vectors $[14, 1, 2]$ and $[15, 1, 1]$. We already know that class c_1 will ultimately win because there are only eight remaining votes that could go to c_2 or c_3. The same reasoning applies to the voting vector $[14, 2, 1]$. Therefore, these voting vectors can be replaced by $[\top, \bot, \bot]$. Once identical leaves are merged and redundant nodes removed, the ADD shrinks from seven nodes to three.

Example 12. Figure 3.8 demonstrates the benefit of replacing the votes of classes that can no longer win with $\bot$. The ADD on the left corresponds to $\mathcal{A}_1 +_A \cdots +_A \mathcal{A}_{18}$, while $\mathcal{A}_{19}, \ldots, \mathcal{A}_{25}$ remain to be added. Because c_3 and c_4 can receive at most seven votes, they cannot catch up

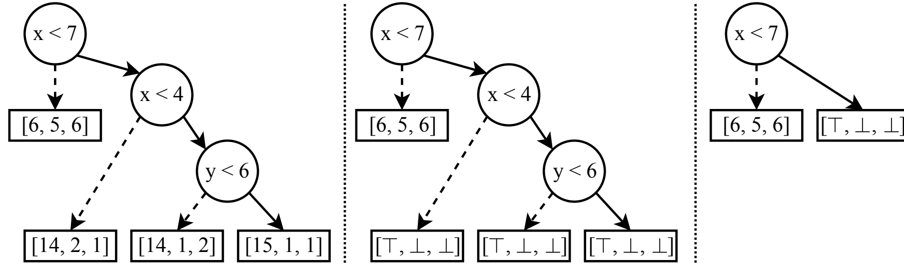


Figure 3.7: Example illustrating how replacing the winning class entry with $\top$ triggers simplifications by eliminating unnecessary nodes in the ADD.

to c_1 when c_1 already has 11 votes. Therefore, we replace their entries with $\bot$. The resulting simplifications are analogous to those shown in Figure 3.7.

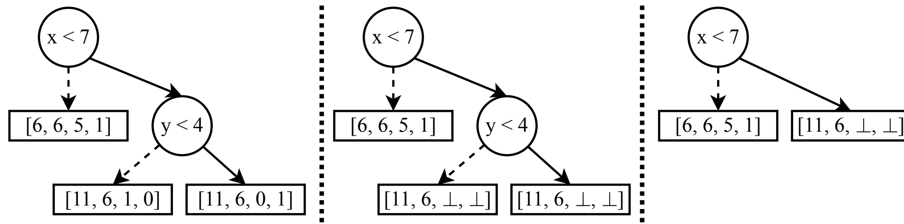


Figure 3.8: Replacing entries of classes that can no longer win with $\bot$ eliminates unnecessary nodes in the ADD.

3.2.1 Stochastic Problem Description

As each decision tree provides exactly one vote, the space of possible voting vectors, after the first k ADDs have been aggregated, is defined as

$$\Omega_k := \left\{ \omega \in \{0, \dots, k\}^K \;\middle|\; \sum_{i=1}^{K} \omega_i = k \right\}$$

which contains $\binom{k+K-1}{k}$ distinct vectors. Given a voting vector $\omega \in \Omega_k$ one can use this information to estimate the probability of class i winning. We can first compute the set of possible voting vectors after the remaining $n-k$ votes have been collected. This is expressed as:

$$\omega + \Omega_{n-k} = \left\{ \omega + v \;\middle|\; \sum_{i=1}^{K} v_i = n-k, v_i \in \mathbb{N} \right\} \subset \Omega_n$$

To compute the probability that class i wins, given the current voting vector ω, we consider all possible ways the remaining $n-k$ votes could be distributed. For each possible distribution $v \in \Omega_{n-k}$, we evaluate:

1. The probability of reaching the outcome $\omega + v$, denoted $P(\omega + v \mid \omega)$, given the current votes ω.

2. The probability that class i wins if the final voting vector is $\omega + v$, denoted $P(R = i \mid \omega + v)$.

By summing over all possible distributions of the remaining votes, we combine these two factors to obtain the overall probability of class i winning given ω:

$$P(R = i \mid \omega) = \sum_{v \in \Omega_{n-k}} P(\omega + v \mid \omega) P(R = i \mid \omega + v)$$

After all remaining votes have been collected, the probability that class i will win is 1 if i has the highest vote count in ω', and 0 otherwise.

$$P(R = i \mid \omega') = \begin{cases} 1 & \text{if } \arg\max_j \omega'_j = i, \\ 0 & \text{otherwise.} \end{cases}$$

Given a voting vector ω, we define two functions $\varphi_\top \colon \Omega_k \times [K] \to (\mathbb{N} \cup \{\top, \bot\})^K$ and $\varphi_\bot \colon \Omega_k \times 2^{[K]} \to (\mathbb{N} \cup \{\top, \bot\})^K$ as:

$$\big(\varphi_\top(\omega, j)\big)_i := \begin{cases} \top & \text{if } i = j, \\ \bot & \text{otherwise.} \end{cases}$$

$$\big(\varphi_\bot(\omega, J)\big)_i := \begin{cases} \bot & \text{if } i \in J, \\ \omega_i & \text{otherwise.} \end{cases}$$

These functions work as follow:

- $\varphi_\top$: Given a voting vector ω and a position j, it turns the entry at position j to $\top$ and all others entries to $\bot$. This function can be applied to declare j as the winning class.
- $\varphi_\bot$: Given a voting vector ω and some positions J, it turns all entries at these positions to $\bot$. This function can be applied to declare all classes in J as losing classes.

Assuming that we have a probability function that returns the probability of class i winning given ω, we define a function $\psi \colon \Omega_k \times [0,1]^2 \to (\mathbb{N} \cup \{\top, \bot\})^K$ for early stopping as:

$$\psi(\omega, \alpha, \beta) := \begin{cases} \varphi_\top(\omega, i) & \text{if } \exists! i : P(R = i \mid \omega) > \alpha, \\ \varphi_\bot(\omega, I) & \text{if } I = \{\, i \mid P(R = i \mid \omega) < \beta \,\}, \\ \omega & \text{otherwise.} \end{cases}$$

This function can be applied even if the probability of a class winning is not 1 or a class losing is 0. Given a voting vector ω, and two thresholds α and β, ψ returns $\varphi_\top(\omega, i)$ if there is exactly one entry i whose probability of winning is larger than α. If no such i exists, ψ turns all entries with a winning probability lower than β to $\bot$ by applying $\varphi_\bot$. Reasonable values satisfy $\beta \leq 0.5 \leq \alpha$.[4]

3.2.2 Semantics-Preserving Early Stopping

General Rules: When one class $i \in [K]$ has more than half of the n total votes, it is guaranteed to win. Similarly, when a class i has fewer than $\lceil \frac{n}{K} \rceil - (n - k)$ votes, it cannot win. Formally, these rules ensure that for every possible configuration of uncounted votes, $P(R = i \mid \omega + v)$ is always 1 or 0, respectively. Therefore, i may immediately be marked with $\top$ or $\bot$.

Example 13. Consider a random forest with $n = 12$ trees and $K = 3$ classes. Suppose the current voting vector is $\omega = [7,\ 1,\ 0]$, corresponding to $k = 8$ votes that have already been cast.

[4] If multiple classes i satisfy $P(R = i \mid \omega) > \alpha$, one could either choose the one with the highest probability or wait for more votes in the hope that the picture will then be clearer.

Because only 4 votes remain, the first class is guaranteed at least 7 total votes, while the other two classes can obtain at most 5 and 4 votes, respectively. Thus, the first class must win. Applying the general rules to ω therefore results in $[\top,\ \bot,\ \bot]$.

Example 14. Again let $n = 12$ and $K = 3$, and suppose $\omega = [v_1,\ v_2,\ 0]$ with $v_1 + v_2 = 9$ (so $k = 9$) and 3 votes remaining. The third class can gain at most 3 additional votes, so it cannot surpass whichever of v_1 or v_2 is leading (which is at least 5). Therefore, the third class is certain to lose, and applying the general rules to ω results in $[v_1,\ v_2,\ \bot]$.

Specific Rules: Extending the general rules, one can also incorporate the votes of the other classes. When $\omega_i > \omega_j + (n-k)$ for all $j \neq i$, then class i will win. Similarly, when $\omega_i + (n-k) < \omega_j$ for some j, then class i will definitely lose. While the general rules above imply these specific cases, the converse is not necessarily true.

Example 15. Let $n = 12$ trees and $K = 4$ classes, and suppose the current voting vector is $\omega = [5,\ 3,\ 2,\ 1]$ with $k = 11$ votes already cast. There is only 1 vote remaining, so the class with 5 votes cannot be overtaken by any other class (even though it does not exceed $\frac{n}{2} = 6$). Therefore, the general rules do not apply here, but the specific rules indicate that the first class is guaranteed to win. Thus, ω simplifies to $[\top,\ \bot,\ \bot,\ \bot]$.

Example 16. Consider a random forest with $n = 60$ trees and $K = 3$ classes, and suppose $\omega = [28,\ 7,\ 0]$ with $k = 35$ votes counted. Although 28 does not exceed half of 60, so the general rules cannot confirm a winner, the third class starting at 0 cannot receive enough votes to surpass 28 (it can receive at most 25 additional votes). Therefore, the specific rules ensure that the third class is certain to lose, and ω becomes $[28,\ 7,\ \bot]$.

3.2.3 Non-Semantics-Preserving Early Stopping

While the general and specific rules always preserve semantics, they can only be applied once at least half of the trees have been aggregated. However, it would be advantageous to allow early stopping before half of the trees have been aggregated, as this would reduce the size of the intermediate ADDs. Although such early stopping might produce outcomes that are not strictly semantics-preserving, this is not always the case.

To illustrate, consider the random forest in Figure 3.9. If *Class B* receives a vote from the first tree, we know $x < 3$ must hold. From this, it follows that $x < 4$ and $x < 5$ must also be true which guarantees that *Class B* receives a vote from each of the remaining two trees. So in this case, we could have determined *Class B* to be the winner after having only collected one vote. In general, a voting vector in an ADD is only reachable under certain conditions, and these conditions restrict the possible votes that the remaining trees can cast.

In the following, we will introduce two, not necessarily semantics-preserving, early stopping approaches. The main idea behind both approaches is to estimate the probability function $P(R = i \mid \omega)$ in order to apply ψ which uses thresholds to apply early stopping.

Lookup Table

Our first non-semantics-preserving approach is data-driven. The general idea is to use the training data and the random forest itself in order to estimate how likely it is for a class to win/lose after k votes. We propose to create two lookup tables W and L that are three dimensional:

- $W(i, h, k)$ counts how often class i eventually wins the majority vote (among all n trees) in the training set, given that after the first k trees it has h votes.

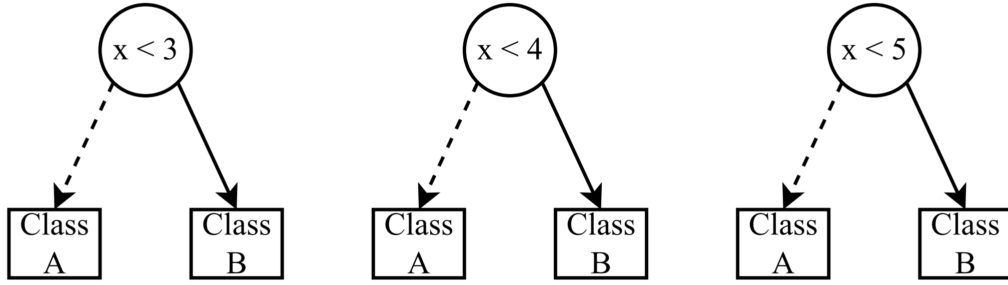


Figure 3.9: A random forest where early stopping can be applied before half of the trees have been aggregated.

- $L(i, h, k)$ counts how often class i eventually loses the majority vote in the training set, given that after the first k trees it has h votes.

Given a sample $x \in \mathbb{F}$, each decision tree T_i is queried to obtain a sequence of votes

$$V = (v_1, v_2, \ldots, v_n) \in [K]^n ,$$

where each $v_i \in [K]$ is the class predicted by tree T_i. We then aggregate those votes into a voting vector ω, whose i-th entry ω_i indicates the number of trees voting for class c_i. To analyze partial outcomes (e.g., after observing only the first k trees), we define:

$$\omega^{\leq k} := \sum_{i=1}^{k} e_{v_i} \in \Omega_k$$

By definition, the random forest's final prediction for the sample x (after all n votes) is given by

$$R(x) = \arg\max_{i \in [K]} \ \omega^{\leq n}{}_i$$

i.e., the class receiving the highest vote count in the final vector $\omega^{\leq n}$.

We use the intermediate voting vectors $\omega^{\leq 1}, \ldots, \omega^{\leq n}$ from each training sample to fill the two tables W and L. If the random forest classifies x as $i = R(x)$, then for each $k \in [n]$:

1. We increment the entry $(i, \omega_i^{\leq k}, k)$ in W by 1 to indicate that class i won.
2. For every other class $j \neq i$, we increment the entry $(j, \omega_i^{\leq k}, k)$ in L by 1 to indicate that these classes lost.

Repeating this procedure for each sample in the training set fully populates W and L.

We can now use these tables to estimate the probability of class i winning given a voting vector ω with $k = \|\omega\|_1$ as follows

$$P(R = i \mid \omega) \approx \frac{W(i, \omega_i, \|\omega\|_1)}{W(i, \omega_i, \|\omega\|_1) + L(i, \omega_i, \|\omega\|_1)}.$$

provided that $W(i, \omega_i, \|\omega\|_1) + L(i, \omega_i, \|\omega\|_1) > 0$. This ratio captures how often class i won in the training set whenever it had ω_i votes out of k at some intermediate stage, relative to how often it lost in that same scenario.

We now instantiate ψ using our look-up table probabilities and refer to the resulting early stopping function as ψ_{LUT}. In other words, ψ_{LUT} uses the same piecewise definition as ψ, but replaces $P(R = i \mid \omega)$ with the empirical estimates derived from the tables W and L.

Machine Learning-Based Approach

Although the lookup table-based approach provides a straightforward way to estimate the probability that a class will win given a voting vector ω, it can suffer from overfitting. Consider the case of $W(i,h,k) = 1$ and $L(i,h,k) = 0$, i.e., there was exactly one training sample where class i won while receiving h out of k votes. In this case, the probability estimation would be 1, even though this may be not very accurate.

As an alternative, we propose a machine learning-based approach that utilizes tables W and L to create a dataset that can then be used to learn a machine learning model. To predict the probability $P(R = i \mid \omega)$, the machine learning model uses the following three features:

- Feature 1: The class i.
- Feature 2: The number of votes class i received.
- Feature 3: The total number of votes it could have received, i.e., $k = \|\omega\|_1$.

To build the training dataset, we take each tuple (i,h,k) from W and create $W(i,h,k)$ copies, using (i,h,k) as the input features and assigning each copy a target label of 1. Similarly, for each (i,h,k) in L, we replicate it $L(i,h,k)$ times with the same input features (i,h,k) but a target label of 0. This way, the frequency of each partial voting scenario (and its outcome) is accurately reflected in the final training set.

Example 17. Given a random forest with 4 trees and three classes, given a sample where the decision trees predict the classes $V = (c_1, c_1, c_2, c_3)$, the training data would look as follows.

Feature 1 (i)	**Feature 2** (k)	**Feature 3** ($\omega_i^{\leq k}$)	**Target**
1	1	1	1
2	1	0	0
3	1	0	0
1	2	2	1
2	2	0	0
3	2	0	0
1	3	2	1
2	3	1	0
3	3	0	0
1	4	2	1
2	4	1	0
3	4	1	0

Given the training dataset, any supervised learning model f such as logistic regression, regression tree, or a neural network can be learned. The probability of class i winning given a partial voting vector $\omega^{\leq k}$ can then be estimated as follows:

$$f(i, \omega_i^{\leq k}, k) \approx P(R = i \mid \omega^{\leq k})$$

Algorithm 3 Abstract Interpretation of a Decision Tree [5]$_{\text{AP}}$.

```
1: procedure AbsIntDT(Node t, Path condition pc)
2:    if t is a leaf then
3:       return {t.class}
4:    end if
5:    if pc ⟹ (t.feature < t.threshold) then
6:       return AbsIntDT(t.true, pc)
7:    else if pc ⟹ ¬(t.feature < t.threshold) then
8:       return AbsIntDT(t.false, pc)
9:    else
10:      trueClasses ← AbsIntDT(t.true, pc)
11:      falseClasses ← AbsIntDT(t.false, pc)
12:      return trueClasses ∪ falseClasses
13:   end if
14: end procedure
```

In the same way as ψ_{LUT}, we can now instantiate ψ with this probability estimation to obtain a new probabilistic early termination function ψ_{ML}.

3.2.4 Abstract Early Stopping

Finally, we present an abstract-interpretation-based method [50] that preserves semantics even when fewer than half the votes have been collected. Previously, we mentioned that the voting vectors of an ADD are only reachable under certain conditions, and that these conditions restrict the possible votes that the remaining trees can cast. Instead of pessimistically assuming that at all times each decision tree may predict any class, we employ an analysis based on abstract interpretation [50]. Specifically, we use the conditions under which a voting vector can be reached to more precisely determine the maximum number of votes each class can potentially receive from the remaining trees. Although this analysis is computationally more expensive than our previously described semantics-preserving early stopping approaches, it enables early stopping before half of the trees have been aggregated while still preserving semantics.

The main idea is that given an ADD, for each path ending in a leaf with voting vector ω, we keep track of the path condition that leads to this path and then for each decision tree that still has to be aggregated, we infer which classes can possibly receive a vote. Algorithm 3 shows how to compute the set of reachable classes in a decision tree given a path condition.

The algorithm takes as input the node t of a decision tree and a path condition pc. If node t is a leaf, the algorithm returns the singleton set containing the leaf's class. Otherwise, the algorithm checks which branches are feasible given the path condition pc[5]. If only the true branch is satisfiable, i.e., $pc \implies (t.feature < t.threshold)$, the algorithm recurses on that branch and returns its set of reachable classes. Analogously, if only the false branch is satisfiable, i.e., $pc \implies \neg(t.feature < t.threshold)$, it recurses on the false branch and returns its result. If both branches are satisfiable, it combines the reachable classes of the true and false branches by taking their union.

Example 18. Figure 3.10 shows an ADD that represents the aggregation of the first two decision trees of a random forest, and the remaining three decision trees of a random forest that still have to be aggregated. For the four paths in the ADD, the three remaining decision trees would return

[5]Note that satisfiability checks and path condition tracking may be done in the same manner as for the infeasible path elimination discussed in Section 3.1.1. However, for clarity, we present a simplified approach here.

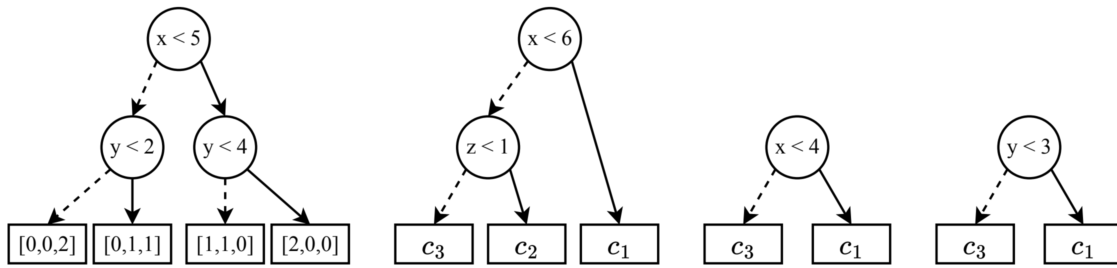


Figure 3.10: An ADD representing the aggregation of 2 decision trees, and 3 additional decision trees that still have to be aggregated.

the following reachable classes using Algorithm 3:

1. $x \geq 5 \wedge y \geq 2$: $\{c_1, c_2, c_3\}$, $\{c_3\}$, and $\{c_1, c_3\}$.
2. $x \geq 5 \wedge y < 2$: $\{c_1, c_2, c_3\}$, $\{c_3\}$, and $\{c_1\}$.
3. $x < 5 \wedge y \geq 4$: $\{c_1\}$, $\{c_1, c_3\}$, and $\{c_3\}$.
4. $x < 5 \wedge y < 4$: $\{c_1\}$, $\{c_1, c_3\}$, and $\{c_1, c_3\}$.

Algorithm 4 shows how Algorithm 3 can be used to apply early stopping before half of the trees have been aggregated while still preserving semantics. It takes as input a random forest $T_1, \ldots, T_n$, an index i representing the number of decision trees that have already been aggregated, a voting vector $votes$, and the path condition pc that led to the voting vector. The algorithm first initializes two arrays $safeVotes$ and $maybeVotes$ that are used to keep track for each class how many votes are guaranteed and how many votes are potentially received from the remaining decision trees $T_{i+1}, \ldots, T_n$.

Next, the algorithm calls ABSINTDT($T_j.root, pc$) for each decision tree T_j that has not yet been aggregated to determine the classes reachable under the given path condition pc. There are two possible cases:

- If exactly one class is reachable, it is guaranteed to receive a vote from that tree, so the algorithm increments the number of guaranteed votes for that class by 1 (Line 8).
- If more than one class is reachable, each of these classes could potentially receive a vote, so $maybeVotes$ is incremented by 1 for each such class (Line 11).

After computing $safeVotes$ and $maybeVotes$, the algorithm first determines the class with the highest amount of guaranteed votes (Lines 15 - 21). Specifically, it sums each class's current votes $votes[c]$ with its guaranteed votes $safeVotes[c]$ to find the maximum guaranteed total. Let that class be $idxMax$.

Finally, the algorithm checks if $idxMax$ can be overtaken by any other class when including potential votes. To do so, it verifies whether its guaranteed votes is strictly larger than $votes[c] + safeVotes[c] + maybeVotes[c]$ for every other class c. If so, the algorithm returns $idxMax$ (Line 30) indicating that $idxMax$ is going to win. Otherwise, the algorithm returns "unsure" (Line 26) indicating that additional votes are needed before a decision can be made.

The following examples illustrate how Algorithm 4 operates on the ADD and decision trees shown in Figure 3.10.

Example 19. For the path $x < 5 \wedge y < 4$, we have already determined that the set of reachable classes are $\{c_1\}$, $\{c_1, c_3\}$, and $\{c_1, c_3\}$ for the remaining three trees. Algorithm 4 uses these sets to compute:

$$safeVotes = [1, 0, 0], \quad maybeVotes = [2, 0, 2].$$

Algorithm 4 Abstract Early Stopping.

```
1:  procedure AbsES(Random Forest T_1, ..., T_n, Index i, Array votes, Path condition pc)
2:      Initialize: Array safeVotes[1 ... K] of integers with 0
3:      Initialize: Array maybeVotes[1 ... K] of integers with 0
4:      for j ← i + 1 to n do
5:          reachableClasses ← AbsIntDT(T_j.root, pc)
6:          if |reachableClasses| = 1 then
7:              c ← getSingleElement(reachableClasses)
8:              safeVotes[c] = safeVotes[c] + 1
9:          else
10:             for c ∈ reachableClasses do
11:                 maybeVotes[c] = maybeVotes[c] + 1
12:             end for
13:         end if
14:     end for
15:     maxSafe, idxMax ← −∞, −1
16:     for c ← 1 to K do
17:         totalVotesSafe ← votes[c] + safeVotes[c]
18:         if totalVotesSafe > maxSafe then
19:             maxSafe, idxMax ← totalVotesSafe, c
20:         end if
21:     end for
22:     for c ← 1 to K do
23:         if c ≠ idxMax then
24:             totalVotesMaybe ← votes[c] + safeVotes[c] + maybeVotes[c]
25:             if totalVotesMaybe ≥ maxSafe then
26:                 return "unsure"
27:             end if
28:         end if
29:     end for
30:     return idxMax
31: end procedure
```

Given that $x < 5 \wedge y < 4$ leads to the voting vector $[2, 0, 0]$, class c_1 is guaranteed to receive at least 3 votes. Since this already exceeds half of the trees, c_1 can be declared the winner without further aggregation.

Example 20. For the path $x < 5 \wedge y \geq 4$, the remaining three trees have the following reachable classes: $\{c_1\}$, $\{c_1, c_3\}$, and $\{c_3\}$. Algorithm 4 uses these sets to compute:

$$\mathit{safeVotes} = [1, 0, 0], \quad \mathit{maybeVotes} = [1, 0, 2].$$

Given that $x < 5 \wedge y < 4$ leads to the voting vector $[1, 1, 0]$, class c_1 is guaranteed to receive at least 2 votes, but class c_3 might receive 2 votes as well. Because no class is definitively in the lead, early stopping is not possible, and the algorithm returns "unsure".

While Algorithm 4 only declares a class as the winner when it is guaranteed to win, it may still return "unsure" in situations where a winner could be determined. This is a consequence of the algorithm processing the remaining decision trees independently and assuming that each class might receive all of its potential votes, even though this may not be possible in practice. One

could of course design a complete method that accounts for dependencies between the remaining decision trees, though this comes at the expense of increased computational overhead, which may not justify the gain in precision.

Although abstract early stopping is more precise than our "general" and "specific" rules presented in Section 3.2.2, it is also computationally more expensive. When we take a look at Algorithm 4, we can see that the algorithm basically consists of three for-loops. The second and third for-loops run exactly K times and only perform constant time operations inside the loop, so their time complexity $O(K)$. The first loop runs $O(n)$ times and inside the loop the ABSINTDT function is called. The function ABSINTDT visits each reachable leaf in the given tree T_j, which has time complexity $O(|T_j|)$ in the worst case, if all leaves are reachable. Inside the for loop there is another for loop that loops over the reachable classes but the number of reachable classes can at most be as large as the number of leaves in T_j. The runtime complexity of the outer loop is $O(n \cdot k)$ where k is the size of the decision tree with the largest size.

Because abstract early stopping requires the path condition, it is best applied during the infeasible path elimination, which already traverses all paths and keeps track of their conditions. In Algorithm 1, when the current node is a leaf, ABSES is called (Line 7). If ABSES can determine a winner, it returns a voting vector with that class's entry set to $\top$. Otherwise, if it returns "unsure", the leaf node remains unchanged. By integrating abstract early stopping with the infeasible path elimination, we avoid traversing all paths in the ADD a second time.

3.3 Evaluation: Transforming Random Forests into ADDs

In this section, we evaluate the ADD-based approach presented in the previous section for transforming random forests into ADDs. We evaluated our approach on a machine with an Intel(R) Xeon(R) Gold 6152 CPU 2.10 GHz with 502 GB of RAM. We implemented our approach in Java using the ADD-Lib [51], a Java library for decision diagrams. The random forests used for this evaluation are exactly those that were used in the evaluation of $[4]_{\text{AP}}$. Essentially, we are performing the same evaluation with the difference that we now additionally evaluate the early stopping approach based on abstract interpretation presented in Section 3.2.4.

Table 3.1 summarizes the datasets and the corresponding random forests, which were trained with 15, 25, 50, or 100 trees, each with a maximum depth of 4, and an 80%/20% train/test split.

We perform measurements with 6 different configurations:

1. *baseline*: Implements the procedure from Section 3.1 (Algorithm 2).
2. *GEN*: Extends *baseline* with the *general rules* (Section 3.2.2).
3. *SPC*: Extends *baseline* with the *specific rules* (Section 3.2.2).
4. *AbsES*: Extends *baseline* with *abstract interpretation* (Section 3.2.4).
5. *LUT*: Extends *baseline* with the *lookup table* (Section 3.2.3), setting $\alpha = 1$ and $\beta = 0$ in the function ψ_{LUT}.
6. *ML*: Extends *baseline* with the machine learning based approach (Section 3.2.3), using an XGBoost [17] model trained on the partial voting vectors. Here, we set $\alpha = 0.999$ and $\beta = 0.001$ in ψ_{ML}.

3.3.1 Experimentation Results

We address the following research questions to evaluate our approach:

Table 3.1: Overview of datasets and the compiled ADD (#F = Number of features, #I = Number of test instances, #C = Number of classes, #T = Number of trees, #N = Number of nodes in the random forest, #P = Number of unique predicates, D = Maximum depth, %A = Accuracy of the random forest on test set).

Dataset	#F	#I	#C	#T	#N	#P	D	%A
ann-thyroid	21	1426	3	25	555	146	4	98
appendicitis	7	22	2	50	722	207	4	90
banknote	4	270	2	100	1998	614	4	97
ecoli	7	66	5	100	2532	379	4	90
glass2	9	33	2	25	445	159	4	87
ionosphere	34	70	2	15	247	101	4	87
iris	4	30	2	100	1200	94	4	93
magic	10	3781	2	25	747	349	4	82
mofn-3-7-10	10	205	2	100	2904	10	4	85
new-thyroid	3	43	3	100	1452	237	4	100
phoneme	5	1080	2	100	2836	957	4	78
ring	20	1480	2	25	625	287	4	83
segmentation	19	42	7	15	329	148	4	92
shuttle	9	11600	7	50	1296	205	4	99
threeOf9	9	103	2	100	1364	9	4	100
twonorm	29	1480	2	15	465	225	4	90
waveform-21	21	1000	3	15	465	214	4	80
wine-recog	13	36	3	25	399	152	4	97
xd6	9	103	2	100	2904	9	4	90

- **RQ1: Do our optimizations reduce the size of the voting ADDs?**
- **RQ2: Do our optimizations reduce the time it takes to compile a random forest into an ADD?**
- **RQ3: Do our probabilistic optimizations preserve the accuracy of the original random forest?**

RQ1: A key motivation for employing early stopping is to limit the size of intermediate ADDs, which are larger than the final majority vote ADD. Constraining their size not only lowers peak memory usage but also reduces overall computation time.

To address **RQ1**, we measured the size of the intermediate ADDs after each aggregation step, specifically, once infeasible paths had been eliminated (see Line 13 in Algorithm 2). Figure 3.11 shows the number of nodes in the vector ADD for each configuration at this point in the process.

Under the *baseline* configuration, the intermediate ADD tends to grow consistenly as additional ADDs are joined. In contrast, our five proposed optimizations almost always produce smaller intermediate ADDs across all datasets. In particular, for *GEN* and *SPC*, the ADD size only starts to shrink once at least half of the trees have been joined, because the general and specific rules cannot be applied earlier. On the other hand, *LUT*, *ML*, and *AbsES* often provide substantial reductions before half of the trees are joined. For *LUT* and *ML*, this is attributed to the high probability of a class winning when it received a majority of the initial votes. For *AbsES*, this shows that our abstract interpretation can indeed often determine the winner before half of the trees have been joined.

For many datasets, e.g., *banknote*, *ecoli*, *iris*, *mofn-3-7-10*, *new-thyroid*, *phoneme*, *threeOf9*, and *xd6*, our optimizations actually reduce the size of the ADD as more ADDs are joined. For other datasets, although the ADD may still grow, it does so more slowly and remains considerably smaller than under the *baseline*. Interestingly, applying *LUT* often stabilizes the ADD size or

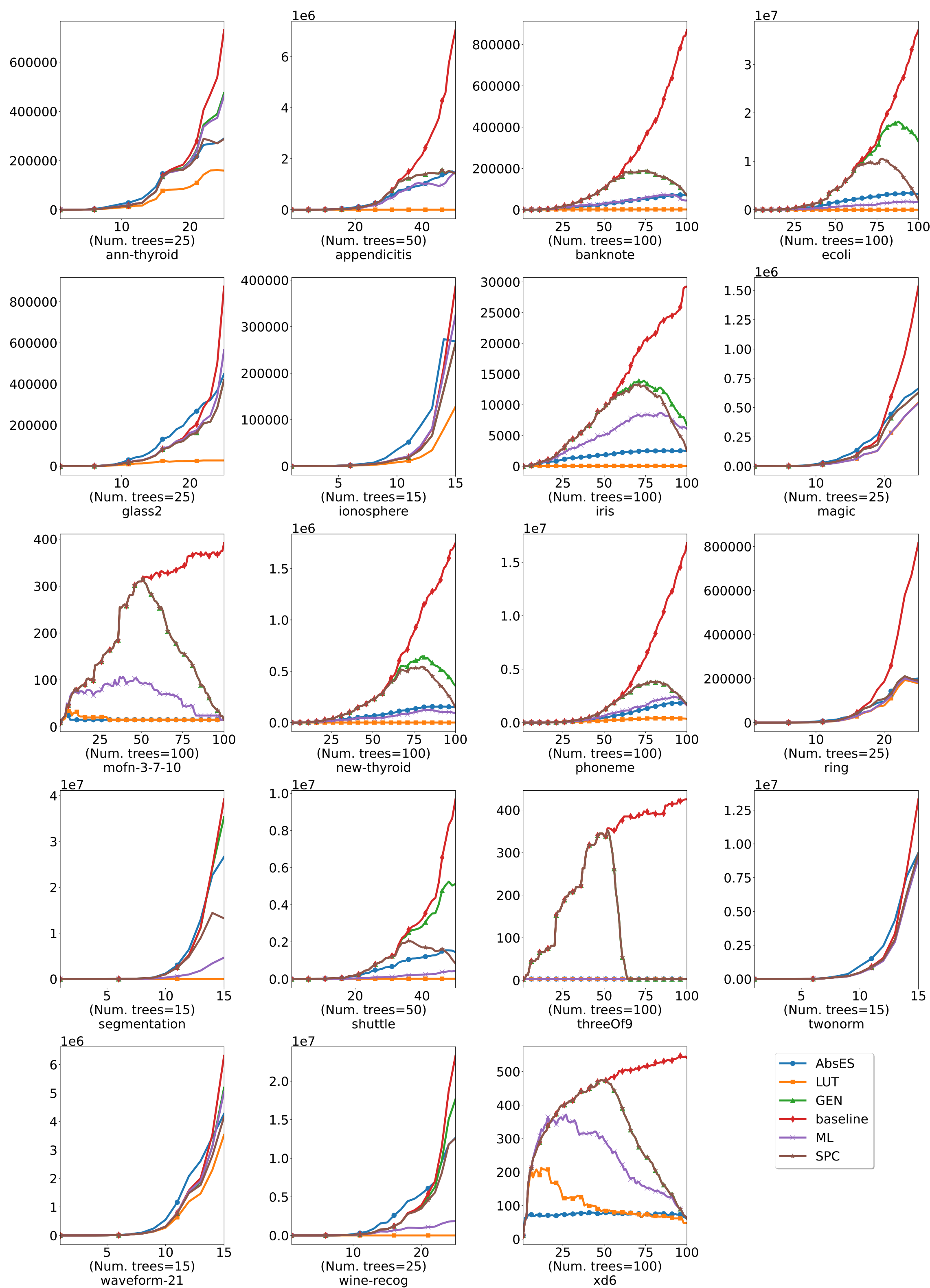


Figure 3.11: Size of intermediate ADDs during the aggregation process.

only increases it marginally (see *appendicitis*, *banknote*, *glass2*, *heart-c*, *iris*, *mofn-3-7-10*, *new-thyroid*, *threeOf9*). Overall, these size reductions greatly decrease peak memory usage and show the effectiveness of our optimizations.

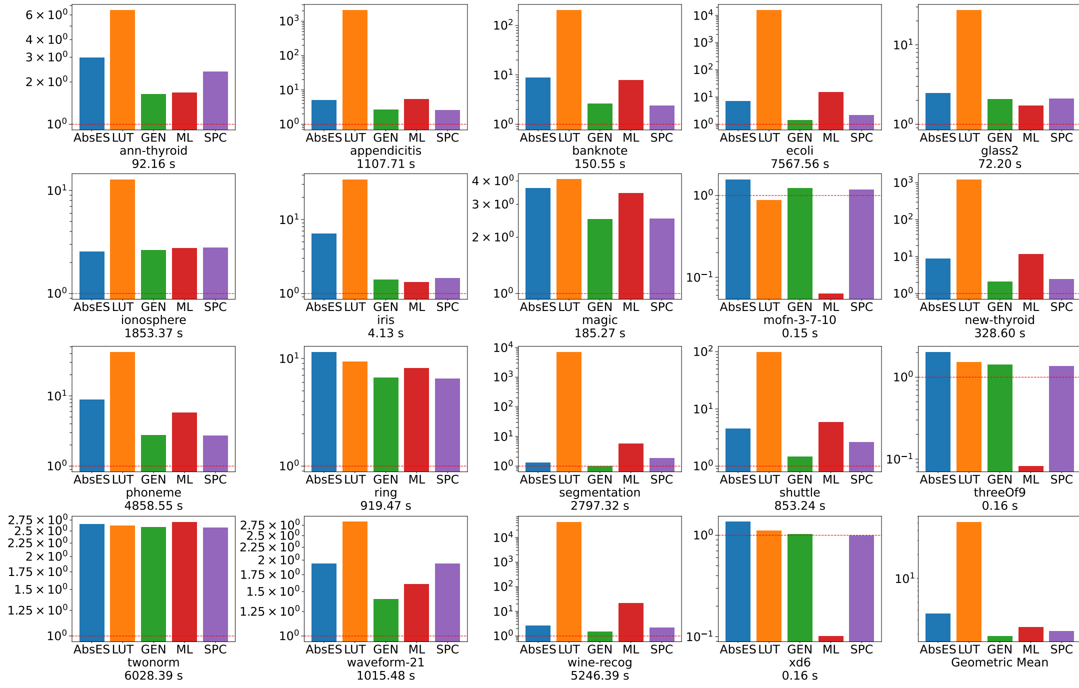


Figure 3.12: Speedups over baseline for the transformation of the random forest to the ADD.

Appendix B, Section B.1, presents detailed results on the class characterization sizes for each class and configuration.

RQ2: Although we have observed that our optimizations reduce the size of the intermediate ADDs, we also need to determine whether these size reductions translate into faster transformation times. Because ADD aggregation scales polynomially with ADD size, it is reasonable to expect that smaller intermediate ADDs will result in shorter transformation times.

Figure 3.12 presents the speedups achieved by each optimization relative to the baseline. For *LUT*, the execution time includes the overhead of constructing the lookup table, and for *ML*, it includes the overhead of training the XGBoost model. Overall, reducing ADD size does indeed lead to improved execution times: the geometric mean speedups for *AbsES*, *LUT*, *GEN*, *ML*, and *SPC* are 3.62×, 51.63×, 1.89×, 2.44×, and 2.18×, respectively.

For many datasets, *LUT* achieves the highest speedup of around 10^3 for *appendicitis* and *new-thyroid*, due to the relatively stable size of intermediate ADDs (see Figure 3.11). The *AbsES* early stopping method, which uses abstract interpretation, generally outperforms both *GEN* and *SPC* because it is more precise and can therefore stop earlier. However, for *segmentation*, *SPC* is slightly better than *AbsES*, indicating that the additional cost of *AbsES*'s precision may not always pay off in terms of speed.

In datasets with more than two classes, *SPC* typically outperforms *GEN*. Meanwhile, if we disregard datasets with very low ADD construction times, *ML* is generally competitive with or slightly better than *SPC*, though not by a large margin.

Finally, *ML* performs poorly on *mofn-3-7-10*, *threeOf9*, and *xd6*, because the overhead of learning the XGBoost model exceeds the baseline ADD transformation time, making it slower than simply using the baseline method.

These results show that our optimizations not only reduce the size of the intermediate ADDs but also accelerate the transformation process.

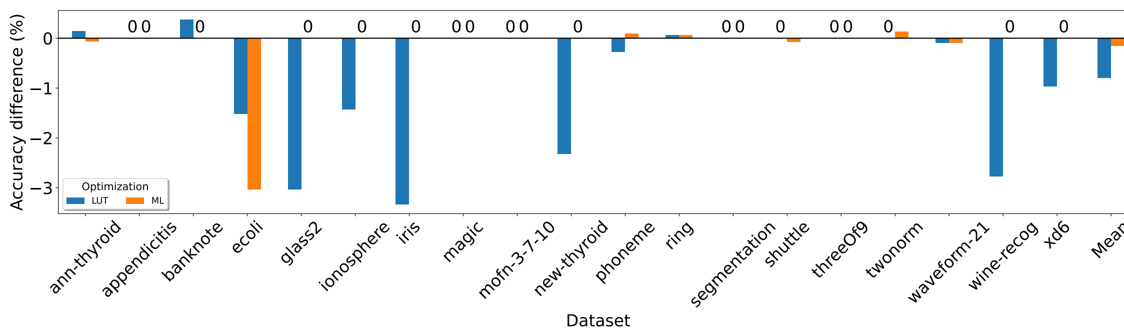


Figure 3.13: Difference in test accuracy compared to the random forest [4]$_{\text{AP}}$.

RQ3: Given the significant speedups provided by *LUT* and *ML*, we must ensure that these probabilistic optimizations do not compromise the accuracy of the original random forest. Figure 3.13 shows the difference in test-set accuracy after applying *LUT* and *ML*[6].

On average, *LUT* and *ML* introduce accuracy differences of -0.8% and -0.16%, respectively. In five datasets, both optimizations result in no loss in accuracy at all, and in six additional datasets, *ML* also causes no drop in accuracy. Overall, the loss in accuracy is relatively small for most datasets.

In some datasets, however, the accuracy loss exceeds 1%. These cases involve very few test samples (fewer than 100), so even a small number of misclassifications can cause a noticeably larger difference in percentage terms. Interestingly, for certain datasets, e.g., *ann-thyroid*, *banknote*, *ring*, our probabilistic optimizations slightly improve accuracy. This gain can occur if the probabilistic simplifications result in a model that generalizes better to unseen data.

In general, there is a trade-off between improving performance (e.g., reducing ADD construction time) and maintaining accuracy. Increasing the aggressiveness of the probabilistic early stopping strategy by decreasing α and/or increasing β can deliver further performance benefits but may also lead to larger accuracy drops. Therefore, these parameters must be tuned carefully to balance speed and accuracy in a way that is acceptable for the target application.

3.4 Transforming Gradient Boosted Trees into Algebraic Decision Diagrams

In this section, we extend the approach introduced in the previous sections to gradient boosted trees. We begin by giving an overview of the transformation procedure, explain how the early stopping approach (cf. Section 3.1) can be generalized to gradient boosted trees, and finally evaluate our method. We explicitly separate the binary and multiclass classification scenarios, because they require different transformations and allow for individual optimization techniques.

3.4.1 Binary Classification

For binary classification, gradient boosted trees consist of n decision trees $T_1, \dots, T_n$, where each leaf contains a real-valued score. The model prediction is obtained by summing the leaf values across all trees and then applying the sigmoid function

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

[6]Since *AbsES*, *GEN*, and *SPC* preserve the exact semantics of the random forest, their accuracy difference is always 0%.

to convert the summed score into a probability. If the sigmoid function returns a value greater than 0.5, the instance is classified as class 1, otherwise as class 0.

The main idea of the transformation is to convert each decision tree T into an ADD $\mathcal{A}$ and then sum the ADDs to obtain a single ADD representing the entire gradient boosted ensemble. Afterwards, the sigmoid function is applied to the resulting ADD, and subsequently a function that maps the real-valued output to a binary class label is applied.

Each T_i can be transformed into an ADD $\mathcal{A}_i$ by following the same procedure as for random forests (see Section 3.1), with the difference that the leaf nodes contain real-valued scores instead of voting vectors. After transforming each T_i into an ADD $\mathcal{A}_i$, a single ADD $\mathcal{A}^+$ is created such that

$$\mathcal{A}^+(\vec{x}) = w(\vec{x}) = \sum_{i=1}^{n} T_i(\vec{x}) \; .$$

This ADD is obtained by adding the ADDs of the individual trees:

$$\mathcal{A}^+ = \mathcal{A}_1 +_A \ldots +_A \mathcal{A}_n$$

where $+_A$ is standard addition lifted onto ADDs. As with the aggregation of ADDs for a random forest, we can apply infeasible path elimination immediately after each aggregation step.

Once $\mathcal{A}^+$ has been constructed, a probabilistic output is derived by applying the sigmoid function, as follows:

$$\mathcal{A}^\sigma(\vec{x}) = \sigma_A(\mathcal{A}^+(\vec{x})) = \frac{1}{1 + e^{-\mathcal{A}^+(\vec{x})}} \; .$$

Subsequently, a function that maps the real-valued output to a binary class label is applied to the resulting ADD:

$$\mathcal{G}(z) = \begin{cases} 1 & \text{if } z > 0.5, \\ 0 & \text{otherwise.} \end{cases}$$

However, because $\sigma(z) > 0.5$ precisely when $z > 0$, the separate sigmoid-then-threshold step can be replaced by a single application of the Heaviside (step) function H, defined as

$$H(z) \;=\; \begin{cases} 1 & z > 0, \\ 0 & \text{otherwise.} \end{cases}$$

Therefore, we may write

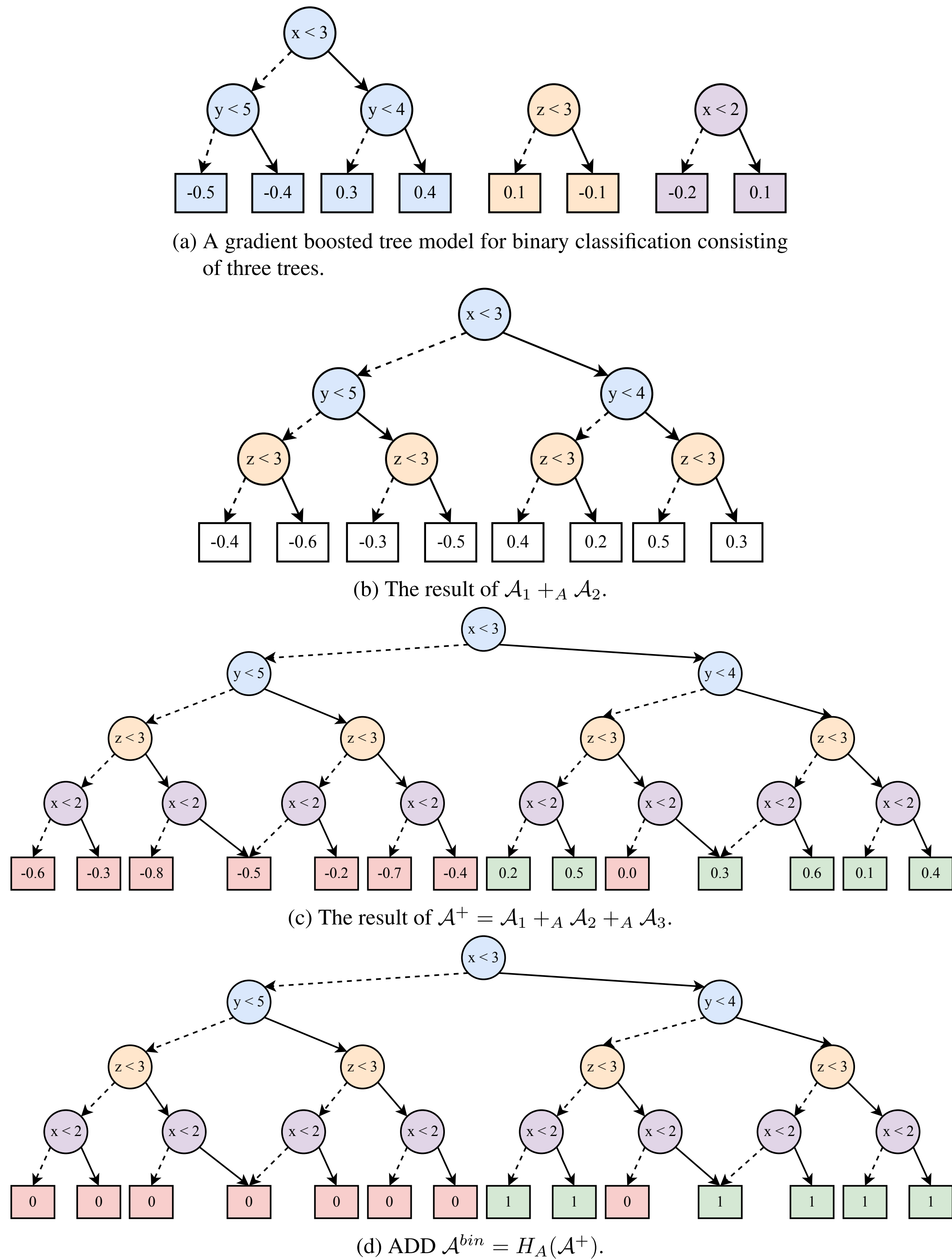
$$\mathcal{A}^{bin}(\vec{x}) \;=\; H_A\big(\mathcal{A}^+(\vec{x})\big),$$

where $\mathcal{A}^+(\vec{x})$ is the summed score of all trees. By applying H as a unary function directly to $\mathcal{A}^+$, we obtain an ADD that outputs the class label (0 or 1) for each input $\vec{x}$. This "one-step" thresholding simplifies both the conceptual and computational process.

Example 21. To illustrate the approach, consider the gradient boosted tree model in Figure 3.14a consisting of three trees for binary classification. These trees T_1, T_2, and T_3, can straightforwardly be transformed into ADDs $\mathcal{A}_1$, $\mathcal{A}_2$, and $\mathcal{A}_3$. Because none of the ADD reduction or reordering rules apply in this particular case, the resulting ADDs maintain the same structure as their respective trees.

Figure 3.14b shows the result of adding $\mathcal{A}_1$ and $\mathcal{A}_2$ using the operator $+_A$. This ADD returns the sum of weights from T_1 and T_2 for any given input. Next, ADD $\mathcal{A}_3$ is added to this intermediate ADD, resulting in $\mathcal{A}^+$ (Figure 3.14c).

Finally, we can apply the Heaviside function H to $\mathcal{A}^+$ resulting in ADD $\mathcal{A}^{bin}$ (Figure 3.14d). For any given input, $\mathcal{A}^{bin}$ returns either 0 or 1, thus reproducing the original gradient boosted



(a) A gradient boosted tree model for binary classification consisting of three trees.

(b) The result of $\mathcal{A}_1 +_A \mathcal{A}_2$.

(c) The result of $\mathcal{A}^+ = \mathcal{A}_1 +_A \mathcal{A}_2 +_A \mathcal{A}_3$.

(d) ADD $\mathcal{A}^{bin} = H_A(\mathcal{A}^+)$.

Figure 3.14: Visualization of the intermediate results of Algorithm 5.

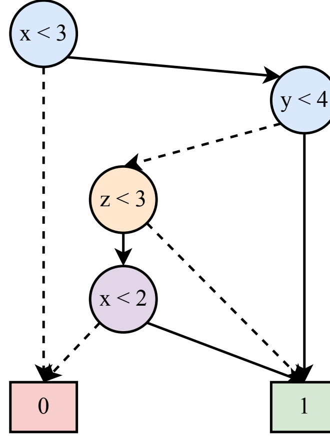


Figure 3.15: The ADD of Figure 3.14d after it has been reduced.

tree model's prediction. Figure 3.15 shows the same ADD after reduction, illustrating how much smaller it becomes once redundant nodes are eliminated.

Algorithm 5 combines all steps discussed so far into a single procedure, that takes gradient boosted trees $T_1, \ldots, T_n$ as input, and outputs $\mathcal{A}^{bin}$.

3.4.2 Multiclass Classification

We now consider the multiclass classification setting, where a gradient boosted model consists of n decision trees *per* class:

$$T_1^1, \ldots, T_n^1, \ \ldots, \ T_1^K, \ldots, T_n^K.$$

Each group of n trees is dedicated to predicting the score for one of the K classes. Just as for binary classification, each decision tree T_i^j is transformed into an ADD $\mathcal{A}_i^j$. It is also straightforward to add the ADDs of the individual trees for each class to obtain a single ADD $\mathcal{A}_j^+$ representing the gradient boosted trees for class j:

$$\mathcal{A}_j^+ = \mathcal{A}_1^j +_A \mathcal{A}_2^j +_A \cdots +_A \mathcal{A}_n^j \ .$$

After obtaining the ADDs $\mathcal{A}_j^+$ for each class $j \in [K]$, the next step is to construct a single ADD $\mathcal{A}^K$ that returns the class with the highest total score in the gradient boosted classifier $\mathfrak{C}$:

$$\mathcal{A}^K(\vec{x}) = \mathfrak{C}(\vec{x}) = \arg\max_{j \in [K]} w_j(\vec{x}) \ .$$

There are different ways to achieve this. One way would be to first combine the ADDs $\mathcal{A}_j^+$ into a single ADD where each leaf contains a vector of scores, with one score per class. Then, the argmax function is applied to the resulting ADD. Another way would be to apply the argmax function iteratively to the ADDs $\mathcal{A}_j^+$. This can be done by applying the argmax function to the first two ADDs, then applying the argmax function to the result and the next ADD, and so on. The leaves of the resulting ADDs need to contain the class label with the highest score and the score itself. This second approach is more efficient in terms of memory and computation time.

To implement the iterative method, one first adapts $\mathcal{A}_j^+$ so that it stores both the score for class j and the class label itself. Next, a function $\circ$ is defined that, given two tuples (w_1, c_1) and

Algorithm 5 Transforming gradient boosted trees for binary classification into an algebraic decision diagram.

```
1: procedure BINARYBOOST2ADD(Gradient Boosted Trees T_1, ..., T_n)
2:   // Step 1: Convert each tree into an ADD
3:   for i ← 1 to n do
4:     A_i ← DT2ADD(T_i)
5:   end for
6:
7:   // Step 2: Sum the individual ADDs
8:   A^+ ← A_1
9:   for i ← 2 to n do
10:    A^+ ← A^+ +_A A_i
11:    // Eliminate infeasible paths (cf. Algorithm 1).
12:    A^+ ← ELIMINFEASPATHS(A^+)
13:  end for
14:
15:  // Step 3: Apply the Heaviside function to obtain a binary classification ADD
16:  A^bin ← H_A(A^+)
17:  return A^bin
18: end procedure
```

(w_2, c_2), returns the tuple with the larger score:

$$\circ\big((w_1, c_1), (w_2, c_2)\big) = \begin{cases} (w_1, c_1) & \text{if } w_1 \geq w_2, \\ (w_2, c_2) & \text{otherwise.} \end{cases}$$

We lift this function to ADDs and apply it to the ADDs $\mathcal{A}_j^+$ iteratively:

$$\mathcal{A}^{wK} = \mathcal{A}_1^+ \circ_A \mathcal{A}_2^+ \circ_A \ldots \circ_A \mathcal{A}_K^+$$

ADD $\mathcal{A}^{wK}$ returns for each instance the class label with the highest score and the score itself. Since the specific score is not needed once we have identified the highest-scoring class, the score can be discarded by lifting the $◑_A$ function to $\mathcal{A}^{wK}$, which only keeps the class label:

$$\mathcal{A}^K = ◑_A(\mathcal{A}^{wK})$$

Having obtained ADD $\mathcal{A}^K$, we can use the techniques described in Section 3.1.2 to create class characterizations for all classes.

Algorithm 6 summarizes the transformation of gradient boosted trees for multiclass classification into an ADD. Note that infeasible paths are eliminated in Line 13 and in Line 20 as this keeps the ADDs small and accelerates the transformation process.

3.5 Early Stopping for Gradient Boosted Trees

One way to reduce the computational cost of transforming boosted trees into ADDs is to stop the transformation process early for certain inputs, similar to the strategy used for random forests in Section 3.2. However, when dealing with gradient boosted trees, each leaf contains a real-valued weight (or "score"), and therefore the implementation of early stopping requires slightly different considerations.

Algorithm 6 Transforming Gradient Boosted Trees for Multiclass Classification into an ADD

1: **procedure** MULTICLASSBOOST2ADD($\{T_1^j, \ldots, T_n^j\}_{j=1}^K$)
2: *// Step 1: Convert each tree into an ADD*
3: **for** $j \leftarrow 1$ **to** K **do**
4: **for** $i \leftarrow 1$ **to** n **do**
5: $\mathcal{A}_i^j \leftarrow \text{DT2ADD}(T_i^j)$
6: **end for**
7: **end for**
8: *// Step 2: Sum the ADDs for each class*
9: **for** $j \leftarrow 1$ **to** K **do**
10: $\mathcal{A}_j^+ \leftarrow \mathcal{A}_1^j$
11: **for** $i \leftarrow 2$ **to** n **do**
12: $\mathcal{A}_j^+ \leftarrow \mathcal{A}_j^+ \ +_A \ \mathcal{A}_i^j$
13: $\mathcal{A}_j^+ \leftarrow$ ELIMINFEASPATHS($\mathcal{A}_j^+$)
14: **end for**
15: **end for**
16: *// Step 3: Compute* $\arg\max$ *among classes*
17: $\mathcal{A}^{wK} \leftarrow \mathcal{A}_1^+$
18: **for** $j \leftarrow 2$ **to** K **do**
19: $\mathcal{A}^{wK} \leftarrow \mathcal{A}^{wK} \ \circ_A \ \mathcal{A}_j^+$
20: $\mathcal{A}^{wK} \leftarrow$ ELIMINFEASPATHS($\mathcal{A}^{wK}$)
21: **end for**
22: $\mathcal{A}^K \leftarrow ◑_A(\mathcal{A}^{wK})$
23: **return** $\mathcal{A}^K$
24: **end procedure**

3.5.1 Early Stopping for Binary Classification

Given gradient boosted trees $T_1, \ldots, T_n$, we define the sum of the first m trees' scores for an input $\vec{x}$ as

$$S_m(\vec{x}) \ = \ \sum_{i=1}^{m} T_i(\vec{x}), \quad \text{for } m = 1, \ldots, n.$$

In binary classification, the final prediction is determined by the sign of $S_n(\vec{x})$: if $S_n(\vec{x}) > 0$, the model predicts class 1 and class 0 otherwise.

During inference, we do not necessarily need to evaluate all n trees if we can determine the final sign of $S_n(\vec{x})$ in advance. Suppose that at step m (having evaluated $T_1(\vec{x}), T_2(\vec{x}), \ldots, T_m(\vec{x})$), we know that:

$$S_m(\vec{x}) + \underline{W}(m+1) \ > \ 0,$$

where $\underline{W}(m+1)$ is the minimal possible sum of the remaining trees $T_{m+1}, \ldots, T_n$. In this case, even under a worst-case scenario for the remaining trees (i.e., each contributing its minimal leaf weight), the total score would still exceed 0. Early stopping can therefore be applied, and the prediction is class 1. On the other hand, if

$$S_m(\vec{x}) + \overline{W}(m+1) \ \leq \ 0,$$

where $\overline{W}(m+1)$ is the maximal possible sum of the remaining trees, then the total weight will not exceed 0, so class 0 can be predicted without evaluating further trees.

Algorithm 7 Computing the minimum and maximum reachable weights in a tree.

```
1:  procedure AbsMinMax(Current node t in tree T, Path condition pc)
2:      if t is a leaf then
3:          return (t.weight, t.weight)
4:      else
5:          if pc ⟹ (t.feature < t.threshold) then
6:              return AbsMinMax(t.true, pc)
7:          else if pc ⟹ ¬(t.feature < t.threshold) then
8:              return AbsMinMax(t.false, pc)
9:          else
10:             (minTrue, maxTrue) ← AbsMinMax(t.true, pc)
11:             (minFalse, maxFalse) ← AbsMinMax(t.false, pc)
12:             return (min(minTrue, minFalse), max(maxTrue, maxFalse))
13:         end if
14:     end if
15: end procedure
```

Each tree T_i is first analyzed to compute its minimum and maximum leaf weights:

$$\underline{w}_i \;=\; \min_{\text{leaf}\in T_i}\{\text{leaf.weight}\}, \quad \overline{w}_i \;=\; \max_{\text{leaf}\in T_i}\{\text{leaf.weight}\}.$$

Subsequently, the following suffix sums are precomputed once, prior to initiating the transformation process:

$$\underline{W}(i) \;=\; \sum_{k=i}^{n} \underline{w}_k, \quad \overline{W}(i) \;=\; \sum_{k=i}^{n} \overline{w}_k.$$

and stored for each $i \in [n]$. Constructing these values has time complexity $O(n \cdot k)$, where k is the maximum size of any single tree (since determining each $\underline{w}_i$ and $\overline{w}_i$ requires $O(k)$, and building the suffix arrays requires $O(n)$). To apply early stopping during the transformation, one can adapt $+_A$ to return $\top$ if $S_m(\vec{x}) + \underline{W}(m+1) \;>\; 0$, and $\bot$ if $S_m(\vec{x}) + \overline{W}(m+1) \;\leq\; 0$, holds.

3.5.2 Abstract Early Stopping for Binary Classification

While the early stopping approach described in the previous section is inexpensive computationally, it could be more precise. Similarly to the abstract early stopping in random forests presented in Section 3.2.4, abstract interpretation can be employed to determine whether early stopping can be applied more precisely. The main idea is to consider the path condition when computing the minimum and maximum weights that can be reached in the remaining trees instead of computing $\overline{W}$ and $\underline{W}$ once at the beginning. The motivation behind this is that we always assume the worst-case scenario when computing $\overline{W}$ and $\underline{W}$, i.e., we assume that it is possible to reach all minimum and maximum weights in the remaining trees.

Algorithm 7 shows how to compute the minimum and maximum reachable weights in a tree T given a path condition pc. If the current node t is a leaf, the minimum and maximum reachable weights are both equal to the weight of the leaf. Otherwise, if it is possible to infer from the path condition if the check of the feature is true ($t.feature < t.threshold$) or false ($t.feature \geq t.threshold$), the algorithm recursively calls itself with the corresponding child node. If it is not possible to infer either case, the algorithm calls itself with both children and combines the results.

To integrate early stopping using this abstract approach, each path from the root to a leaf is traversed while tracking the path condition, analogous to infeasible path elimination (see Sec-

tion 3.1.1). When a leaf node is reached, the maximum and minimum sums of weights in the remaining trees are computed by calling Algorithm 7 for each unprocessed tree. The early stopping conditions from Section 3.5.1 can then be checked. For efficiency reasons, the infeasible path elimination can be combined with the application of abstract early stopping since the infeasible path elimination has to visit all paths anyway and we apply the infeasible path elimination after each aggregation.

3.5.3 Early Stopping for Multiclass Classification

This section discusses early stopping for multiclass classification. The ADDs $\mathcal{A}_j^+$ can become quite large, especially when the number of trees per class or the number of nodes in the trees is high. Therefore it is crucial to use early stopping to stop the transformation early if possible. In the multiclass setting, only a path condition-based early stopping strategy is presented here, leveraging path conditions to refine the bounds on reachable weights.

Recall that the following computation must be performed:

$$\mathcal{A}^{wK} = \mathcal{A}_1^+ \circ_A \mathcal{A}_2^+ \circ_A \ldots \circ_A \mathcal{A}_K^+.$$

Remember that we adapt $\mathcal{A}_i^+$ to not only return the weight but also the class. We can think of the resulting ADD of $\mathcal{A}_1^+ \circ_A \mathcal{A}_2^+$ as the ADD that for an input returns the class with the larger weight and the weight itself. If it can be shown that one class's weight exceeds all reachable weights in the remaining ADDs, that class is guaranteed to win. Similarly, if all remaining classes have weights provably greater, the current class can be discarded.

If we want to know whether all reachable weights in the remaining ADDs are larger or smaller, we can also just compute the minimum and maximum reachable weight in the remaining ADDs. To compute the maximum and minimal reachable weight in an ADD given a path condition, we can simply adapt Algorithm 7 to work on ADDs. Algorithm 8 shows how given the leaf (w, c) of the ADD $\mathcal{A}_1^+ \circ_A \mathcal{A}_2^+ \circ_A \ldots \circ_A \mathcal{A}_i^+$, and the path condition leading to that leaf, one can check if we can apply early stopping. The algorithm determines whether all reachable weights in the remaining ADDs are smaller or larger. If all weights are smaller, then we know that the class c will definitely win, so the algorithm returns $\top_c$. If all weights are larger, we know that some other class among $i + 1 \ldots K$ will win, so the algorithm returns $\bot$. Otherwise, the algorithm returns (w, c) indicating that early stopping cannot be applied yet.

For ADDs, early stopping for multiclass classification without the path condition is not particularly beneficial. In principle, one could apply early stopping in two ways:

- During the aggregation, i.e., while computing of $\mathcal{A}_1^+ \circ_A \mathcal{A}_2^+$. However, this would require a custom ADD implementation specifically tailored to this use case, which is not practical due to its complexity

- After the aggregation, which, without a path condition, would be very imprecise and therefore not particularly helpful. Without a path condition, one must assume all weights are reachable at all times, meaning early stopping is only applicable if one weight is strictly smaller than all others across all classes (or vice versa).

Though, in Chapter 4 we present an alternative approach that does not make use of ADDs that allows for the application of early stopping for multiclass classification without using the path condition.

Algorithm 8 Early Stopping for Multiclass Classification

```
1: procedure ADDMcEarlyStopping(ADDs A_1^+, ..., A_K^+, Index i, Path Condition pc,
   Weight w, Class c)
2:   allSmaller, allLarger ← True, True
3:   for j ← i + 1 to K do
4:     (absMin, absMax) ← AbsMinMax(A_j^+, pc)
5:     allSmaller ← allSmaller and (absMax < w)
6:     allLarger ← allLarger and (absMin > w)
7:     if ¬allSmaller and ¬allLarger then
8:       break
9:     end if
10:  end for
11:  if allSmaller then
12:    // Class c is the winner
13:    return ⊤_c
14:  end if
15:  if allLarger then
16:    // Another class among i + 1 ... K will definitively win
17:    return ⊥
18:  end if
19:  return (w, c)
20: end procedure
```

3.6 Evaluation: Transforming Gradient Boosted Trees into ADDs

In this section we evaluate our approach for transforming gradient boosted trees into ADDs. We evaluated our approach on a machine with an Intel(R) Xeon(R) Gold 6152 CPU 2.10 GHz with 502 GB of RAM. Our implementation is written in Java using the ADD-Lib [51], a java library for decision diagrams.

Each gradient boosted model was trained with 50 trees per class using an 80%/20% train/test split. The maximum depth is set to either 3 or 4 for all gradient boosted tree models. Table 3.2 provides an overview of the datasets and the learned tree ensembles.

Experimentation Results

In this evaluation, we evaluate the time it takes to transform the gradient boosted tree models into ADDs, and the sizes of the ADDs. We perform measurements with the following configurations:

- *ADD*: The ADD-based approach as described in Section 3.4 without early stopping.
- *ES*: The ADD-based approach with early stopping, as described in Section 3.5.1 for binary classification.
- *PCES*: The ADD-based approach with path condition-based early stopping (Section 3.5.2 for binary classification and Section 3.5.3 for multiclass classification).

Transformation time: Tables 3.3 and 3.4 list the time (in seconds) required to compile each gradient boosted model into an ADD. For *ADD*, the transformation times range from 0.01 seconds for *promoters* to 4062.44 seconds for *ann-thyroid*. We can also see that early stopping and

Table 3.2: Overview of datasets and the learned tree ensembles (#F = Number of features, #C = Number of classes, #N = Number of nodes in the tree ensemble, #P = Number of unique predicates in the tree ensemble, D = Maximum depth, %A = Accuracy on test set).

Dataset	**#F**	**#C**	**#N**	**#P**	**D**	**%A**
ann-thyroid	21	3	1302	113	3	100
appendicitis	7	2	328	48	3	91
divorce	54	2	88	16	3	100
ecoli	7	5	2370	184	3	85
glass2	9	2	540	107	4	88
promoters	58	2	39	1	3	100
shuttle	9	7	2026	143	3	100
threeOf9	9	2	72	3	3	100
wine-recog.	13	3	552	76	3	97
zoo	16	7	772	16	4	83

path condition-based early stopping almost always improve the transformation time (except for *promoters* where the transformation time is low already).

In most cases, *PCES* is more effective than *ES*. For *appendicitis*, for example, *ADD* needs 10.14 s, while *ES* and *PCES* finish in 4.51 s and 1.06 s, respectively (speedups of $2.25\times$ and $9.56\times$). Similarly, for *glass2*, *ES* provides a speedup of $2.99\times$, and *PCES* achieves $18.53\times$. For the binary datasets, the geometric mean speedups for *ES* and *PCES* are $1.43\times$ and $2.77\times$, which shows that both optimizations are effective at accelerating the transformation, with *PCES* being more effective.

Table 3.3: Transformation time in seconds for binary classification datasets, with speedup over *ADD* in parentheses.

Dataset	appendicitis	divorce	glass2	promoters	threeOf9	GeoMean
ADD	10.14	0.101	1057.39	0.012	0.019	
ES	4.51 (2.25×)	0.076 (1.33×)	354.12 (2.99×)	0.019 (0.63×)	0.018 (1.06×)	1.43×
PCES	1.06 (9.56×)	0.086 (1.17×)	57.07 (18.53×)	0.017 (0.71×)	0.017 (1.12×)	2.77×

For multiclass datasets, the geometric mean speedup for *PCES* is $1.77\times$. Although the *zoo* dataset shows a slight time increase (from 212 ms to 218 ms), all other cases, especially those with longer baseline times (*ann-thyroid*, *ecoli*, *shuttle*), benefit significantly.

Table 3.4: Transformation time in seconds for multi classification datasets, with speedup over *ADD* in parentheses.

Dataset	ann-thyroid	ecoli	shuttle	wine-recog.	zoo	GeoMean
ADD	4062.44	2808.10	700.69	125.02	0.212	
PCES	2719.29 (1.49×)	798.20 (3.52×)	243.35 (2.88×)	106.77 (1.17×)	0.218 (0.97×)	1.77×

Size: Tables 3.5 and 3.6 summarize the final ADD sizes, i.e., the number of nodes in the ADD. For the binary class datasets (Table 3.5), sizes range from 3 to 312644 nodes. Note that *ADD* and *ES* result in identical sizes, whereas *PCES* sometimes produces slightly larger ADDs (e.g., *appendicitis* and *glass2*).

For multiclass datasets (Table 3.6), the sizes vary from 290 up to 166284. For most datasets, there are only slight differences between *ADD* and *PCES*, except for *shuttle* where the ADD produced by *PCES* is 10.51% smaller compared to the ADD produced by *ADD*. In Chapter 5, we investigate whether these size differences impact explanation generation times.

Table 3.5: Number of nodes in the ADD for each binary classification dataset, with percentage changes relative to *ADD* in parentheses.

Dataset	appendicitis	divorce	glass2	promoters	threeOf9
ADD	8287	332	312 644	3	3
ES	8287 (0.0%)	332 (0.0%)	312 644 (0%)	3 (0.0%)	3 (0.0%)
PCES	8465 (+2.15%)	332 (0.0%)	313 093 (+0.14%)	3 (0.0%)	3 (0.0%)

Table 3.6: Number of nodes in the ADD for each multiclass dataset, with percentage changes relative to *ADD* in parentheses.

Dataset	ann-thyroid	ecoli	shuttle	wine-recog.	zoo
ADD	162 056	102 640	60 251	29 742	290
PCES	166 284 (+2.6%)	103 462 (+0.8%)	53 921 (-10.51%)	28 860 (-2.97%)	290 (0.0%)

Appendix B, Section B.2, presents detailed results on the class characterization sizes for each class and configuration.

Chapter 4

Transforming Tree Ensembles into Decision Trees

In this chapter, we present an alternative approach to transform random forests and gradient boosted trees into a unified structure. Instead of relying on Algebraic Decision Diagrams (ADDs), we directly construct decision trees and directed acyclic graphs (DAGs) as the target models. We achieve this by performing a depth-first search (DFS)-based traversal of the trees of the ensemble while creating a new tree at the same time.

First, we discuss some of the inefficiencies of the ADD-based approach.

- The ADD-based approach iteratively constructs large intermediate results that are larger than the final output (cf. Figure 3.8). Constructing these intermediate results is costly in both time and space.
- Infeasible path elimination is only possible as a post-process. Although it can substantially reduce the size of the ADDs, the approach first builds a structure containing infeasible paths and then incurs additional cost to remove them.
- The infeasible path elimination is repetitive in certain scenarios, as the same sub-paths might be visited multiple times. For instance, in the ADD-based approach for random forests (Section 3.1), the first two ADDs are aggregated, infeasible paths are removed, and then the result is aggregated with a third ADD. When infeasible paths are eliminated after this aggregation, the traversal restarts from the root, possibly revisiting sub-paths that were already processed in the first step.
- Maintaining ADDs in an ordered form is computationally expensive. Although standard ADD libraries enforce ordering after each operation, this overhead offers limited benefit for the scenarios considered in this dissertation.
- For gradient boosted trees, the score ADDs A^+ are often essentially tree-shaped in practice, since real-valued weights rarely allow subtree sharing.
- For some optimizations, it is advantageous to store additional node-level information. Standard ADD libraries do not directly support this, and storing metadata externally (e.g., in a hash map) can introduce significant overhead.

The core idea of our new approach is to directly construct the final result by traversing the ensemble's trees recursively while creating a new tree at the same time. This procedure avoids certain inefficiencies of the ADD-based method:

- By directly constructing the final result, the approach avoids creating large intermediate results.
- The target model is a DAG without explicit variable ordering, avoiding the overhead associated with maintaining a strict ordering.

Algorithm 9 RF2DT: Transforming a random forest into a single decision tree $[5]_{\text{AP}}$.

```
1:  procedure RF2DT(Random forest T_1, ..., T_n, Current node t_i in tree T_i, Array votes)
2:      Create new node newNode
3:      if t_i is a leaf then
4:          // Increase vote count for the class in the current leaf
5:          votes[t_i.class] ← votes[t_i.class] + 1
6:          if i = n then
7:              // If processing the last tree, assign the majority class
8:              newNode.class ← arg max_class votes
9:          else
10:             // Recursively process next tree in the list
11:             newNode ← RF2DT(T_1, ..., T_n, T_{i+1}.root, votes)
12:         end if
13:         votes[t_i.class] ← votes[t_i.class] − 1
14:     else
15:         newNode.feature ← t_i.feature
16:         newNode.threshold ← t_i.threshold
17:         newNode.true ← RF2DT(T_1, ..., T_n, t_i.true, votes)
18:         newNode.false ← RF2DT(T_1, ..., T_n, t_i.false, votes)
19:     end if
20:     return newNode
21: end procedure
```

- Infeasible paths are detected and avoided during traversal, eliminating the need for an expensive post-processing step.
- A custom DAG data structure allows integration of additional node-level information, enabling domain-specific optimizations.
- In the case of boosted trees, where subtrees rarely share weights, decision trees can be used instead of DAGs, removing the overhead of attempting to merge identical subtrees.

This approach applies to both random forests and gradient boosted trees. Section 4.1 introduces our algorithm $[5]_{\text{AP}}$, which transforms random forests into decision trees, followed by an evaluation in Section 4.2. Similarly, Section 4.3 extends this method to gradient boosted trees, with its evaluation presented in Section 4.4.

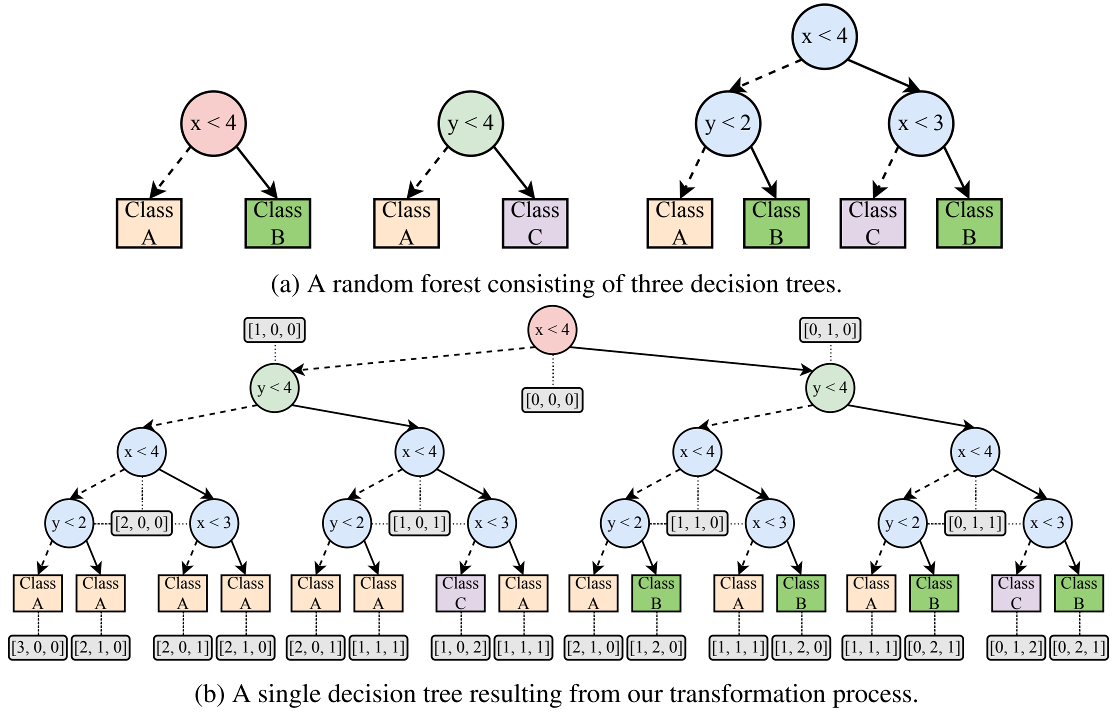
4.1 Transforming Random Forests to Decision Trees

This section explains how to transform a random forest into a single decision tree. The transformation traverses each tree in the ensemble while constructing a new unified tree at the same time. We begin with a straightforward implementation before presenting various optimizations.

4.1.1 Baseline Transformation Method

Algorithm 9 shows a basic procedure for converting a random forest into a single decision tree. The algorithm's inputs include a random forest $T_1, \ldots, T_n$, the current node t_i from the i-th tree, and an array $votes$ that tracks class votes.

- At the start, a new node is created (Line 2).



(a) A random forest consisting of three decision trees.

(b) A single decision tree resulting from our transformation process.

Figure 4.1: Two figures illustrating the transformation process from random forest into a single decision tree.

- If t_i is a leaf, the corresponding class count in $votes$ is incremented (Line 5).
- If t_i is a leaf in the final tree T_n, the algorithm determines the majority class and assigns it to the new node as its class (Line 8). Otherwise, it recursively processes the next tree in the ensemble (Line 11), then backtracks by decrementing the class vote (Line 13).
- If t_i is an inner node, a new node with the same feature and threshold is created (Lines 15–16), and the procedure is applied recursively to both true and false successors (Lines 17–18).

Example 22. We illustrate the transformation process with a simple example. Consider the random forest consisting of three decision trees in Figure 4.1a[1], which is the same random forest as in Figure 3.1. Applying our transformation process to this random forest results in a single decision tree, as shown in Fig. 4.1b, which given an input directly returns the majority vote class. Each node in the resulting tree is annotated with a vote vector $[v_A, v_B, v_C]$, where each entry indicates how many votes have been collected for each class *A*, *B*, and *C*.

Consider the rightmost path in the decision tree in Fig. 4.1b. When we follow the true edge of $x < 4$ in the first tree, we reach the leaf node with class *Class B*, so the vote count for *Class B* is increased by one, resulting in the voting vector $[0, 1, 0]$. We now continue with the second tree, where we follow the true edge of $y < 4$ and reach the leaf node with class *Class C*, so the vote count for *Class C* is increased by one to $[0, 1, 1]$. Finally, we reach the last tree, where we follow the true edge of $x < 4$ and subsequently the true edge of $x < 3$ and reach the leaf node with class *Class B*, so the vote count for *Class B* is increased by one to $[0, 2, 1]$. Since we are processing the last tree, we create a leaf node with the majority class, which is *Class B* in this case.

[1]Dotted edges represent false branches, while solid edges represent true branches.

While Algorithm 9 is simple, it is not very efficient. The algorithm visits each node in the first tree once, then for each leaf node in the first tree, it visits each node in the second tree and so on. Each operation in the algorithm runs in constant time except the recursive calls, and the $\arg\max$ in Line 8 which takes $O(K)$ time. The runtime can be expressed as $O(|T_1| + (|T_1| \cdot |T_2|) + ((|T_1| \cdot |T_2| \cdot |T_3|)) \ldots)$, which is equivalent to the following summation:

$$O(\sum_{i=1}^{n} \prod_{j=1}^{i} |T_j|)$$

The trees of a random forest are often similar in size, so an upper bound for the runtime complexty is $O(|T_i|^n)$ for some i. Since the algorithm creates a copy of each visited node (except the leaf nodes of every tree except the last one), an upper bound for the space complexity is $O(|T_i|^n)$ as well. In the remainder of this section, we present several optimizations that aim to make the computation efficient in practice.

4.1.2 Optimizations

There are several ways in which we can optimize the transformation algorithm.

- **Using a DAG:** Although Algorithm 9 constructs a decision tree, merging identical subtrees into a single DAG node (via hashing) can significantly reduce memory usage. This introduces some overhead during construction but can benefit analyses that operate on the final model.
- **Infeasible Path Elimination:** The infeasible path elimination discussed in Section 3.1.1 can be incorporated to skip any subtrees shown to be unreachable under the path condition. This reduces the number of paths that the algorithm has to traverse and can significantly accelerate the transformation process and reduce the size of the resulting model.
- **Early Stopping:** Methods from Chapter 3 (Section 3.2.2 and Section 3.2.4) can stop the transformation when a class is guaranteed to win, therefore avoiding unnecessary traversal of subsequent trees.
- **Dynamic Tree Ordering:** A heuristic that processes smaller trees first can further reduce transformation time, particularly when early stopping conditions are often satisfied.
- **Tree Simplification:** Pre-simplifying each tree using the path condition (Algorithm 10) can eliminate redundant predicates and reduce the number of feasible paths.
- **Class Characterizations:** We can also generate class characterizations for the decision-tree-based approach presented in this chapter. Similar to Section 3.1.2, the idea is to pick the class of interest c and replace any leaf containing c with a leaf of value 1, and all other leaves with 0. This converts each resulting leaf into a simple binary outcome and simplifies the analysis to a single "is class c or not" question. Afterward, we perform standard DAG optimizations, merging identical subtrees and removing any redundant nodes, to obtain a more compact representation.

4.1.3 Using a DAG

Algorithm 9 uses a decision tree as the target model, though there can be many common subtrees that could be unified to a single tree. By using a DAG, one could reduce the amount of memory needed and potentially accelerate algorithms that perform analyses on the target model by avoiding redundant computations in identical subtrees. Our approach works in the following way. We

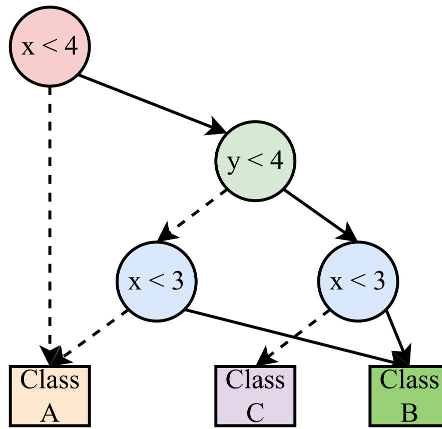


Figure 4.2: The DAG obtained by applying our optimizations.

use a hash map to keep track of all unique subtrees. Every time a new subtree has been created (after Line 18), we check whether the newly created subtree already exists. If it does exist, we return the subtree stored in the hash map. Otherwise, we put the newly created subtree into the hash map and return it. Overall, this can have a slightly negative effect on the transformation time caused by the overhead of using the hash map, though it can greatly reduce the amount of memory needed.

4.1.4 Infeasible Path Elimination

The infeasible path elimination presented in Section 3.1.1 for ADDs can be applied straightforwardly to the approach presented in this section. For the ADD-based approach it was applied as a post-process, e.g., after each aggregation or after the majority vote ADD has been created. For the approach presented in this section, the transformation and infeasible path elimination can be performed at the same time. It is crucial to reduce the number of paths that our approach has to traverse. E.g., in Figure 4.1b the $x < 4$ predicate appears twice on each path as it is both in the first and third tree. When we follow the false edge of $x < 4$ in the first tree, we do not need to follow the true edge of the $x < 4$ node in the third tree. The same holds when we follow the true edge of $x < 4$.

Figure 4.2 illustrates the impact of applying infeasible path elimination and DAG merging on the random forest in Figure 4.1a, producing a smaller DAG.

4.1.5 Early Stopping

The early stopping approaches presented in Chapter 3 can also be applied in a straightforward way to the approach presented in this section. The semantics-preserving early stopping approaches presented in Section 3.2.2, i.e., our “general”, and “specific” rules can be applied whenever Algorithm 9 visits a leaf. These semantics-preserving early stopping approaches only require the current number of votes and the number of trees in total which our approach always keeps track of.

The abstract early stopping approach presented in Section 3.2.4 can also be applied straightforwardly. If we add the infeasible path elimination to our approach, the algorithm will have to keep track of the path condition, so whenever a leaf node is reached, Algorithm 4 can be run to apply abstract early stopping.

Applying early stopping in our approach is crucial because it limits the number of nodes that must be visited. However, when a DAG-based representation is used, early stopping does not reduce the final model size. This happens because early stopping is applied only if a particular

class is guaranteed to win, which implies that all leaves in that sub-DAG would map to the same class label. Therefore, those identical leaves can be collapsed into a single leaf node in the DAG. For example, in Figure 4.1b, if *Class A* accumulates two votes while the other classes have none, every leaf reachable at that point corresponds to *Class A*. In such a situation, the internal node $x < 4$ can be replaced by a single *Class A* leaf, avoiding further branching and maintaining the overall DAG size.

4.1.6 Abstract Early Stopping Heuristic

The abstract early stopping method in Section 3.2.4 has time complexity $O(n \cdot k)$, where k is the largest tree still to be processed. While the potential savings can be large when abstract early stopping is applicable, we only want to try to apply it if it is likely to be applicable. Therefore, we implemented a simple heuristic that decides whether to try to apply abstract early stopping that works as follows:

- The heuristic only tries abstract early stopping if less than half of the trees have been processed. Otherwise, non-abstract early stopping is applied. The motivation behind this is that the earlier abstract early stopping can be applied, the higher the savings. Non-abstract early stopping is cheaper, i.e. $O(K)$ where K is the number of classes.
- The heuristic only tries abstract early stopping if the leading class holds at least 70% of the potential votes.

4.1.7 Tree Order Heuristic

While our procedure processes the random forest trees in the given order, the trees can be processed in any sequence, and this sequence can significantly affect both transformation time and the size of the resulting model. Multiple strategies can be considered:

- *Static ordering*: Determine a fixed order for processing all trees before the transformation begins.
- *Dynamic ordering*: Decide, during the transformation, which tree to process next based on current conditions or heuristics.

We propose a greedy dynamic heuristic that leverages the path condition to decide which tree to process next. Whenever a leaf is reached in the current tree and a subset of unprocessed trees $T' \subset \{T_1, \dots, T_n\}$ remains, the heuristic evaluates, for each $t' \in T'$, how many leaves are reachable under the current path condition. The algorithm then proceeds with the tree in T' that has the smallest number of reachable leaves.

The motivation is to minimize the number of paths to traverse, which reduces both runtime and model size. Additionally, recall that some early stopping methods (Section 3.2) require at least half of the trees to have been processed to be applied. Also, remember that for a random forest with trees $T_1, \dots, T_n$ where each tree T_i has m_i leaf nodes, the tree resulting from our approach will have $\prod_{i=1}^{n} m_i$ leaf nodes. Processing smaller trees early may allow early stopping to take effect more quickly, avoiding unnecessary processing of larger trees. Each additional tree processed increases the likelihood of satisfying early stopping criteria.

A slight overhead arises from both tracking which trees have already been processed and computing the number of reachable leaves for each remaining tree at every leaf node. If m is the size of a tree, determining its reachable leaves under the current path condition costs $O(m)$ time. However, prioritizing smaller trees helps keep this overhead manageable.

Algorithm 10 Simplifying a tree under a given path condition.

```
 1: procedure SMPLFY(Node t, Path condition pc)
 2:   newNode ← new Node()
 3:   if t is a leaf then
 4:     newNode.class ← t.class
 5:     return newNode
 6:   end if
 7:   if pc ⟹ (t.feature < t.threshold) then
 8:     return SMPLFY(t.true, pc)
 9:   else if pc ⟹ ¬(t.feature < t.threshold) then
10:     return SMPLFY(t.false, pc)
11:   else
12:     trueChild ← SMPLFY(t.true, pc)
13:     falseChild ← SMPLFY(t.false, pc)
14:     if trueChild and falseChild are leaves and trueChild.class = falseChild.class then
15:       newNode.class ← trueChild.class
16:     else
17:       newNode.feature ← t.feature
18:       newNode.threshold ← t.threshold
19:       newNode.true ← trueChild
20:       newNode.false ← falseChild
21:     end if
22:     return newNode
23:   end if
24: end procedure
```

Finally, this heuristic highlights the adaptability of the path-based transformation approach described in this chapter, as incorporating an order-based heuristic is straightforward. In contrast, the ADD-based aggregation method from Chapter 3 operates on pairs of ADDs and does not naturally support such ordering heuristics.

4.1.8 Tree Simplification

One important goal is to reduce the number of paths that must be traversed. Consider a random forest with 50 trees and assume we are currently processing tree T_5. Each leaf node in T_5 potentially leads to the exploration of all feasible paths in the remaining trees $T_6, \ldots, T_{50}$. Ideally, we want to minimize these paths to avoid exponential growth in the traversal.

A useful strategy is to simplify a tree before processing it, leveraging the path condition. The core idea is to generate a new, simplified decision tree by applying the following simplification rule:

$$\text{if } a \text{ then } B \text{ else } B \implies B$$

While standard decision trees are typically learned without such redundancies, certain predicates might become redundant under a given path condition. In those cases, applying the above rule can be advantageous. Specifically, if the condition implies a is always satisfied (or always false), then one branch becomes irrelevant.

Applying this simplification rule has the following effect on our transformation algorithm:

- By eliminating redundant predicates, the number of leaf nodes in the simplified tree decreases, which reduces the number of recursive calls that must be performed.
- Although our algorithm unifies common sub-trees to a DAG, applying the simplification rule can significantly reduce the size. This is a result of the infeasible path elimination.

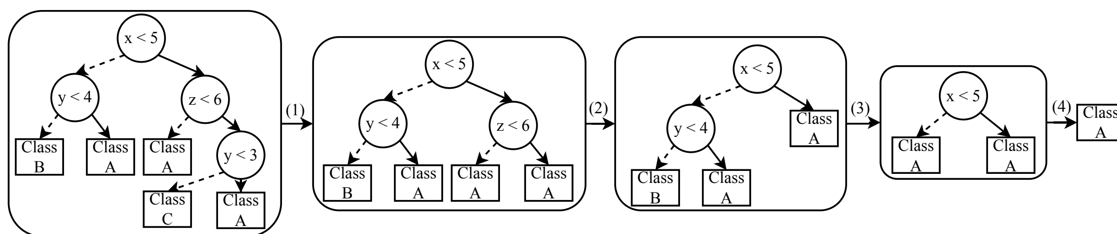


Figure 4.3: Example of tree simplification with path condition $\{x \in [-\infty, +\infty), y \in [-\infty, 2), z \in [-\infty, +\infty)\}$.

When our algorithm recursively processes the then branch, the predicate a is added to the path condition, while processing the else branch adds $\neg a$, potentially resulting in distinct subtrees that cannot be unified.

- A minor downside of the simplification is that neither a nor $\neg a$ are added to the path condition, which might slightly increase the time for the recursive call as there may be more feasible paths. However, the benefit of reducing the number of paths that we have to traverse usually outweighs this downside.

Algorithm 10 shows how a simplified decision tree can be created given a decision tree and a path condition pc. The procedure SMPLFY work as follows:

1. If the node t is a leaf, we simply create and return a new leaf node with the same class label.

2. Otherwise, if the path condition implies that $t.feature < t.threshold$ is always true (i.e., $pc \implies (t.feature < t.threshold)$), the false branch can be skipped and the true branch is simplified recursively.

3. If instead $pc \implies \neg(t.feature < t.threshold)$ is always true, then the true branch is skipped and only the false branch is simplified recursively.

4. If neither branch is implied by the path condition, both children are simplified recursively. After simplification, if both children are leaves with the same class label, they are unified to a single leaf.

By repeatedly applying this procedure from the root down to every child, a simplified tree is obtained.

Example 23. Figure 4.3 shows how a decision tree can be simplified using Algorithm 10 with the path condition $\{x \in [-\infty, +\infty), y \in [-\infty, 2), z \in [-\infty, +\infty)\}$. In the first step, the subtree rooted at $y < 3$ can be replaced with the leaf node *Class A* since $y \in [-\infty, 2) \implies y < 3$ and so the false branch is infeasible. In the second step, the subtree rooted at $z < 6$ can be replaced with *Class A* since both successors are *Class A*. In the third step, just as $y < 3$, the subtree rooted at $y < 4$ can be replaced with *Class A*. Finally, the root node can be replaced with *Class A* since both successors are *Class A*. So, the resulting decision tree consists only of a single leaf node *Class A*. If we had not performed this simplification, the transformation algorithm would have to process all remaining trees for the three reachable leaf nodes *Class A*. Instead, now it has to process all remaining trees only once.

Algorithm 11 Transforming a random forest into a DAG with optimizations [5]$_{\text{AP}}$.

```
1: procedure RF2DTOpt(Random forest T_1,...,T_n, Current node t_i in tree T_i, Array votes, Path condition pc, Hash map uniqueMap)
2:     Create new node newNode
3:     if t_i is a leaf then
4:         // Increase vote count for the class in the current leaf
5:         votes[t_i.class] ← votes[t_i.class] + 1
6:         if i = n then
7:             // If processing leaf in the last tree, create leaf with the majority class
8:             newNode.class ← arg max_class votes
9:         else
10:            safeWinner ← earlyStop(T_1,...,T_n, t_i, votes, pc)
11:            if safeWinner ≠ "unsure" then
12:                newNode.class ← safeWinner
13:            else
14:                // Simplify next tree in the list and pass it to the recursive call
15:                nextTreeSimplified ← SMPLFY(T_{i+1}.root, pc)
16:                newNode ← RF2DTOpt(T_1,...,T_n, nextTreeSimplified, votes, pc, uniqueMap)
17:            end if
18:        end if
19:        // After recursive call, decrease the vote count
20:        votes[t_i.class] ← votes[t_i.class] − 1
21:    else
22:        if pc ⟹ t_i.feature < t_i.threshold then
23:            newNode ← RF2DTOpt(T_1,...,T_n, t_i.true, votes, pc, uniqueMap)
24:        else if pc ⟹ ¬(t_i.feature < t_i.threshold) then
25:            newNode ← RF2DTOpt(T_1,...,T_n, t_i.false, votes, pc, uniqueMap)
26:        else
27:            newNode.feature ← t_i.feature
28:            newNode.threshold ← t_i.threshold
29:            pc_t ← pc ∧ (t_i.feature < t_i.threshold)
30:            newNode.true ← RF2DTOpt(T_1,...,T_n, t_i.true, votes, pc_t, uniqueMap)
31:            pc_f ← pc ∧ ¬(t_i.feature < t_i.threshold)
32:            newNode.false ← RF2DTOpt(T_1,...,T_n, t_i.false, votes, pc_f, uniqueMap)
33:            if newNode.true == newNode.false then
34:                newNode ← newNode.true
35:            end if
36:        end if
37:    end if
38:    if uniqueMap.contains(newNode) then
39:        return uniqueMap.get(newNode)
40:    else
41:        uniqueMap.put(newNode, newNode)
42:        return newNode
43:    end if
44: end procedure
```

4.1.9 Optimized Approach

Algorithm 11 contains the previously presented optimizations, except for the dynamic tree order heuristic. In addition to the random forest $T_1, \ldots, T_n$, the node currently being processed t_i, the $votes$ array, the algorithm now also has the path condition pc and a hash map $uniqueMap$, which stores all unique DAGs, as the input. In comparison to our simple approach presented in Algorithm 9, the algorithm contains the following changes:

- When the currently processed node is a leaf, in Line 10 early stopping is applied. This can either be the abstract early stopping approach (see Algorithm 4) or the non-abstract early stopping approach. If early stopping is successful, the class of the newly created node can be set to the winning class (Line 12).

- If early stopping cannot be applied, the algorithm now first simplifies the next tree to be processed before performing the recursive call (Line 15).

- When the current node is an inner node, the algorithm checks, just like the infeasible path

elimination introduced in Section 3.1.1, if only the true or only the false successor are feasible, and only proceeeds with the feasible successor.

- If both successors are feasible, the algorithm recursively processes both successors. It then checks whether both successors are the same, and if that is the case, the new node is set to the result of the recursive call with the true successor.

- Before returning the result, the algorithm first checks whether the DAG rooted at the newly created node already exists. If it does the algorithm returns the instance stored in the hash map. Otherwise, it puts the new unique DAG into the hash map and then returns the result.

4.2 Evaluation: Transforming Random Forests into Decision Trees

In this section, we evaluate the techniques introduced in the previous section. We compare these techniques with the ADD-based approach presented in Section 3.1 and we use exactly the same random forests that have been trained on datasets from the UCI Machine Learning Repository [52]. Our implementation was done in Java and we performed the evaluation on the same machine that was used for the evaluation in Section 3.3, an Intel(R) Xeon(R) Gold 6152 CPU 2.10 GHz with 502 GB of RAM.

All configurations discussed here integrate infeasible path elimination and the DAG-based optimization by default, as described in Sections 3.1.1 and 4.1. The following list summarizes the additional features that differentiate each configuration:

- *DFS*: Our baseline approach (Algorithm 9), already incorporating infeasible path elimination and the DAG construction.

- *ES*: The baseline extended with early stopping based on the "specific rules" introduced in Section 3.2.2.

- *AbsES*: The baseline plus abstract early stopping (Algorithm 4).

- *HEUR*: The baseline plus a heuristic that decides whether to apply abstract early stopping (Section 4.1.6).

- *ORD*: The *HEUR* configuration extended by the tree-ordering heuristic, which processes next the tree with the fewest reachable leaves (Section 4.1.7).

- *+S*: Each of the above five configurations can be further combined with tree simplification (Subsection 4.1.8). For instance, *DFS*+S corresponds to the baseline approach with tree simplification.

4.2.1 Experimentation Results

Our evaluation aims to address the following research questions:

- **RQ1: Are our optimizations effective in terms of improving transformation time and size of the resulting DAG?**

- **RQ2: Can our DFS-based approach transform random forests into a DAG more efficiently than the ADD-based approach?**

RQ1: To answer *RQ1*, we analyse the transformation time, i.e., the time it takes to transform a random forests into a DAG, and the size of the resulting DAG.

Transformation Time: Figure 4.4 illustrates the speedup of all configurations relative to the *DFS* baseline. In nearly all cases, our optimizations, i.e., early stopping, heuristic ordering, and tree simplification, result in significant improvements. Configurations marked with *+S* perform significantly better than their non-simplified counterparts. For instance, *HEUR+S* achieves the highest geometric mean speedup (5.22×), followed by *ORD+S* (4.62×).

The largest speedups can be observed for datasets such as *magic*, *phoneme*, and *twonorm*, where *ORD+S* achieves factors of 51.84×, 59.65×, and 16.4×, respectively. In the *ring* dataset, *DFS* does not complete within 72 hours, whereas approaches like *ORD+S* finish in only 7.4 seconds, mainly due to tree simplification.

In contrast, *segmentation* sees limited improvement. Additionally, small datasets (e.g., *mofn-3-7-10*, *threeOf9*, *xd6*) may show minimal slowdowns with certain optimizations, but the absolute time remains under 30 ms, so this is not a practical concern.

Size: Table 4.1 shows the size of the final DAGs. Because *ES*, *AbsES*, and *HEUR* do not affect the DAG size, we limit our size comparisons to *DFS*, *ORD*, and their *+S* variants. Each entry shows the relative change from *DFS*.

On average, *ORD* produces DAGs 190.19% larger than *DFS*, due to the changed processing order that reduces subtree-sharing opportunities. Meanwhile, *DFS*+S results in DAGs 33.19% smaller than *DFS*, and *ORD*+S results in DAGs 9% smaller. Several datasets show especially large differences. For instance, *magic*, *ring*, *segmentation*, *twonorm*, and *waveform-21* can more than double in size when using *ORD*. However, *ORD* also completes faster for many of these datasets, indicating a trade-off between model size and transformation time.

Tree simplification substantially reduces both size and runtime. For example, *DFS*+S reduces the final DAG size by more than 50% in datasets like *ann-thyroid*, *magic*, *phoneme*, *ring*, and *twonorm*, with a particularly strong effect on *ring* (−93.32%). *ORD*+S typically produces larger DAGs than *DFS*+S, yet still smaller than or comparable to *DFS*. Overall, these results confirm that tree simplification results in significant benefits, and the order-based heuristic can speed up transformations when memory overhead is not the primary concern.

Appendix B (Section B.1), presents detailed results on the class characterization sizes for each class and configuration.

RQ2: For *RQ2*, we compare our DFS-based approach against the most efficient ADD-based variant. We found that *AbsES* was the fastest ADD-based configuration, whereas *HEUR+S* performed best among our DFS-based strategies. Table 4.2 summarizes the results.

Overall, the DFS-based approach consistently outperforms the ADD-based approach in terms of transformation time. The geometric mean speedup is 20.49×. For example, for *ecoli* the DFS-based approach is 52 times faster and reduces the transformation time from 1089 to only 20.87 seconds.

On average, the DFS-based approach results in a 2.63% reduction in the final DAG size. For many datasets, the DFS-based approach results in significant savings, such as *ann-thyroid*, *ecoli*, *new-thyroid*, *phoneme*, and *wine-recog*, ranging between 20% and 36%. However, there are also instances where the DFS-based approaches produces larger DAGs, such as for *ionosphere*, *ring*, and *twonorm*. Despite these outliers, the overall trend indicates that, in addition to speeding up compilation, the DFS-based approach often provides a smaller or comparably sized final model. This combination of faster runtime and compact representation makes the DFS-based method a promising alternative for large-scale or time-sensitive tasks.

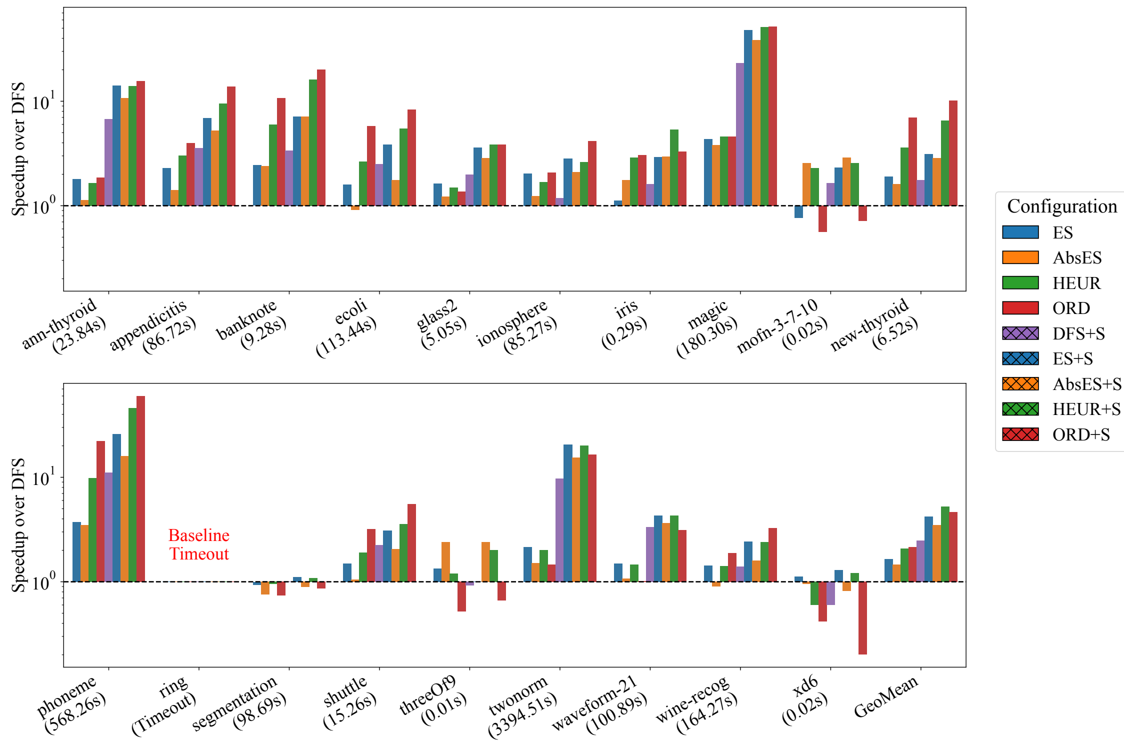


Figure 4.4: Speedups over *DFS*. Baseline times, i.e., transformation time of *DFS* in seconds, are shown in parentheses next to dataset names.

Table 4.1: Number of nodes in the resulting DAGs.

Dataset	DFS	ORD	DFS+S	ORD+S
ann-thyroid	399 123	752 390 (+88.51%)	168 791 (-57.71%)	240 857 (-39.65%)
appendicitis	1 943 572	2 931 476 (+50.83%)	1 161 957 (-40.22%)	1 307 657 (-32.72%)
banknote	75 318	84 462 (+12.14%)	46 417 (-38.37%)	44 700 (-40.65%)
ecoli	1 096 500	1 181 757 (+7.78%)	848 456 (-22.62%)	853 732 (-22.14%)
glass2	530 575	859 870 (+62.06%)	322 062 (-39.30%)	400 259 (-24.56%)
ionosphere	486 411	1 064 976 (+118.95%)	319 255 (-34.37%)	706 200 (+45.19%)
iris	1823	1765 (-3.18%)	1734 (-4.88%)	1436 (-21.23%)
magic	1 728 499	5 920 778 (+242.54%)	605 848 (-64.95%)	964 578 (-44.20%)
mofn-3-7-10	22	23 (+4.55%)	24 (+9.09%)	26 (+18.18%)
new-thyroid	81 797	86 085 (+5.24%)	66 287 (-18.96%)	63 038 (-22.93%)
phoneme	1 977 030	2 340 429 (+18.38%)	954 830 (-51.70%)	988 824 (-49.98%)
ring	5 089 861	106 994 692 (+2002.11%)	340 078 (-93.32%)	974 490 (-80.85%)
segmentation	9 862 431	22 309 988 (+126.21%)	8 341 151 (-15.43%)	19 300 003 (+95.69%)
shuttle	443 504	580 946 (+30.99%)	367 195 (-17.21%)	418 842 (-5.56%)
threeOf9	3	3 (0.00%)	3 (0.00%)	3 (0.00%)
twonorm	37 524 747	223 709 342 (+496.16%)	11 170 913 (-70.23%)	37 589 007 (+0.17%)
waveform-21	5 348 612	18 727 728 (+250.14%)	2 769 993 (-48.21%)	7 122 294 (+33.16%)
wine-recog	7 801 905	15 506 552 (+98.75%)	6 293 801 (-19.33%)	9 914 530 (+27.08%)
xd6	67	68 (+1.49%)	65 (-2.99%)	63 (-5.97%)
Mean		**+190.19%**	**−33.19%**	**−9.00%**

Finally, we also tested the method from [53], which transforms a random forest into a semantically equivalent, minimal-size decision tree. However, due to its high memory requirements, it was only able to complete on *banknote*, *iris*, *mofn-3-7-10*, *new-thyroid*, *threeOf9*, and *xd6*. For these six datasets, our method was between 30× and 832× faster, and produced smaller models by using DAGs instead of decision trees.

Table 4.2: Compilation time and size comparison of ADD- and DFS-based approaches.

Dataset	Transformation Time		Speedup	DAG Size		Size Reduction (%)
	ADD	DFS		ADD	DFS	
ann-thyroid	31.36	1.715	18.29×	212 766	168 791	-20.67%
appendicitis	224.463	9.202	24.39×	1 148 415	1 161 957	+1.18%
banknote	17.357	0.58	29.93×	58 747	46 417	-20.99%
ecoli	1089.779	20.879	52.19×	1 134 199	848 456	-25.19%
glass2	30.276	1.322	22.90×	448 542	322 062	-28.20%
ionosphere	726.348	32.761	22.17×	268 210	319 255	+19.03%
iris	0.647	0.054	11.98×	1805	1734	-3.93%
magic	52.166	3.515	14.84×	662 797	605 848	-8.59%
mofn-3-7-10	0.094	0.009	10.44×	15	24	+60.00%
new-thyroid	36.812	1.005	36.63×	98 191	66 287	-32.49%
phoneme	548.798	12.346	44.45×	1 495 068	954 830	-36.13%
ring	80.331	10.94	7.34×	200 528	340 078	+69.59%
segmentation	2236.004	91.331	24.48×	9 177 945	8 341 151	-9.12%
shuttle	191.587	4.294	44.62×	484 730	367 195	-24.25%
threeOf9	0.078	0.006	13.00×	3	3	0.00%
twonorm	2299.646	170.141	13.52×	9 334 318	11 170 913	+19.68%
waveform-21	536.906	23.659	22.69×	3 464 949	2 769 993	-20.06%
wine-recog	2060.718	68.784	29.96×	9 148 528	6 293 801	-31.2%
xd6	0.121	0.015	8.07×	46	65	+41.3%
Aggregate			**20.49×**			**−2.63%**

4.3 Transforming Gradient Boosted Trees into Decision Trees

In this section, we show to extend the approach of transforming random forests into a single decision tree (see Section 4.1) to the case of gradient boosted trees. The main difference between random forests and gradient boosted trees is that, in the latter, leaf nodes hold real-valued weights rather than class labels. We handle the case of binary and multiclass classification separately since there are differences in the structure of the trees that enable distinct optimizations.

4.3.1 Binary Classification

A gradient boosted tree model for binary classification consist of n trees $T_1, \ldots, T_n$. The way a gradient boosted tree model makes its prediction, given an input, is to obtain the weights predicted by the individual boosted trees, sum them up, apply the sigmoid function, and choose class 1 if the result is larger than 0.5 and 0 otherwise[2].

The proposed approach again traverses every path while creating a copy of each tree structure. The main difference is the treatment of the leaves. Since the leaves of a boosted tree hold real-valued weights, the approach accumulates the weights as it traverses the sequence of trees. When a leaf node in the last tree is reached, the class label of the new leaf node is set to 1 if the accumulated weight is greater than 0 and 0 otherwise.

Algorithm 12 describes the approach for transforming gradient boosted trees $T_1, \ldots, T_n$ into a single decision tree. The algorithm takes as inputs the boosted trees $T_1, \ldots, T_n$, the current node t_i in tree T_i (initially the root node of T_1), and the accumulated weight w, which is initialized with 0. If the current node t_i is a leaf, the weight of the leaf is added to the accumulated weight. If T_i is the last tree, the class of the new node is set to 1 if the accumulated weight is greater than 0 and 0 otherwise. Otherwise, the algorithm is called recursively with the root of the next tree T_{i+1} and the updated accumulated weight. If the current node is not a leaf, the feature, and threshold of the current node are copied to the new node. The algorithm then recursively calls itself with the children of the current node, and assigns the results to the true and false children of the new node. In the end, the new node is returned.

The time and space complexity of Algorithm 12 is the same as for Algorithm 9, although here the winning class can be determined by checking the sign of the cumulative weight instead

[2] In Section 3.4.1 we noted that one can equivalently check whether the summed value is larger than 0.

Algorithm 12 Transforming gradient boosted trees for binary classification into a single decision tree.

1: **procedure** BT2DT(Gradient Boosted Trees $T_1, \ldots, T_n$, Current node t_i in tree T_i, Accumulated weight w)
2: Create new node $newNode$
3: **if** t_i is a leaf **then**
4: $w \leftarrow w + t_i.weight$
5: **if** $i = n$ **then**
6: $newNode.class \leftarrow 1$ **if** $w > 0$ **else** 0
7: **else**
8: $newNode \leftarrow$ BT2DT($T_1, \ldots, T_n, T_{i+1}.root, w$)
9: **end if**
10: **else**
11: $newNode.feature \leftarrow t_i.feature$
12: $newNode.threshold \leftarrow t_i.threshold$
13: $newNode.true \leftarrow$ BT2DT($T_1, \ldots, T_n, t_i.true, w$)
14: $newNode.false \leftarrow$ BT2DT($T_1, \ldots, T_n, t_i.false, w$)
15: **end if**
16: **return** $newNode$
17: **end procedure**

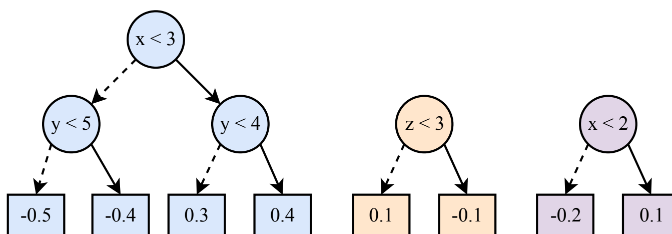


Figure 4.5: A gradient boosted tree model for binary classification consisting of three trees.

of applying an $\arg\max$.

Optimizations from Section 4.1.2, such as the elimination of infeasible paths, the unification of identical sub-trees to produce DAGs, and early stopping, can also be employed.

Example 24. Figure 4.5 shows a gradient boosted tree model for binary classification that consists of three trees[3]. By applying Algorithm 12 to this model, one obtains the single decision tree shown in Figure 4.6. In Figure 4.6, the accumulated weight is displayed below the horizontal line at the time each node is created.

We illustrate the transformation by examining the rightmost path. Initially, the algorithm copies the root node of the first tree, which has the predicate $x < 3$. In the beginning, the accumulated weight is 0.0. The algorithm then visits the true child node $y < 4$, so the accumulated weight remains 0.0. The next true child node is a leaf node with weight 0.4, which increases the accumulated weight to 0.4.

The algorithm proceeds with the second tree, whose root node is $z < 3$. This node is copied, and the accumulated weight is still 0.4. The true child of this node is a leaf with weight -0.1, causing the accumulated weight to decrease to 0.3.

Finally, the algorithm visits the root of the last tree, labeled $x < 2$, and copies it. Its true child is a leaf node with weight 0.1, leading to an accumulated weight of 0.4. Since the final

[3]This is the same gradient boosted tree model shown in Figure 3.14a in Section 3.4.1

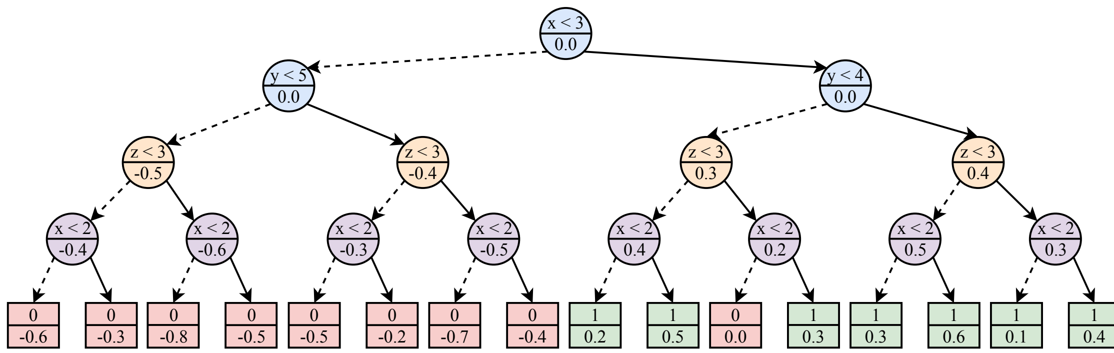


Figure 4.6: The decision tree resulting from applying Algorithm 12 to the gradient boosted tree model in Figure 4.5.

accumulated weight exceeds 0, the class of the resulting leaf is set to 1.

After transforming this tree into a DAG, the resulting DAG, shown in Figure 3.15, is identical to the one produced using the ADD-based approach.

4.3.2 Binary Classification with Early Stopping

As with the ADD-based approach, early stopping is applicable and improves the efficiency of the transformation. The transformation introduced in the previous section can be extended by using the early stopping strategy in Section 3.5.1. Algorithm 13 shows how to modify Algorithm 12 for this purpose.

Before initiating the transformation, suffix sums $\overline{W}$ and $\underline{W}$ must be computed (see Section 3.5.1). Recall that for each $i \in [n]$, these arrays respectively store the maximum and minimum sums of weights that can be obtained by the remaining boosted trees $T_i, \ldots, T_n$. If $w + \underline{W}[i+1] > 0$, the class of the new node can safely be set to 1 because the cumulative weight cannot become negative. Analogously, if $w + \overline{W}[i+1] \leq 0$, the class 0 is assigned, given that the cumulative weight cannot exceed 0 in the subsequent trees.

For the gradient boosted tree model in Figure 4.5, the array $\underline{W}$ is $[-0.8, -0.3, -0.2]$, and $\overline{W}$ is $[0.6, 0.2, 0.1]$. Using these arrays, one can immediately conclude that class 0 wins if the -0.5 or -0.4 leaves in the first tree are reached, because the remaining trees contribute at most 0.2, so the final sum of weights cannot exceed 0. Similarly, if the leaf with weight 0.4 is reached in the first tree, class 1 wins, since the minimum possible contribution from subsequent trees is -0.3, resulting in a total of at least 0.1. Only for the leaf with weight 0.3 no immediate decision can be made, as neither bound fully rules out the possibility of crossing the 0 threshold.

Binary Classification with Abstract Early Stopping

The abstract early stopping approach for binary classification boosted trees (Section 3.5.2) can also be applied to the decision-tree-based procedure described here. Algorithm 14 shows how to modify the transformation to use abstract early stopping.

Whenever a leaf node is reached, the minimum and maximum reachable weights in the remaining trees are computed with AbsMinMax (Algorithm 7). These bounds are then employed to decide whether early stopping is possible, analogous to Algorithm 13. This approach is more precise but also more computationally expensive, since each call to Algorithm 7 has time complexity $O(k)$, where k is the number of nodes in a tree, and the for-loop in Algorithm 14 adds a factor of n for n boosted trees, resulting in a time complexity of $O(n \cdot k)$. Thus, it is often advantageous to compute these bounds only when early stopping is likely.

Algorithm 13 Transforming gradient boosted trees for binary classification into a single decision tree with early stopping.

1: **procedure** BT2DTES($T_1, \ldots, T_n$, Current node t_i in tree T_i, Accumulated weight w)
2: Create new node $newNode$
3: **if** t_i is a leaf **then**
4: $w \leftarrow w + t_i.weight$
5: **if** $i = n$ **then**
6: $newNode.class \leftarrow 1$ **if** $w > 0$ **else** 0
7: **else if** $w + \underline{W}[i+1] > 0$ **then**
8: $newNode.class \leftarrow 1$
9: **else if** $w + \overline{W}[i+1] \leq 0$ **then**
10: $newNode.class \leftarrow 0$
11: **else**
12: $newNode \leftarrow$ BT2DTES($T_1, \ldots, T_n, T_{i+1}.root, w$)
13: **end if**
14: **else**
15: $newNode.feature \leftarrow t_i.feature$
16: $newNode.threshold \leftarrow t_i.threshold$
17: $newNode.true \leftarrow$ BT2DTES($T_1, \ldots, T_n, t_i.true, w$)
18: $newNode.false \leftarrow$ BT2DTES($T_1, \ldots, T_n, t_i.false, w$)
19: **end if**
20: **return** $newNode$
21: **end procedure**

Abstract Early Stopping Heuristic

In order to decide when it is worthwhile to compute the minimum and maximum reachable weights at each leaf node, a heuristic is required. One possibility is to use the abstract early stopping approach only when fewer than half of the trees have been processed. In these earlier stages, pruning large subtrees can result in significant reductions for the remaining transformation steps, compensating for the additional overhead of the abstract analysis. However, nodes in later trees are visited more often and therefore early stopping is also employed more often. The cost of repeatedly computing precise bounds can become prohibitive if it does not frequently result in pruning. In these later stages, the simpler early stopping approach from Section 4.3.2 is often more efficient, as its overhead is smaller.

4.3.3 Multiclass Classification

The case of gradient boosted trees for multiclass classification is now considered. In this setting, there are n trees per class, resulting in $n \cdot K$ trees in total, where K is the number of classes. The proposed approach is divided into two steps:

1. Combine the trees of each class into a single tree, resulting in K trees (one per class). Each such tree outputs the sum of the weights for its class.

2. Merge these K trees into a single decision tree that outputs the class with the highest total weight.

Algorithm 14 Transforming gradient boosted trees for binary classification into a single decision tree using abstract early stopping.

```
1: procedure BT2DTAbsES(T_1, ..., T_n, Current node t_i in tree T_i, Accumulated weight w,
   Path condition pc)
2:   Create new node newNode
3:   if t_i is a leaf then
4:     w ← w + t_i.weight
5:     minWeight, maxWeight ← (0, 0)
6:     for j ← i + 1 to n step 1 do
7:       (Δ_min, Δ_max) ← AbsMinMax(T_j.root, pc)
8:       (minWeight, maxWeight) ← (minWeight + Δ_min, maxWeight + Δ_max)
9:     end for
10:    if w + minWeight > 0 then
11:      newNode.class ← 1
12:    else if w + maxWeight ≤ 0 then
13:      newNode.class ← 0
14:    else if i = n then
15:      newNode.class ← 1 if w > 0 else 0
16:    else
17:      newNode ← BT2DTAbsES(T_1, ..., T_n, T_{i+1}.root, w, pc)
18:    end if
19:  else
20:    newNode.feature ← t_i.feature
21:    newNode.threshold ← t_i.threshold
22:    newNode.true ← BT2DTAbsES(T_1, ..., T_n, t_i.true, w, pc)
23:    newNode.false ← BT2DTAbsES(T_1, ..., T_n, t_i.false, w, pc)
24:  end if
25:  return newNode
26: end procedure
```

4.3.4 Score Decision Trees

The first step combines the n trees of each class into a single *score* tree where every leaf outputs the sum of the weights of those n trees. This is similar to the transformation of boosted trees for binary classification (Algorithm 12), except that the leaf nodes store the accumulated weight rather than a final class. Algorithm 15 shows how to transform the trees of a single class into one tree. When a leaf node in the last tree is reached, the weight of the new leaf node is set to the accumulated weight, and the class is recorded.

The infeasible path elimination from Section 4.1.2 can also be applied here. Identical subtrees can be unified to form DAGs, although in practice, real-valued leaves often limit merging opportunities.

Example 25. Algorithm 15 is illustrated by applying it to the gradient boosted tree model of Figure 4.7. E.g., the leaf reached by the path $y < 5 \wedge x < 5$ in the tree for *Class C* is 0.5 because the sum of weights contained in the leaves along that path are 0.2 and 0.3. Note that infeasible paths are already removed, e.g., $x < 3 \implies x < 4$, so no copy of $x < 4$ is created when it is visited after following the true edge of $x < 3$.

Algorithm 15 Transforming the boosted trees of a class into a single decision tree.

```
1: procedure BT2ScoreDT(T_1, ..., T_n, Current node t_i in tree T_i, Accumulated weight w,
   Class c)
2:   Create new node newNode
3:   if t_i is a leaf then
4:     w ← w + t_i.weight
5:     if i = n then
6:       newNode.weight ← w
7:       newNode.class ← c
8:     else
9:       newNode ← BT2ScoreDT(T_1, ..., T_n, T_{i+1}.root, w, c)
10:    end if
11:  else
12:    newNode.feature ← t_i.feature
13:    newNode.threshold ← t_i.threshold
14:    newNode.true ← BT2ScoreDT(T_1, ..., T_n, t_i.true, w, c)
15:    newNode.false ← BT2ScoreDT(T_1, ..., T_n, t_i.false, w, c)
16:  end if
17:  return newNode
18: end procedure
```

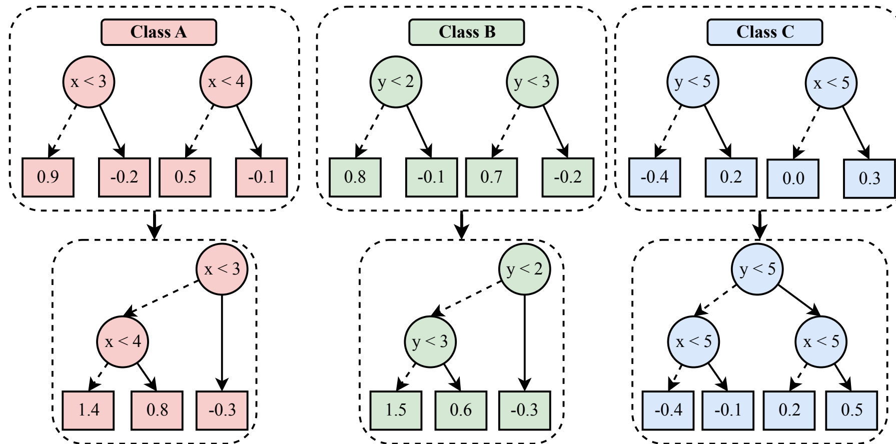


Figure 4.7: A gradient boosted tree model and the results of applying Algorithm 15 to this model.

4.3.5 Majority Vote Decision Trees

Algorithm 16 presents the merging process, which operates on the score trees $T_1, \ldots, T_K$, the current node t_i in tree T_i, the current maximum weight w (initialized to $-\infty$), and the class c associated with that maximum weight (initialized to $\perp$). When a leaf node is reached, its weight is compared with the current maximum w. If it is greater, w and c are updated. If the current tree is not the last tree, the algorithm proceeds with the root of T_{i+1}. When a leaf in the last tree is reached, the new leaf node is assigned class c.

Example 26. Figure 4.8 illustrates the result of applying Algorithm 16 to the score trees from Figure 4.7. Each internal node appears in the same color as its corresponding node in Figure 4.7, while each leaf node is colored according to the winning class along that path. For clarity, each node is annotated with a box containing the current maximum weight observed at that position, as well as the class to which that weight belongs.

Left-most path: Initially, the maximum weight is $-\infty$ and the class is $\perp$. Following the

Algorithm 16 Transforming the score decision trees into a single decision tree.

```
1: procedure MULTICLSBT2DT(Score decision trees T_1, ..., T_K, Current node t_i in tree T_i,
   Weight w, Class c)
2:   Create new node newNode
3:   if t_i is a leaf then
4:     if t_i.weight > w then
5:       w ← t_i.weight
6:       c ← c_i
7:     end if
8:     if i = K then
9:       newNode.class ← c
10:    else
11:      newNode ← MULTICLSBT2DT(T_1, ..., T_K, T_{i+1}.root, w, c)
12:    end if
13:  else
14:    newNode.feature ← t_i.feature
15:    newNode.threshold ← t_i.threshold
16:    newNode.true ← MULTICLSBT2DT(T_1, ..., T_K, t_i.true, w, c)
17:    newNode.false ← MULTICLSBT2DT(T_1, ..., T_K, t_i.false, w, c)
18:  end if
19:  return newNode
20: end procedure
```

false branches of $x < 3$ and $x < 4$, a leaf node with weight 1.4 is reached. Since $1.4 > -\infty$, the algorithm updates the maximum weight to 1.4 and sets c to c_A. Next, the algorithm proceeds with the root node of *Class B* and follows the false branch twice arriving at a leaf with weight 1.5. Because $1.5 > 1.4$, the maximum weight and winning class are updated to 1.5 and c_B. Finally, following the false branches of $y < 5$ and $x < 5$ in the score tree of *Class C* leads to a leaf with weight -0.4, which is less than the current maximum. No update is performed, and since this leaf appears in the last class's tree, the final leaf node in the new decision tree is assigned the winning class c_B.

Right-most path: Following the true branch of $x < 3$ in the first tree, ends in a leaf with weight -0.3 which is the new maximum. Continuing with the root node of *Class B* and following its true branch leads to another leaf with weight -0.3. Since this is equal to the current maximum (and earlier classes are prioritized if weights are tied), the algorithm does not perform an update. Note that for the remaining tree (*Class C*), the path constraints $x < 3$ and $y < 2$ imply $x < 5$ and $y < 5$, so a leaf with weight 0.5 is reached. Since $0.5 > -0.3$, the maximum weight and winning class are now updated to 0.5 and c_C. Reaching the final leaf, the algorithm assigns the newly created leaf node to c_C

Overall, each path in the final merged tree corresponds to a sequence of branching decisions across the K classes, and the leaf node reflects whichever class receives the largest accumulated weight. Figure 4.9 shows the tree of Figure 4.8 after common sub-trees have been merged (for clarity, the leaf nodes are not merged).

4.3.6 Multiclass Early Stopping

Score decision trees for multiclass classification can become quite large if the number of trees per class or the number of nodes in each tree is high. Early stopping is therefore crucial to improving efficiency, and multiple approaches, both with and without taking the path condition into account, are proposed.

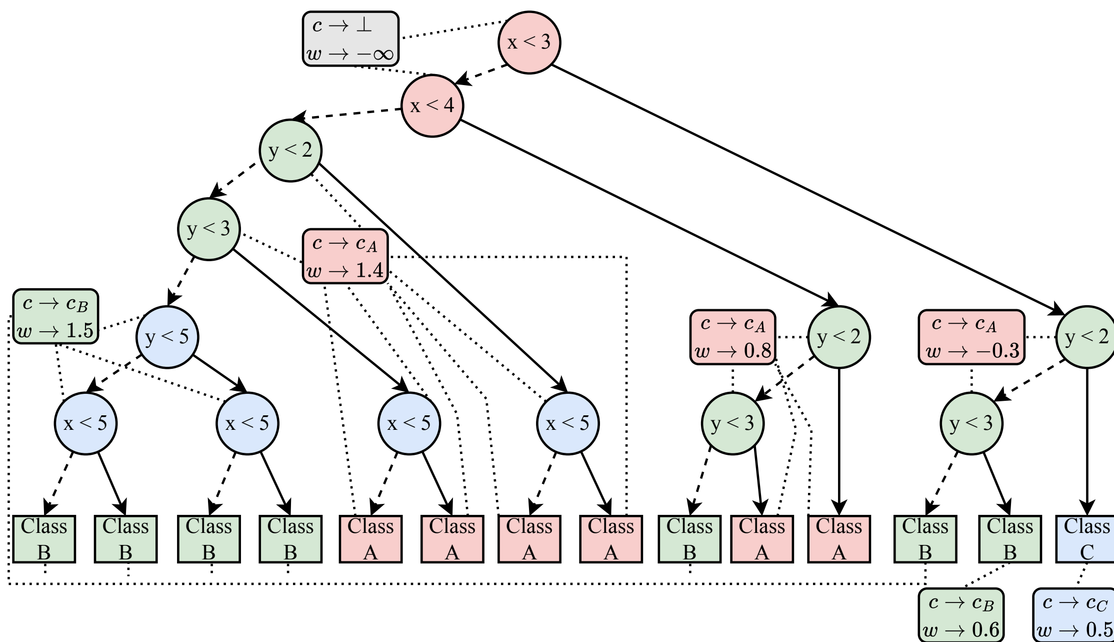


Figure 4.8: Result of applying Algorithm 16 to the score decision trees of Figure 4.7.

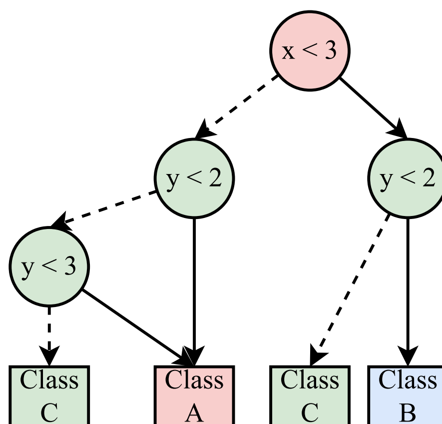


Figure 4.9: A simplified version of the decision tree from Figure 4.8.

Precomputation of Reachable Weights: Our key observation is that if we know, for any node of a score decision tree, the minimum and maximum weight that can be reached below it, then we can safely prune subtrees that cannot affect the final decision. This information enables constant-time checks for two scenarios:

1. All possible weights below the current node are strictly smaller than the best weight seen so far along the path. In this case, the current class cannot win, so we can stop exploring this entire subtree.

2. We are processing the last class's tree, and all possible weights below the current node are strictly greater than the best weight seen so far. In this scenario, the current class will win along every continuation of this subtree, so we can immediately assign the current class as the winning class without further traversal.

To enable constant-time checks of these two conditions, we precompute for each node the min-

Algorithm 17 Transforming score decision trees into a single decision tree using early stopping.

```
 1: procedure MULTICLSBT2DTES(Score decision trees T_1, ..., T_K, Current node t_i in tree T_i, Weight w, Class c)
 2:     Create new node newNode
 3:     if t_i is a leaf then
 4:         if t_i.weight > w then
 5:             w ← t_i.weight
 6:             c ← c_i
 7:         end if
 8:         if i = K then
 9:             newNode.class ← c
10:         else
11:             newNode ← MULTICLSBT2DTES(T_1, ..., T_K, T_{i+1}.root, w, c)
12:         end if
13:     else
14:         if i = K ∧ t_i.minRW > w then
15:             newNode.class ← c_i
16:         else if t_i.maxRW < w then
17:             if i = K then
18:                 newNode.class ← c
19:                 return newNode
20:             end if
21:             newNode ← MULTICLSBT2DTES(T_1, ..., T_K, T_{i+1}.root, w, c)
22:         else
23:             // Same as Algorithm 16 Lines 14-17.
24:         end if
25:     end if
26:     return newNode
27: end procedure
```

imum and maximum reachable weights in its subtree and store these values in the fields *minRW* and *maxRW*. This precomputation requires only one traversal per score decision tree, with a time complexity of $O(n)$ for a tree of n nodes. Afterwards, retrieving the minimum or maximum reachable weight for any node takes $O(1)$.

Algorithm with Early Stopping: Whenever a non-leaf node is reached:

- If it is a node in the last tree and its minimum reachable weight *minRW* is larger than the maximum weight observed so far, the procedure terminates at this node, and a new leaf node is created with the class assigned to the current class.
- If the maximum reachable weight *maxRW* is smaller than the maximum weight observed so far, the current class cannot win. If the current tree is the last tree, a new leaf node with the class assigned to c is created. Otherwise, the next score decision tree is traversed.

If neither condition is satisfied, we continue with the standard recursive procedure as before.

4.3.7 Path-Sensitive Multiclass Early Stopping

Path conditions can result in tighter bounds for *minRW* and *maxRW* by ruling out unreachable leaves. Although precomputing *minRW* and *maxRW* once per tree is relatively inexpensive, path-sensitive computation allows pruning even more aggressively, at the cost of additional overhead.

Two Additional Forms of Early Stopping: Before exploring a tree T_i, we recompute or refine *minRW* and *maxRW* for all reachable nodes under pc. This already allows us to apply early stopping earlier since *minRW* and *maxRW* are more accurate.

We also add two additional forms of early stopping:

Algorithm 18 Transforming the score decision trees into a single decision tree using path condition-based early stopping.

```
1: procedure MultiClsBT2DTPCES(Score decision trees T_1, ..., T_K, Current node t_i in tree T_i, Weight w,
   Class c, Path Condition pc)
2:     Create new node newNode
3:     if t_i is a leaf then
4:         HandleLeafCase(t_i, w, c, i, K, newNode, T_1, ..., T_K, pc)
5:     else
6:         if i = K ∧ t_i.minRW > w then
7:             newNode.class ← c_i
8:         else if t_i.maxRW < w then
9:             if i = K then
10:                newNode.class ← c
11:                return newNode
12:            end if
13:            computeAbsMaxMin(T_{i+1}, pc)
14:            newNode ← MultiClsBT2DTPCES(T_1, ..., T_K, t_{i+1}.root, w, c, pc)
15:        else if i < K ∧ t_i.maxRW < −0.5 then
16:            maybeNode ← CheckSNR(T_1, ..., T_K, i, pc, w, c, t_i.maxRW)
17:            if maybeNode ≠ null then
18:                return maybeNode
19:            end if
20:        else if t_i.minRW > w ∧ t_i.minRW > 1 then
21:            maybeNode ← CheckGIR(T_1, ..., T_K, i, pc, w, c, newNode, t_i.minRW)
22:            if maybeNode ≠ null then
23:                return maybeNode
24:            end if
25:        else
26:            newNode.feature ← t_i.feature
27:            newNode.threshold ← t_i.threshold
28:            pc_t = pc ∧ (t_i.feature < t_i.threshold)
29:            newNode.true ← MultiClsBT2DTPCES(T_1, ..., T_K, t_i.true, w, c, pc_t)
30:            pc_f = pc ∧ ¬(t_i.feature < t_i.threshold)
31:            newNode.false ← MultiClsBT2DTPCES(T_1, ..., T_K, t_i.false, w, c, pc_f)
32:        end if
33:    end if
34:    return newNode
35: end procedure
```

1. **Safe Pruning of "Loser" Subtrees:** In Algorithm 17 in Line 16, the check $t_i.maxRW < w$ can never be true for the first tree since w is initialized with $-\infty$. Though, in the first tree it can be possible that c_1 will never win at some subtree t_i if $t_i.maxRW$ is sufficiently small. Suppose the current class's *maxRW* under path condition *pc* is α. If there is at least one future tree T_j $(j > i)$ whose reachable weights under *pc* are all strictly greater than α, then class c_i is guaranteed to lose on this path. We can thus skip the entire subtree of t_i and continue with the next tree T_{i+1}.

2. **Early Determination of a "Winning" Subtree:** In Algorithm 17 in Line 14, we only check if $t_i.minRW > w$ if the current tree is the last tree. If we additionally check that the current node's *minRW* under *pc* cannot be exceeded by any tree remaining to be processed, we can also apply early stopping if $i < K$.

On-Demand Computation of Bounds: We implement the path-sensitive approach by calling `computeAbsMaxMin`(T_i, *pc*) on-demand, whenever a tree T_i is to be processed. This function recomputes *minRW* and *maxRW* for nodes reachable under the current path condition *pc*. Although it adds some additional cost, tighter bounds frequently result in substantial savings by pruning entire subtrees.

Algorithmic Sketch: Algorithm 18 maintains the structure of Algorithm 17 but includes:

Algorithm 19 Leaf-case handling.

```
1:  procedure HandleLeafCase(t_i, w, c, i, K, newNode, T_1, ..., T_K, pc)
2:      if t_i.weight > w then
3:          w ← t_i.weight
4:          c ← c_i
5:      end if
6:      if i = K then
7:          newNode.class ← c
8:      else
9:          computeAbsMaxMin(T_{i+1}, pc)
10:         newNode ← MultiClsBT2DTPCES(T_1, ..., T_K, t_{i+1}.root, w, c, pc)
11:     end if
12: end procedure
```

- A path condition *pc* that collects the feature constraints along the current path.
- Calls to `computeAbsMaxMin`(T_{i+1}, pc) (e.g., Lines 4 and 13) before processing the next tree T_{i+1}.
- Two helper functions:
  - CheckSNR (Algorithm 20), which checks if there is at least one future tree whose path-feasible weights all exceed the current subtree's *maxRW*.
  - CheckGIR (Algorithm 21), which checks if no future tree can surpass the current class's *minRW*.

Heuristic Thresholds: Although we could always check for possible early stopping by invoking CheckSNR or CheckGIR, in the worst case these functions might visit every node in all remaining trees. In practice, path conditions help reduce this cost, but it can still be expensive to run these checks at every step. To mitigate this overhead, we introduce simple numeric thresholds (e.g., *maxRW* < -0.5 or *minRW* > 1) that help us decide when these checks are likely to succeed. While these thresholds are simple, they work well in practice by avoiding expensive computations when early stopping is unlikely to be triggered. In real-world settings, they can be tuned to balance the runtime overhead against the potential gains from early pruning.

Putting It All Together: By tightening bounds via the path condition and adding these extra early stopping checks, we can prune large portions of the merged tree. While this introduces some overhead (e.g., multiple calls to `computeAbsMaxMin`), the net effect typically reduces the final tree size and transformation time. Therefore, in scenarios with many large boosted trees, path-sensitive early stopping can significantly improve overall efficiency.

4.4 Evaluation: Transforming Gradient Boosted Trees into Decision Trees

In this section, we evaluate the various optimizations to transforming gradient boosted trees into decision trees. We compare our approach to our ADD-based approach presented in Section 3.4 and evaluate the impact of our optimizations and heuristics on the transformation time and the size of the resulting decision tree.

We want to answer the following research questions:

Algorithm 20 Checks if there is at least one tree in $T_{i+1}, \ldots, T_K$ whose reachable weights are all larger than $maxRW$.

```
1: procedure CHECKSNR(T_1, ..., T_K, i, pc, w, c, maxRW)
2:     for j ← i + 1 to K step 1 do
3:         if smallerNotReachable(T_j, pc, maxRW) then
4:             computeAbsMaxMin(T_{i+1}, pc)
5:             return MULTICLSBT2DTPCES(T_1, ..., T_K, T_{i+1}.root, w, c, pc)
6:         end if
7:     end for
8:     return null
9: end procedure
```

Algorithm 21 Check if all reachable weights in trees $T_{i+1}, \ldots, T_K$ are smaller than $minRW$.

```
1: procedure CHECKGIR(T_1, ..., T_K, i, pc, w, c, newNode, minRW)
2:     foundGreater ← false
3:     for j ← i + 1 to K step 1 do
4:         if greaterIsReachable(T_j, pc, minRW) then
5:             foundGreater ← true
6:             break
7:         end if
8:     end for
9:     if ¬foundGreater then
10:        newNode.class ← c_i
11:        return newNode
12:    end if
13:    return null
14: end procedure
```

- **RQ1: Are our optimizations effective in terms of improving transformation time and size of the resulting DAG?**

- **RQ2: Can our DFS-based approach transform gradient boosted trees into a DAG more efficiently than our ADD-based approach?**

4.4.1 Experimental Setup

We evaluated our approach on a machine with an Intel(R) Xeon(R) Gold 6152 CPU 2.10 GHz with 502 GB of RAM. We implemented our approach in Java and used exactly the same gradient boosted trees as in Section 3.6 to ensure a fair comparison.

4.4.2 Experimentation Results

All configurations discussed here integrate infeasible path elimination and the DFS-based optimization by default. The following list summarizes the additional features that differentiate each configuration:

- *DFS*: Our approach without early stopping (Algorithm 12 for binary classification, and Algorithms 15 and 16 for multiclass classification).

- *ES*: Our approach with early stopping (Algorithm 13 for binary classification, and Algorithm 17 for multiclass classification).

Table 4.3: Number of nodes in the DAG for each binary classification dataset.

Dataset	appendicitis	divorce	glass2	promoters	threeOf9
DFS/*ES*/*PCES*/*HEUR*	7 744	350	267 799	3	3

Table 4.4: Number of nodes in the DAG for each multi-classification dataset, with percentage changes relative to *DFS* in parentheses.

Dataset	ann-thyroid	ecoli	shuttle	wine-recog.	zoo
DFS	179 260	112 208	73 300	27 722	264
ES	144 866 (-19.1%)	104 863 (-6.5%)	60 519 (-17.4%)	27 830 (+0.3%)	257 (-2.6%)
PCES	128 834 (-28.1%)	90 717 (-19.1%)	47 527 (-35.1%)	27 661 (-0.2%)	250 (-5.3%)

- *PCES*: Our approach with path condition-based early stopping (Algorithm 14 for binary classification, and Algorithm 18 for multiclass classification).
- *HEUR*: Our approach with the heuristic for abstract early stopping.

RQ1: Are our optimizations effective in terms of improving transformation time and size of the resulting DAG?

Transformation Time: First, we examine the time required to transform the gradient boosted trees to a DAG for each configuration. Figure 4.10 shows the speedup over the *DFS*-configuration. The first 5 datasets are binary classification datasets, while the other 5 datasets are multiclass classification datasets. Note that the *HEUR*-configuration only exists for binary classification.

For binary classification (*appendicitis*, *glass2*, *divorce*, *promoters*, and *threeOf9*), *ES*, *PCES*, and *HEUR* result in improvements over *DFS*, with significant speedups for *appendicitis* and *glass2*. Specifically, *PCES* achieves a speedup of $1.59\times$ on *appendicitis*, whereas *ES* achieves a speedup of $2.2\times$ on *glass2*. Although results are more mixed on the remaining datasets (due to their very low overall transformation times), the geometric mean speedups for *ES*, *PCES*, and *HEUR* are $1.12\times$, $1.07\times$, and $1.37\times$, respectively.

For multiclass classification (*ann-thyroid*, *ecoli*, *shuttle*, *wine-recog*, and *zoo*), the effect of early stopping is more significant. Both *ES* and *PCES* improve upon *DFS* in all cases, except for *PCES* on *zoo*. On *ann-thyroid*, *wine-recog*, and *zoo*, *ES* outperforms *PCES*, while on *ecoli* and *shuttle*, *PCES* provides higher speedups. The geomean speedups for *ES* and *PCES* are $2.14\times$ and $1.79\times$, indicating that both methods effectively reduce the transformation time for these datasets.

Size: We next evaluate how these configurations affect the size of the resulting DAG. Table 4.3 reports the DAG sizes (in terms of node count) for the binary classification datasets. As mentioned in Section 4.3.2, the size remains unchanged in *ES*, *PCES*, and *HEUR* relative to *DFS*. For instance, the DAGs for *promoters* and *threeOf9* each contain only one inner node and two leaves (i.e., three total nodes), while *glass2* generates a large DAG of 267,799 nodes but still completes in under two seconds using *ES*.

Table 4.4 presents absolute and relative DAG sizes for the multiclass datasets. Overall, *ES* and *PCES* reduce the DAG size compared to *DFS*, apart from a slight increase (0.3%) for *wine-recog* under *ES*. Consistent with the higher precision of *PCES*, it generally leads to even greater size reductions (e.g., up to -35.1% on *shuttle*), demonstrating that our optimizations are effective at limiting the final DAG complexity.

Appendix B (Section B.2), presents detailed results on the class characterization sizes for each class and configuration.

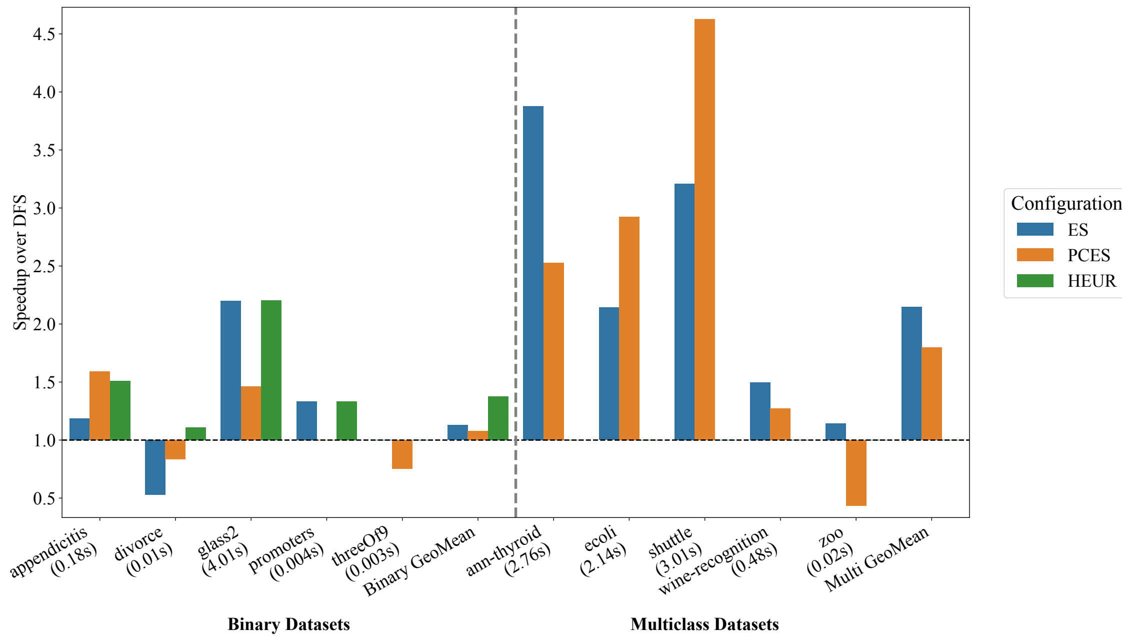


Figure 4.10: Speedup over the *DFS* configuration.

Table 4.5: Transformation times and DAG sizes for the ADD- vs. DFS-based approaches.

Dataset	Transformation Time		Speedup	Size		Size Reduction (%)
	ADD	DFS		ADD	DFS	
ann-thyroid	2719.291	1.091	2492.48×	166284	128834	-22.52%
appendicitis	1.062	0.121	8.78×	8465	7744	-8.52%
divorce	0.086	0.009	9.56×	332	350	5.42%
ecoli	798.2	0.731	1091.93×	103462	90717	-12.32%
glass2	57.076	1.818	31.39×	313093	267799	-14.47%
promoters	0.017	0.003	5.67×	3	3	0.0%
shuttle	243.351	0.65	374.39×	53921	47527	-11.86%
threeOf9	0.017	0.003	5.67×	3	3	0.0%
wine-recog	106.772	0.376	283.97×	28860	27661	-4.15%
zoo	0.218	0.037	5.89×	290	250	-13.79%
Aggregate			**51.98×**			**−8.22%**

RQ2: Can our DFS-based approach transform gradient boosted trees into a DAG more efficiently than our ADD-based approach?

Finally, we compare our DFS-based approach to the ADD-based technique in terms of transformation time and the resulting DAG size. We selected *PCES* as the best-performing ADD-based method and used *HEUR* for binary datasets and *PCES* for multiclass datasets in the DFS-based approach. Table 4.5 shows that the DFS-based procedure achieves a geometric mean speedup of $51.98\times$, indicating substantially faster transformation than the ADD-based approach. In particular, *ann-thyroid* and *ecoli* show major speedups, dropping from thousands of seconds to under two seconds of transformation time.

A likely explanation for the high transformation times in the ADD-based approach is the overhead associated with computing

$$\mathcal{A}^{wK} = \mathcal{A}_1^+ \circ_A \mathcal{A}_2^+ \circ_A \ldots \circ_A \mathcal{A}_K^+$$

Appendix C provides detailed results of the intermediate ADD sizes before and after infeasible path elimination. In some instances, the ADD is reduced to just 1% of its original size once the elimination has been applied. In contrast, the DFS-based approach avoids this problem by performing infeasible path elimination on the fly.

Regarding DAG size, the DFS-based configurations also reduce node count by 8.22% on average relative to the ADD-based baseline. For instance, *ann-thyroid* and *ecoli* show size reductions of 22.5% and 12.3%, respectively. The only exception is *divorce*, where a 5.42% increase is observed. Overall, the combination of significantly faster compilation times and smaller or comparable DAG sizes makes this DFS-based solution an alternative for efficient transformations.

Chapter 5

Analyzing Tree Ensembles

In this chapter, we present how to generate explanations, verify pre-/postcondition-based queries, and check the equivalence of two tree ensembles. All of these approaches leverage the transformation techniques introduced in the previous chapters. By converting a tree ensemble into a single, semantically equivalent directed acyclic graph (DAG), we simplify the process of analyzing tree ensembles.

In Section 5.1, we describe how to generate abductive and inflated explanations, followed by an evaluation of these methods in Section 5.2. Then, in Section 5.3, we describe our approach for verifying pre- and postcondition-based queries, as well as for checking the equivalence of two tree ensembles.

5.1 Explaining Tree Ensemble Decisions

We now describe how to generate abductive [35] and inflated explanations [38] for tree ensembles by leveraging our transformation-based approaches. We assume that a random forest or gradient boosted tree has already been transformed into a DAG G using the approaches presented in Chapters 3 and 4. Furthermore, for each class c_i, we assume that a corresponding class characterization graph G^{c_i} has been constructed. These graphs enable us to unify the explanation process for both random forests and gradient boosted trees. This section is largely based on the work presented in [3]$_{\text{AP}}$.

5.1.1 Generating Abductive Explanations

We begin by describing how to generate Abductive Explanations (AXps), which also provides a basis for generating inflated explanations. To do so, we employ the method from [54].

Given an input instance $\vec{v}$, we determine its predicted class $G(\vec{v})$ by traversing the DAG G from the root to a leaf. Suppose the prediction is class c_i. The features referenced in the predicates along this traversal form what is called the path explanation [55] of $\vec{v}$. A more concise path explanation can be obtained by considering the path taken by $\vec{v}$ in G^{c_i} (the class characterization graph for class c_i), since it contains only those predicates necessary to distinguish class c_i from all other classes.

It is possible to compute an abductive explanation in polynomial time for decision graphs [54]. Since the graph constructed by our approach is itself a type of decision graph, we can apply that algorithm directly. The procedure requires an initial seed of features $A \subseteq \mathcal{F}$, leaving $\mathcal{F} \setminus A$ as the unset variables. Although one could take $A = \mathcal{F}$ (the entire feature set), we instead initialize A to the path explanation of $\vec{v}$. Moreover, the algorithm checks whether each feature in A is necessary and removes any that are not. This path explanation is typically smaller than the full set $\mathcal{F}$ and thus provides a more efficient seed. However, in the worst case, it can be as large as $\mathcal{F}$.

5.1.2 Generating Inflated Explanations

We now describe how to generate inflated explanations for tree ensembles by extending the SMT-based approach from [38] to operate on a single graph. As before, we assume that for each class c_i, a class characterization graph G^{c_i} has already been constructed.

Consider an input instance $\vec{v} \in \mathbb{F}$ with predicted class $c_i = G(\vec{v})$. Using G^{c_i}, we first compute an abductive explanation, which provides a minimal set of necessary features $\mathcal{X} \subseteq \mathcal{F}$. Our goal is then to inflate each feature's range to make the explanation as large as possible, while still preserving the model's prediction of c_i.

Let feature $j \in \mathcal{F}$ be a fixed feature. Suppose $d_1, \ldots, d_m$ are the thresholds used by the decision trees for feature j in ascending order. These thresholds partition the domain D_j into intervals

$$(d_0, d_1), \quad [d_1, d_2), \quad \ldots, \quad [d_m, d_{m+1}),$$

where $d_0 = -\infty$ and $d_{m+1} = \infty$. Within a single interval, all feature values are indistinguishable to the ensemble, since the decision trees do not split further within that range.

We construct an initial interval assignment $\mathbb{E}_j$ for each feature j as follows:

- If $j \in \mathcal{X}$, then $\mathbb{E}_j$ is the unique interval $[d_k, d_{k+1})$ containing v_j (the value of j in $\vec{v}$).
- If $j \notin \mathcal{X}$, we initialize $\mathbb{E}_j$ to $[d_0,\ d_{m+1})$, i.e., the entire range of possible values for feature j.

Because $\mathcal{X}$ is an AXp for $\vec{v}$, this initial assignment preserves the prediction c_i, but the intervals may not be maximal yet.

Inflation Process

To inflate an interval $\mathbb{E}_j$, we iteratively increase its lower or upper bound to include neighboring threshold ranges, stopping whenever the assignment would allow a different class to become feasible. Specifically, at each step we check if Equation (2.1) still holds for the updated interval. We use the decision graph G^{c_i} to perform this check efficiently. If expanding an interval $\mathbb{E}_j$ would make it possible to reach a leaf labeled 0 (representing a class other than c_i), we revert to the previous interval. Otherwise, we keep the larger interval and continue.

Algorithm 22 shows how to verify Equation (2.1): it performs a depth-first search on G^{c_i} to see if there is a path leading to a 0-valued leaf. If such a path exists, the current interval assignment would no longer guarantee class c_i.

Algorithm 23 then systematically inflates each interval by attempting to lower and raise its bounds.

- Algorithm 24 searches thresholds downward to find the smallest feasible lower bound.
- Algorithm 25 searches upward among thresholds to find the largest feasible upper bound.

Instead of a linear search, one could also use binary search, which is better in the worst-case, but linear search can terminate faster in the best case.

By the end of this inflation procedure:

- Each feature $j \in \mathcal{X}$ has an interval $\mathbb{E}_j$ that cannot be further extended without losing the guarantee of predicting c_i.
- Features $j \notin \mathcal{X}$ remain at their full range $[d_0, d_{m+1})$, indicating that these features do not affect the class c_i for this explanation.

Therefore, the final assignment $\mathbb{E}_j$ satisfies Equation (2.1) for all j and forms a valid inflated explanation under Definition 2.

Below, we highlight several practical considerations for improving performance:

Algorithm 22 Checking the validity of equation (2.1) $[3]_{\text{AP}}$.

1: **procedure** PATHTOZERO(G^{c_i}, $(\mathcal{X}, \mathbb{X})$)
2: **if** $G^{c_i}.isLeaf()$ **then**
3: **return** $G^{c_i}.isZero()$
4: **end if**
5: $j \leftarrow G^{c_i}.feature$
6: $d \leftarrow G^{c_i}.threshold$
7: $[d_l, d_u) \leftarrow \mathbb{E}_j$
8: $G_t^{c_i} \leftarrow G^{c_i}.trueChild$
9: $G_f^{c_i} \leftarrow G^{c_i}.falseChild$
10: **if** $d_l \leq d < d_u$ **then**
11: **return** *PathToZero*($G_t^{c_i}$, $(\mathcal{X}, \mathbb{X})$) **OR** *PathToZero*($G_f^{c_i}$, $(\mathcal{X}, \mathbb{X})$)
12: **else if** $d \leq d_l$ **then**
13: **return** *PathToZero*($G_f^{c_i}$, $(\mathcal{X}, \mathbb{X})$)
14: **else**
15: **return** *PathToZero*($G_t^{c_i}$, $(\mathcal{X}, \mathbb{X})$)
16: **end if**
17: **end procedure**

- **Search Order:** We process features $\mathcal{X}$ in a fixed order, expanding each interval independently. The order in which the features are processed can affect the final inflated explanation, and the efficiency of its computation.
- **Binary vs. Linear Search:** Let m be the number of thresholds used by the tree ensemble for a specific feature. While linear search can terminate immediately after encountering a single failure, binary search ensures $O(\log m)$ complexity for finding bounds. One can choose based on the typical size of m.
- **Caching Results:** Since the graph G^{c_i} is a DAG, it is possible to visit the same node twice during one call of Algorithm 22. Although one could cache visited nodes (since each node is unique in our DAG structure), in practice we found the overhead of maintaining a visited set outweighed any performance gains.

5.2 Evaluation: Generating Abductive and Inflated Explanations for Tree Ensembles

Now, we evaluate our approach for generating abductive and inflated explanations using the ADD-based and depth-first search (DFS)-based transformations from Chapters 3 and 4. Our main objective is to measure how efficiently these methods can generate explanations once a tree ensemble has been transformed into its DAG representation.

Implementation Details and Measurement Setup: All runtimes reported here exclude the one-time overhead of transforming a tree ensemble into the DAG. This separation isolates the cost of explanation generation alone and enables more straightforward comparisons. Moreover, the transformation cost can be amortized when many explanations are required, especially for datasets with larger numbers of instances. The random forests and gradient boosted trees used for the evaluation are exactly those that were also used to evaluate the transformation techniques in Sections 3.3, 3.6, 4.2 and 4.4.

Algorithm 23 Inflating explanations [3]$_{\text{AP}}$.

```
1:  procedure INFLATEINTERVALS(G^{c_i}, v⃗)
2:      𝒳 ← AbductiveExplanation(G^{c_i}, v⃗)
3:      𝕏 ← ∅
4:      for j ∈ ℱ \ 𝒳 do
5:          𝔼_j ← [d_0, d_{m+1})
6:          𝕏 ← 𝕏 ∪ (j, 𝔼_j)
7:      end for
8:      for j ∈ 𝒳 do
9:          𝔼_j ← [d_k, d_{k+1})
10:         𝕏 ← 𝕏 ∪ (j, 𝔼_j)
11:     end for
12:     for j ∈ 𝒳 do
13:         k_0 ← InflateLowerBound(G^{c_i}, (𝒳, 𝕏), 𝔼_j)
14:         𝔼_j ← [d_{k_0}, d_{k+1})
15:         k_1 ← InflateUpperBound(G^{c_i}, (𝒳, 𝕏), 𝔼_j)
16:         𝔼_j ← [d_{k_0}, d_{k_1})
17:     end for
18: end procedure
```

Algorithm 24 Find smallest lower bound [3]$_{\text{AP}}$.

```
1:  procedure INFLATELOWERBOUND(G^{c_i}, (𝒳, 𝕏), 𝔼_j = [d_k, d_{k+1}))
2:      for k' = k − 1 down to 0 do
3:          𝔼_j ← [d_{k'}, d_{k+1})
4:          if PathToZero(G^{c_i}, (𝒳, 𝕏)) then
5:              return k' + 1
6:          end if
7:      end for
8:      return 0
9:  end procedure
```

Our implementation is written in Java, and for each instance (in both training and test sets), we run 10 "warm-up" iterations (to let the JVM optimize common execution paths), followed by 40 timing runs. We then take the median of these 40 runs as the final time for that instance. Finally, we aggregate the medians across all instances (e.g., by averaging or reporting maxima/minima) to evaluate overall performance.

5.2.1 Generating Explanations for Random Forests

We now evaluate our approach for generating abductive and inflated explanations by applying it on the DAGs obtained by applying our ADD-based and DFS-based approaches from Chapters 3 and 4 to random forests. With this evaluation, we want to answer the following research questions:

- **RQ1: Do our optimizations affect the generation of abductive/inflated explanation with the ADD-based approach?**
- **RQ2: Do our optimizations affect the generation of abductive/inflated explanation with the DFS-based approach?**
- **RQ3: Can our approaches generate explanations for random forests efficiently?**

Algorithm 25 Find largest upper bound [3]$_{\text{AP}}$.

1: **procedure** INFLATEUPPERBOUND(G^{c_i}, $(\mathcal{X}, \mathbb{X})$, $\mathbb{E}_j = [d_{k'}, d_{k+1})$)
2: **for** k'' = $k + 2$ **to** $m + 1$ **do**
3: $\mathbb{E}_j \leftarrow [d_{k'}, d_{k''})$
4: **if** *PathToZero*(G^{c_i}, $(\mathcal{X}, \mathbb{X})$) **then**
5: **return** $k'' - 1$
6: **end if**
7: **end for**
8: **return** $m + 1$
9: **end procedure**

Table 5.1: Average time (in milliseconds) and speedup comparisons for abductive explanations generated using the ADD-based approach.

Dataset	ADD Avg.	BWL avg.	Speedup BWL	SWL Avg.	Speedup SWL	AbsES Avg.	Speedup AbsES
ann-thyroid	0.0007	0.0007	0.98×	0.0007	0.98×	0.0007	0.98×
appendicitis	0.0017	0.0016	1.09×	0.0016	1.09×	0.0018	0.95×
banknote	0.0005	0.0005	0.97×	0.0005	0.98×	0.0005	0.97×
ecoli	0.0013	0.0012	1.04×	0.0012	1.08×	0.0011	1.11×
glass2	0.003	0.0029	1.04×	0.0027	1.10×	0.0032	0.94×
ionosphere	0.0353	0.033	1.07×	0.0328	1.08×	0.0341	1.03×
iris	0.0007	0.0006	1.21×	0.0006	1.11×	0.0007	0.94×
magic	0.0046	0.0048	0.97×	0.0049	0.95×	0.0048	0.97×
mofn-3-7-10	0.0003	0.0003	1.10×	0.0004	0.94×	0.0003	1.02×
new-thyroid	0.0012	0.0014	0.88×	0.0015	0.83×	0.0014	0.83×
phoneme	0.0006	0.0007	0.99×	0.0006	1.01×	0.0006	1.02×
ring	0.0045	0.0045	1.00×	0.0045	0.99×	0.0045	1.00×
segmentation	0.0086	0.0084	1.02×	0.0082	1.05×	0.0088	0.98×
shuttle	0.0003	0.0003	1.06×	0.0003	1.04×	0.0003	1.04×
threeOf9	0.0006	0.0004	1.33×	0.0006	0.94×	0.0003	1.90×
twonorm	0.0493	0.0499	0.99×	0.0496	0.99×	0.0471	1.05×
waveform-21	0.0132	0.0131	1.01×	0.0128	1.03×	0.0133	1.00×
wine-recog	0.1278	0.1329	0.96×	0.1302	0.98×	0.1299	0.98×
xd6	0.0005	0.0006	0.93×	0.0006	0.92×	0.0005	1.05×
Geomean			**1.03×**		**1.00×**		**1.02×**

RQ1: Do our optimizations affect the generation of abductive/inflated explanation with the ADD-based approach?

First, we want to see whether our transformation optimizations affect the generation of abductive/inflated explanations with our ADD-based approach. We compare the time it takes to generate abductive and inflated explanations by applying the algorithms presented in this section on the ADDs obtained with our ADD-based transformation approach. The four configurations are *ADD* (our baseline), *BWL*, *SWL*, and *AbsES*. We do not evaluate *ML* and *LUT* as these optimizations are not semantics-preserving.

Abductive Explanations with the ADD-based approach (Table 5.1): Table 5.1 reports the average runtime (in milliseconds) for producing abductive explanations across all training and test instances of each dataset. All datasets require under 1 ms on average to generate an abductive explanation. Since runtimes are already so low, none of the optimizations lead to large improvements in absolute terms. The geometric mean speedups for *BWL*, *SWL*, *AbsES* are 1.03×, 1.00×, and 1.02× respectively. Improvements become more noticeable only in a few specific

Table 5.2: Average time (in milliseconds) and speedup comparisons for inflated explanations generated using the ADD-based approach.

Dataset	**ADD Avg.**	**BWL avg.**	**Speedup BWL**	**SWL Avg.**	**Speedup SWL**	**AbsES Avg.**	**Speedup AbsES**
ann-thyroid	0.0043	0.0043	1.00×	0.0042	1.01×	0.0043	0.99×
appendicitis	0.039	0.0366	1.07×	0.0312	1.25×	0.0377	1.03×
banknote	0.0156	0.0164	0.95×	0.0157	0.99×	0.0161	0.97×
ecoli	0.0422	0.0367	1.15×	0.0346	1.22×	0.0345	1.22×
glass2	0.0657	0.0624	1.05×	0.0633	1.04×	0.0603	1.09×
ionosphere	0.1069	0.1059	1.01×	0.1051	1.02×	0.1102	0.97×
iris	0.0088	0.0099	0.89×	0.0105	0.83×	0.0085	1.03×
magic	0.0955	0.0946	1.01×	0.0965	0.99×	0.0918	1.04×
mofn-3-7-10	0.0009	0.0008	1.06×	0.0009	0.95×	0.0009	1.02×
new-thyroid	0.0117	0.0126	0.93×	0.0123	0.95×	0.0123	0.95×
phoneme	0.1844	0.184	1.00×	0.1816	1.02×	0.1866	0.99×
ring	0.0268	0.0261	1.02×	0.027	0.99×	0.0265	1.01×
segmentation	0.0649	0.0639	1.01×	0.0639	1.01×	0.0653	0.99×
shuttle	0.0011	0.001	1.05×	0.001	1.05×	0.001	1.03×
threeOf9	0.0011	0.0009	1.17×	0.0012	0.91×	0.0007	1.49×
twonorm	0.9019	0.9055	1.00×	0.8979	1.00×	0.8671	1.04×
waveform-21	0.2204	0.2216	0.99×	0.2197	1.00×	0.2262	0.97×
wine-recog	1.335	1.2749	1.05×	1.2328	1.08×	1.2223	1.09×
xd6	0.0017	0.0016	1.03×	0.0015	1.12×	0.0015	1.11×
Geomean			**1.02×**		**1.02×**		**1.05×**

cases, such as *iris*, where *BWL* and *SWL* are 1.21× and 1.11× faster, and *threeOf9*, where *AbsES* shows a 1.9× speedup. Even in those cases, the overall computation time remains well below 1 ms, so the absolute gains are modest.

Inflated Explanations with the ADD-based approach (Table 5.2): Table 5.2 shows the average runtime for generating inflated explanations, which are more complex to produce than abductive explanations. Although inflated explanations are harder to generate than abductive explanations, most datasets require under 1 ms of computation with the ADD-based approach. The only exception is *wine-recog*, which averages approximately 1.33 ms. The geometric mean speedups for *BWL*, *SWL*, and *AbsES* are 1.02×, 1.02×, and 1.05×, respectively. Some datasets see greater relative improvements, such as ecoli, where speedups of up to 1.22× are observed, and *threeOf9*, which achieves a 1.49× improvement with *AbsES*. Even in these cases, the total time stays well below 1 ms.

Overall, since runtimes are already extremely low, selecting a specific optimization (from *BWL*, *SWL*, and *AbsES*) does not significantly affect performance in practice.

RQ2: Do our optimizations affect the generation of abductive/inflated explanation with the DFS-based approach?

We perform the same experiments as in **RQ1**, except that we now use our DFS-based approach to transform random forests into DAGs and then use these DAGs to generate abductive and inflated explanations.

We only compare the configurations *DFS*, *ORD*, *DFS+S*, and *ORD+S*, as the configurations *DFS*, *ES*, *AbsES*, and *HEUR* all produce exactly the same DAG.

Abductive Explanations with the DFS-based approach (Table 5.3): Table 5.3 shows the average time it takes to generate abductive explanations with our DFS-based approach. Sim-

Table 5.3: Average time (in milliseconds) and speedup comparisons for abductive explanations generated using the DFS-based approach.

Dataset	DFS Avg.	ORD avg.	Speedup ORD	DFS+S Avg.	Speedup DFS+S	ORD+S Avg.	Speedup ORD+S
ann-thyroid	0.0062	0.0043	1.45×	0.0023	2.76×	0.0007	8.51×
appendicitis	0.0109	0.0091	1.20×	0.0086	1.28×	0.0056	1.94×
banknote	0.0005	0.0004	1.06×	0.0006	0.79×	0.0005	0.99×
ecoli	0.0021	0.0026	0.78×	0.0021	0.97×	0.0027	0.78×
glass2	0.0063	0.0058	1.09×	0.0064	0.98×	0.0042	1.49×
ionosphere	0.1165	0.0828	1.41×	0.0572	2.04×	0.0376	3.09×
iris	0.0006	0.0009	0.67×	0.0007	0.88×	0.0007	0.80×
magic	0.0247	0.0101	2.44×	0.0262	0.94×	0.0039	6.28×
mofn-3-7-10	0.0004	0.0004	1.05×	0.0003	1.41×	0.0003	1.59×
new-thyroid	0.0016	0.0016	0.98×	0.0015	1.05×	0.0013	1.20×
phoneme	0.0023	0.0016	1.44×	0.0025	0.94×	0.003	0.78×
ring	0.5686	0.0402	14.14×	4.4099	0.13×	0.0289	19.71×
segmentation	0.023	0.0211	1.09×	0.023	1.00×	0.0203	1.13×
shuttle	0.0017	0.0014	1.20×	0.0017	0.99×	0.0014	1.20×
threeOf9	0.0004	0.0004	1.15×	0.0004	1.22×	0.0003	1.31×
twonorm	0.3062	0.1099	2.79×	0.7155	0.43×	0.1309	2.34×
waveform-21	0.0621	0.0312	1.99×	0.1348	0.46×	0.0411	1.51×
wine-recog	0.0677	0.0575	1.18×	0.0304	2.23×	0.0259	2.61×
xd6	0.0007	0.0006	1.31×	0.0007	1.10×	0.0005	1.53×
Geomean			**1.44×**		**0.96×**		**1.91×**

ilar to the results of our ADD-based approach, we can see that we can leverage our DFS-based approach to generate abductive explanations efficiently. For all datasets and configurations the time it takes to generate an abductive explanation is less than 1 ms, except for the *ring* dataset with the *DFS+S* configuration.

We can see slight differences between the configurations. The order based heuristic seems to accelerate the generation of abductive explanations, as *ORD* and *ORD+S* are 44% and 91% faster. In contrast, *DFS+S* is slightly slower on average, i.e., 4% slower than *DFS+S*. One likely reason for the speedup in *ORD* and *ORD+S* is that they use a smaller initial seed of features on average (see Appendix A.1 for details), reducing the cost of generating explanations.

Some datasets, such as *ring*, show particularly large gains for *ORD+S* (e.g., 19.71×), while others (e.g., *ecoli*) show more modest or even negative speedups.

Inflated Explanations with the DFS-based approach (Table 5.4): Table 5.4 shows the average time it takes to generate inflated explanations with our DFS-based approach. While the time it takes to generate inflated explanations is higher, since it is a harder problem, our approach is still efficient at generating explanations. In most cases, it still takes less than 1 ms, e.g., for *ORD+S* only for the *twonorm* dataset it takes longer than 1 ms to generate an inflated explanation.

Similar to abductive explanations, *ORD* and *ORD+S* are more efficient at generating inflated explanations, achieving speedups of 2.18× and 2.94× respectively. Meanwhile, *DFS+S* is slightly slower, leading to a slowdown of 0.87.

The *ring* dataset is particularly interesting, because it takes *DFS* and *DFS+S* 12.56 and 61.71 milliseconds on average to generate an inflated explanation, while *ORD* and *ORD+S* are 64.81× and 95.23× times faster with 0.19 and 0.13 ms respectively.

In summary, if the primary concern is to generate explanations as quickly as possible, the combined *ORD+S* approach offers the most significant speed improvements on average across our tested datasets.

Table 5.4: Average time (in milliseconds) and speedup comparisons for inflated explanations generated using the DFS-based approach.

Dataset	**DFS Avg.**	**ORD avg.**	**Speedup ORD**	**DFS+S Avg.**	**Speedup DFS+S**	**ORD+S Avg.**	**Speedup ORD+S**
ann-thyroid	0.0768	0.0408	1.88×	0.0602	1.28×	0.0079	9.73×
appendicitis	0.7839	0.3287	2.38×	0.7854	1.0×	0.192	4.08×
banknote	0.1137	0.067	1.70×	0.1053	1.08×	0.0381	2.98×
ecoli	0.223	0.1522	1.47×	0.2513	0.89×	0.1612	1.38×
glass2	0.5032	0.278	1.81×	0.5323	0.95×	0.1578	3.19×
ionosphere	1.3258	0.6608	2.01×	0.4967	2.67×	0.2809	4.72×
iris	0.0067	0.0098	0.68×	0.0053	1.26×	0.0051	1.31×
magic	1.779	0.2246	7.92×	4.3389	0.41×	0.1287	13.83×
mofn-3-7-10	0.0011	0.001	1.05×	0.0008	1.35×	0.0008	1.41×
new-thyroid	0.045	0.0351	1.28×	0.0342	1.31×	0.0235	1.92×
phoneme	0.4388	0.1418	3.09×	0.4435	0.99×	0.192	2.29×
ring	12.565	0.1939	64.81×	61.7171	0.20×	0.1319	95.23×
segmentation	0.4007	0.3589	1.12×	0.5152	0.78×	0.4472	0.90×
shuttle	0.0224	0.015	1.50×	0.0646	0.35×	0.0282	0.80×
threeOf9	0.001	0.0008	1.30×	0.0007	1.32×	0.001	0.97×
twonorm	18.9836	4.0028	4.74×	42.57	0.45×	4.5863	4.14×
waveform-21	2.224	0.7864	2.83×	4.4828	0.50×	0.8834	2.52×
wine-recog	1.3979	1.1822	1.18×	0.8484	1.65×	0.4356	3.21×
xd6	0.0017	0.0013	1.35×	0.0016	1.06×	0.0013	1.32×
Geomean			**2.18×**		**0.87×**		**2.94×**

RQ3: Can our approaches generate explanations for random forests efficiently?

Tables 5.5 (abductive explanations) and 5.6 (inflated explanations) compare the efficiency of three methods for generating explanations:

- *rfxpl*: The SAT-based state-of-the-art approach presented in [38].
- *ADD*: The ADD-based method with the *AbsES* configuration.
- *DFS*: Our DFS-based method with the *ORD+S* configuration.

For the ADD-based approach, we take the times obtained with the *AbsES* version, while for the DFS-based approach we take the times obtained with the *ORD+S* configuration, as these were the fastest at generating explanations. Each table shows, for each dataset, the total number of instances (#inst.) and, for each method, the maximum, minimum, and average time (in milliseconds) required to generate an explanation. We also report the speedup of the ADD-based approach and DFS-based approach over *rfxpl*.

Abductive Explanations (Table 5.5): In Table 5.5, we compare abductive explanations generated by *rfxpl*, *ADD*, and *DFS*. Overall, both *ADD* and *DFS* significantly outperform the SAT-based *rfxpl* approach, often by orders of magnitude. The geometric mean of the speedups across all datasets is approximately 7314× for *ADD* and 4771× for *DFS*. For instance, in the *iris* dataset, the average time for *rfxpl* is around 52 ms, whereas *ADD* and *DFS* each require only 0.0007 ms on average, resulting in speedups on the order of tens of thousands. Datasets such as *banknote*, *ecoli*, and *mofn-3-7-10* show similarly large or even higher speedups.

A few cases show more moderate speedups, e.g., *segmentation*, *ionosphere*, *twonorm*, but even there, *DFS* and *ADD* typically reduce explanation times to well under a millisecond on average. Overall, these results confirm that both *ADD* and *DFS* can efficiently generate abductive explanations, with *ADD*'s *AbsES* variant offering the largest overall speed gains.

Table 5.5: Average time (in milliseconds) and speedup comparisons for abductive explanations generated using rfxpl, and ADD- and DFS-based approaches.

		rfxpl			ADD				DFS			
Dataset	**#inst.**	**max.**	**min.**	**avg.**	**max.**	**min.**	**avg.**	**Speedup**	**max.**	**min.**	**avg.**	**Speedup**
ann-thyroid	7129	32.0	6.0	14.61	0.021	0.0004	0.0007	20799.65×	0.027	0.0002	0.0007	20029.19×
appendicitis	106	16.0	6.0	10.96	0.016	0.0004	0.0018	6091.04×	0.0132	0.0009	0.0056	1944.58×
banknote	1348	45.0	9.0	23.94	0.0064	0.0002	0.0005	48126.34×	0.0122	0.0002	0.0005	49647.29×
ecoli	327	195.0	57.0	111.31	0.0153	0.0003	0.0011	97952.63×	0.014	0.0006	0.0027	41891.09×
glass2	162	7.0	1.0	5.3	0.0373	0.0004	0.0032	1650.36×	0.0228	0.0006	0.0042	1257.70×
ionosphere	350	6.0	0.0	3.45	1.3672	0.0008	0.0341	100.92×	0.9251	0.0016	0.0376	91.53×
iris	149	71.0	39.0	52.74	0.0108	0.0004	0.0007	71268.88×	0.0068	0.0002	0.0007	71813.38×
magic	18905	32.0	0.0	9.68	0.1163	0.0004	0.0048	2037.89×	0.0687	0.0006	0.0039	2466.95×
mofn-3-7-10	1024	41.0	10.0	22.38	0.0066	0.0001	0.0003	65600.38×	0.0043	0.0001	0.0003	83621.27×
new-thyroid	215	108.0	47.0	66.4	0.0316	0.0004	0.0014	45984.90×	0.008	0.0004	0.0013	49552.44×
phoneme	5349	92.0	22.0	43.75	0.0077	0.0001	0.0006	69421.76×	0.019	0.0005	0.003	14818.30×
ring	7400	21.0	1.0	8.25	0.0256	0.0001	0.0045	1842.16×	0.7578	0.0013	0.0289	286.03×
segmentation	210	15.0	3.0	10.56	0.1767	0.0008	0.0088	1204.17×	0.2325	0.0015	0.0203	521.05×
shuttle	58000	184.0	46.0	101.14	0.018	0.0002	0.0003	319528.65×	0.0758	0.0003	0.0014	72413.09×
threeOf9	512	27.0	5.0	13.12	0.0022	0.0002	0.0003	42063.52×	0.0021	0.0001	0.0003	39904.21×
twonorm	7400	16.0	0.0	5.55	1.3541	0.0009	0.0471	117.90×	1.5613	0.0012	0.1309	42.42×
waveform-21	5000	20.0	3.0	8.77	0.2328	0.0006	0.0133	661.14×	1.0744	0.0008	0.0411	213.36×
wine-recog	178	26.0	5.0	12.91	1.343	0.0006	0.1299	99.36×	0.3152	0.0011	0.0259	497.64×
xd6	512	35.0	10.0	19.5	0.0281	0.0001	0.0005	37649.34×	0.0092	0.0001	0.0005	39833.33×
Geomean								**7314.95×**				**4771.36×**

Table 5.6: Average time (in milliseconds) and speedup comparisons for inflated explanations generated using rfxpl, and ADD- and DFS-based approaches.

		rfxpl			ADD				DFS			
Dataset	**#inst.**	**max.**	**min.**	**avg.**	**max.**	**min.**	**avg.**	**Speedup**	**max.**	**min.**	**avg.**	**Speedup**
ann-thyroid	7129	45.0	9.0	23.38	0.4079	0.001	0.0043	5399.88×	1.025	0.0007	0.0079	2963.32×
appendicitis	106	62.0	19.0	28.81	0.7515	0.0039	0.0377	763.42×	2.9882	0.0033	0.192	150.06×
banknote	1348	182.0	92.0	127.89	0.3376	0.0009	0.0161	7935.17×	0.3678	0.0008	0.0381	3356.25×
ecoli	327	1382.0	546.0	805.59	0.9118	0.001	0.0345	23371.03×	1.7956	0.0079	0.1612	4997.93×
glass2	162	16.0	3.0	10.72	0.9542	0.001	0.0603	177.89×	0.6826	0.0026	0.1578	67.96×
ionosphere	350	10.0	1.0	4.78	5.4881	0.0031	0.1102	43.34×	6.5652	0.0067	0.2809	17.01×
iris	149	155.0	77.0	110.17	0.0944	0.0014	0.0085	12926.07×	0.0342	0.0011	0.0051	21674.95×
magic	18905	57.0	7.0	23.66	4.9371	0.0012	0.0918	257.62×	2.1353	0.006	0.1287	183.84×
mofn-3-7-10	1024	41.0	10.0	22.68	0.0107	0.0004	0.0009	25713.09×	0.0101	0.0003	0.0008	29566.71×
new-thyroid	215	382.0	193.0	284.34	0.1651	0.003	0.0123	23105.24×	0.1012	0.0054	0.0235	12120.29×
phoneme	5349	632.0	161.0	261.0	15.3799	0.0005	0.1866	1398.58×	5.0266	0.0124	0.192	1359.47×
ring	7400	32.0	4.0	15.18	0.9635	0.0004	0.0265	572.53×	5.404	0.006	0.1319	115.08×
segmentation	210	34.0	17.0	24.2	1.4175	0.0054	0.0653	370.5×	5.1711	0.0069	0.4472	54.1×
shuttle	58000	325.0	115.0	211.29	0.0475	0.0004	0.001	204269.46×	0.3537	0.0019	0.0282	7486.81×
threeOf9	512	28.0	4.0	13.05	0.0051	0.0005	0.0007	17863.97×	0.0056	0.0005	0.001	12956.46×
twonorm	7400	21.0	2.0	9.84	76.4441	0.0024	0.8671	11.35×	51.433	0.0362	4.5863	2.15×
waveform-21	5000	32.0	10.0	19.53	7.4995	0.0014	0.2262	86.34×	15.7674	0.0107	0.8834	22.11×
wine-recog	178	41.0	17.0	26.63	11.5938	0.0014	1.2223	21.79×	8.4068	0.0065	0.4356	61.15×
xd6	512	38.0	9.0	19.73	0.0401	0.0005	0.0015	13199.33×	0.0207	0.0006	0.0013	15098.86×
Geomean								**1537.85×**				**666.02×**

Inflated Explanations (Table 5.6): Table 5.6 extends the comparison to inflated explanations. Again, *rfxpl* requires substantially more time in most datasets, whereas both *ADD* and *DFS* often run in just fractions of a millisecond. The geometric mean of speedups for *ADD* is about 687×, and for *DFS* is about 297×, indicating both methods are still consistently faster than *rfxpl*, although by a somewhat smaller margin on average than in the abductive scenario.

Nonetheless, the speedups remain impressively high for specific datasets. For example, in *mofn-3-7-10*, *DFS* achieves up to 29566× faster average performance compared to *rfxpl*, and in *iris* we see a speedup of more than $10^4\times$. In some cases, e.g., *twonorm*, *waveform-21*, the overhead for inflated explanations is larger, resulting in more modest, but still substantial, speedups.

To summarize the results regarding generating explanations for random forests:

1. Both *ADD* and *DFS* approaches consistently outperform the SAT-based *rfxpl* across all tested datasets, for both abductive and inflated explanations.

2. *ADD* in the *AbsES* configuration is the fastest among the tested methods in the abductive setting, with a geometric mean speedup of roughly 7314×.

3. Inflated explanations incur a bit more overhead than abductive explanations, but *ADD* and *DFS* still maintain large speedups over *rfxpl*.

Overall, these results demonstrate the scalability and efficiency of the ADD- and DFS-based techniques compared to the state-of-the-art SAT-based approach, making them well-suited for generating explanations efficiently in real-world scenarios.

For additional information on abductive explanation sizes and the initial feature seed A, see Appendix A, Section A.1.

5.2.2 Generating Explanations for Gradient Boosted Trees

We now evaluate our approach for generating abductive and inflated explanations by applying it on the DAGs obtained by applying our ADD-based and DFS-based approaches from Chapters 3 and 4 to gradient boosted trees. With this evaluation, we want to answer the following research questions:

- **RQ1: Do our optimizations affect the generation of abductive/inflated explanation with the ADD-based approach?**
- **RQ2: Do our optimizations affect the generation of abductive/inflated explanation with the DFS-based approach?**
- **RQ3: Can our approaches generate explanations for gradient boosted trees efficiently?**

RQ1: Do our optimizations affect the generation of abductive/inflated explanation with the ADD-based approach?

We first take a look at the time it takes to generate abductive and inflated explanations with our ADD-based approach to see how the optimizations affect the generation of explanations. For ADDs we only compare the baseline configuration with the *PCES* because *ES* produced the same ADDs for the binary datasets and it cannot be applied to multiclass datasets.

Abductive and inflated explanations with the ADD-based approach (Table 5.7): Table 5.7 shows the effect of the *PCES* optimization on the time required to generate both abductive and inflated explanations with our ADD-based approach for gradient boosted trees. For all datasets, it always takes less than 1 ms on average to generate abductive and inflated explanations. Overall, the geomean values at the bottom of the table indicate that *PCES* achieves a mean speedup of $1.00\times$ for abductive explanations and $0.96\times$ for inflated explanations, indicating that *PCES* is, on average, slightly slower than the baseline ADD across these particular datasets. Nevertheless, there are individual cases, such as the *threeOf9* dataset, where *PCES* outperforms the baseline (e.g., $1.38\times$ speedup for abductive explanations). The variability in results highlights that the benefits of *PCES* can depend on dataset characteristics and the specific structure of the underlying gradient boosted trees. Therefore, *PCES* may be well-suited for specific applications, while in other contexts, the baseline ADD could remain competitive or offer better performance.

RQ2: Do our optimizations affect the generation of abductive/inflated explanation with the DFS-based approach?

Now, we evaluate our DFS-based approach. The configurations that we evaluate are *DFS*, *ES*, and *PCES*. For binary datasets all configurations produce the same DAG, so we only evaluate *DFS* for binary datasets.

Table 5.7: Average time (in milliseconds) and speedup comparisons for abductive and inflated explanations generated using the ADD-based approach for gradient boosted trees.

	Abductive			Inflated		
Dataset	**ADD Avg.**	**PCES Avg.**	**PCES Speedup**	**ADD Avg.**	**PCES Avg.**	**PCES Speedup**
ann-thyroid	0.0012	0.0014	0.85×	0.0231	0.0314	0.74×
appendicitis	0.001	0.001	0.92×	0.0058	0.0064	0.92×
divorce	0.0008	0.0008	0.94×	0.0021	0.0022	0.96×
ecoli	0.0037	0.0036	1.05×	0.1722	0.1706	1.01×
glass2	0.0033	0.0033	1.01×	0.036	0.0341	1.05×
promoters	0.0002	0.0002	0.90×	0.0006	0.0006	0.91×
shuttle	0.0009	0.0008	1.08×	0.2592	0.2484	1.04×
threeOf9	0.0003	0.0002	1.38×	0.0006	0.0005	1.22×
wine-recog	0.052	0.0523	0.99×	0.1664	0.1618	1.03×
zoo	0.0014	0.0015	0.97×	0.0038	0.0046	0.82×
Geomean			**1.00×**			**0.96×**

Table 5.8: Average time (in milliseconds) and speedup comparisons for abductive and inflated explanations generated using the DFS-based approach for gradient boosted trees.

	Abductive					Inflated				
Dataset	**DFS Avg.**	**ES avg.**	**Speedup ES**	**PCES Avg.**	**Speedup PCES**	**DFS Avg.**	**ES Avg**	**Speedup ES**	**PCES Avg.**	**Speedup PCES**
ann-thyroid	0.0057	0.0061	0.94×	0.0006	10.38×	0.0601	0.0645	0.93×	0.0094	6.36×
appendicitis	0.001	-	-	-	-	0.0068	-	-	-	-
divorce	0.0007	-	-	-	-	0.0023	-	-	-	-
ecoli	0.0023	0.0023	0.98×	0.002	1.13×	0.1165	0.1286	0.91×	0.0863	1.35×
glass2	0.0021	-	-	-	-	0.0286	-	-	-	-
promoters	0.0002	-	-	-	-	0.0006	-	-	-	-
shuttle	0.001	0.001	0.95×	0.0011	0.91×	0.168	0.2015	0.83×	0.1738	0.97×
threeOf9	0.0002	-	-	-	-	0.0005	-	-	-	-
wine-recog	0.0431	0.0512	0.84×	0.0493	0.87×	0.1403	0.1611	0.87×	0.1543	0.91×
zoo	0.0014	0.0017	0.79×	0.0017	0.82×	0.0046	0.0043	1.08×	0.0039	1.18×
Geomean			**0.9×**		**1.5×**			**0.92×**		**1.55×**

Abductive and inflated explanations with the DFS-based approach (Table 5.8): Table 5.8 shows time required to generate both abductive and inflated explanations with our DFS-based approach for gradient boosted trees. Overall, we can see that our DFS-based approach always takes less than 0.1 ms for abductive explanations and less than 1 ms for inflated explanations. For abductive explanations, the optimizations do not have a large effect except for *PCES* on the *ann-thyroid* dataset where it is 10.38 times faster. For inflated explanations, the results look similar where *PCES* is 6.36× and 1.35× faster on the *ann-thyroid* and *ecoli* datasets. Overall, these results show that *PCES* enables the most efficient generation of explanations.

RQ3: Can our approaches generate explanations for gradient boosted trees efficiently?

Tables 5.9 (abductive explanations) and 5.10 (inflated explanations) compare the efficiency of four methods for generating explanations:

- *SMT*: The SMT-based approach presented in [40].
- *MaxSAT*: The MaxSAT-based approach presented in [41][1].
- *ADD*: Our ADD-based method with the baseline configuration.
- *DFS*: Our DFS-based method with the *PCES* configuration.

[1] While there are also other approaches to generating abductive explanations such as [56] that is able to achieve slight performance improvement over [41], our approach remains significantly faster by several orders of magnitude compared to [41].

Table 5.9: Average time (in milliseconds) and speedup comparisons for abductive explanations generated using the SMT-, MaxSAT-, ADD- and DFS-based approaches.

		SMT			MaxSAT			ADD				DFS			
Dataset	**#inst.**	**max.**	**min.**	**avg.**	**max.**	**min.**	**avg.**	**max.**	**min.**	**avg.**	**Speedup**	**max.**	**min.**	**avg.**	**Speedup**
ann-thyroid	7129	1430.9	26.0	60.0	653.6	37.3	77.1	0.5224	0.0001	0.0014	54392×	0.468	0.0001	0.0006	139887×
appendicitis	106	483.1	18.0	60.0	78.4	21.0	35.2	0.0053	0.0003	0.001	33772×	0.0057	0.0005	0.0009	37145×
divorce	150	82.6	23.7	40.0	17.7	4.5	9.0	0.0231	0.0004	0.0008	11051×	0.0038	0.0003	0.0006	15471×
ecoli	327	4716.8	76.9	510.0	1120.5	268.4	719.2	0.0334	0.0004	0.0036	202106×	0.009	0.0005	0.002	362535×
glass2	162	1605.1	27.9	220.0	251.6	116.6	177.5	0.0127	0.0006	0.0033	54393×	0.0117	0.0006	0.0021	85932×
promoters	106	84.1	16.3	40.0	0.8	0.2	0.4	0.0013	0.0002	0.0002	1895×	0.0041	0.0002	0.0003	1583×
shuttle	58000	9144.2	136.9	450.0	1596.9	74.7	239.3	0.0284	0.0003	0.0008	284138×	0.0257	0.0002	0.0011	222820×
threeOf9	512	43.6	3.9	10.0	1.0	0.0	0.4	0.0013	0.0001	0.0002	2197×	0.0016	0.0001	0.0003	1501×
wine-recog	178	254.1	33.7	60.0	150.7	37.1	67.2	0.2429	0.0006	0.0523	1285×	0.1848	0.0007	0.0493	1362×
zoo	59	2049.5	104.6	350.0	58.9	5.6	16.4	0.0074	0.0006	0.0015	11210×	0.0104	0.0005	0.0017	9865×
Geomean											**18115**×				**21140**×

For the ADD-based approach, we take the times obtained with the baseline version, while for the DFS-based approach we take the times obtained with the *PCES* configuration, as these were the fastest at generating explanations. Each table shows, for each dataset, the total number of instances (#inst.) and, for each method, the maximum, minimum, and average time (in milliseconds) required to generate an explanation. We also report the speedup of the ADD-based approach and DFS-based approach over *MaxSAT*.

We used the same set of 21 gradient boosted tree models from [41]. Because our transformation approaches exceeded a reasonable time limit on 11 of these models, we limit our evaluation to the 10 that could be handled witin a reasonable time limit. Optimizing our approach to handle these more challenging models remains an open challenge for subsequent research.

Abductive Explanations (Table 5.9): In Table 5.9, we compare the abductive explanations generated by *SMT*, *MaxSAT*, *ADD*, and *DFS*. Overall, both *ADD* and *DFS* significantly outperform *SMT* and *MaxSAT*, often by orders of magnitude. The geometric mean of the speedups across all datasets is approximately 18115× for *ADD* and 21140× for *DFS*. For instance, in the *ecoli* dataset, the average time for *MaxSAT* is around 719 ms, whereas *ADD* and *DFS* each require only about 0.002–0.0036 ms on average, resulting in speedups on the order of tens of thousands. Datasets such as *shuttle*, *ann-thyroid*, and *glass2* show similarly large or even higher speedups.

A few cases show more moderate speedups, e.g., *wine-recog*, *promoters*, *threeOf9*, but even there, *DFS* and *ADD* achieve speedups of more than 3 orders of magnitude. Overall, these results confirm that both *ADD* and *DFS* can efficiently generate abductive explanations for gradient boosted trees, with *DFS*'s *PCES* variant offering the largest overall speed gains.

Inflated Explanations (Table 5.10): Table 5.10 extends the comparison to inflated explanations. The SMT, and MaxSAT approaches cannot be used to generate inflated explanations, and we are not aware of any tool that, as of today, is able to generate inflated explanations. Instead, we compare the time it takes *ADD* and *DFS* to generate inflated explanations to the time it takes *SMT* and *MaxSAT*. Note that generating inflated explanations is harder than generating abductive explanations.

Even though generating inflated explanations is harder, *ADD* and *DFS* still outperform *SMT MaxSAT* significantly. The geometric mean of speedups for ADD is about 1976×, and 2544× for *DFS*, indicating both methods are still consistently faster than *MaxSAT*. For datasets such as *ecoli* and *glass2* where it takes *MaxSAT* 719 and 177 milliseconds respectively, *ADD* and *DFS* show significant speedups of more than 4000×.

For a few datasets such as *promoters*, *threeOf9* and *wine-recog* the speedups are slightly lower, but they are still higher than two orders of magnitude. Overall, we can say that although we compare our approaches for generating inflated explanations with approaches that generate abductive explanations, our approaches are still significantly faster which shows that our approaches

Table 5.10: Average time (in milliseconds) and speedup comparisons for inflated explanations generated using the SMT-, MaxSAT-, ADD- and DFS-based approaches. The times for the SMT-, and MaxSAT-based approaches are those for generating abductive explanations, as these tools are are unable to generate inflated explanations.

		SMT			MaxSAT			ADD				DFS			
Dataset	**#inst.**	**max.**	**min.**	**avg.**	**max.**	**min.**	**avg.**	**max.**	**min.**	**avg.**	**Speedup**	**max.**	**min.**	**avg.**	**Speedup**
ann-thyroid	7129	1430.9	26.0	60.0	653.6	37.3	77.1	13.2012	0.0004	0.0314	2458×	11.5587	0.0004	0.0094	8161×
appendicitis	106	483.1	18.0	60.0	78.4	21.0	35.2	0.0449	0.0018	0.0064	5535×	0.0478	0.0017	0.006	5874×
divorce	150	82.6	23.7	40.0	17.7	4.5	9.0	0.0128	0.0013	0.0022	4100×	0.0101	0.001	0.002	4555×
ecoli	327	4716.8	76.9	510.0	1120.5	268.4	719.2	1.3361	0.0019	0.1706	4214×	0.852	0.0019	0.0863	8334×
glass2	162	1605.1	27.9	220.0	251.6	116.6	177.5	0.1997	0.0047	0.0341	5205×	0.2163	0.0035	0.0286	6203×
promoters	106	84.1	16.3	40.0	0.8	0.2	0.4	0.0035	0.0004	0.0006	617×	0.0038	0.0004	0.0007	586×
shuttle	58000	9144.2	136.9	450.0	1596.9	74.7	239.3	4.4883	0.0033	0.2484	963×	3.9662	0.0027	0.1738	1376×
threeOf9	512	43.6	3.9	10.0	1.0	0.0	0.4	0.0033	0.0002	0.0005	842×	0.0042	0.0002	0.0006	684×
wine-recog	178	254.1	33.7	60.0	150.7	37.1	67.2	0.677	0.0076	0.1618	415×	0.4292	0.0078	0.1543	435×
zoo	59	2049.5	104.6	350.0	58.9	5.6	16.4	0.037	0.0011	0.0046	3579×	0.02	0.0012	0.0039	4182×
Geomean											**1976×**				**2544×**

are well-suited for generating inflated explanations.

To summarize the results regarding generating explanations for gradient boosted trees:

1. Both *ADD* and *DFS* consistently outperform the SMT- and MaxSAT-based approaches across all tested datasets, for both abductive and inflated explanations.

2. *DFS* in the *PCES* configuration is the fastest among the tested methods in the abductive setting, with a geometric mean speedup of roughly $21140\times$.

3. Inflated explanations incur a bit more overhead than abductive explanations, but *ADD* and *DFS* still maintain large speedups over *SMT* and *MaxSAT*.

For additional information on abductive explanation sizes and the initial feature seed A, see Appendix A, Section A.2.

Compilation-based vs. Solver-based Approaches: The experimental results presented in this chapter illustrate the main strengths and limitations of our compilation-based methods for generating explanations of tree ensembles. On one hand, once a random forest or gradient boosted tree has been successfully transformed into a compact, semantically equivalent DAG, computing both abductive and inflated explanations becomes highly efficient, often faster by orders of magnitude compared to solver-based approaches (SMT, SAT, MaxSAT). This speedup is particularly valuable when multiple explanation queries or different kinds of analysis are required, or when fast, nearly instantaneous feedback is needed. In such situations, the one-time cost of transforming the model is offset by the ability to answer queries very quickly afterward.

On the other hand, the transformation itself can be expensive or even infeasible for some large or structurally complex models. In our experiments on 21 gradient boosted trees, for example, we could only compile 10 models within a reasonable time. For the remaining 11 models, the compilation process was prohibitively slow or infeasible given standard resource limits. In contrast, solver-based methods, which focus on just a single explanation or a small number of them at a time, do not require a full semantic unification of the model and therefore can handle these very large ensembles, even though at a slower per-query speed.

These observations highlight a fundamental trade-off between one-time compilation versus on-demand encoding into a solver:

- **Compilation-Based Strengths:**
  - **Fast Query Answering After Compilation:** Once a model is transformed into a DAG or ADD, each explanation query can be performed efficiently.

  - **Interactive/Real-Time Scenarios:** In settings where users frequently request explanations, e.g., interactive systems, compilation-based methods can greatly reduce the response time and make repeated querying feasible at scale.
  - **Support for Analyses Beyond Explanations:** The compiled representation can also be reused for other queries (e.g., model equivalence, pre-/postcondition checks) without having to invoke a solver again.

- **Solver-Based Strengths:**
  - **No Large Upfront Cost:** Solver-based approaches simply encode the current explanation query into a SAT or SMT instance. This makes them more suitable if the model is very large and only a small number of explanation queries need to be answered.
  - **Robustness for Hard-to-Compile Models:** For ensembles that are difficult or impossible to transform within practical time/memory limits, solver-based methods provide a fallback. Their local, query-focused approach avoids the exponential explosion that can occur when attempting to combine all trees into a single structure.

An ideal workflow might combine the two approaches by leveraging their strengths. For smaller or moderately sized models, compilation-based methods are effective by offering rapid, repeated queries. For extremely large or complex models, solver-based methods can still provide explanations, although more slowly, without the risk of infeasible compilation times. Future work could refine this synergy even further by introducing partial or approximate compilation techniques for large ensembles, or by identifying structural properties that help predict whether a given model can be compiled efficiently (see Section 8.1 for more details).

5.3 Verification of Tree Ensembles

In this section, we sketch possible approaches to verifying queries that specify both pre- and postconditions for random forests and gradient boosted trees, leveraging the transformations presented in Chapters 3 and 4. This section is largely based on the work presented in $[2]_{\mathrm{AP}}$.

5.3.1 Formulating Verification Queries

Let $M : \mathbb{F} \to K$ be a tree ensemble model. A verification query typically consists of:

- **Precondition $\phi(\vec{x})$:** We constrain each feature x_i (for $i \in \mathcal{F}$) to lie within a fixed interval $[l_i, u_i]$:
$$\phi(\vec{x}) \;\equiv\; \bigwedge_{i \in \mathcal{F}} \big(l_i \leq x_i \leq u_i\big).$$
  This corresponds to an axis-aligned hyperrectangle in $\mathbb{F}$ (one interval per feature).

- **Postcondition $\psi(y)$:** This formula specifies an acceptable subset of classes in K. For example, ψ might assert that y belongs to a particular subset $S \subseteq K$, i.e.,
$$\psi(y) \;\equiv\; y \in S.$$

The verification goal is to prove that, under the specified precondition,
$$\forall \vec{x} \in \mathbb{F}, \quad \phi(\vec{x}) \implies \psi\big(M(\vec{x})\big),$$

i.e., there is no $\vec{x}$ in the region defined by ϕ that causes M to produce an output violating ψ. Equivalently, we want to show

$$\neg\Big(\exists \vec{x} \in \mathbb{F} : \phi(\vec{x}) \;\wedge\; \neg\psi\big(M(\vec{x})\big)\Big).$$

A particularly important special case is local robustness, where we verify that small perturbations around a given reference input $\vec{x}_{\text{ref}} \in \mathbb{F}$ do not change the model's prediction. Specifically, we define

$$\phi_{\vec{x}_{\text{ref}},\boldsymbol{\delta}}(\vec{x}) \;\equiv\; \bigwedge_{i \in \mathcal{F}} \big(x_i \in \big[x_{\text{ref},i} - \delta_i,\; x_{\text{ref},i} + \delta_i\big]\big),$$

where each $\delta_i \geq 0$ specifies the maximum allowable perturbation for feature i. If all δ_i are the same value δ, then $\phi_{\vec{x}_{\text{ref}},\boldsymbol{\delta}}$ describes an L_∞-ball. If they differ, we obtain an axis-aligned hyperrectangle that can account for varying feature scales or sensitivities.

The postcondition ψ then enforces that the predicted class remains the same as $M\big(\vec{x}_{\text{ref}}\big)$. Formally, verifying local robustness requires showing:

$$\forall \vec{x} \in \mathbb{F}, \quad \phi_{\vec{x}_{\text{ref}},\boldsymbol{\delta}}(\vec{x}) \implies M(\vec{x}) = M\big(\vec{x}_{\text{ref}}\big),$$

which ensures that no point within these bounds leads to a different prediction.

5.3.2 Verifying Queries via Compilation-Based Approaches

In $[2]_{\text{AP}}$, we proposed an approach for solving verification queries by transforming random forests into ADDs. The main idea is to incorporate the precondition into the path condition that is used to eliminate infeasible paths (see Section 3.1.1). Recall that the path condition is simply a hyperrectangle, which, in the infeasible path elimination, is initialized with $(-\infty, \infty)$ for each feature. To solve verification problems, we can instead initialize the path condition with the precondition, which is also simply a hyperrectangle. Concretely:

1. Initialize the path condition to be the hyperrectangle specified by $\phi(\vec{x})$.
2. Apply any of our tree ensemble into ADD (or tree ensemble into DAG) transformations, ensuring that only those paths consistent with $\phi(\vec{x})$ remain in the model.
3. Check whether all leaves of this reduced model satisfy ψ. If every leaf meets the postcondition, the verification succeeds. If any leaf violates ψ, the verification fails.

Because this method incorporates the precondition at an early stage, it keeps the size of the DAG small and the transformation efficient. However, this approach may be less efficient when the user needs to solve multiple verification queries, each with different pre-/postconditions.

5.3.3 Verifying Queries via Compilation-Based Approaches for Multiple Queries

When verifying a large number of queries, repeatedly transforming the tree ensemble into an ADD/DAG for each pre-/postcondition can become expensive. Instead, we can:

1. Use our standard transformation algorithm once to convert the entire tree ensemble into a DAG representation, without initially restricting the domain via a precondition.
2. For a new query $\langle\phi, \psi\rangle$, traverse the DAG to identify all feasible paths under ϕ.
3. For each feasible leaf node, check if it satisfies ψ. If all such leaves satisfy ψ, the verification query holds. Otherwise, if at least one leaf violates ψ, the query fails immediately.

Although it requires an initial DAG transformation step, this method can be advantageous if one is interested in solving many verification problems on the same model: the one-time cost of DAG construction can be amortized over all subsequent queries.

In summary, both methods rely on our transformation approaches (Chapters 3 and 4), but they differ in when and how the precondition $\phi(\vec{x})$ is introduced and in how they scale to scenarios involving a large number of verification queries. If we only have a single query, incorporating the precondition from the start reduces the DAG size. On the other hand, if we must answer many distinct queries, building the DAG once without constraints can be more efficient overall, because we can quickly evaluate each new pre-/postcondition against that same DAG.

5.3.4 Verifying Equivalence of Tree Ensembles

In addition to verifying pre-/postcondition queries, a common scenario is to check whether two different tree ensembles always produce the same predictions. Such scenarios occur when a model is transformed, e.g., pruned or otherwise compressed, and one wants to ensure that the new, smaller model remains functionally equivalent to the original. For instance, [57] demonstrates a pruning approach for boosted trees that preserves the model's predictive behavior.

Equivalence between two ensembles M_1, M_2 means that both models predict the same class for every input $\vec{x} \in \mathbb{F}$. Formally, we want to show

$$\forall \vec{x} \in \mathbb{F}, \quad M_1(\vec{x}) \;=\; M_2(\vec{x}).$$

Equivalently, we seek to rule out the existence of any $\vec{x}$ such that $M_1(\vec{x}) \;\neq\; M_2(\vec{x})$. In [2]$_{\text{AP}}$, we proposed an approach for verifying the equivalence of two random forests RF_1 and RF_2. Concretely, we define the function

$$\delta_{\text{eq}}(c, c') \;=\; \begin{cases} \mathit{true} & \text{if } c = c', \\ \mathit{false} & \text{otherwise.} \end{cases}$$

The following steps outline how we use this function to check equivalence:

1. Transform RF_1 and RF_2 into ADDs A_1 and A_2 using our ADD-based transformation approach.
2. Combine A_1 and A_2 via δ_{eq}, resulting in a new ADD A_{eq}. Conceptually, $A_{\text{eq}}(\vec{x})$ evaluates to true if and only if $RF_1(\vec{x})$ and $RF_2(\vec{x})$ produce the same class.
3. Eliminate all infeasible paths in A_{eq} to obtain a reduced diagram A'_{eq}.
4. Check the result:
   - If $A'_{\text{eq}} \;\equiv\; \mathit{true}$, then for every feasible path, the outputs of RF_1 and RF_2 match. Therefore, RF_1 and RF_2 are equivalent.
   - Otherwise, if there exists any leaf that evaluates to false, we have a concrete counterexample where RF_1 and RF_2 disagree on at least one input $\vec{x}$.

This approach can also be applied to gradient boosted trees by using the techniques introduced in Section 3.4 to transform them into ADDs. Additionally, one could use the transformation-based approaches from Chapter 4 (which do not use ADDs) to verify the equivalence of two ensembles. A straightforward way would be:

1. Transform the ensemble to a DAG.
2. Convert this DAG into an ADD.

3. Apply the δ_{eq}-based procedure on the resulting ADD.

Another way to verify equivalence, which avoids an explicit δ_{eq}-based combination, is described below:

1. Transform M_1 and M_2 (which may be random forests or gradient boosted trees) into DAGs G_1 and G_2.
2. For each path in G_1 that ends in a leaf, record (i) the path condition and (ii) the resulting predicted class c.
3. Check whether there exists a path in G_2 that is feasible under the same path condition but leads to a different class $c' \neq c$.
   - If such a path in G_2 exists, there is a counterexample where M_1 and M_2 disagree.
   - If no disagreeing path exists for any path condition of G_1, then M_1 and M_2 predict the same class for all feasible inputs, and are therefore equivalent.

Comparison of Equivalence-Checking Approaches. When verifying the equivalence of two ensembles, the choice between the δ_{eq}-based approach and the path-based approach involves a trade-off in efficiency. On the one hand, the δ_{eq}-based method combines both models into a single ADD and prunes infeasible paths before checking for a ‘false’ leaf. This can simplify later analyses if one needs a unified data structure or wants to answer further equivalence-related queries. However, constructing and simplifying the combined ADD may be expensive upfront, especially if the models are large.

On the other hand, the path-based approach can potentially discover mismatches quickly: by iterating over each path in G_1 and testing it against G_2, one may stop immediately once a single path is found to produce conflicting class labels. While this early-exit capability can be highly efficient when a counterexample is easy to find, enumerating many paths to confirm a complete absence of mismatches may become more costly than using a single unified structure. Therefore, which approach is more efficient often depends on whether one expects a mismatch (and expects to detect it fast) or needs a definitive proof of equivalence that benefits from conducting all checks within a single combined diagram.

Chapter 6

Forest GUMP: A Tool For Explainability

In this chapter, we present *Forest GUMP*, a tool for explainability and verification of random forests. Forest GUMP[1] (Generalized Unifying Merge Process) illustrates how *algebraic aggregation* can be leveraged to optimize, verify, and explain random forests. We designed it to be accessible to everyone, including users without a background in IT or machine learning, by implementing it as a simple-to-use web application. The tool demonstrates the methods described in Section 3.1 to transform random forests into ADDs, and then use the ADD for explanation, verification, and equivalence checking. The following sections are largely based on the work presented in $[1]_{\mathrm{AP}}$ and $[2]_{\mathrm{AP}}$.

6.1 Overview of Forest GUMP

As shown in Figure 6.1, Forest GUMP's user interface is divided into two main areas. On the left side, users can perform the following tasks:

- **Upload or select a dataset:** Users may upload a custom dataset or choose from one of six provided datasets. The chosen dataset will be used to train the random forest (cf. (1) in Figure 6.1).
- **Set hyperparameters:** Users can specify parameters such as the number of decision trees to be learned (cf. (2) in Figure 6.1).
- **Select an aggregation method:** Multiple aggregation methods are available, including those introduced in Section 3.1 (cf. (3) in Figure 6.1). Users can also enable and configure the elimination of infeasible paths (stepwise or after merging all ADDs).
- **Classify a sample:** By entering feature values, users can classify a new instance. The path from root to leaf will be highlighted in green (satisfied predicates) and red (unsatisfied predicates) to illustrate the decision process.
- **Check equivalence:** One can verify the equivalence of two random forests.
- **Verify with pre-/postconditions:** One can apply pre-/postcondition-based verification to the learned random forest.
- **Export the model:** The currently visualized ADD can be exported to Java, C++, Python, GraphViz's `dot` format, or as an SVG file for local viewing (cf. (4) in Figure 6.1).

On the right side, Forest GUMP displays the chosen representation of the random forest as an Algebraic Decision Diagram (ADD). The gray rectangle (cf. (5) in Figure 6.1) indicates the root of this ADD, and zoom controls (cf. (6) in Figure 6.1) help navigate larger diagrams. In the top-left corner (cf. (7) in Figure 6.1), the interface shows both the total number of nodes and the length

[1] A running instance of Forest GUMP is available at `https://gitlab.com/scce/forest-gump`.

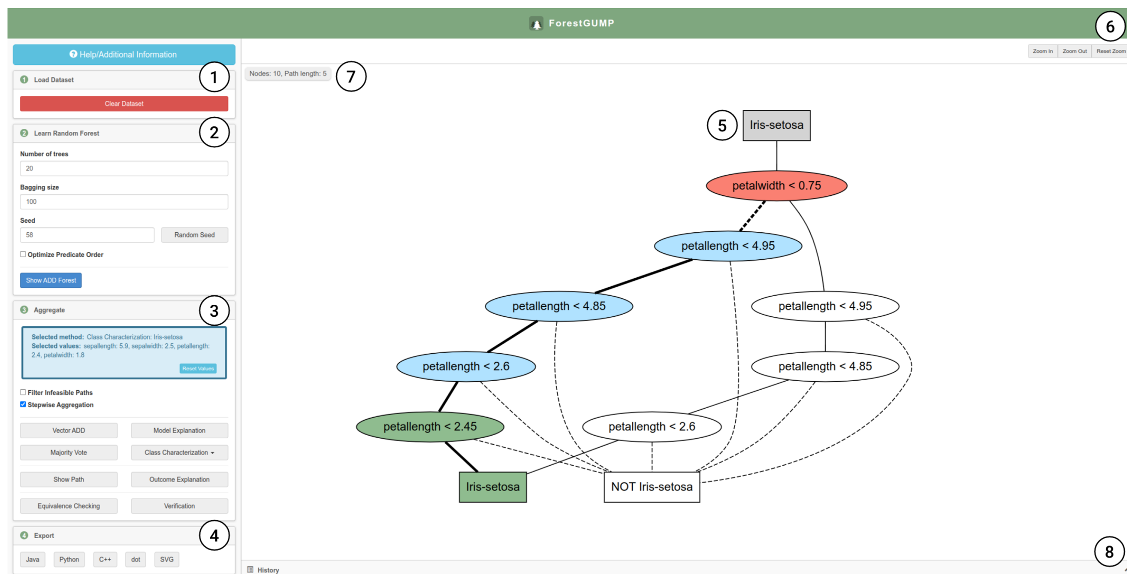


Figure 6.1: Overview of Forest GUMP. The visualized ADD is our solution to the class characterization problem (cf. Sect. 6.2.3) for the class Iris-Setosa (10 nodes, highlighted path of length 5) $[2]_{AP}$.

History

Clear History | Generate CSV

ADD Variant	Number of nodes	Maximum depth	Number of trees	Bagging size	Seed	Filter unsat. paths	Dataset	Opt. predicate order
Class Charac. Iris-setosa (filtered)	8	5	20	100	58	true	IRIS	true
Model Explanation (filtered)	196	17	20	100	58	true	IRIS	true
Class Charac. Iris-setosa (filtered)	10	5	20	100	58	true	IRIS	false
Model Explanation (filtered)	310	19	20	100	58	true	IRIS	false
ADD Forest	191	9	20	100	58	false	IRIS	false

Figure 6.2: The execution history in Forest GUMP: The user can re-execute previous setups and export the history as CSV $[2]_{AP}$.

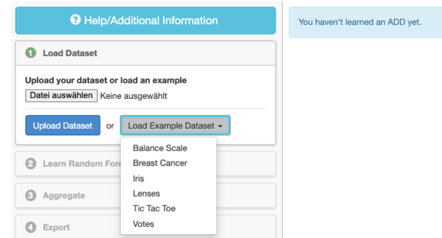


Figure 6.3: Users can choose to upload their own dataset or select one of six exemplary datasets $[2]_{AP}$.

of the currently highlighted path. A history of all visualized representations can be accessed at the bottom-right (cf. (8) in Figure 6.1).

6.2 Forest GUMP in Action

In this section, we will first illustrate the complexity of understanding a random forest's decision, and then illustrate the effects of the three explainability problems and the verification methods. The three explainability problems are:

- **Model Explanation:** The problem of making the model as a whole interpretable is solved in terms of an ADD that specifies precisely the same classification function as the original Random Forest (cf. Section 3.1).

- **Class Characterization:** The problem, given a class c, characterizing the set of all samples that are classified by the Random Forest as c. This problem is solved in terms of a BDD which precisely characterizes this set of samples (cf. Section 3.1.2).

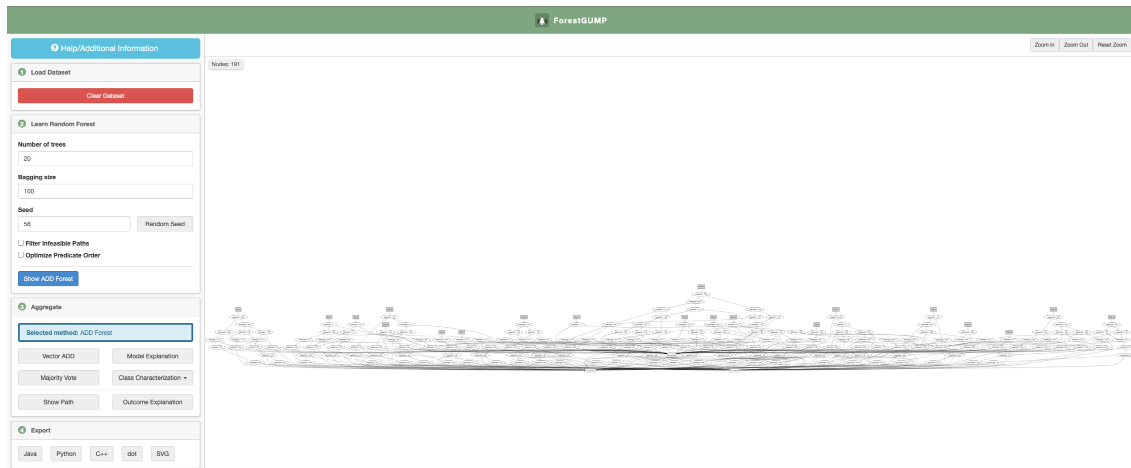


Figure 6.4: A random forest consisting of 20 individual decision trees (191 nodes, the longest path consists of 9 nodes). Note that each decision tree is represented as an ADD and that all ADDs share common subfunctions, i.e. it is essentially a shared ADD forest. The actual random forest, where nothing is shared, contains 284 nodes $[2]_{\text{AP}}$.

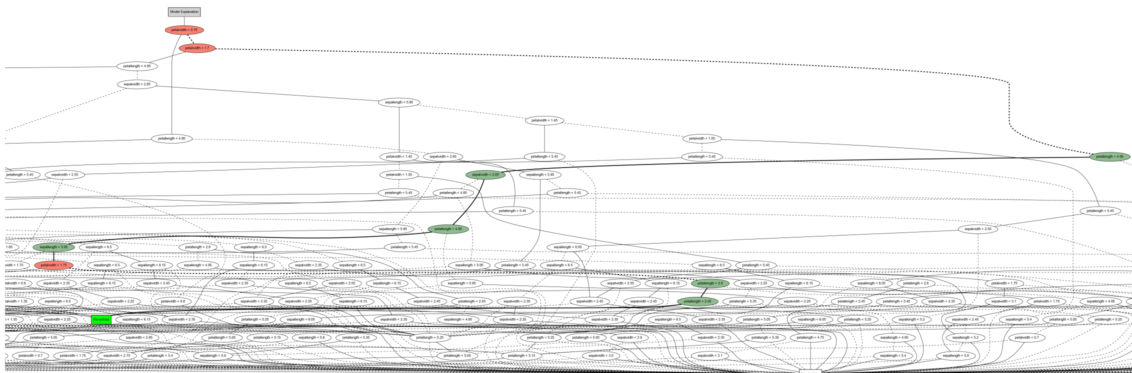


Figure 6.5: An extract of the model explanation. The ADD is constructed from the most frequent label abstraction of the aggregated random forest following an elimination of all infeasible paths (310 nodes, the longest path with length 19, the highlighted path has a length of 9) $[2]_{\text{AP}}$.

- **Outcome Explanation:** The problem of explaining a concrete classification. In Forest GUMP, this is solved by highlighting the path in the ADD that corresponds to the classification of the concrete sample and removing redundant predicates.

6.2.1 Learning a Random Forest

To begin, we need a trained random forest. In Forest GUMP, users can either upload their own dataset in the Attribute-Relation File Format (ARFF) [58] or select one of six preloaded datasets to start immediately (see Figure 6.3).

After choosing a dataset, the user must specify several hyperparameters to train the random forest (Figure 6.4):

- **Number of trees**: how many decision trees to learn within the forest.
- **Bagging size**: the fraction of samples used to learn each tree.
- **Seed**: a random seed to ensure the results can be reproduced.

Additionally, Forest GUMP provides two optional settings:

- **Elimination of infeasible paths**, which can significantly reduce the size of the resulting ADDs (see Section 3.1.1).
- **Predicate order optimization**, which can also greatly impact the size of the ADDs. By default, the predicate order is fixed, but users can enable Forest GUMP to optimize the predicate order which can result in more compact diagrams.

A random forest trained on the *Iris* dataset is shown in Figure 6.4. It consists of 20 trees[2], and was trained with the bagging size set to 100% and the seed set to 58.

When classifying a new input, each tree predicts a class label by traversing from its root to a leaf. The forest's final decision is the label predicted most frequently. While understanding how a large ensemble of trees arrives at a specific decision can be challenging in standard practice, Forest GUMP offers tools and visualizations that make this process more transparent. In the following sections, we illustrate how to leverage these capabilities to better understand, explain, and verify random forests.

6.2.2 Model Explanation

A concise white-box model corresponding to the random forest in Figure 6.4 can be constructed using the approach presented in Section 3.1. This addresses the *Model Explanation Problem* by transforming the black-box ensemble into a single ADD.

Figure 6.5 illustrates the result, producing a model with 310 nodes. Although this model may still appear overwhelming at first glance, it enables direct tracing of the classification steps. For the Iris dataset, the diagram reveals that at most 19 decisions (involving petal and sepal features) are required to reach a conclusion. These decisions constitute our set of predicates, whose conjunction forms a solution to the *Outcome Explanation Problem*. However, more succinct explanations can be found in the class characterization BDD discussed in the following section.

To demonstrate how the model explains specific outcomes, consider a sample with the features $petallength = 2.4$, $petalwidth = 1.8$, $sepallength = 5.9$, $sepalwidth = 2.5$. The outcome explanation given by the model explanation consists of the following 9 predicates (in Figure 6.5 satisfied predicates are highlighted in green, unsatisfied predicates are highlighted in red):

$$\begin{aligned}
\neg(petalwidth < 0.75) \quad &\wedge \quad \neg(petalwidth < 1.7) \quad \wedge \\
(petallength < 4.95) \quad &\wedge \quad (sepalwidth < 2.65) \quad \wedge \\
(petallength < 4.85) \quad &\wedge \quad (sepallength < 5.95) \quad \wedge \\
\neg(petalwidth < 1.75) \quad &\wedge \quad (petallength < 2.6) \quad \wedge \\
(petallength < 2.45) \quad &
\end{aligned}$$

This single-model approach is already an improvement over inspecting 20 separate decision trees, but as the following sections show, we can optimize it further.

6.2.3 Class Characterization

The class characterization problem is especially interesting because it reverses the classification perspective. Whereas the direct question is, "Given a sample, what is its predicted class?", the

[2] Note that this is a shared representation where each decision tree is represented by its own root in the shared ADD structure.

Figure 6.6: The Forest GUMP verification interface, where users can define a precondition and a postcondition.

reverse question is, "Given a class, what are the characteristics of all the samples belonging to this class?". Section 3.1.2 describes how to construct a class characterization BDD for a given majority-vote ADD.

Figure 6.1 shows a BDD that characterizes all samples guaranteed to be classified as *Iris-Setosa*. This reversal of the learned classification function has practical utility. For instance, in marketing research, where customers are profiled and matched to product offerings, reversing the mapping allows the marketing team to identify all customers considered most likely to favor a given product.

Figure 6.1 shows the highlighted path for the same sample $petallength = 2.4$, $petalwidth = 1.8$, $sepallength = 5.9$, $sepalwidth = 2.5$. Here, each node is colored in green in case the predicate holds and in red otherwise. Unlike the model explanation path of length 9, this BDD path has length 5:

$$\begin{aligned} \neg \quad &(petalwidth < 0.75) &\wedge (petallength < 4.95) \quad &\wedge \\ &(petallength < 4.85) &\wedge (petallength < 2.6) \quad &\wedge \\ &(petallength < 2.45) \end{aligned}$$

6.2.4 Outcome Explanation Problem

The classification formula from the class characterization BDD precisely captures why the sample belongs to the specified class. Even though the BDD is relatively succinct overall, we can still see redundancies within the specific local path. For example, the predicate $petallength < 2.45$ implies $petallength < 2.6$, $petallength < 4.85$ and $petallength < 4.95$. This redundancy results from the requirement in BDDs that predicates must be listed in a strict, fixed order. By removing these redundant predicates, we obtain a minimal outcome explanation for this sample:

$$\neg(petalwidth < 0.75) \wedge (petallength < 2.45).$$

In Forest GUMP, these redundant predicates appear in blue (see Figure 6.1). Thus, we have distilled the path from 9 predicates in the original model explanation, down to 5 predicates in the

class characterization BDD, and finally to just 2 predicates in the minimal outcome explanation.

6.2.5 Verification

In Section 5.3 we presented an approach for solving pre-/postcondition based verification problems using ADDs, which is also supported in Forest GUMP. Once an ADD has been constructed, the user can specify a precondition and a postcondition via the interface shown in Figure 6.6. The precondition is defined as a list of predicates, each comprising a feature, a comparison operator (e.g., ==, $<$, $\leq$, $>$, or $\geq$), and a numeric value. The precondition is then the conjunction of these predicates, which can be represented as a hyperrectangle. The postcondition is simply a subset of all classes present in the majority vote ADD, which the user selects. After the user provides the pre- and postcondition, clicking the *Verify* button constructs the ADD using the δ_{eq} operator, as described in Section 5.3.

Chapter 7

Related Work

In this chapter, we review related work on explainability approaches for tree ensemble methods. We categorize these approaches into heuristic methods that approximate model behavior, logic-based methods that apply formal reasoning, and compilation-based techniques that transform models into interpretable forms. We also provide an overview of methods for tree ensemble verification, and discuss tools for explainability.

7.1 Heuristic Explainability Approaches

Heuristic approaches, such as LIME [20], SHAP [34], and Anchor [59], aim to explain the predictions of black-box models by approximating their decision boundaries or behaviors, either locally (around specific instances) or globally (across the entire input space). These methods are typically model-agnostic and can be applied to a wide range of machine learning models, including random forests and gradient boosted trees. However, because they rely on approximations, the explanations they provide may fail to capture the full complexity of the underlying model, potentially leading to incomplete or misleading explanations.

Recent work has highlighted the limitations of heuristic approaches and underscored the need for more rigorous, logic-based explainability methods. For example, [60, 61] demonstrate that existing definitions of SHAP values can result in misleading conclusions about feature importance. Furthermore, while interpretable models, such as decision trees, are often recommended to enhance transparency [62, 63, 64], other studies [55, 65] have shown that, in practice, the paths in a decision tree can exhibit what is known as *explanation redundancy*. In such cases the logically sound explanation for a prediction is more succinct than the path itself suggests. This redundancy highlights the potential for more compact and precise explanations than those offered by many heuristic approaches, motivating the development of logic-based techniques for interpretability.

7.2 Logic-based Explainability Approaches

Unlike heuristic methods, logic-based approaches aim to provide formal and rigorous explanations by considering the internal structure of the model. One popular example is the use of abductive explanations [35, 36, 37], which identify minimal subsets of feature assignments sufficient to obtain a particular prediction. In the context of tree ensembles, abductive explanations pinpoint the smallest set of input features that, once fixed, guarantee the same outcome for a specific instance, thus highlighting which features are truly essential.

To generate abductive explanations, algorithms often rely on formal reasoning tools such as SAT [66], SMT [67], or MaxSAT [68] solvers. In these methods, the model's decision behavior is encoded into logical constraints, and the solvers are then used to determine minimal satisfying assignments that correspond to valid explanations. For example, [39] presents a SAT-based method tailored to random forests, while [41] introduces a MaxSAT solver approach for boosted trees. An optimization over the latter method, aimed at improving the transformation efficiency,

is discussed in [56].

Another class of logic-based explanations is that of inflated explanations, introduced in [38]. Whereas abductive explanations focus strictly on identifying a minimal set of necessary features, inflated explanations additionally specify an interval around each feature value that cannot be altered without changing the prediction. By including these intervals, the method offers additional information about each feature's contribution to the prediction. In [38], the authors describe an SAT-based approach to compute such explanations for random forests. A related concept, named general sufficient reasons [69], likewise extends the granularity of abductive explanations by providing more information about the conditions under which a prediction holds.

Beyond abductive and inflated explanations, numerous other logic-based methods have been proposed. For example, δ-relevant sets [70, 71] and probabilistic abductive explanations [72, 73] identify subsets of features that ensure a specific prediction with high probability. These approaches are motivated by use-cases where abductive explanations can be too large to be interpreted effectively. Similarly, [74] tackles the challenge posed by an exponential number of minimal abductive explanations in tree ensembles by introducing algorithms to compute preferred majoritary reasons. These reasons take into account user-defined preferences, thus focusing on explanations that are more likely to be interpretable and aligned with human cognitive constraints.

The availability of multiple valid abductive explanations can further complicate interpretability. To address this, [75] proposes aggregating these explanations into robust feature-importance scores through game-theoretic and causal-strength techniques, offering robustness against adversarial attacks that exploit methods such as SHAP and LIME. In [76], the authors introduce a Counter-Example Guided Abstract Refinement (CEGAR) framework that generates example-based abductive explanations covering multiple reference instances and outlines how to derive subset-minimal explanations.

Beyond explanation methods tailored to classification-based tree ensembles, there has also been work on explaining tree ensembles for regression tasks. For instance, [77] leverages Mixed-Integer Linear Programming (MILP) constraints to derive contrastive explanations for boosted regression trees, while [78] applies a MILP-based technique to generate abductive explanations.

Although we focused on abductive and inflated explanations in this thesis, the directed acyclic graphs (DAGs) produced by our transformations are applicable to generating other logic-based explanation approaches as well.

7.3 Compilation-based Explainability Approaches

Another class of methods addresses interpretability by transforming complex models, such as tree ensembles or neural networks, in a more tractable or interpretable form, for example as decision diagrams or decision trees. This approach is closely related to the broader field of knowledge compilation [79], in which logical formulas or probabilistic models are compiled into target languages that allow certain queries to be answered efficiently. However, these techniques can be computationally demanding and often struggle to scale to very large models [43].

One early example in the Bayesian network setting is [80], which compiles Bayesian network classifiers into decision graphs to enable model-level reasoning. Similarly, [81] proposes a compilation-based approach for binary neural networks into Binary Decision Diagrams or Sentential Decision Diagrams [82, 83, 84], allowing efficient analysis of the compiled models.

In the case of tree ensembles, [42, 85, 86, 87, 88] show how to compile random forests into Algebraic Decision Diagrams (ADDs). A closely related line of work focuses on transforming random forests into semantically equivalent but more compact decision trees. For example, [53] proposes an approach that leverages dynamic programming to transform a random forest into a single, minimal-size decision tree. This transformation aims to preserve the predictive behavior of

the original ensemble while resulting in a more transparent model structure. Given that finding a strictly minimal decision tree is NP-hard, [53] also provides heuristic algorithms that are typically faster but do not guarantee minimality.

In [89, 90, 91, 92], the authors present various approaches to transform piecewise linear neural networks into semantically equivalent decision trees. Their transformation approaches are based on symbolic execution and are similar to our transformation approaches presented in Chapter 4. The authors leverage their approach for interpretability and verification of neural networks.

Early Stopping: Early stopping is a common regularization technique used during the training of gradient boosted trees and neural networks to mitigate overfitting. By monitoring the performance on a validation set, the training process is terminated when further improvement stagnates, which prevents excessive optimization on the training data. To the best of our knowledge, no compilation-based approaches utilize early stopping in the same manner as our method during the compilation process. A related concept is presented in [93], where the authors propose a deep neural network architecture that enables early exits from the network to make predictions when high confidence is achieved. However, this does not directly tackle the complexity of compiling the entire model into an interpretable format. In contrast, in Chapters 3 and 4, we introduce a form of early stopping specifically tailored to the transformation of tree ensembles into DAGs, terminating the compilation process as soon as the final prediction can be determined with certainty, therefore avoiding unnecessary computation while preserving semantics.

7.4 Tree Ensemble Verification

A variety of methods have been proposed to verify the robustness and correctness of tree ensemble models [94, 95, 96, 97, 98, 99]. One class of approaches formulates the verification problem as a MILP that can be solved with off-the-shelf MILP solvers [94, 100]. Another line of work relies on SMT solvers to establish robustness guarantees [96, 101]. A third category employs abstract interpretation to derive safe over-approximations of the ensemble's behavior, enabling formal verification [102, 97, 98].

In contrast, our approach transforms the entire tree ensemble into a single, semantically equivalent DAG. This DAG representation enables direct verification of the model by leveraging analysis techniques that operate on graph structures while preserving the exact behavior of the original ensemble.

7.5 Visualization Techniques

A number of approaches have focused on visual representations and interactive interfaces for tree-based models to further enhance interpretability.

In [103], the authors propose an XAI protocol for tree-based models that supports iterative interactions between users and models. The approach allows users to validate, correct, and extend the model's knowledge using their domain expertise. Conversely, it also aids users in refining their own understanding through insights derived from the model's predictions.

PyXAI [104] is a Python library designed for working with decision trees, random forests, and boosted trees. It provides functionality for generating logic-based explanations, including abductive and contrastive explanations. The library also supports incorporating domain knowledge to refine and correct learned models. As stated by the authors, their tool 'is suited to users who are not machine learning specialists' [104], a principle that also serves as a key motivation for Forest GUMP.

In this context, Forest GUMP advances the state of the art by offering a platform for the explainability and verification of random forests. By leveraging ADDs, Forest GUMP transforms complex tree ensembles into a single interpretable representation. It enables tasks like model explanation, equivalence checking, and verification against pre-/postconditions through an intuitive web interface.

Chapter 8

Conclusion and Future Work

In this thesis, we addressed a central challenge in machine learning: improving the transparency of high-performing yet opaque tree ensemble models. Motivated by the shortcomings of traditional black-box solutions (e.g., LIME and SHAP) that only offer approximate explanations, we introduced compilation-based approaches to transform random forests and gradient boosted trees into a single semantically equivalent directed acyclic graph (DAG). This unified representation is often interpretable on its own while also supporting formal analyses such as abductive and inflated explanations, pre-/postcondition verification, and equivalence checks.

The main challenge lies in the transformation process, which can be extremely expensive for large and complex ensembles. In fact, existing approaches [42] sometimes take hours to compile the entire model, limiting real-world applicability. To mitigate this challenge, we introduced a range of optimization techniques. Early stopping optimizations terminate the compilation whenever sufficient information is collected to determine the outcome, potentially cutting the overhead significantly while preserving either exact or near-exact semantics, depending on the application's precision needs. Additionally, we compared Algebraic Decision Diagram (ADD)-based approaches against a novel depth-first search (DFS)-based method that does not rely on ADDs, enabling optimizations that were unavailable to the previous state of the art.

Once the transformation procedures were finalized, we investigated how to exploit the resulting DAG for various analytical tasks. In particular, abductive and inflated explanations become straightforward to compute from the DAG without specialized algorithms, and our experiments show performance gains of several orders of magnitude. We also showcased how this single structure supports pre-/postcondition verification and equivalence checks. Building on these ideas, we presented Forest GUMP, a web application that provides interactive compilation, explanation generation, verification, code export, and more, all within a user-friendly framework.

Despite the significant progress in scaling our compilation approach and transforming large models more rapidly than previous methods, there are still situations where certain ensembles prove too large or complex to be feasibly compiled. In those instances, solver-based approaches that encode each query individually may be more practical because they avoid the overhead of unifying the entire model, even if each query runs more slowly. For models that can be compiled, however, our methods excel at delivering many explanations or rapid feedback. Future research may explore hybrid strategies that combine partial or approximate compilation with solver-based reasoning, further extending the range of model sizes and complexities that can be handled.

Before closing, we review the main performance gains of the methods discussed in this thesis:

- **Speeding up ADD-based compilation for random forests:** By incorporating our proposed optimizations into the method from [42], which transforms random forests into ADDs, we achieve a $3.62\times$ speedup.

- **Introducing a new DFS-based transformation:** We present a novel approach based on DFS that generates a DAG instead of an ADD. On average, this DFS-based method is $20.49\times$ faster and produces a representation 2.63% smaller than the optimized ADD-based approach [42].

- **Advancing compilation for gradient boosted trees:** We propose a new method for transforming gradient boosted trees, improving upon our earlier ADD-based approach [3]$_{\text{AP}}$ by $51.98\times$ in speed and producing DAGs that are 8.22% smaller.
- **Accelerating explanation generation for random forests:** In comparison to the state-of-the-art SAT-based solution [38], for random forests our approach generates abductive explanations $7314\times$ faster and inflated explanations $687\times$ faster.
- **Delivering efficient explanations for gradient boosted trees:** Our method outperforms the MaxSAT-based solution [41] by $21140\times$ in abductive explanation speed and uniquely supports inflated explanations[1]. Although inflated explanations are harder to compute, we still generate them $2544\times$ faster than [41] can compute abductive explanations.

In conclusion, the techniques outlined in this thesis substantially advance the explainability and verification of tree ensemble models. Our approach simplifies complex ensemble logic into concise, query-efficient DAG representations, enabling greater transparency and trustworthiness in AI systems. While certain scalability challenges remain, the proposed methods offer a robust foundation for future optimizations and analyses, with the potential to extend the applicability of machine learning in safety-critical and regulated domains.

8.1 Future Work

This section outlines several potential directions that build upon the methods and results presented in this thesis. While the proposed compilation techniques and analyses have been extensively evaluated on both random forests and gradient boosted trees, additional research opportunities remain for extending these approaches to other model classes, refining existing optimizations, and enhancing tool support.

Compiling Different Input Models

The compilation techniques introduced in this thesis primarily target classification-focused random forests and gradient boosted trees. However, adapting these transformations to other machine learning models represents a promising direction for future research. In particular, random forests or gradient boosted trees for regression tasks (i.e., models that predict continuous values rather than discrete classes) could be compiled into ADDs or DAGs with only minor adjustments. For random forests, each tree outputs a continuous value, and the final prediction is given by the average (or sum) of all individual tree outputs. For gradient boosted trees, the final output often corresponds to the sum of each tree's prediction, which can similarly be encoded in a single, semantically equivalent DAG.

Such expansions would enable explanation generation and formal verification for a wider range of models, including those involving real-valued targets. For example, abductive/contrastive explanations for regression-based ensembles [77] could be directly computed once the entire model is transformed into a single DAG, making analyses of large ensembles significantly more efficient and straightforward.

Exploring Additional Explanation Types

The analyses performed in this thesis, i.e., abductive and inflated explanation generation, pre-/postcondition-based verification, and equivalence checking, represent only a subset of the pos-

[1] To our knowledge, no other approach supports inflated explanations for boosted trees, so we compare our method's inflated-explanation time to the abductive-explanation time of [41] for reference.

sible operations on tree-based models. A wide variety of additional explanation types exist, including contrastive and probabilistic abductive explanations [72], which would be interesting to explore in future work.

For abductive explanations, there can exist exponentially many different explanations [74]. Instead of computing a single abductive explanation, one might aim to identify a minimal-size explanation. Although such a requirement generally increases computational complexity, the advantages of our unified DAG representation could prove beneficial in keeping these costs low. Contrastive explanations, which aim to determine why one outcome occurs instead of another, could also be straightforwardly integrated into the existing DFS-based framework, without the need for specialized algorithms.

As new explanation methodologies continue to emerge, the ability to compile models into compact DAGs enables straightforward and efficient computation of these explanations. In particular, the overhead of generating the DAG is incurred only once regardless of the analysis method, highlighting the advantages of DAG-based representations for potentially expensive or sophisticated analyses.

Additional Optimizations

Despite the efficiency gains realized through early stopping and other compilation-based techniques, compiling extremely large random forests and gradient boosted trees remains challenging. Further scaling could be achieved by dividing an ensemble into multiple sub-models and compiling each subset of trees into a separate DAG. For instance, a random forest containing 100 trees might be partitioned into four groups of 25 trees, each compiled into its own DAG.

Although this multi-DAG strategy may increase the complexity of subsequent analyses (e.g., abductive or inflated explanation generation), it represents a viable trade-off between initial compilation cost and analysis overhead. Given that the approach developed in this thesis can generate explanations within milliseconds, a modest increase in runtime might be acceptable if it enables the handling of larger-scale models. Investigating further heuristics, e.g., directly computing a class characterization combined with a parallel implementation to accelerate the compilation process, represents another direction for future optimization research.

Visualization and Tool Support

Chapter 6 introduced *Forest GUMP*, a platform that enables the practical application of the proposed ADD-based transformations for random forests. Several extensions to this tool could be envisioned. First, supporting gradient boosted trees would broaden its applicability beyond the current random forest focus. Second, additional compilation methods (like those in Chapter 4) could be incorporated to offer a wider array of transformation strategies.

Forest GUMP already provides outcome explanations (as in [42]), but adding support for abductive and inflated explanations would extend the tool's range of analyses. Enhanced visualization tools could also be introduced to help users interpret and interact with the compiled DAGs, increasing accessibility for end-users without formal-methods expertise.

Overall, these directions suggest a wealth of opportunities for future research, including support for additional input models, new analysis capabilities, further optimizations, and improved tool support. By extending the scope of DAG-based compilation and analysis, the methods proposed in this thesis have the potential to address emerging challenges in AI explainability and verification for increasingly complex machine learning models.

List of Abbreviations

ADD Algebraic Decision Diagram

AXp Abductive Explanation

BDD Binary Decision Diagram

DAG directed acyclic graph

DFS depth-first search

LIME Local Interpretable Model-Agnostic Explanations

MaxSAT Maximum Satisfiability

MILP Mixed-Integer Linear Programming

ROBDD Reduced Ordered Binary Decision Diagram

SAT Boolean Satisfiability

SHAP SHapley Additive exPlanations

SMT Satisfiability Modulo Theories

XAI Explainable Artificial Intelligence

References

[1] Alnis Murtovi, Alexander Bainczyk, and Bernhard Steffen. “Forest GUMP: A Tool for Explanation”. In: *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part II*. Ed. by Dana Fisman and Grigore Rosu. Vol. 13244. Lecture Notes in Computer Science. Springer, 2022, pp. 314–331. DOI: 10.1007/978-3-030-99527-0_17.

[2] Alnis Murtovi, Alexander Bainczyk, Gerrit Nolte, Maximilian Schlüter, and Bernhard Steffen. “Forest GUMP: a tool for verification and explanation”. In: *Int. J. Softw. Tools Technol. Transf.* 25.3 (2023), pp. 287–299. DOI: 10.1007/S10009-023-00702-5.

[3] Alnis Murtovi, Maximilian Schlüter, and Bernhard Steffen. “Computing Inflated Explanations for Boosted Trees: A Compilation-Based Approach”. In: *The Combined Power of Research, Education, and Dissemination - Essays Dedicated to Tiziana Margaria on the Occasion of Her 60th Birthday*. Ed. by Mike Hinchey and Bernhard Steffen. Vol. 15240. Lecture Notes in Computer Science. Springer, 2025, pp. 183–201. DOI: 10.1007/978-3-031-73887-6_14.

[4] Alnis Murtovi, Maximilian Schlüter, and Bernhard Steffen. “Voting-Based Shortcuts through Random Forests for Obtaining Explainable Models”. In: *Real Time and Such - Essays Dedicated to Wang Yi to Celebrate His Scientific Career*. Ed. by Susanne Graf, Paul Pettersson, and Bernhard Steffen. Vol. 15230. Lecture Notes in Computer Science. Springer, 2025, pp. 135–153. DOI: 10.1007/978-3-031-73751-0_11.

[5] Alnis Murtovi, Maximilian Schlüter, and Bernhard Steffen. “An Efficient Compilation-Based Approach to Explaining Random Forests Through Decision Trees”. In: *Proceedings of the 17th International Conference on Agents and Artificial Intelligence, ICAART 2025 - Volume 2, Porto, Portugal, February 23-25, 2025*. Ed. by Ana Paula Rocha, Luc Steels, and H. Jaap van den Herik. SCITEPRESS, 2025, pp. 484–495. DOI: 10.5220/0013188600003890. URL: https://doi.org/10.5220/0013188600003890.

[6] Alnis Murtovi, Giorgis Georgakoudis, Konstantinos Parasyris, Chunhua Liao, Ignacio Laguna, and Bernhard Steffen. “Enhancing Performance Through Control-Flow Unmerging and Loop Unrolling on GPUs”. In: *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2024, Edinburgh, United Kingdom, March 2-6, 2024*. Ed. by Tobias Grosser, Christophe Dubach, Michel Steuwer, Jingling Xue, Guilherme Ottoni, and ernando Magno Quintão Pereira. IEEE, 2024, pp. 106–118. DOI: 10.1109/CGO57630.2024.10444819.

[7] Yann LeCun, Yoshua Bengio, and Geoffrey E. Hinton. “Deep learning”. In: *Nat.* 521.7553 (2015), pp. 436–444. DOI: 10.1038/NATURE14539. URL: https://doi.org/10.1038/nature14539.

[8] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “Imagenet classification with deep convolutional neural networks”. In: *Advances in neural information processing systems* 25 (2012).

[9] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. “Attention is All you Need”. In: *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*. Ed. by Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett. 2017, pp. 5998–6008. URL: https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

[10] Alex Graves, Abdel-rahman Mohamed, and Geoffrey E. Hinton. “Speech recognition with deep recurrent neural networks”. In: *IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2013, Vancouver, BC, Canada, May 26-31, 2013*. IEEE, 2013, pp. 6645–6649. DOI: 10.1109/ICASSP.2013.6638947.

[11] Leo Breiman. “Random Forests”. In: *Mach. Learn.* 45.1 (2001), pp. 5–32. DOI: 10.1023/A:1010933404324.

[12] Jerome H Friedman. “Greedy function approximation: a gradient boosting machine”. In: *Annals of statistics* (2001), pp. 1189–1232.

[13] Léo Grinsztajn, Edouard Oyallon, and Gaël Varoquaux. “Why do tree-based models still outperform deep learning on typical tabular data?” In: *NeurIPS*. 2022. URL: http://papers.nips.cc/paper_files/paper/2022/hash/0378c7692da36807bdec87ab043cdadc-Abstract-Datasets_and_Benchmarks.html.

[14] Ravid Shwartz-Ziv and Amitai Armon. “Tabular data: Deep learning is not all you need”. In: *Inf. Fusion* 81 (2022), pp. 84–90. DOI: 10.1016/J.INFFUS.2021.11.011.

[15] Vadim Borisov, Tobias Leemann, Kathrin Seßler, Johannes Haug, Martin Pawelczyk, and Gjergji Kasneci. “Deep Neural Networks and Tabular Data: A Survey”. In: *IEEE Transactions on Neural Networks and Learning Systems* (2022), pp. 1–21. DOI: 10.1109/TNNLS.2022.3229161.

[17] Tianqi Chen and Carlos Guestrin. “XGBoost: A Scalable Tree Boosting System”. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*. Ed. by Balaji Krishnapuram, Mohak Shah, Alexander J. Smola, Charu C. Aggarwal, Dou Shen, and Rajeev Rastogi. ACM, 2016, pp. 785–794. DOI: 10.1145/2939672.2939785.

[19] Thomas G. Dietterich. “Ensemble Methods in Machine Learning”. In: *Multiple Classifier Systems, First International Workshop, MCS 2000, Cagliari, Italy, June 21-23, 2000, Proceedings*. Ed. by Josef Kittler and Fabio Roli. Vol. 1857. Lecture Notes in Computer Science. Springer, 2000, pp. 1–15. DOI: 10.1007/3-540-45014-9_1.

[20] Marco Túlio Ribeiro, Sameer Singh, and Carlos Guestrin. “"Why Should I Trust You?": Explaining the Predictions of Any Classifier”. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*. Ed. by Balaji Krishnapuram, Mohak Shah, Alexander J. Smola, Charu C. Aggarwal, Dou Shen, and Rajeev Rastogi. ACM, 2016, pp. 1135–1144. DOI: 10.1145/2939672.2939778.

[21] Zachary C. Lipton. “The Mythos of Model Interpretability”. In: *ACM Queue* 16.3 (2018), p. 30. DOI: 10.1145/3236386.3241340.

[22] Riccardo Guidotti, Anna Monreale, Salvatore Ruggieri, Franco Turini, Fosca Giannotti, and Dino Pedreschi. “A Survey of Methods for Explaining Black Box Models”. In: *ACM Comput. Surv.* 51.5 (2019), 93:1–93:42. DOI: 10.1145/3236009.

[23] Finale Doshi-Velez and Been Kim. "Towards A Rigorous Science of Interpretable Machine Learning". In: (2017). arXiv: 1702.08608 [stat.ML]. URL: https://arxiv.org/abs/1702.08608.

[24] Solon Barocas, Moritz Hardt, and Arvind Narayanan. *Fairness and Machine Learning: Limitations and Opportunities*. MIT Press, 2023.

[25] Alvin Rajkomar, Jeffrey Dean, and Isaac Kohane. "Machine learning in medicine". In: *New England Journal of Medicine* 380.14 (2019), pp. 1347–1358. DOI: 10.1056/NEJMra1814259.

[26] Sana Tonekaboni, Shalmali Joshi, Melissa D. McCradden, and Anna Goldenberg. "What Clinicians Want: Contextualizing Explainable Machine Learning for Clinical End Use". In: *Proceedings of the Machine Learning for Healthcare Conference, MLHC 2019, 9-10 August 2019, Ann Arbor, Michigan, USA*. Ed. by Finale Doshi-Velez, Jim Fackler, Ken Jung, David C. Kale, Rajesh Ranganath, Byron C. Wallace, and Jenna Wiens. Vol. 106. Proceedings of Machine Learning Research. PMLR, 2019, pp. 359–380. URL: http://proceedings.mlr.press/v106/tonekaboni19a.html.

[27] Andreas Fuster, Paul Goldsmith-Pinkham, Tarun Ramadorai, and Ansgar Walther. "Predictably Unequal? The Effects of Machine Learning on Credit Markets". In: *The Journal of Finance* 77.1 (2022), pp. 5–47. DOI: https://doi.org/10.1111/jofi.13090.

[28] Sandra Wachter, Brent Mittelstadt, and Luciano Floridi. "Why a right to explanation of automated decision-making does not exist in the general data protection regulation". In: *International data privacy law* 7.2 (2017), pp. 76–99. DOI: 10.2139/ssrn.2903469.

[29] Reuben Binns. "Fairness in Machine Learning: Lessons from Political Philosophy". In: *Conference on Fairness, Accountability and Transparency, FAT 2018, 23-24 February 2018, New York, NY, USA*. Ed. by Sorelle A. Friedler and Christo Wilson. Vol. 81. Proceedings of Machine Learning Research. PMLR, 2018, pp. 149–159. URL: http://proceedings.mlr.press/v81/binns18a.html.

[30] Anna Jobin, Marcello Ienca, and Effy Vayena. "The global landscape of AI ethics guidelines". In: *Nat. Mach. Intell.* 1.9 (2019), pp. 389–399. DOI: 10.1038/S42256-019-0088-2.

[31] Amina Adadi and Mohammed Berrada. "Peeking Inside the Black-Box: A Survey on Explainable Artificial Intelligence (XAI)". In: *IEEE Access* 6 (2018), pp. 52138–52160. DOI: 10.1109/ACCESS.2018.2870052.

[32] Leilani H. Gilpin, David Bau, Ben Z. Yuan, Ayesha Bajwa, Michael A. Specter, and Lalana Kagal. "Explaining Explanations: An Overview of Interpretability of Machine Learning". In: *5th IEEE International Conference on Data Science and Advanced Analytics, DSAA 2018, Turin, Italy, October 1-3, 2018*. Ed. by Francesco Bonchi, Foster J. Provost, Tina Eliassi-Rad, Wei Wang, Ciro Cattuto, and Rayid Ghani. IEEE, 2018, pp. 80–89. DOI: 10.1109/DSAA.2018.00018.

[33] Matthew D. Zeiler and Rob Fergus. "Visualizing and Understanding Convolutional Networks". In: *Computer Vision - ECCV 2014 - 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part I*. Ed. by David J. Fleet, Tomás Pajdla, Bernt Schiele, and Tinne Tuytelaars. Vol. 8689. Lecture Notes in Computer Science. Springer, 2014, pp. 818–833. DOI: 10.1007/978-3-319-10590-1_53.

[34] Scott M. Lundberg and Su-In Lee. "A Unified Approach to Interpreting Model Predictions". In: *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*. Ed. by Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett. 2017, pp. 4765–4774. URL: https://proceedings.neurips.cc/paper/2017/hash/8a20a8621978632d76c43dfd28b67767-Abstract.html.

[35] Alexey Ignatiev, Nina Narodytska, and João Marques-Silva. "Abduction-Based Explanations for Machine Learning Models". In: *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*. AAAI Press, 2019, pp. 1511–1519. DOI: 10.1609/AAAI.V33I01.33011511.

[36] Andy Shih, Arthur Choi, and Adnan Darwiche. "A Symbolic Approach to Explaining Bayesian Network Classifiers". In: *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*. Ed. by Jérôme Lang. ijcai.org, 2018, pp. 5103–5111. DOI: 10.24963/IJCAI.2018/708.

[37] Adnan Darwiche and Auguste Hirth. "On the Reasons Behind Decisions". In: *ECAI 2020 - 24th European Conference on Artificial Intelligence, 29 August-8 September 2020, Santiago de Compostela, Spain, August 29 - September 8, 2020 - Including 10th Conference on Prestigious Applications of Artificial Intelligence (PAIS 2020)*. Ed. by Giuseppe De Giacomo, Alejandro Catalá, Bistra Dilkina, Michela Milano, Senén Barro, Alberto Bugarín, and Jérôme Lang. Vol. 325. Frontiers in Artificial Intelligence and Applications. IOS Press, 2020, pp. 712–720. DOI: 10.3233/FAIA200158. URL: https://doi.org/10.3233/FAIA200158.

[38] Yacine Izza, Alexey Ignatiev, Peter J. Stuckey, and João Marques-Silva. "Delivering Inflated Explanations". In: *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence, IAAI 2024, Fourteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2014, February 20-27, 2024, Vancouver, Canada*. Ed. by Michael J. Wooldridge, Jennifer G. Dy, and Sriraam Natarajan. AAAI Press, 2024, pp. 12744–12753. DOI: 10.1609/AAAI.V38I11.29170.

[39] Yacine Izza and João Marques-Silva. "On Explaining Random Forests with SAT". In: *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021*. Ed. by Zhi-Hua Zhou. ijcai.org, 2021, pp. 2584–2591. DOI: 10.24963/IJCAI.2021/356.

[40] Alexey Ignatiev, Nina Narodytska, and João Marques-Silva. "On Validating, Repairing and Refining Heuristic ML Explanations". In: *CoRR* abs/1907.02509 (2019). arXiv: 1907.02509. URL: http://arxiv.org/abs/1907.02509.

[41] Alexey Ignatiev, Yacine Izza, Peter J. Stuckey, and João Marques-Silva. "Using MaxSAT for Efficient Explanations of Tree Ensembles". In: *Thirty-Sixth AAAI Conference on Artificial Intelligence, AAAI 2022, Thirty-Fourth Conference on Innovative Applications of Artificial Intelligence, IAAI 2022, The Twelveth Symposium on Educational Advances in Artificial Intelligence, EAAI 2022 Virtual Event, February 22 - March 1, 2022*. AAAI Press, 2022, pp. 3776–3785. DOI: 10.1609/AAAI.V36I4.20292.

[42] Frederik Gossen and Bernhard Steffen. "Algebraic aggregation of random forests: towards explainability and rapid evaluation". In: *Int. J. Softw. Tools Technol. Transf.* 25.3 (2023), pp. 267–285. DOI: 10.1007/S10009-021-00635-X.

[43] João Marques-Silva. "Logic-Based Explainability: Past, Present & Future". In: *CoRR* abs/2406.11873 (2024). DOI: 10.48550/ARXIV.2406.11873. arXiv: 2406.11873.

[44] J. Ross Quinlan. "Induction of Decision Trees". In: *Mach. Learn.* 1.1 (1986), pp. 81–106. DOI: 10.1023/A:1022643204877.

[45] Chang-Yeong Lee. "Representation of switching circuits by binary-decision programs". In: *The Bell System Technical Journal* 38.4 (1959), pp. 985–999.

[46] Akers. "Binary decision diagrams". In: *IEEE Transactions on computers* 100.6 (1978), pp. 509–516.

[47] Bryant. "Graph-Based Algorithms for Boolean Function Manipulation". In: *IEEE Transactions on Computers* C-35.8 (1986), pp. 677–691. DOI: 10.1109/TC.1986.1676819.

[48] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. "Efficient Implementation of a BDD Package". In: *Proceedings of the 27th ACM/IEEE Design Automation Conference. Orlando, Florida, USA, June 24-28, 1990*. Ed. by Richard C. Smith. IEEE Computer Society Press, 1990, pp. 40–45. DOI: 10.1145/123186.123222.

[49] R. Iris Bahar, Erica A. Frohm, Charles M. Gaona, Gary D. Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. "Algebraic Decision Diagrams and Their Applications". In: *Formal Methods Syst. Des.* 10.2/3 (1997), pp. 171–206. DOI: 10.1023/A:1008699807402.

[50] Patrick Cousot and Radhia Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In: *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*. Ed. by Robert M. Graham, Michael A. Harrison, and Ravi Sethi. ACM, 1977, pp. 238–252. DOI: 10.1145/512950.512973. URL: https://doi.org/10.1145/512950.512973.

[51] Frederik Gossen, Alnis Murtovi, Philip Zweihoff, and Bernhard Steffen. "ADD-Lib: Decision Diagrams in Practice". In: *CoRR* abs/1912.11308 (2019). arXiv: 1912.11308. URL: http://arxiv.org/abs/1912.11308.

[53] Thibaut Vidal and Maximilian Schiffer. "Born-Again Tree Ensembles". In: *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*. Vol. 119. Proceedings of Machine Learning Research. PMLR, 2020, pp. 9743–9753. URL: http://proceedings.mlr.press/v119/vidal20a.html.

[54] Xuanxiang Huang, Yacine Izza, Alexey Ignatiev, and João Marques-Silva. "On Efficiently Explaining Graph-Based Classifiers". In: *Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning, KR 2021, Online event, November 3-12, 2021*. Ed. by Meghyn Bienvenu, Gerhard Lakemeyer, and Esra Erdem. 2021, pp. 356–367. DOI: 10.24963/KR.2021/34.

[55] Yacine Izza, Alexey Ignatiev, and João Marques-Silva. "On Tackling Explanation Redundancy in Decision Trees". In: *J. Artif. Intell. Res.* 75 (2022), pp. 261–321. DOI: 10.1613/JAIR.1.13575.

[56] Gilles Audemard, Jean-Marie Lagniez, Pierre Marquis, and Nicolas Szczepanski. "Computing Abductive Explanations for Boosted Trees". In: *International Conference on Artificial Intelligence and Statistics, 25-27 April 2023, Palau de Congressos, Valencia, Spain*. Ed. by Francisco J. R. Ruiz, Jennifer G. Dy, and Jan-Willem van de Meent. Vol. 206. Proceedings of Machine Learning Research. PMLR, 2023, pp. 4699–4711. URL: https://proceedings.mlr.press/v206/audemard23a.html.

[57] Youssouf Emine, Alexandre Forel, Idriss Malek, and Thibaut Vidal. "Free Lunch in the Forest: Functionally-Identical Pruning of Boosted Tree Ensembles". In: *CoRR* abs/2408.16167 (2024). DOI: 10.48550/ARXIV.2408.16167. arXiv: 2408.16167. URL: https://doi.org/10.48550/arXiv.2408.16167.

[58] Ian H. Witten, Eibe Frank, and Mark A. Hall. *Data mining: practical machine learning tools and techniques, 3rd Edition*. Morgan Kaufmann, Elsevier, 2011. ISBN: 9780123748560. URL: https://www.worldcat.org/oclc/262433473.

[59] Marco Túlio Ribeiro, Sameer Singh, and Carlos Guestrin. "Anchors: High-Precision Model-Agnostic Explanations". In: *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*. Ed. by Sheila A. McIlraith and Kilian Q. Weinberger. AAAI Press, 2018, pp. 1527–1535. DOI: 10.1609/AAAI.V32I1.11491.

[60] Xuanxiang Huang and João Marques-Silva. "On the failings of Shapley values for explainability". In: *Int. J. Approx. Reason.* 171 (2024), p. 109112. DOI: 10.1016/J.IJAR.2023.109112.

[61] João Marques-Silva and Xuanxiang Huang. "Explainability Is *Not* a Game". In: *Commun. ACM* 67.7 (2024), pp. 66–75. DOI: 10.1145/3635301.

[62] Cynthia Rudin. "Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead". In: *Nat. Mach. Intell.* 1.5 (2019), pp. 206–215. DOI: 10.1038/S42256-019-0048-X. URL: https://doi.org/10.1038/s42256-019-0048-x.

[63] Christoph Molnar. *Interpretable Machine Learning. A Guide for Making Black Box Models Explainable*. 2nd ed. 2022. URL: https://christophm.github.io/interpretable-ml-book.

[64] Cynthia Rudin, Chaofan Chen, Zhi Chen, Haiyang Huang, Lesia Semenova, and Chudi Zhong. "Interpretable Machine Learning: Fundamental Principles and 10 Grand Challenges". In: *CoRR* abs/2103.11251 (2021). arXiv: 2103.11251.

[65] João Marques-Silva and Alexey Ignatiev. "No silver bullet: interpretable ML models must be explained". In: *Frontiers Artif. Intell.* 6 (2023). DOI: 10.3389/FRAI.2023.1128212.

[66] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, eds. *Handbook of Satisfiability - Second Edition*. Vol. 336. Frontiers in Artificial Intelligence and Applications. IOS Press, 2021. ISBN: 978-1-64368-160-3. DOI: 10.3233/FAIA336.

[67] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. "Satisfiability modulo theories: introduction and applications". In: *Commun. ACM* 54.9 (2011), pp. 69–77. DOI: 10.1145/1995376.1995394.

[68] António Morgado, Federico Heras, Mark H. Liffiton, Jordi Planes, and João Marques-Silva. "Iterative and core-guided MaxSAT solving: A survey and assessment". In: *Constraints An Int. J.* 18.4 (2013), pp. 478–534. DOI: 10.1007/S10601-013-9146-2.

[69] Chunxi Ji and Adnan Darwiche. "A New Class of Explanations for Classifiers with Non-binary Features". In: *Logics in Artificial Intelligence - 18th European Conference, JELIA 2023, Dresden, Germany, September 20-22, 2023, Proceedings*. Ed. by Sarah Alice Gaggl, Maria Vanina Martinez, and Magdalena Ortiz. Vol. 14281. Lecture Notes in Computer Science. Springer, 2023, pp. 106–122. DOI: 10.1007/978-3-031-43619-2_8. URL: https://doi.org/10.1007/978-3-031-43619-2_8.

[70] Stephan Wäldchen. "Towards explainable artificial intelligence: interpreting neural network classifiers with probabilistic prime implicants". PhD thesis. Technical University of Berlin, Germany, 2022.

[71] Stephan Wäldchen, Jan MacDonald, Sascha Hauch, and Gitta Kutyniok. "The Computational Complexity of Understanding Binary Classifier Decisions". In: *J. Artif. Intell. Res.* 70 (2021), pp. 351–387. DOI: 10.1613/JAIR.1.12359. URL: https://doi.org/10.1613/jair.1.12359.

[72] Yacine Izza, Xuanxiang Huang, Alexey Ignatiev, Nina Narodytska, Martin C. Cooper, and João Marques-Silva. "On computing probabilistic abductive explanations". In: *Int. J. Approx. Reason.* 159 (2023), p. 108939. DOI: 10.1016/J.IJAR.2023.108939.

[73] Yacine Izza, Kuldeep S. Meel, and João Marques-Silva. "Locally-Minimal Probabilistic Explanations". In: *ECAI 2024 - 27th European Conference on Artificial Intelligence, 19-24 October 2024, Santiago de Compostela, Spain - Including 13th Conference on Prestigious Applications of Intelligent Systems (PAIS 2024)*. Ed. by Ulle Endriss, Francisco S. Melo, Kerstin Bach, Alberto José Bugarín Diz, Jose Maria Alonso-Moral, Senén Barro, and Fredrik Heintz. Vol. 392. Frontiers in Artificial Intelligence and Applications. IOS Press, 2024, pp. 1092–1099. DOI: 10.3233/FAIA240601.

[74] Gilles Audemard, Steve Bellart, Louenas Bounia, Frédéric Koriche, Jean-Marie Lagniez, and Pierre Marquis. "On Preferred Abductive Explanations for Decision Trees and Random Forests". In: *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI 2022, Vienna, Austria, 23-29 July 2022*. Ed. by Luc De Raedt. ijcai.org, 2022, pp. 643–650. DOI: 10.24963/IJCAI.2022/91.

[75] Gagan Biradar, Yacine Izza, Elita Lobo, Vignesh Viswanathan, and Yair Zick. "Axiomatic Aggregations of Abductive Explanations". In: *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence, IAAI 2024, Fourteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2014, February 20-27, 2024, Vancouver, Canada*. Ed. by Michael J. Wooldridge, Jennifer G. Dy, and Sriraam Natarajan. AAAI Press, 2024, pp. 11096–11104. DOI: 10.1609/AAAI.V38I10.28986.

[76] Gilles Audemard, Jean-Marie Lagniez, Pierre Marquis, and Nicolas Szczepanski. "On the Computation of Example-Based Abductive Explanations for Random Forests". In: *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI 2024, Jeju, South Korea, August 3-9, 2024*. ijcai.org, 2024, pp. 3679–3687. URL: https://www.ijcai.org/proceedings/2024/407.

[77] Gilles Audemard, Jean-Marie Lagniez, and Pierre Marquis. “On the Computation of Contrastive Explanations for Boosted Regression Trees”. In: *ECAI 2024 - 27th European Conference on Artificial Intelligence, 19-24 October 2024, Santiago de Compostela, Spain - Including 13th Conference on Prestigious Applications of Intelligent Systems (PAIS 2024)*. Ed. by Ulle Endriss, Francisco S. Melo, Kerstin Bach, Alberto José Bugarín Diz, Jose Maria Alonso-Moral, Senén Barro, and Fredrik Heintz. Vol. 392. Frontiers in Artificial Intelligence and Applications. IOS Press, 2024, pp. 1083–1091. DOI: 10.3233/FAIA240600.

[78] Gilles Audemard, Steve Bellart, Jean-Marie Lagniez, and Pierre Marquis. “Computing Abductive Explanations for Boosted Regression Trees”. In: *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI 2023, 19th-25th August 2023, Macao, SAR, China*. ijcai.org, 2023, pp. 3432–3441. DOI: 10.24963/IJCAI.2023/382.

[79] Adnan Darwiche and Pierre Marquis. “A Knowledge Compilation Map”. In: *J. Artif. Intell. Res.* 17 (2002), pp. 229–264. DOI: 10.1613/JAIR.989.

[80] Andy Shih, Arthur Choi, and Adnan Darwiche. “Compiling Bayesian Network Classifiers into Decision Graphs”. In: *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*. AAAI Press, 2019, pp. 7966–7974. DOI: 10.1609/AAAI.V33I01.33017966.

[81] Weijia Shi, Andy Shih, Adnan Darwiche, and Arthur Choi. “On Tractable Representations of Binary Neural Networks”. In: *Proceedings of the 17th International Conference on Principles of Knowledge Representation and Reasoning, KR 2020, Rhodes, Greece, September 12-18, 2020*. Ed. by Diego Calvanese, Esra Erdem, and Michael Thielscher. 2020, pp. 882–892. DOI: 10.24963/KR.2020/91.

[82] Adnan Darwiche. “SDD: A New Canonical Representation of Propositional Knowledge Bases”. In: *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*. Ed. by Toby Walsh. IJCAI/AAAI, 2011, pp. 819–826. DOI: 10.5591/978-1-57735-516-8/IJCAI11-143.

[83] Arthur Choi and Adnan Darwiche. “Dynamic Minimization of Sentential Decision Diagrams”. In: *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence, July 14-18, 2013, Bellevue, Washington, USA*. Ed. by Marie desJardins and Michael L. Littman. AAAI Press, 2013, pp. 187–194. DOI: 10.1609/AAAI.V27I1.8690.

[84] Yexiang Xue, Arthur Choi, and Adnan Darwiche. “Basing Decisions on Sentences in Decision Diagrams”. In: *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, July 22-26, 2012, Toronto, Ontario, Canada*. Ed. by Jörg Hoffmann and Bart Selman. AAAI Press, 2012, pp. 842–849. DOI: 10.1609/AAAI.V26I1.8221.

[85] Frederik Gossen, Tiziana Margaria, and Bernhard Steffen. “Towards Explainability in Machine Learning: The Formal Methods Way”. In: *IT Prof.* 22.4 (2020), pp. 8–12. DOI: 10.1109/MITP.2020.3005640. URL: https://doi.org/10.1109/MITP.2020.3005640.

[86] Frederik Gossen, Tiziana Margaria, and Bernhard Steffen. “Formal Methods Boost Experimental Performance for Explainable AI”. In: *IT Prof.* 23.6 (2021), pp. 8–12. DOI: 10.1109/MITP.2021.3123495. URL: https://doi.org/10.1109/MITP.2021.3123495.

[87] Frederik Gossen and Bernhard Steffen. "Large Random Forests: Optimisation for Rapid Evaluation". In: *CoRR* abs/1912.10934 (2019). arXiv: 1912.10934. URL: http://arxiv.org/abs/1912.10934.

[88] Frederik Jakob Gossen. "Aggressive aggregation: (Domain-specific) program optimisation with algebraic decision diagrams". PhD thesis. Technical University of Dortmund, Germany, 2021. URL: http://hdl.handle.net/2003/40625.

[89] Maximilian Schlüter and Bernhard Steffen. "Affinitree: A Compositional Framework for Formal Analysis and Explanation of Deep Neural Networks". In: *Tests and Proofs - 18th International Conference, TAP 2024, Milan, Italy, September 9-10, 2024, Proceedings*. Ed. by Marieke Huisman and Falk Howar. Vol. 15153. Lecture Notes in Computer Science. Springer, 2024, pp. 148–167. DOI: 10.1007/978-3-031-72044-4_8.

[90] Maximilian Schlüter, Gerrit Nolte, Alnis Murtovi, and Bernhard Steffen. "Towards rigorous understanding of neural networks via semantics-preserving transformations". In: *Int. J. Softw. Tools Technol. Transf.* 25.3 (2023), pp. 301–327. DOI: 10.1007/S10009-023-00700-7.

[91] Gerrit Nolte, Maximilian Schlüter, Alnis Murtovi, and Bernhard Steffen. "The power of typed affine decision structures: a case study". In: *Int. J. Softw. Tools Technol. Transf.* 25.3 (2023), pp. 355–374. DOI: 10.1007/S10009-023-00701-6.

[92] Maximilian Schlüter and Gerrit Nolte. "Introduction to Symbolic Execution of Neural Networks - Towards Faithful and Explainable Surrogate Models". In: *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.* 82 (2022). DOI: 10.14279/TUJ.ECEASST.82.1225.

[93] Surat Teerapittayanon, Bradley McDanel, and H. T. Kung. "BranchyNet: Fast inference via early exiting from deep neural networks". In: *23rd International Conference on Pattern Recognition, ICPR 2016, Cancún, Mexico, December 4-8, 2016*. IEEE, 2016, pp. 2464–2469. DOI: 10.1109/ICPR.2016.7900006. URL: https://doi.org/10.1109/ICPR.2016.7900006.

[94] Alex Kantchelian, J. D. Tygar, and Anthony D. Joseph. "Evasion and Hardening of Tree Ensemble Classifiers". In: *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*. Ed. by Maria-Florina Balcan and Kilian Q. Weinberger. Vol. 48. JMLR Workshop and Conference Proceedings. JMLR.org, 2016, pp. 2387–2396. URL: http://proceedings.mlr.press/v48/kantchelian16.html.

[95] Hongge Chen, Huan Zhang, Si Si, Yang Li, Duane S. Boning, and Cho-Jui Hsieh. "Robustness Verification of Tree-based Models". In: *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*. Ed. by Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett. 2019, pp. 12317–12328. URL: https://papers.nips.cc/paper/2019/hash/cd9508fdaa5c1390e9cc329001cf1459-Abstract.html.

[96] Gil Einziger, Maayan Goldstein, Yaniv Sa'ar, and Itai Segall. "Verifying Robustness of Gradient Boosted Models". In: *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*. AAAI Press, 2019, pp. 2446–2453. DOI: 10.1609/aaai.v33i01.33012446.

[97] Francesco Ranzato and Marco Zanella. “Abstract Interpretation of Decision Tree Ensemble Classifiers”. In: *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*. AAAI Press, 2020, pp. 5478–5486. URL: https://ojs.aaai.org/index.php/AAAI/article/view/5998.

[98] John Törnblom and Simin Nadjm-Tehrani. “Formal verification of input-output mappings of tree ensembles”. In: *Sci. Comput. Program.* 194 (2020), p. 102450. DOI: 10.1016/J.SCICO.2020.102450.

[99] Laurens Devos, Lorenzo Cascioli, and Jesse Davis. “Robustness Verification of Multi-Class Tree Ensembles”. In: *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence, IAAI 2024, Fourteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2014, February 20-27, 2024, Vancouver, Canada*. Ed. by Michael J. Wooldridge, Jennifer G. Dy, and Sriraam Natarajan. AAAI Press, 2024, pp. 21019–21028. DOI: 10.1609/AAAI.V38I19.30093.

[100] Chong Zhang, Huan Zhang, and Cho-Jui Hsieh. “An Efficient Adversarial Attack for Tree Ensembles”. In: *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*. Ed. by Hugo Larochelle, Marc’Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin. 2020. URL: https://proceedings.neurips.cc/paper/2020/hash/ba3e9b6a519cfddc560b5d53210df1bd-Abstract.html.

[101] Laurens Devos, Wannes Meert, and Jesse Davis. “Verifying Tree Ensembles by Reasoning about Potential Instances”. In: *Proceedings of the 2021 SIAM International Conference on Data Mining, SDM 2021, Virtual Event, April 29 - May 1, 2021*. Ed. by Carlotta Demeniconi and Ian Davidson. SIAM, 2021, pp. 450–458. DOI: 10.1137/1.9781611976700.51.

[102] Stefano Calzavara, Pietro Ferrara, and Claudio Lucchese. “Certifying Decision Trees Against Evasion Attacks by Program Analysis”. In: *Computer Security - ESORICS 2020 - 25th European Symposium on Research in Computer Security, ESORICS 2020, Guildford, UK, September 14-18, 2020, Proceedings, Part II*. Ed. by Liqun Chen, Ninghui Li, Kaitai Liang, and Steve A. Schneider. Vol. 12309. Lecture Notes in Computer Science. Springer, 2020, pp. 421–438. DOI: 10.1007/978-3-030-59013-0_21.

[103] Gilles Audemard, Sylvie Coste-Marquis, Pierre Marquis, Mehdi Sabiri, and Nicolas Szczepanski. “Designing an XAI Interface for Tree-Based ML Models”. In: *ECAI 2024 - 27th European Conference on Artificial Intelligence, 19-24 October 2024, Santiago de Compostela, Spain - Including 13th Conference on Prestigious Applications of Intelligent Systems (PAIS 2024)*. Ed. by Ulle Endriss, Francisco S. Melo, Kerstin Bach, Alberto José Bugarín Diz, Jose Maria Alonso-Moral, Senén Barro, and Fredrik Heintz. Vol. 392. Frontiers in Artificial Intelligence and Applications. IOS Press, 2024, pp. 1075–1082. DOI: 10.3233/FAIA240599.

[104] Gilles Audemard, Jean-Marie Lagniez, Pierre Marquis, and Nicolas Szczepanski. “PyXAI: An XAI Library for Tree-Based Models”. In: *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI 2024, Jeju, South Korea, August 3-9, 2024*. ijcai.org, 2024, pp. 8601–8605. URL: https://www.ijcai.org/proceedings/2024/989.

Online References

[16] Kaggle. *Kaggle: Your Machine Learning and Data Science Community*. Accessed: 2025-01-13. URL: https://www.kaggle.com/.

[18] RK Kaggle. *State of Data Science and Machine Learning 2021*. Accessed: 2025-01-13. 2021. URL: https://www.kaggle.com/kaggle-survey-2021.

[52] Arthur Asuncion, David Newman, et al. *UCI machine learning repository*. Accessed: 2025-01-13. 2007. URL: https://archive.ics.uci.edu/.

Appendix A

Abductive Explanation Sizes

A.1 Random Forests

Tables A.1 and A.2 compare the *absolute* size of abductive explanations (Table A.1) and the *relative* size (Table A.2) for several methods: the SAT-based approach (*rfxpl*), the ADD-based approach (ADD), and the DFS-based approach (DFS), along with the proposed optimizations (BWL, SWL, AbsES, ORD, etc.). Table A.1 shows how large the explanations are on average for each dataset, while Table A.2 provides the ratio of each method's explanation size relative to *rfxpl*, making it clear which methods produce larger or smaller explanations compared to the SAT-based baseline.

Tables A.3 and A.4 focus on the initial seed of features, i.e., the features on the root-to-leaf path for each instance (see Section 5.1.1). Table A.3 presents the average number of features encountered along that path, whereas Table A.4 shows how each method's path length compares to the ADD baseline. This path length is particularly important because it defines the starting set of features (A) from which abductive explanations are derived.

Table A.1: Random Forests: Average size of abductive explanations.

		ADD				DFS			
Dataset	**rfxpl**	**ADD**	**BWL**	**SWL**	**AbsES**	**DFS**	**ORD**	**DFS+S**	**ORD+S**
ann-thyroid	2.59	8.68	8.68	8.68	8.68	2.6	2.61	2.61	2.56
appendicitis	3.69	6.47	6.47	6.47	6.47	3.76	3.66	3.8	3.76
banknote	2.09	2.78	2.78	2.78	2.78	2.06	2.06	2.06	2.07
ecoli	3.49	4.56	4.56	4.56	4.56	3.41	3.52	3.5	3.55
glass2	4.8	5.98	5.98	5.98	5.96	4.72	4.82	4.69	4.75
ionosphere	12.13	13.3	13.3	13.3	13.27	10.54	9.52	10.54	9.67
iris	2.06	3.01	3.01	3.01	3.01	2.06	2.06	2.06	2.06
magic	4.98	6.91	6.91	6.91	6.91	4.06	4.26	4.05	4.0
mofn-3-7-10	2.31	2.45	2.45	2.45	2.45	2.3	2.3	2.3	2.3
new-thyroid	2.83	4.6	4.6	4.6	4.6	2.99	2.99	2.99	2.99
phoneme	2.51	3.92	3.92	3.92	3.92	2.61	2.59	2.58	2.62
ring	9.44	12.17	12.17	12.17	12.2	-	8.96	8.92	8.94
segmentation	8.27	11.69	11.69	11.69	11.69	8.26	8.21	8.26	8.23
shuttle	2.79	4.6	4.6	4.6	4.6	2.77	2.88	2.77	2.76
threeOf9	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
twonorm	9.47	12.44	12.44	12.44	12.44	9.24	9.21	9.12	9.0
waveform-21	7.03	9.71	9.71	9.71	9.71	6.84	6.97	6.88	6.81
wine-recog	4.67	8.4	8.4	8.4	8.4	6.08	5.68	6.1	5.6
xd6	3.36	4.88	4.88	4.88	4.88	3.27	3.27	3.26	3.27

A.2 Gradient Boosted Trees

Similar to the random forests results, we now compare abductive explanation sizes and root-to-leaf path lengths for gradient boosted trees. Tables A.5 and A.6 report absolute and relative explanation sizes (here, relative to the SMT approach), while Tables A.7 and A.8 focus on the average number of features along the path from root to leaf.

Table A.2: Random forests: Ratio of each method's abductive explanation size to the rfxpl baseline.

	ADD				DFS			
Dataset	**ADD**	**BWL**	**SWL**	**AbsES**	**DFS**	**ORD**	**DFS+S**	**ORD+S**
ann-thyroid	3.35	3.35	3.35	3.35	1.01	1.01	1.01	0.99
appendicitis	1.75	1.75	1.75	1.75	1.02	0.99	1.03	1.02
banknote	1.33	1.33	1.33	1.33	0.99	0.99	0.99	0.99
ecoli	1.31	1.31	1.31	1.31	0.98	1.01	1.0	1.02
glass2	1.25	1.25	1.25	1.24	0.98	1.01	0.98	0.99
ionosphere	1.1	1.1	1.1	1.09	0.87	0.79	0.87	0.8
iris	1.46	1.46	1.46	1.46	1.0	1.0	1.0	1.0
magic	1.39	1.39	1.39	1.39	0.81	0.86	0.81	0.8
mofn-3-7-10	1.06	1.06	1.06	1.06	1.0	1.0	1.0	1.0
new-thyroid	1.62	1.62	1.62	1.62	1.06	1.05	1.06	1.05
phoneme	1.56	1.56	1.56	1.56	1.04	1.03	1.03	1.04
ring	1.29	1.29	1.29	1.29	-	0.95	0.95	0.95
segmentation	1.41	1.41	1.41	1.41	1.0	0.99	1.0	1.0
shuttle	1.65	1.65	1.65	1.65	0.99	1.03	0.99	0.99
threeOf9	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
twonorm	1.31	1.31	1.31	1.31	0.98	0.97	0.96	0.95
waveform-21	1.38	1.38	1.38	1.38	0.97	0.99	0.98	0.97
wine-recog	1.8	1.8	1.8	1.8	1.3	1.22	1.31	1.2
xd6	1.45	1.45	1.45	1.45	0.97	0.97	0.97	0.97
Geomean	$\mathbf{1.44\times}$	$\mathbf{1.44\times}$	$\mathbf{1.44\times}$	$\mathbf{1.44\times}$	$\mathbf{0.99\times}$	$\mathbf{0.99\times}$	$\mathbf{0.99\times}$	$\mathbf{0.98\times}$

Table A.3: Random forests: Average number of features on path.

	ADD				DFS			
Dataset	**ADD**	**BWL**	**SWL**	**AbsES**	**DFS**	**ORD**	**DFS+S**	**ORD+S**
ann-thyroid	8.83	8.83	8.83	8.83	6.18	4.16	6.22	3.1
appendicitis	6.66	6.66	6.66	6.66	6.97	6.18	6.85	6.38
banknote	3.31	3.31	3.31	3.31	3.2	3.55	3.23	3.24
ecoli	5.17	5.17	5.17	5.17	4.64	5.79	4.98	5.72
glass2	7.23	7.23	7.23	7.21	7.09	7.19	7.05	7.23
ionosphere	15.27	15.27	15.27	15.24	16.3	14.07	15.68	13.47
iris	3.68	3.68	3.68	3.68	3.87	2.74	3.87	2.42
magic	8.02	8.02	8.02	8.02	7.49	8.12	7.65	6.7
mofn-3-7-10	4.3	4.3	4.3	4.3	3.83	3.92	3.83	3.92
new-thyroid	5.0	5.0	5.0	5.0	5.0	4.73	4.98	4.74
phoneme	4.35	4.35	4.35	4.35	4.29	4.43	4.08	4.67
ring	15.14	15.14	15.14	15.17	-	14.39	15.39	13.38
segmentation	12.31	12.31	12.31	12.31	11.55	10.83	11.55	10.85
shuttle	4.6	4.6	4.6	4.6	5.06	5.6	5.06	5.37
threeOf9	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
twonorm	15.11	15.11	15.11	15.11	15.86	15.85	15.11	14.5
waveform-21	11.93	11.93	11.93	11.93	12.05	12.37	11.85	11.57
wine-recog	10.69	10.69	10.69	10.69	10.53	8.73	10.53	8.52
xd6	5.57	5.57	5.57	5.57	6.01	5.33	5.82	5.16

Table A.4: Random forests: Ratio of each method's average path length (number of features) to the ADD baseline.

	ADD			DFS			
Dataset	**BWL**	**SWL**	**AbsES**	**DFS**	**ORD**	**DFS+S**	**ORD+S**
ann-thyroid	1.0×	1.0×	1.0×	0.7×	0.47×	0.7×	0.35×
appendicitis	1.0×	1.0×	1.0×	1.05×	0.93×	1.03×	0.96×
banknote	1.0×	1.0×	1.0×	0.97×	1.07×	0.97×	0.98×
ecoli	1.0×	1.0×	1.0×	0.9×	1.12×	0.96×	1.1×
glass2	1.0×	1.0×	1.0×	0.98×	0.99×	0.98×	1.0×
ionosphere	1.0×	1.0×	1.0×	1.07×	0.92×	1.03×	0.88×
iris	1.0×	1.0×	1.0×	1.05×	0.74×	1.05×	0.66×
magic	1.0×	1.0×	1.0×	0.93×	1.01×	0.95×	0.83×
mofn-3-7-10	1.0×	1.0×	1.0×	0.89×	0.91×	0.89×	0.91×
new-thyroid	1.0×	1.0×	1.0×	1.0×	0.95×	1.0×	0.95×
phoneme	1.0×	1.0×	1.0×	0.99×	1.02×	0.94×	1.07×
ring	1.0×	1.0×	1.0×	-	0.95×	1.02×	0.88×
segmentation	1.0×	1.0×	1.0×	0.94×	0.88×	0.94×	0.88×
shuttle	1.0×	1.0×	1.0×	1.1×	1.22×	1.1×	1.17×
threeOf9	1.0×	1.0×	1.0×	1.0×	1.0×	1.0×	1.0×
twonorm	1.0×	1.0×	1.0×	1.05×	1.05×	1.0×	0.96×
waveform-21	1.0×	1.0×	1.0×	1.01×	1.04×	0.99×	0.97×
wine-recog	1.0×	1.0×	1.0×	0.98×	0.82×	0.98×	0.8×
xd6	1.0×	1.0×	1.0×	1.08×	0.96×	1.05×	0.93×
Geomean	**1.00×**	**1.00×**	**1.00×**	**0.98×**	**0.93×**	**0.97×**	**0.89×**

Table A.5: Gradient boosted trees: Average size of abductive explanations.

			ADD			DFS		
Dataset	**SMT**	**MaxSAT**	**ADD**	**ES**	**PCES**	**DFS**	**ES**	**PCES**
ann-thyroid	1.52	1.42	1.46	-	1.46	1.55	1.55	1.43
appendicitis	3.61	3.48	3.41	3.41	3.41	3.17	3.17	3.17
divorce	5.41	4.38	3.64	3.64	3.64	3.24	3.24	3.24
ecoli	3.4	3.4	3.44	-	3.44	3.48	3.48	3.49
glass2	4.66	4.64	4.42	4.42	4.42	4.18	4.18	4.18
promoters	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
shuttle	3.7	3.28	3.22	-	3.22	3.23	3.23	3.24
threeOf9	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
zoo	4.14	3.74	4.47	-	4.47	4.19	4.19	4.14

Table A.6: Gradient boosted trees: Ratio of each method's abductive explanation size to the *SMT* baseline.

		ADD			DFS		
Dataset	**MaxSAT**	**ADD**	**ES**	**PCES**	**DFS**	**ES**	**PCES**
ann-thyroid	0.93	0.96	-	0.96	1.02	1.02	0.94
appendicitis	0.96	0.94	0.94	0.94	0.88	0.88	0.88
divorce	0.81	0.67	0.67	0.67	0.6	0.6	0.6
ecoli	1.0	1.01	-	1.01	1.02	1.02	1.03
glass2	1.0	0.95	0.95	0.95	0.9	0.9	0.9
promoters	1.0	1.0	1.0	1.0	1.0	1.0	1.0
shuttle	0.89	0.87	-	0.87	0.87	0.87	0.88
threeOf9	1.0	1.0	1.0	1.0	1.0	1.0	1.0
zoo	0.9	1.08	-	1.08	1.01	1.01	1.0
Geomean	**0.94×**	**0.94×**	**0.90×**	**0.94×**	**0.91×**	**0.91×**	**0.90×**

Table A.7: Gradient boosted trees: Average number of features on path.

	ADD			DFS		
Dataset	**ADD**	**ES**	**PCES**	**DFS**	**ES**	**PCES**
ann-thyroid	2.65	-	2.65	6.06	6.06	2.43
appendicitis	5.43	5.43	5.43	4.75	4.75	4.75
divorce	4.91	4.91	4.91	4.9	4.9	4.9
ecoli	4.78	-	4.78	5.1	5.1	5.12
glass2	7.15	7.15	7.15	6.44	6.44	6.44
promoters	1.0	1.0	1.0	1.0	1.0	1.0
shuttle	4.38	-	4.34	4.32	4.32	4.38
threeOf9	1.0	1.0	1.0	1.0	1.0	1.0
wine-recog	6.78	-	6.79	6.74	6.74	6.77
zoo	7.0	-	7.0	7.54	7.64	7.64

Table A.8: Gradient boosted trees: Ratio of each method's average path length (number of features) to the ADD baseline.

	ADD		DFS		
Dataset	**ES**	**PCES**	**DFS**	**ES**	**PCES**
ann-thyroid	-	1.0×	2.28×	2.28×	0.91×
appendicitis	1.0×	1.0×	0.88×	0.88×	0.88×
divorce	1.0×	1.0×	1.0×	1.0×	1.0×
ecoli	-	1.0×	1.07×	1.07×	1.07×
glass2	1.0×	1.0×	0.9×	0.9×	0.9×
promoters	1.0×	1.0×	1.0×	1.0×	1.0×
shuttle	-	0.99×	0.99×	0.99×	1.0×
threeOf9	1.0×	1.0×	1.0×	1.0×	1.0×
wine-recog	-	1.0×	0.99×	0.99×	1.0×
zoo	-	1.0×	1.08×	1.09×	1.09×
Geomean	**1.00×**	**1.00×**	**1.07×**	**1.07×**	**0.98×**

Appendix B

Class Characterization Sizes

B.1 Random Forests

Explanation of Tables B.1–B.8. These tables present the class characterization sizes for the random forests evaluated in Sections 3.3 and 4.2 under the proposed optimizations. The second column shows the total size of the final ADD/DAG, while the subsequent columns list each class's characterization size, with the percentage in parentheses indicating its fraction of the final model size. For example, "Class 0 = 139288 (65.47%)" indicates that the Class 0 characterization consists of 139288 nodes, which represents 65.47% of the model's total size.

Table B.1: Class characterization sizes for random forests for the *ann-thyroid* dataset.

Config.	DAG	Class 0	Class 1	Class 2
ADD				
AbsES	212766	139288 (65.47%)	92185 (43.33%)	207314 (97.44%)
LUT	114228	67324 (58.94%)	59582 (52.16%)	107698 (94.28%)
GEN	210832	137551 (65.24%)	91906 (43.59%)	205379 (97.41%)
ADD	210832	137551 (65.24%)	91906 (43.59%)	205379 (97.41%)
ML	181102	105575 (58.30%)	87575 (48.36%)	174139 (96.16%)
SPC	210832	137551 (65.24%)	91906 (43.59%)	205379 (97.41%)
DFS				
ES	399123	237095 (59.40%)	192421 (48.21%)	385129 (96.49%)
ORD	752390	464027 (61.67%)	336571 (44.73%)	725922 (96.48%)
ES+S	168791	111174 (65.86%)	72860 (43.17%)	163639 (96.95%)
ORD+S	240857	173845 (72.18%)	84119 (34.92%)	229866 (95.44%)

Table B.2: Class characterization sizes for random forests for the *ecoli* dataset.

Config.	DAG	Class 0	Class 1	Class 2	Class 3	Class 4
ADD						
AbsES	1134199	473228 (41.72%)	794761 (70.07%)	342562 (30.20%)	193953 (17.10%)	496353 (43.76%)
LUT	1026	429 (41.81%)	932 (90.84%)	678 (66.08%)	27 (2.63%)	197 (19.20%)
GEN	1112004	467556 (42.05%)	777373 (69.91%)	331511 (29.81%)	190268 (17.11%)	489221 (43.99%)
ADD	1112296	467918 (42.07%)	777465 (69.90%)	331385 (29.79%)	190518 (17.13%)	489206 (43.98%)
ML	465828	366708 (78.72%)	289234 (62.09%)	126001 (27.05%)	26939 (5.78%)	131455 (28.22%)
SPC	1109022	465834 (42.00%)	775668 (69.94%)	331115 (29.86%)	190218 (17.15%)	487529 (43.96%)
DFS						
ES	1096500	444462 (40.53%)	798158 (72.79%)	335058 (30.56%)	181391 (16.54%)	466361 (42.53%)
ORD	1181757	501579 (42.44%)	829175 (70.16%)	357477 (30.25%)	203597 (17.23%)	509703 (43.13%)
ES+S	848456	344197 (40.57%)	610092 (71.91%)	254929 (30.05%)	147355 (17.37%)	367112 (43.27%)
ORD+S	853732	351680 (41.19%)	588813 (68.97%)	253302 (29.67%)	156552 (18.34%)	386272 (45.25%)

Table B.3: Class characterization sizes for random forests for the *iris* dataset.

Config.	**DAG**	**Class 0**	**Class 1**	**Class 2**
ADD				
AbsES	1805	534 (29.58%)	1720 (95.29%)	1362 (75.46%)
LUT	51	10 (19.61%)	50 (98.04%)	49 (96.08%)
GEN	1802	523 (29.02%)	1718 (95.34%)	1361 (75.53%)
ADD	1802	523 (29.02%)	1718 (95.34%)	1361 (75.53%)
ML	1384	272 (19.65%)	1307 (94.44%)	978 (70.66%)
SPC	1802	523 (29.02%)	1718 (95.34%)	1361 (75.53%)
DFS				
ES	1823	493 (27.04%)	1746 (95.78%)	1407 (77.18%)
ORD	1765	498 (28.22%)	1706 (96.66%)	1321 (74.84%)
ES+S	1734	486 (28.03%)	1667 (96.14%)	1319 (76.07%)
ORD+S	1436	417 (29.04%)	1380 (96.10%)	1054 (73.40%)

Table B.4: Class characterization sizes for random forests for the *new-thyroid* dataset.

Config.	**DAG**	**Class 0**	**Class 1**	**Class 2**
ADD				
AbsES	98191	96797 (98.58%)	24921 (25.38%)	75374 (76.76%)
LUT	390	388 (99.49%)	56 (14.36%)	339 (86.92%)
GEN	94752	93356 (98.53%)	24612 (25.98%)	72223 (76.22%)
ADD	94752	93356 (98.53%)	24612 (25.98%)	72223 (76.22%)
ML	36687	26677 (72.72%)	6354 (17.32%)	28173 (76.79%)
SPC	94747	93351 (98.53%)	24612 (25.98%)	72218 (76.22%)
DFS				
ES	81797	80718 (98.68%)	20590 (25.17%)	62773 (76.74%)
ORD	86085	84954 (98.69%)	21699 (25.21%)	65885 (76.53%)
ES+S	66287	65320 (98.54%)	17869 (26.96%)	49813 (75.15%)
ORD+S	63038	62050 (98.43%)	16961 (26.91%)	47375 (75.15%)

Table B.5: Class characterization sizes for random forests for the *segmentation* dataset.

Config.	**DAG**	**Class 0**	**Class 1**	**Class 2**	**Class 3**	**Class 4**	**Class 5**	**Class 6**
ADD								
AbsES	9177945	3210555 (34.98%)	2401751 (26.17%)	3500737 (38.14%)	6215994 (67.73%)	3024966 (32.96%)	2885906 (31.44%)	997034 (10.86%)
LUT	9161	5939 (64.83%)	5 (0.05%)	7279 (79.46%)	2087 (22.78%)	7525 (82.14%)	4 (0.04%)	6 (0.07%)
GEN	9174027	3210472 (35.00%)	2400791 (26.17%)	3497814 (38.13%)	6211830 (67.71%)	3023990 (32.96%)	2884890 (31.45%)	996339 (10.86%)
ADD	9174027	3210472 (35.00%)	2400791 (26.17%)	3497814 (38.13%)	6211830 (67.71%)	3023990 (32.96%)	2884890 (31.45%)	996339 (10.86%)
ML	2736407	319999 (11.69%)	377556 (13.80%)	995540 (36.38%)	1899603 (69.42%)	824561 (30.13%)	708676 (25.90%)	232789 (8.51%)
SPC	9216490	3232465 (35.07%)	2419053 (26.25%)	3520514 (38.20%)	6251387 (67.83%)	3068395 (33.29%)	2891019 (31.37%)	1008230 (10.94%)
DFS								
ES	9862431	2991678 (30.33%)	1860568 (18.87%)	4090934 (41.48%)	6504728 (65.95%)	4141799 (42.00%)	2935420 (29.76%)	1138192 (11.54%)
ORD	22309988	6743286 (30.23%)	3761766 (16.86%)	8584273 (38.48%)	13602388 (60.97%)	9098427 (40.78%)	5995684 (26.87%)	2603727 (11.67%)
ES+S	8341151	2540151 (30.45%)	1617964 (19.40%)	3346095 (40.12%)	5472974 (65.61%)	3507507 (42.05%)	2444290 (29.30%)	968464 (11.61%)
ORD+S	19300003	5831829 (30.22%)	3421259 (17.73%)	7333447 (38.00%)	11811932 (61.20%)	7916756 (41.02%)	5109501 (26.47%)	2133934 (11.06%)

B.2 Gradient Boosted Trees

As in the previous section, these tables show class characterization sizes for gradient boosted trees under the proposed optimizations. The second column provides the total size of the final ADD/DAG, and the subsequent columns give each class characterization size, along with its percentage of the overall model. For instance, “Class 0 = 139288 (65.47%)” indicates that the Class 0 diagram consists of 139288 nodes, which represents 65.47% of the model’s total size.

Table B.6: Class characterization sizes for random forests for the *shuttle* dataset.

Config.	DAG	Class 0	Class 1	Class 2	Class 3	Class 4	Class 5
ADD							
AbsES	484730	287270 (59.26%)	206 (0.04%)	192695 (39.75%)	348562 (71.91%)	169758 (35.02%)	
LUT	13790	2735 (19.83%)	46 (0.33%)	5565 (40.36%)	6581 (47.72%)	969 (7.03%)	13226 (95.91%)
GEN	478586	282374 (59.00%)	207 (0.04%)	192231 (40.17%)	346189 (72.34%)	165154 (34.51%)	
ADD	478586	282374 (59.00%)	207 (0.04%)	192231 (40.17%)	346189 (72.34%)	165154 (34.51%)	
ML	69863	48760 (69.79%)		24779 (35.47%)	46238 (66.18%)	20546 (29.41%)	
SPC	477618	282042 (59.05%)	206 (0.04%)	191630 (40.12%)	345802 (72.40%)	164474 (34.44%)	
DFS							
ES	443504	269112 (60.68%)	247 (0.06%)	178149 (40.17%)	315069 (71.04%)	149636 (33.74%)	
ORD	580946	362438 (62.39%)	242 (0.04%)	238720 (41.09%)	417086 (71.79%)	184515 (31.76%)	
ES+S	367195	227275 (61.89%)	229 (0.06%)	147709 (40.23%)	261686 (71.27%)	119576 (32.56%)	
ORD+S	418842	254420 (60.74%)	233 (0.06%)	174352 (41.63%)	300631 (71.78%)	136859 (32.68%)	

Table B.7: Class characterization sizes for random forests for the *waveform-21* dataset.

Config.	DAG	Class 0	Class 1	Class 2
ADD				
AbsES	3464949	2913108 (84.07%)	2540810 (73.33%)	2111047 (60.93%)
LUT	2766328	2267391 (81.96%)	1971188 (71.26%)	1826336 (66.02%)
GEN	3382403	2849085 (84.23%)	2478054 (73.26%)	2048336 (60.56%)
ADD	3382382	2849064 (84.23%)	2478057 (73.26%)	2048310 (60.56%)
ML	3157854	2434283 (77.09%)	2112672 (66.90%)	2125694 (67.31%)
SPC	3382041	2849444 (84.25%)	2478921 (73.30%)	2048493 (60.57%)
DFS				
ES	5348612	4004112 (74.86%)	3904922 (73.01%)	3717370 (69.50%)
ORD	18727728	13851842 (73.96%)	12814761 (68.43%)	12945995 (69.13%)
ES+S	2769993	2101337 (75.86%)	2024991 (73.10%)	1947810 (70.32%)
ORD+S	7122294	5224057 (73.35%)	4958112 (69.61%)	4869780 (68.37%)

Table B.8: Class characterization sizes for random forests for the *wine-recog* dataset.

Config.	DAG	Class 0	Class 1	Class 2
ADD				
AbsES	9148528	4546431 (49.70%)	8633723 (94.37%)	6208418 (67.86%)
LUT	502	360 (71.71%)	444 (88.45%)	232 (46.22%)
GEN	9124128	4534845 (49.70%)	8608889 (94.35%)	6194715 (67.89%)
ADD	9124141	4534845 (49.70%)	8608902 (94.35%)	6194728 (67.89%)
ML	1179552	783326 (66.41%)	989703 (83.90%)	666383 (56.49%)
SPC	9124186	4534841 (49.70%)	8608949 (94.35%)	6194788 (67.89%)
DFS				
ES	7801905	4011244 (51.41%)	7387077 (94.68%)	5082841 (65.15%)
ORD	15506552	7595292 (48.98%)	14510164 (93.57%)	9859062 (63.58%)
ES+S	6293801	3159437 (50.20%)	5938057 (94.35%)	4244379 (67.44%)
ORD+S	9914530	4416620 (44.55%)	9226630 (93.06%)	6807420 (68.66%)

Table B.9: Class characterization sizes for gradient boosted trees for the *ann-thyroid* dataset.

Config.	DAG	Class 0	Class 1	Class 2
ADD				
ADD	162056	149857 (92.47%)	12205 (7.53%)	161217 (99.48%)
PCES	166284	149857 (90.12%)	16433 (9.88%)	165140 (99.31%)
DFS				
DFS	179260	160868 (89.74%)	18391 (10.26%)	178393 (99.52%)
ES	144866	126474 (87.30%)	18391 (12.70%)	144185 (99.53%)
PCES	128834	126474 (98.17%)	2359 (1.83%)	128719 (99.91%)

Table B.10: Class characterization sizes for gradient boosted trees for the *ecoli* dataset.

Config.	DAG	Class 0	Class 1	Class 2	Class 3	Class 4
ADD						
ADD	102640	33229 (32.37%)	64548 (62.89%)	41117 (40.06%)	27731 (27.02%)	52098 (50.76%)
PCES	103462	33725 (32.60%)	64511 (62.35%)	40738 (39.37%)	28065 (27.13%)	53167 (51.39%)
DFS						
DFS	112208	38568 (34.37%)	62029 (55.28%)	46315 (41.28%)	26623 (23.73%)	61852 (55.12%)
ES	104863	31923 (30.44%)	60240 (57.45%)	46894 (44.72%)	26502 (25.27%)	60083 (57.30%)
PCES	90717	28219 (31.11%)	54668 (60.26%)	43838 (48.32%)	21632 (23.85%)	50112 (55.24%)

Table B.11: Class characterization sizes for gradient boosted trees for the *shuttle* dataset.

Config.	DAG	Class 0	Class 1	Class 2	Class 3	Class 4	Class 5	Class 6
ADD								
ADD	60251	31612 (52.47%)	17154 (28.47%)	7746 (12.86%)	31069 (51.57%)	26804 (44.49%)	7373 (12.24%)	1866 (3.10%)
PCES	53921	29563 (54.83%)	14329 (26.57%)	7257 (13.46%)	25663 (47.59%)	25055 (46.47%)	6523 (12.10%)	2012 (3.73%)
DFS								
DFS	73300	31711 (43.26%)	16981 (23.17%)	17359 (23.68%)	28225 (38.51%)	37028 (50.52%)	6854 (9.35%)	4668 (6.37%)
ES	60519	27672 (45.72%)	13351 (22.06%)	14253 (23.55%)	27077 (44.74%)	29765 (49.18%)	6778 (11.20%)	4439 (7.33%)
PCES	47527	25291 (53.21%)	10695 (22.50%)	8286 (17.43%)	21463 (45.16%)	23855 (50.19%)	5856 (12.32%)	3077 (6.47%)

Table B.12: Class characterization sizes for gradient boosted trees for the *wine-recog* dataset.

Config.	DAG	Class 0	Class 1	Class 2
ADD				
ADD	29742	27113 (91.16%)	29292 (98.49%)	3724 (12.52%)
PCES	28860	26265 (91.01%)	28352 (98.24%)	3708 (12.85%)
DFS				
DFS	27722	25173 (90.81%)	27180 (98.04%)	3258 (11.75%)
ES	27830	25282 (90.84%)	27180 (97.66%)	3616 (12.99%)
PCES	27661	25105 (90.76%)	27179 (98.26%)	3744 (13.54%)

Table B.13: Class characterization sizes for gradient boosted trees for the *zoo* dataset.

Config.	DAG	Class 0	Class 1	Class 2	Class 3	Class 4	Class 5	Class 6
ADD								
ADD	290	57 (19.66%)	12 (4.14%)	58 (20.00%)	140 (48.28%)	77 (26.55%)	48 (16.55%)	135 (46.55%)
PCES	290	57 (19.66%)	12 (4.14%)	58 (20.00%)	140 (48.28%)	77 (26.55%)	48 (16.55%)	135 (46.55%)
DFS								
DFS	264	52 (19.70%)	11 (4.17%)	55 (20.83%)	125 (47.35%)	92 (34.85%)	64 (24.24%)	123 (46.59%)
ES	257	52 (20.23%)	12 (4.67%)	55 (21.40%)	120 (46.69%)	105 (40.86%)	77 (29.96%)	118 (45.91%)
PCES	250	52 (20.80%)	19 (7.60%)	47 (18.80%)	112 (44.80%)	110 (44.00%)	81 (32.40%)	110 (44.00%)

Appendix C

Transforming Gradient Boosted Trees into Algebraic Decision Diagrams

Explanation of Tables C.1–C.5. Each table shows intermediate ADD sizes for a given dataset as we construct $\mathcal{A}^{wK} = \mathcal{A}_1^+ \circ_A \cdots \circ_A \mathcal{A}_K^+$. For $i = 1$, the size of ADD $\mathcal{A}_1^+$ is shown. For $i > 1$, each row i corresponds to the result of joining $\mathcal{A}_i^+$ with the diagram from the previous step. For both *ADD* and *PCES*, the **Join** column shows the size immediately after combining partial diagrams, while the **Elim** column shows the size after removing infeasible paths. For *PCES*, the infeasible path elimination is combined with path condition-based early stopping, allowing the size to be reduced even for $i = 1$. Finally, the **%** column is simply $\frac{\text{Elim size}}{\text{Join size}} \cdot 100$ for that step, so it indicates what fraction of the post-Join size remains after elimination in each iteration. Note that in the final join ($i = K$), rather than storing both the weight and the class in each leaf, only the class with the highest weight is kept, because the weight is no longer needed once the final classification outcome has been determined.

Table C.1: Sizes of the intermediate results during the construction of $\mathcal{A}^{wK}$ for *ann-thyroid*.

Size	**ADD**			**PCES**		
	Join	**Elim**	**%**	**Join**	**Elim**	**%**
1	491 783	491 783	100.0	491 783	481 636	97.94
2	30 666 933	735 254	2.40	30 379 029	302 835	1.00
3	974 118	162 056	16.64	932 633	166 284	17.83

Table C.2: Sizes of the intermediate results during the construction of $\mathcal{A}^{wK}$ for *ecoli*.

Size	**ADD**			**PCES**		
	Join	**Elim**	**%**	**Join**	**Elim**	**%**
1	57 065	57 065	100.0	57 065	21 964	38.49
2	75 813 237	365 169	0.48	19 784 889	187 339	0.95
3	111 223 219	409 184	0.37	40 702 909	131 425	0.32
4	27 496 974	485 369	1.77	6 636 720	104 905	1.58
5	12 502 242	102 640	0.82	3 161 712	103 462	3.27

Table C.3: Sizes of the intermediate results during the construction of $\mathcal{A}^{wK}$ for *shuttle*.

Size	**ADD**			**PCES**		
	Join	**Elim**	**%**	**Join**	**Elim**	**%**
1	443 312	443 312	100.0	443 312	155 499	35.08
2	1 651 135	457 026	27.68	979 089	171 757	17.54
3	879 140	448 405	51.0	588 715	162 354	27.58
4	8 333 636	618 133	7.42	3 572 731	252 079	7.06
5	14 696 053	806 804	5.49	5 267 784	139 683	2.65
6	3 117 642	846 511	27.15	387 524	64 011	16.52
7	110 955	60 251	54.3	53 811	53 921	100.2

Table C.4: Sizes of the intermediate results during the construction of $\mathcal{A}^{wK}$ for *wine-recog*.

Size	**ADD**			**PCES**		
	Join	**Elim**	**%**	**Join**	**Elim**	**%**
1	28 444	28 444	100.0	28 444	22 690	79.77
2	11 456 364	330 506	2.88	10 048 932	82 659	0.82
3	51 395	29 742	57.87	34 788	28 860	82.96

Table C.5: Sizes of the intermediate results during the construction of $\mathcal{A}^{wK}$ for *zoo*.

Size	**ADD**			**PCES**		
	Join	**Elim**	**%**	**Join**	**Elim**	**%**
1	79	79	100.0	79	79	100.0
2	120	120	100.0	120	120	100.0
3	163	147	90.18	163	147	90.18
4	165	165	100.0	165	165	100.0
5	338	306	90.53	338	306	90.53
6	633	633	100.0	633	258	40.76
7	290	290	100.0	290	290	100.0