Karlsruher Institut für Technologie

Bachelor thesis

Local Search with Knowledge Compilation Powered Machine Learning

Kornelius Benjamin Rohrschneider

Date: 15. April 2025

Supervisors: Prof. Dr. Peter Sanders
Dr. rer. nat. Markus Iser

Institute of Theoretical Informatics, Algorithm Engineering
Department of Informatics
Karlsruhe Institute of Technology

Abstract

The SAT problem is omnipresent in the field of computer science. Due to decades of research, SAT solvers have become a core technology many applications are built upon. This thesis explores a novel approach with the potential to fundamentally improve SAT solvers. We propose a new workflow that utilizes knowledge compilation to efficiently extract relevant data and learn hidden patterns from real-world instances. Furthermore, we introduce specialized algorithms designed to reduce the barriers to research, making this approach more accessible and enable future advancements in SAT solving.

Acknowledgments

Thanks to

- Dr. Markus Iser for his guidance as my supervisor and giving me all the support I could have asked for.
- Professor Sanders for the opportunity to work on such an interesting research topic.
- My sister for lectoring my thesis and giving me additional advice.
- My friends and family, who unconditionally supported me.
- The authors and contributors of all the software I relied on, as project dependencies and to perform evaluations.
- Numerous contributors on forums such as Stack Overflow, that provided helpful advice and explanations on how to accomplish certain tasks.

In this study, ChatGPT was employed for research purposes and to assist with debugging and the composition of evaluation scripts. The implementation of the presented components and algorithms themselves was entirely handwritten. The LaTeX code was also handwritten, using only occasional phrasing support from search engines, forums and AI.

I hereby declare that I have written this thesis independently and that I have not used any sources and aids other than those specified, that I have marked the passages taken verbatim or with regard to content as such and that I have observed the statutes of the Karlsruhe Institute of Technology for safeguarding good scientific practice in the currently valid version.

Karlsruhe, 15th of April, 2025

Contents

1 Introduction

1.1 Motivation

Many difficult problems in a wide range of applications are solved by transforming them into a SAT problem. Therefore, solving a SAT problem as fast as possible is very important as it allows to solve many problems in different fields more efficiently.

The Boolean formulas that appear in real-world applications usually have a certain structure and contain specific patterns, which allows modern CDCL SAT solvers to solve most of them very efficiently, despite SAT being NP-complete.

Despite that, local search solvers still mostly use manually configured heuristics that have proven to work well. However, these manual heuristics are unable to find complex patterns that are hidden in the data and cannot easily be manually detected. As using such patterns drives the efficiency of modern SAT solvers, this leaves room for massive improvements to local search solvers.

Therefore, machine learning models that can find and take advantage of such hidden patterns have the potential to create better heuristics and improve local search solvers to faster converge to the solution.

As formula instances from different application areas can have a different structure with different patterns, the quality of human-made heuristics also depends on the application area and instance. A machine learning model can account for this as it learns the different types of patterns and how they affect the satisfying assignments, automating a (previously manual) heuristic selection.

Despite CDCL solvers being mostly used nowadays, local search solvers are still highly relevant, as the biggest current performance improvements result from the development of hybrid solvers that integrate local search solvers as subroutines into CDCL solvers, like Kissat [1]. Additionally, local search solvers can, on their own, be faster than CDCL solvers for "easy" instances which are known to have many satisfying assignments. Improving local search solving can therefore have a significant impact on state of the art SAT solvers and their performance.

1.2 Contribution

This thesis represents an initial research into this novel approach to integrate a knowledge powered machine learning model into a local search solver in order to improve its performance.

It introduces fundamental research on the workflow to generate a data set, train a machine learning model with it and integrate it into a local search solver. To do so, it evaluates the advantages and disadvantages of various approaches and algorithms. It also provides several components that integrate with each other and are optimized to seamlessly execute this entire workflow.

By comparing different algorithms and approaches, we identify those that are more and less effective. Future researchers can benefit from these insights, as they won't have to repeat similar analyses, but can instead adopt the best found solution or (in some cases) explore entirely different approaches.

Additionally, future research can be based on this initial groundwork by adapting the provided framework to their needs, without having to implement everything from scratch themselves.

This thesis not only benefits future research, but also presents implementation-level breakthroughs: One proposed algorithm achieves both a lower theoretical complexity and a better real-world performance than existing alternatives.

1.3 Structure of Thesis

Chapter 2 explains the background information required to understand the research presented in this thesis.

Chapter 3 then gives an overview on the entire workflow required for this novel approach as well as the implementation provided to execute it. This implementation is split up into several distinct components: A model counter and a formula analyzer (both programmed in Rust) that are responsible for high-performance calculations, a framework (programmed in Python) that connects the other components and provides an interface to a machine learning model, and the local search solver itself, that uses the framework to execute the meachine learning model.

As multiple of these components should be evaluated in order to find the most effective solution, this thesis also contains several evaluations. Chapter 4 presents the different evaluation and their results.

After explaining the approach, showcasing the implementation and evaluating the analyses results, the thesis discusses the implications of the research and its results as well as the future works that can build on it and provide further research into this novel area in Chapter 5.

2 Preliminaries

2.1 Fundamentals

2.1.1 SAT Solving

The Boolean satisfiability problem (SAT) refers to deciding whether a Boolean function is satisfiable (i.e. that at least one assignment of variables exists under which the Boolean function is true).

SAT is NP-complete. This means that all problems in NP can be reduced within polynomial time to a SAT problem. Many real-world algorithms utilize this and solve a problem by transforming them into a SAT problem. This can often be more performant than implementing a custom solution as SAT solvers contain various optimizations and have been researched for a long time. Therefore, solving a SAT problem as fast as possible and improving SAT solvers is very important as it allows to solve many problems in different fields more efficiently.

SAT solvers have been developed since 1960 [2]. While no polynomial algorithm to solve all SAT instances can exist (if $P \neq NP$), modern solvers can solve most real-world instances efficiently by taking advantage of the fact that such instances have a certain structure and contain specific patterns, which allows to solve them much faster than arbitrary random instances.

SAT solvers usually take a formula as CNF (conjunctive normal form) or initially convert it into that form [3].

A generalization of SAT is the #SAT problem, which does not just inquire whether a Boolean function is satisfiable, but specifically how many different assignments exist which satisfy the Boolean function. In this context, the term "model" (of a Boolean function) refers to such an assignment of variables which satisfy the function. Therefore, solving this problem for an instance is also called model counting. #SAT can further be generalized to take a partial assignment and determine how many full assignments exist that do not contradict the given partial assignment and still satisfy the Boolean function. Solving this problem can also be referred to as model counting under given constraints [4].

Local search solvers start with an initial assignment of variables. They then iteratively flip the assignments of single variables based on internal heuristics in order to improve the current position in the search space of possible assignments (towards a model). In order to avoid a circle of flips, it keeps a so-called "history" of the last k variables, which it does not

flip. The variables not included in the history in a specific search step can also be called the "free" variables.

While CDCL solvers are also able to prove that a formula does not have any satisfying assignments, local search solvers can only find satisfying assignments. However, they can be faster than CDCL solvers for "easy" instances which are known to have many satisfying arguments.

TaSSAT is a modern local search solver that has been developed in 2024, and built on top of YalSAT [5].

2.1.2 Knowledge Compilation

The term knowledge compilation describes the process of pre-"compiling" data into a different data form, which can then be used by an algorithm to process it more efficiently. This means that the expensive extraction of the important data parts only has to be done once before the data can be used numerous times in different cheap queries. This way, the actual algorithm working on the transformed data can be much more efficient and simple, reducing the overall time to process the data [6].

In this context, knowledge compilation refers to transforming the original CNF formula representation of a Boolean function into a different representation, called d-DNNF, which is optimized to count its models (optionally under a given partial assignment) very efficiently. The extraction of the important data parts that are required to count the models is therefore only done once, instead of for each repeated query. While model counting is an NP-hard problem on the original CNF formulas, this knowledge compilation enables us to count the models in linear time on its d-DNNF representation. This compilation has initially been done by the c2d compiler [7], with a recent improvement being the d4 compiler [8][9]. Those compilers use different output formats for the resulting d-DNNF formulas, which we call C2D-NNF respectively D4-NNF.

A d-DNNF formula is a deterministic (d) decomposable (D) formula that is in the negation normal form (NNF). If a d-DNNF formula is also smooth (s), it is also called an sd-DNNF formula. The following paragraphs explain the definition for each of these terms:

Negation normal form A formula is in the negation normal form (NNF) if and only if it only consists of the logical operators $\wedge$ (And), $\vee$ (Or) and $\neg$ (Not), the Boolean values $\top$ (True), $\bot$ (False) and literals, and if negations ($\neg$) only appear directly in front of Boolean values and literals [10, page 5].

Decomposability "A formula is decomposable (D) if and only if for each conjunction $C = C_1 \wedge C_2 \wedge \cdots \wedge C_n$, the sets of variables of each conjunct C_i are disjunct." [10, page 5]

Decomposability implies that an assignment that satisfies a specific conjunct does not influence any of the other conjuncts. Therefore, all assignments that satisfy one of the conjuncts *of (only) the variables of that conjunct* can be combined, and the model count of a conjunction is the product of the model counts of the conjuncts *of (only) the variables of that conjunct* ($C_{count} = C_{1,\text{ count}} * C_{2,\text{ count}} * \cdots * C_{m,\text{ count}}$).

Determinism "A formula is deterministic (d) if and only if for each disjunction $D = D_1 \vee D_2 \vee \cdots \vee D_m$, the disjuncts share no common satisfying arguments." [10, page 6]

Determinism implies that there can be no assignment that satisfies more than one disjunct. Therefore, it causes the model count of a disjunction to be the sum of the model counts of the disjuncts *of all variables of the disjunction* ($D_{count} = D_{1,\text{ count}} + D_{2,\text{ count}} + \cdots + D_{m,\text{ count}}$).

Smoothness "A formula is smooth (s) if and only if for each disjunction $D = D_1 \vee D_2 \vee \cdots \vee D_m$, the disjuncts have the same set of variables." [10, page 6]

Our approach to use d-DNNF formulas to efficiently count models (under given partial assignments) has previously been researched by Darwiche, Adnan and others [10]. They have implemented such an algorithm in a tool called ddnnife, which can be used for efficient model counting.

2.1.3 Supervised Machine Learning

Supervised machine learning is the task to learn how to map known input values to known output values and is able to generalize from that known data to predict other output values based on their input values. Instead of explicitly programming a mapping algorithm, a machine learning model learns the relations between input and output autonomously by receiving the training data. That way, it is able to grasp deeper correlations and interconnections in the training data that are possibly too hard to be noticed, understood and manually programmed by a human.

A machine learning model is called a *classification* model if each output values is one of several distinct classes, and a *regression* model if each output value is a continous number.

When training machine learning models, we separate between parameters and hyperparameters. The hyperparameters define the structure of the model that learns such a correlation (e.g. how detailed the correlations it learns can be, for how long it will be trained or how much it is affected by outliers). The parameters are the actual values the model that is defined by given hyperparameters learns autonomously.

A random forest is a specific machine learning model. It works by creating and training many decision trees independently with random subsets of the training data, and uses their average results as prediction (in the regression case). Random forests can be trained and evaluated very quickly as these operations can be parallelized over the trees.

A deep neural network is a different machine learning model. It consists of multiple layers of interconnected neurons. Each neuron receives its input from neurons in the previous layer and applies a linear combination to them, using learned weights and a bias (the parameters of the model). The result is then transformed by non-linear activation function and passed to next layer of neurons. Neural networks are very flexible and can learn much more complex patterns than random forests. However, they take much longer to train and require a more careful configuration and tuning of hyperparameters in order to produce good results.

As we rely on frameworks to create and train such machine learning models, we do not need to know or implement the details on how exactly the internal structure of a model resp. the parameters are trained.

In order to train a supervised machine learning model, one needs to have a precalculated data set, which contains the known input and output values. This data set is usually split into a training set, a validation set and a testing set:

The **training set** is the set which is primarily used to train a model. The model learns the patterns and correlations in its data.
The **validation set** is then used to test the generalization of the trained model. While most trained models can easily predict the output values of their training set, using unseen data allows to judge how well the model is able to generalize from the known training data to predict other output values based on their input values. As the training process includes finding the best model type and hyperparameters, the validation set is used to compare the generalization performance of different models in order to pick the best one.
The **testing set** is then finally used to calculate the true generalization performance on entirely unseen data. This is necessary as the model selection is influenced by the validation set, which might result with a model being selected that performs particularly good on this specific validation set.

In order for the trained model to perform well in the application it is used for, it is important that the distribution of the real-world input data (meaning its type, properties and the patterns in it) is roughly equal to the distributions of the input data in the training, testing and validation set.

2.2 Related Work

While the specific idea to use a knowledge compilation powered machine learning model in order to improve the performance of a local search solver is, to our knowledge, a novel approach, there has been much prior research on generally using machine learning in order to speed up SAT solving algorithms.

Most research on this topic has been spent to improve CDCL solvers and their heuristics, specifically branching decisions:

Jia Hui Liang has researched the potential of using machine learning to create branching heuristics and restart policies that can be used in CDCL solvers. Using this approach, they could create heuristics that (at least) matched the current state-of-the-art ones [11]. Modern CDCL solvers use a very high volume of branching decisions. Despite their machine learning models being called for each branching decision, they have proven to be successful in competitions. This is due to them being low-capacity and queried only with local information (and not with information about the entire formula), making their inference extremely efficient [12].

In contrast to that, Selsam et al have developed NeuroCore, querying a neural network on the entire problem. Due to the more expensive inference, this is not done for each branching decision, but rather periodically. It is therefore used to predict globally-informed data that can improve the existing heuristics [13].

Jesse Han builds up on that approach to periodically query a neural network on the entire problem to improve the existing branching heuristic. By predicting different values that can help the heuristics more effectively, they could improve the performance of a state-of-the-art solver [12].

Haoze Wu has used a more similar approach to ours, using a machine learning model to generate an initial assignment of variables that can then be used in a CDCL solver. Instead of using knowledge compilation to train the machine learning model to predict model counts, they have trained a regression model to predict a satisfiability confidence score for given assignments. They used this model to predict an initial assignment by repeatedly calculating the SAT confidence scores of different random assignments and then picking the mean score of all assignments that contain a literal positively resp. negatively. With this model, they could reduce the runtime of a SAT solver by 23%, however, adding an even larger amount of preprocessing time [14].

But there has also been research on local search solvers: Zheng et al present a machine learning - powered approach to improve the local search solver heuristics determining which variable to flip while solving a Weighted Partial MaxSAT problem (WPMS). Partial MaxSAT is a SAT generalization which aims to determine the maximum amount of "soft" clauses that can be satisfied while satisfying all "hard" clauses. It should be noted that a local search solver cannot be guaranteed to produce the optimal solution of such a problem. While their approach is customized towards this problem and the maximization of the soft clause weights, a key part of it is to also improve the algorithm to find a satisfying solution for all hard clauses more effectively. In order to do that, their algorithm uses reinforcement learning to learn which of the literals in any hard clause *(actions)* should be set to satisfy that hard clause in order to achieve the highest share of newly satisfied hard clauses *(reward)*. While satisfying the hard clauses is usually not the main challenge for WPMS solvers, and other parts of the proposed algorithm might have a bigger impact, it could still significantly improve state-of-the-art local search WPMS solves [15].

3 Approach

3.1 Workflow

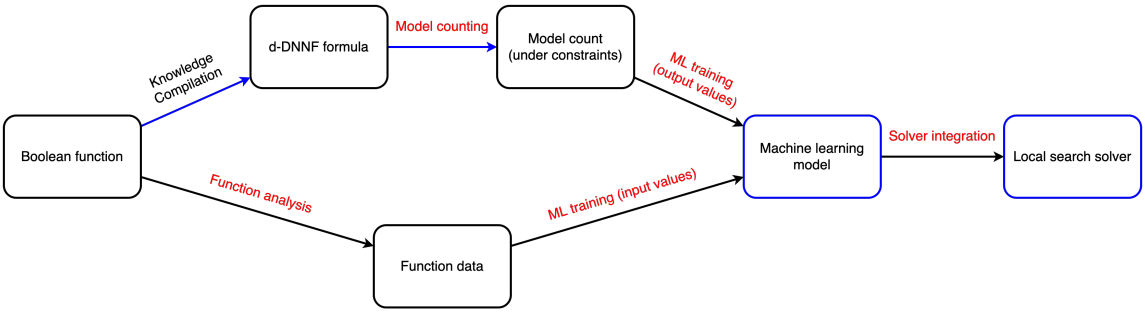

Figure 3.1: The workflow of this novel approach to integrate a knowledge powered machine learning model into a local search solver. The custom implementations are marked in red and the evaluated steps resp. components are marked in blue.

The primary idea of this thesis is to integrate a machine learning model into a local search solver in order to improve its performance.

There are two core ways in which the integration of a machine learning model can do so: Firstly, it should give a good initial assignment of variables. This causes the local search solver to start in a better position in the search space, requiring less flips to find a model. And secondly, it should give a good prediction of which variable assignment to change for each step. This improves the quality of the flips and makes the solver converge more quickly towards a model, again requiring less flips to find a model.

Both of this can be achieved by a machine learning model which learns to accurately predict the amount of models of a formula under specific given constraints. In order to get a good initial assignment, such a model could predict the amount of models for each variable assigned to true respectively false, which would be used to assign the variables accordingly. And in order to get a selected variable to flip, it could predict the amount of models (given the current fixed history) for each free variable assigned to true respectively false. This would then be used to flip a variable for which the opposite assignment is predicted to lead to a significantly larger number of models under the current history.

In order to train a machine learning model accordingly, we first need to calculate a respective data set (which we can split into a training, validation and testing set). However,

calculating the amount of models of a formula under given constraints is, like SAT itself, NP-complete and very expensive. Therefore, we use the concept of knowledge compilation and first "compile" each Boolean function into a different representation which is optimized to counting its models very efficiently. By using this approach, we only have to transform each Boolean function once before we can count its models under many different partial assignments efficiently. Being able to calculate those model counts very quickly is especially important as it also allows for the project to be used for reinforcement learning instead of supervised learning in the future.

Thus, the entire workflow (displayed in Figure 3.1) is the following: We compile the Boolean functions that are present as CNF formulas into d-DNNF formulas. We then generate a data set by analyzing the function (for input values) and counting the models on the compiled formula (for output values). This data set gets split into a training and testing set and is used to train a machine learning model with it. And finally, we integrate this model into an existing local search solver and combine the model predictions with the existing heuristics of the solver.

Due to the unique structure of the workflow required for this approach, this chapter consists of one section for each of the custom components that are used in different parts of the workflow:

- **d-DNNF Model Counter** (Section 3.2)
  A model counter that efficiently counts the models of pre-compiled d-DNNF formulas under given constraints. It is used as a dependency by the SAT-Training framework.

- **CNF Analyzer** (Section 3.3)
  An analyzer that extracts relevant data from a CNF formatted Boolean function. It is used as a dependency by the SAT-Training framework.

- **SAT-Training** (Section 3.4)
  A framework that integrates and combines the other components in order to serve as the unifying user interface to execute any part of this workflow. It can generate different types of data sets (using the CNF Analyzer for input values and the d-DNNF Model Counter for output values), train machine learning models with such a generated data set and perform a hyperparameter search. It also provides a way to execute a trained model to predict output values, which is used to integrate it into local search solvers like TaSSAT.

- **TaSSAT Integration** (Section 3.5)
  A local search solver that integrates a trained machine learning model in order to improve its performance.

3.2 d-DNNF Model Counter

The d-DNNF Model Counter is a library that can parse pre-compiled C2D-NNF or D4-NNF formatted Boolean functions and count their models for a given partial assignment efficiently. It therefore represents an alternative to ddnnife. While it can be used on its own, in this thesis, it is mainly used as a dependency by the SAT-Training framework in order to execute the workflow.

3.2.1 Model Counting without Smoothing

The d-DNNF Model Counter constitutes a significant research result, as it uses a new improved model counting algorithm which has a lower theoretical complexity bound. This comes from the fact that, unlike ddnnife, the d-DNNF Model Counter does not require any smoothing of the given formulas.

Disadvantages of Smoothing

The ddnnife model counting algorithm requires the formula to be smooth, i.e. to be an sd-DNNF, and to contain all variables. This allows for a simple model counting algorithm. As d-DNNF formulas generally cannot be expected to be smooth and to contain all variables, ddnnife contains an integrated smoothing algorithm, which it applies to a formula before invoking the model counting algorithm [10].

However, smoothing a formula has several negative effects. Firstly, it is computationally expensive. While the model counting algorithm itself is in linear time (in regards to the amount of nodes of the formula), the smoothing algorithm takes at least $O(\#\text{vars}*\#\text{nodes})$. While all Boolean functions have to be initially converted into d-DNNF formulas, invoking ddnnife once to just smooth the function is an extra step. More importantly though, it also increases the size of the graph and number of nodes (see Figure 3.2). The number of nodes it adds to the graph can be up to $O(\#\text{vars} * \#\text{nodes})$, causing slower graph traversals [10, page 13].

ddnnife Algorithm

If a d-DNNF formula is smooth, we know that the disjunctions have the same set of variables. This means that the variable set of the disjunction equals the variable set of each disjunct. In that case, we only have to add the model counts of the disjuncts *of (only) the variables of that disjunct* to get the model count of the disjunction. As we only have to consider the variables of the current child node of each disjunction and conjunction to calculate its models, we only have to consider the single variable for each literal, resulting in the trivial model count 1 for each literal.

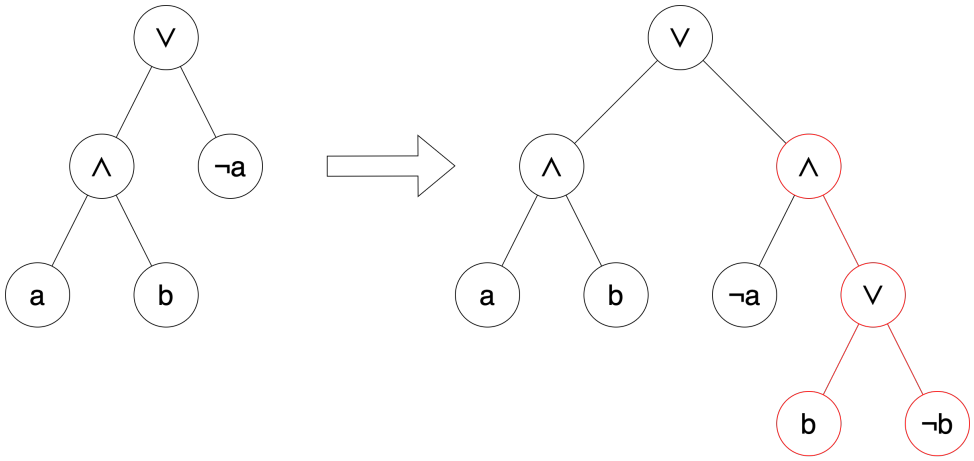

Figure 3.2: A d-DNNF formula before and after smoothing.

Therefore, we can traverse the graph bottom-up, use the model count 1 for each literal and add respectively multiply the child model counts for each disjunction respectively conjunction. Figure 3.3 illustrates this algorithm on the example formula. The final model count of the root node equals the model count of the formula of all variables in it. As the pre-processed formula contains all variables, this equals the total model count. This is the model counting algorithm ddnnf uses internally [10].

d-dNNF Model Counter Algorithm

Unlike the ddnnife model counting algorithm, the algorithm used by the d-DNNF Model Counter does not require the formula to be smooth. Therefore, it does not contain a smoothing algorithm. However, this requires the algorithm to be more complex. As the algorithm operates on the formula in a graph representation, corresponding terminology is used to describe it. This means that a conjunction is described as an "Or node" and a disjunction as an "And node" with the "children" being the conjuncts respectively disjuncts.

The fact that a formula does not have to be smooth means that the child nodes of an Or node don't necessarily contain the same set of variables; some children might have different variables than others. In this case, we cannot simply add the model counts of the children, as the missing variables (compared to all other children) can be assigned both true and false each. To calculate the child's model count *(considering all variables of the parent node)*, we have to multiply it by two for each missing variable to account for the different possible assignments to them (see Figure 3.4).

However, knowing how many variables are missing compared to the other children would require each node to list all variables (which would result in a superlinear time for graph construction: $O(\#\text{vars} * \#\text{nodes})$, the same theoretical constraint ddnnife uses for smoothing). Therefore, we use another approach, which allows us to both create the graph and

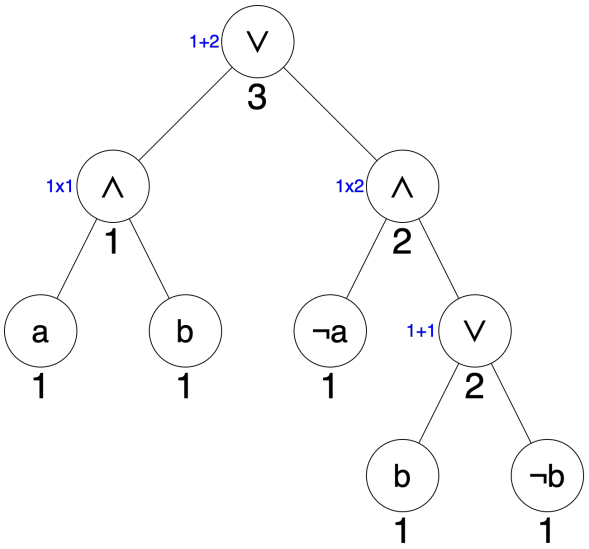

Figure 3.3: The model counting algorithm for sd-DNNF formulas. The model count of each node (of the variables of that node) is displayed under the node and its calculation (if applicable) to the left of it.

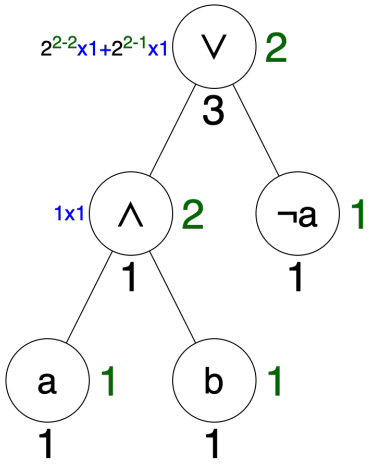

Figure 3.4: Initial idea for a model counting algorithm for d-DNNF formulas. The model count of each node (of the variables of that node) is displayed under the node, its calculation (if applicable) to the left of it and its variable amount (in green) to the right of it.

count the models in linear time (in regards to the amount of nodes, counting 2^x as an $O(1)$ operation):

The basic idea of the approach is that we do not need to know *exactly* how many variables are missing, we just need to know the *relative difference* of missing variables between the children of an Or node. Instead of calculating the true amount of variables, we count the so-called amount of "relevant variables". For any Or node, this number is the maximum amount of any children's "relevant variables". (And like with the true amount of variables, this number is the sum of all children's relevant variables for any And node and 1 for any literal.)

For each Or node, we then calculate the amount of missing variables of each child by using the difference between the children's relevant variables and the parents' relevant variables, and adapt the model count accordingly.

This amount of relevant variables might be lower than the real amount of variables (e.g. if multiple children miss another child's variable). However, in that case, all of the children have the same amount of additional missing variables (called x) that are not accounted for. And if the amount of relevant variables of an Or node is x smaller than the true amount of variables, this will cause its own amount of missing variables (at the next Or node parent) to be x higher than when using the true amount of variables.

Because of the distributive law (see the following equation), the effect of the additional different variables each child is missing is the same as the effect of the increased amount of missing variables of the Or node itself:

$$C_{1,\,\text{count}} * 2^x + C_{2,\,\text{count}} * 2^x + \cdots + C_{m,\,\text{count}} * 2^x = (C_{1,\,\text{count}} + C_{2,\,\text{count}} + \cdots + C_{m,\,\text{count}}) * 2^x$$

Therefore, calculating the model count via the amount of relevant variables will lead to the same (correct) result as calculating it via the amount of total variables. Figure 3.5 illustrates this model counting algorithm that uses the amount of relevant variables using a new formula with nodes that have a different amount of relevant variables than their true amount of variables.

The only additional thing that needs to be considered is that this does not account for the difference between the total and relevant amount of variables of the highest Or node (as it does not have a parent Or node to have missing variables in comparison to). Therefore, after traversing the final graph until the root node, we have to calculate one final global amount of missing variables by comparing the calculated amount of relevant variables of the root node with the known amount of total variables of the formula (and multiplying the total model count by two for each missing variable) (see Figure 3.6).

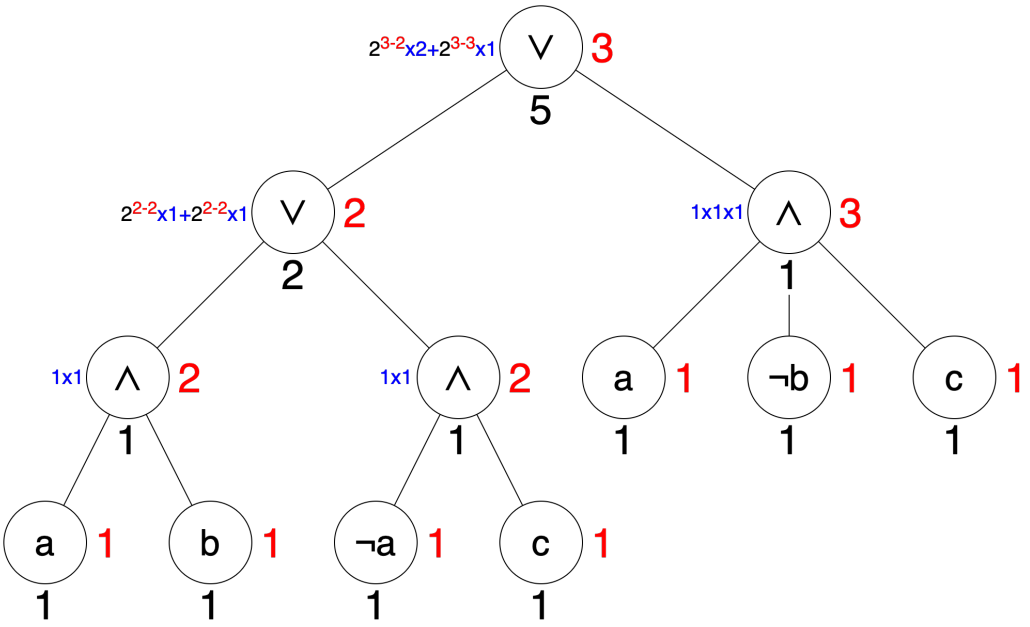

Figure 3.5: The proposed model counting algorithm for d-DNNF formulas. The model count of each node (of the variables of that node) is displayed under the node, its calculation (if applicable) to the left of it and its relevant variable amount (in red) to the right of it.

3.2.2 Optimization Techniques

Core Features

Positive and negative core features are the features that are included respectively excluded in every single model of a Boolean function. Therefore, an optimization to count the models more efficiently can be to directly return zero if any positive core feature is excluded or negative core feature is included in the given assignment.

Like ddnnife, the d-DNNF Model Counter implements this optimization technique.

ddnnife can detect the core features very quickly by determining the literals that only appear positively resp. negatively in the formula during the parsing. This works due to the smoothness of the formula.

As the formulas the d-DNNF Model Counter operates on are not guaranteed to be smooth, detecting the core features requires a more expensive precalculation that traverses the entire formula graph and determines the exact missing variables.

Partial Traversals

Like ddnnife, the d-DNNF Model Counter supports the optimization technique of using partial traversals.

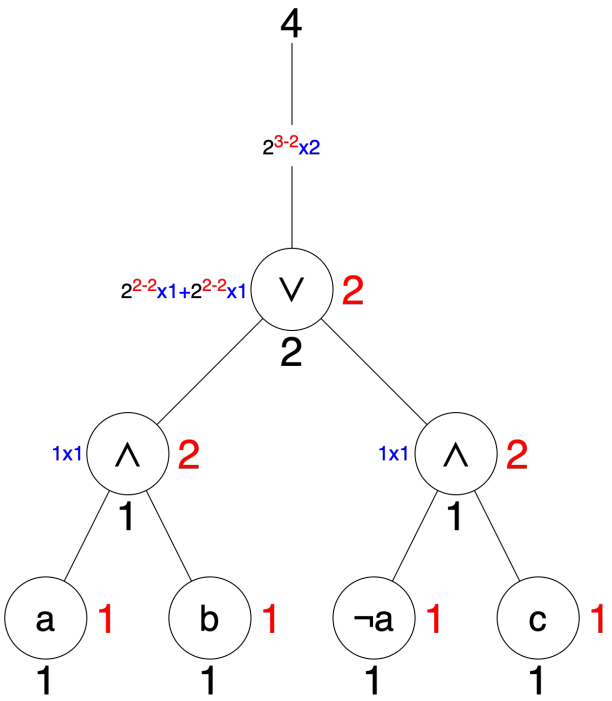

Figure 3.6: The proposed model counting algorithm for d-DNNF formulas on a graph that requires the global missing variables amount. The model count of each node (of the variables of that node) is displayed under the node, its calculation (if applicable) to the left of it and its relevant variable amount (in red) to the right of it.

The core idea of a partial traversal is as follows: If the model count of every node is cached, the model count for a given partial assignment can be calculated more quickly by only calculating the model counts for nodes that are affected by the partial assignment, i.e. nodes that have any assigned variable as its descendant, and using the known model counts otherwise.

Ddnnife implements this by initially traversing the graph to identify which nodes are influenced by given constraints, before then traversing the graph again to actually calculate their model counts for each count request. This is done (only) if 20 or less variables have been assigned in the given partial assignment.

The d-DNNF Model Counter uses the same core idea, but builds up on it to improve its effectiveness: It can optionally perform a one-time precalculation to cache the variables that affect each node. This way, that information can directly be used for many different queries without requiring an additional graph traversal for each. However, the d-DNNF Model Counter also provides the capability to use an explicit initial graph traversal like ddnnife, which can be preferred for some use-cases.

Another key improvement is that the d-DNNF Model Counter does not require the known model count to be the model count without any constraints. The partial traversal optimization works better the fewer additional constraints are given, that affect nodes and require new model count calculations. However, if one wants to count the models for many different partial assignments with a large consistent subset, the core approach (ddnnife uses) would calculate the model counts of the nodes that are affected by any part of the assignment, including the identical subsets, for each request. The d-DNNF Model Counter allows to individually configure which known model counts (calculated under which constraints) should be cached, which can provide a massive performance improvement for such use-cases.

On top of that, the d-DNNF Model Counter also allows to use multiple known model counts. This increases the probability that a big part of the nodes' model counts have already been calculated and cached, and can further improve the performance. Use-cases that have to count models for random combinations of multiple assignments can especially benefit from this.

It should also be noted that this optimization approach generally benefits from the fact that the d-DNNF Model Counters does not require the formulas to be smoothed. As smoothing requires all missing variables to be added, it adds many leaves to the graph and causes much more nodes to have certain variables as their descendants. Therefore, this process also causes those partial traversals having to traverse a bigger part of the graph (which is influenced by the changed constraints).

3.2.3 Implementation Details

The d-DNNF Model Counter leaves the users the choice which exact precalculations and optimizations to use. This allows for users to choose the configuration that is most per-

formant for their use-case, as the cost of a precalculation required for an optimization techinque might not be worth it for a small number of requests.

This requires the user to explicitly request the precalculation (of core features and node variables) and caching of calculated model counts if that is wanted. To allow the user to rely on the requested optimizations being used, the model counter returns an error if one of them cannot be used due to any missing prior calculation.

Structure

The d-DNNF Model Counter is split up into multiple parts. It contains a parser (which parses a C2D-NNF or D4-NNF formatted Boolean function and creates its graph structure), a pre-processor for D4-NNF formatted Boolean functions (which simplifies the graph structure and removes redundant parts like nodes without or with only one child or Boolean values; those redundant nodes exist due to the D4-NNF format), the actual model counter that analyzes the graph structure and counts the models under a given partial assignment and a benchmarking module using Criterion [16] which can accurately measure its performance. It also provides a way to print a parsed formula which can help to understand its structure.

The model counter is programmed in a very maintainable and encapsulated way, which makes it easy to adapt it for future needs.

Bugfixes

In comparison to ddnnife, there are several minor bugs that have been fixed:

- ddnnife only invokes the smoother as pre-processor when the input file has been generated by d4 (the files that d4 and c2d produce have a different format, here called D4-NNF and C2D-NNF). However, the C2D-NNF formatted files c2d produces are not smoothed per default [17]. Therefore, the results ddnnife produces when using such a file are wrong and can even contradict each other.
- While ddnnife parses and counts the models in files that have been generated by D4 correctly, it apparently relies on not only the D4-NNF format but also some patterns that always occur in the files specifically generated by D4. We discovered some custom modified D4-NNF formatted files that which were matched the format description, but could not be parsed by ddnnife.

Both of these bugs have not been present in the d-DNNF Model Counter. However, fixing those bugs should not require big structural changes; therefore ddnnife could be improved to not contain them anymore as well.

Usage as Dynamic C Library

In order to use the d-DNNF Model Counter as a library from other programming languages, it contains a C interface and gets compiled to a dynamic C library. Code in other programming languages can then use an FFI (foreign function interface) to call those C functions. An FFI module handles the conversion of data types and (de-)allocation.

As in real-world applications, model counts of the same formula will be calculated under many different partial assignments, and as precalculations can reduce the cost of repeated model counts, it would be unnecessarily expensive to parse and count the same formula from scratch for each call. Therefore, the model counter keeps the formula struct allocated in between the FFI calls. In order to do this, it provides several FFI functions: One to parse a file and return a handle on the formula struct, one to count the models of a previously parsed function under a given partial assignment and one to deallocate the formula struct.

3.3 CNF Analyzer

The CNF Analyzer is a library that extracts requested relevant data about each variable from a CNF formatted Boolean function and return it as a single structured vector that can help a machine learning model to learn the model counts of that funcition. This data can consist of the number of clauses a variable appears in, the normalized number of clauses a variable appears in, the number of horn clauses a variable appears in, the number of inverted horn clauses a variable appears in and (given a partial assignment) the number of unsatisfied clauses under the partial assignment the variable would satisfy if set (each of those for both positively and negatively). It can also contain an exists bit and (given a partial assignment) an assignment bit if requested. Like the d-DNNF Model Counter, the CNF Analyzer is, in this thesis, mainly used as a dependency by the SAT-Training framework in order to execute the workflow.

3.3.1 Implementation Details

Structure

Similarly to the d-DNNF Model Counter, the CNF Analyzer is also split up into multiple parts, namely a parser (which parses a DICOM formatted Boolean function) and the actual analyzer (which calculates and provides the requested analysis results).

The analyzer is programmed in a maintainable and configurable way so that it can easily be adapted to the needs of other projects, e.g. those that require a different custom combination of analyses results.

Usage as Dynamic C Library

As with the d-DNNF Model Counter, the CNF Analyzer contains a C interface and gets compiled to a dynamic C library in order to be used from other programming languages. An FFI module handles the conversion of data types and (de-)allocation.

The CNF Analyzer also uses the same principle of keeping the internal state allocated in between the FFI calls as the same formula will be analyzed in regard to many different partial assignments in real-world applications. This results in a big speedup as most of the analysis results are independent from the partial assignment and therefore only have to be calculated once. Like the d-DNNF Model Counter, it provides one function to parse a file and return a handle to the analyzer, one to get the requested analysis results and one to deallocate the analyzer.

3.4 SAT-Training

The SAT-Training component is a framework that integrates and combines the other components in order to seamlessly execute the workflow of this approach. It therefore serves as the unifying user interface.

It can generate different types of data sets (using the CNF Analyzer for input values and the d-DNNF Model Counter for output values), train machine learning models with such a generated data set and perform a hyperparameter search. It also provides a way to execute a trained model to predict output values, which can be used to integrate it into local search solvers like TaSSAT.

The SAT-Training framework allows to perform these different steps of the workflow on request as independent tasks, without them all being hardcoded in a specific order. This also provides much flexibility, e.g. to perform several tests with different settings.

3.4.1 Calculating Data Points

The first task is to calculate the data points of previously compiled formulas, that a machine learning model should later be trained on.

There are two possible types of data points that can be calculated. For more information on the structure of the data points and how the data they contain can be used to improve a local search solver, see Section 3.5.1, which explains the data points and how they are integrated into TaSSAT. This task can create data points of both of these types.

It uses the CNF Analyzer as a dependency to calculate input values of the requested data points. The d-DNNF Model Counter is used to calculate the matching output values of the requested data points. In order to do this, the task requires the formulas to be precompiled in the C2D-NNF or D4-NNF format as well as to exist as CNF formulas.

One factor that is important to consider when calculating data points is that most machine learning models (including the ones used in this thesis) expect a fixed number of input and output values. As the amount of variables in a formula is variable and can be arbitrary large or small, and as our data points contain input and output values for each variable, this poses a number of challenges for us. The way we solve this problem is by restricting our model to a maximum number of variables and filling up the input values for each missing variable with zeroes. To help the model understand which variables exist and which don't (after all, some input values might also be zero for existing variables), we introduce an "exists bit" as additional input value for each variable. This bit is set to true (1) for each existing variable, and set to false (0) for each missing variable (just like all other input values). Nevertheless, this approach contains some pitfalls, such as the model being restricted to formulas with their variable amounts within a specific window, or scoring metrics needing to be adapted. For more information on these problems, see sections 4.3.2 and 4.3.4.

As the different families of data points may contain different structures and patterns a machine learning must learn, it is important that the distribution of the different families is the same in the training, testing and validation set as well as in the real data points the trained model is used on. Therefore, the SAT-Training framework ensures this to be the case when splitting the data points into those different sets.

Implementation Details

Currently, the benchmark database [18] is used to provide a high-quality set of real-world formulas. This allows to utilize the additional information it contains about the formulas.

Therefore, this task currently also requires the formulas to be in the benchmark database, as it uses statistics that are stored about a formula in it (which are, among many other values, its amount of clauses and the minimum, average and maximum degree of all clauses in the clause graph) as a part of the data point input values. However, this is implemented in such an encapsulated way that an alternative (calculation-based) provider of these input values can easily be added for real-world usage.

The d-DNNF Model Counter and CNF Analyzer are called via an FFI module with encapsulated classes that handle the conversion of data types and (de-)allocation.

In order to finish the calculations as quickly as possible, this task uses a given amount of processes in parallel.

After the calculation has finished, the data points are saved automatically. In order to try out different sizes of training, validation and testing sets, they are not automatically split (but only split by an additional user request). Instead, they are saved as a so-called "data point collection", which means that they are mapped to their family, so that they can be split with an equal family distribution.

When the user requests the data points to be split, they will initially only be split up into a testing set and a data set (with both training and validation data). The data set will later be

split up when requesting the actual machine learning model training. This allows to easily try out different validation split strategies, e.g. cross-validation instead of a fixed validation set.

Another important implementation detail is that the family information (whose size in the data points is the amount of families in the total data set) is only added to the data points when the user requests to perform the initial test split. This way, we can calculate data points over multiple requests, even if they are of new families, before splitting the data set when the family information is added to all of them.

For clarity, the data sets (and information about the models trained on them) are saved in a folder hierarchy, with a folder for each test split and a sub-folder in it for each trained model. This is not only more manageable but also allows to use the project for many different data sets and experiments.

3.4.2 Model Training

The SAT-Training framework can also perform the tasks to train and test a machine learning model as well as to use a random search for the best model hyperparameters. As these tasks behave very similarly, they are discussed together in this section.

The only significant difference between them is which models they train. While the train/validate tasks uses arbitrary given hyperparameters to train & test a single model with them, the hyperparameter search samples a given amount of random hyperparameter combinations and trains & tests a model for each of them. It then returns the model with the best generalization, meaning the one that performed best on the validation set.

Possible Model Types

The component had initially used random forests (using scikit-learn [19]) as machine learning models for training, testing and hyperparameter search. Random forests have been chosen as they usually provide good results in many applications and have rather few hyperparameters, and are therefore well suited as a first model type to deliver good initial prediction with relatively few efforts. Nevertheless, the component had been built in a way that makes it easy to change the machine learning model or extend the component to support multiple model types.

Due to the evaluation of the random forest performance (see Section 4.10), the decision has been made to also try deep neural network (using PyTorch [20]) as they are not only able to learn much more complex relations and patterns and have more configuration options, but also have structural advantages over random forest regressors for this specific application (see Section 4.3.4).

Due to time constraints and other priorities, only the train/validate task with an explicit validation set, but not the hyperparameter search, could also be implemented for neural networks.

Other Implementation Details

Requesting the training of a machine learning model requires a test split. The request can then be configured to either use cross-validation or split the data set into an explicit training and validation set. In our experience, using an explicit validation set has produced better results as the cross-validation implementation does not take the family information into account to perform equal distributions.

When using an explicit validation set, the final returned model is re-trained on both the training and the validation set (after testing it). This makes it consistent with the cross-validation option and uses as much data as possible to train the final version in order to get the best results. As the exact train/validation-split is not saved anyways, and only the testing set is used after training as unseen data, this does not distort test results.

Like the data calculation task, these tasks can use a given number of processes in parallel.

After the training has finished, they then save both the resulting model itself, its training and validation scores, hyperparameters and additional information (on the hyperparameter search) in the folder for the respective model. Multiple scoring metrics can be selected for the scoring. For more information on the exact options and their advantages resp. disadvantages, see Section 4.3.2. The scores can, in any case, then be used for evaluation and comparison.

3.4.3 General Implementation Details

The SAT-Training framework provides a CLI as the user interface to configure and execute the individual steps of this workflow. Each user request is implemented as a CLI command.

As it depends on the libraries being present, it contains a shell script which has to be run once to setup the project, creating required folders, building Rust dependencies and installing Python dependencies.

It also contains a scoring module which helps to automatically calculate arbitrarily many different scores for each data point. It provides functions to let scikit-learn use selected scoring metrics and read the calculated scores from a scikit-learn result, as well as to calculate the scores itself for given scoring metrics, data sets and predicted outputs. This helps to reduce unnecessary and duplicated code.

The scoring module also provides custom scoring metrics that are better suited for this specific application than the default ones and can judge the true performance more accurately. For more information on that, see Section 4.3.2 in the evaluation chapter.

3.5 Integration into TaSSAT

TaSSAT is a local search solver (which is built on top of YalSAT). Like other local search solvers, it iteratively flips variables of an internal assignment [5]. It marks an important

piece of groundwork for this research as we alter it to incorporate a machine learning model.

As described in the workflow section, the values predicted by the machine learning model can then be used in combination with the solver's own heuristics to decide on (1.) an initial variable assignment and (2.) a variable to flip in each search step. This machine learning model integration should improve the solver's performance.

The output values that a machine learning model is trained on, and that it should predict, are customized to these two scenarios. Therefore, two different types of data points exist:

3.5.1 Data Point Structure

The model can either be trained on data points with or without history information (meaning a partial assignment). The model trained without history information can only be used to generate an initial assignment of variables, while the model trained with history information can be used for each flip to generate a recommendation on which variables to flip.

In both cases, the model gets information on the formula (and, depending on the case, on the current history) that can easily be calculated while trying to solve the formula. It is then supposed to output data that can help to decide on these problems (which initial assignment to use and which variable to flip). As the calculated (training) data points define the predictions of the model, they have to be structured in a way that allows to make these decisions.

Data Points without History Information

The **input values** of a data point without history information consist of general information on the formula (e.g. its amount of clauses) and specific information on each variable of the formula (e.g. its number of appearances) that a machine learning model should be able to use to predict model counts as accurately as possible.

When calculating data points without history information, the goal is to predict how each variable should be initially assigned for the highest chance to get close to a model in the search space. Therefore, the **output values** of a data point consist of one value for each variable, describing which share of models of the formula contain it positively respectively negatively. This information can directly be used to generate an initial assignment.

As these input and output values are fixed for any formula, exactly one data point should be calculated for each formula.

Data Points with History Information

The **input values** of a data point with history information contains general information on the formula (e.g. its amount of clauses) and specific information on each variable of the

formula (e.g. its number of appearances) that a machine learning model should be able to use to predict model counts under a given history as accurately as possible. Ths history itself is given as one input value for each variable, describing whether the history contains it positively, negatively or not at all.

When calculating data points with history information, the main goal is to predict which variable should be flipped (and included in the history) next in order to find a model as fast as possible. Therefore, the **output values** of a data point consist of one value for each variable. If the variable is not part of the history, its value describes which share of models of the formula *under the current history* contain it positively respectively negatively. The output values of the history variables are irrelevant for this goal; in order to optimally use the machine learning model's capacity, they can be set to zero.

This information can then be used to select a variable to flip (and include in the history) in each step by choosing a variable (that is not in the history) for which the opposite assignment is predicted to lead to a significantly larger number of models under the current history.

The reason why the data points only contain the assignment of the history variables, and not the full assignment of the local search solver, is that the other assignments cannot be utilized. In order to find any amount of models, a large number of variables have to be unassigned (if the model would be counted under the entire assignment, the result would basically always be zero). Since the history is defined as the variables that will not be considered for a flip in the current step, it is reasonable to pick it as the constraints for the model counting to find a different variable to flip.

As these input and output values depend on a history, a number of histories should randomly be generated for each formula, and exactly one data point should be calculated for each formula-history pair.

However, additionally to learning the variable that should be flipped, there could be another goal for the model to predict: In some states, the local search solver might face a situation where the current history is so unfortunate that it excludes (almost) every model. While this would be solved by additional variable flips that move those badly assigned variables out of the fixed history, it would be a further improvement to detect such situations and introduce an escape mechanisms, which allows to exceptionally flip a part of the history in such a case. A way how this could be implemented with a machine learning model would be to also predict the total amount of models under the current history as one general output value. A repeatedly low estimate could trigger such an escape mechanism. In order to decide on which history variable to flip, the output values of history variables could describe how flipping it would change the amount of models under the history, instead of being set to zero.

While the SAT-Training framework is capable to generate such data points with an escape mechanism, we decided that it is more important to focus on producing good prediction values for the core approach; therefore, we have not pursued this idea any longer. Nevertheless, the existing architecture can easily be used to approach this in future research.

3.5.2 Implementation Details

In order to integrate a trained machine learning model into TaSSAT, we modified its source code to call the SAT-Training framework to execute the specified model on the given input values and use its results in combination with its own heuristics. This can also be used as an example of how to integrate a machine learning model into other local search solvers.

4 Experimental Evaluation

As the workflow consists of several distinct steps and components that should all be evaluated in order to find the most effective solutions (see Section 3.1), this chapter contains multiple evaluations, separated in different sections. Each evaluation compares different hyperparameters that had to be decided on in order to process the data and continue with the next step:

- **Compiler evaluation** (Section 4.1)
  An evaluation of different knowledge compilers, which compares their performances and results.
- **Model Counting evaluation** (Section 4.2)
  An evaluation of different model counting algorithms, which compares their performances for different applications.
- **Machine Learning Evaluation** (Section 4.3)
  An evaluation of the different machine learning models that have been trained to predict model counts under given constraints, which compares their prediction accuracy.

In order to carry out the evaluations, we used Spack [21], a package manager for high-performance computing, in combination with Slurm [22], a job scheduler.

4.1 Compiler Evaluation

The first evaluation compares c2d, d4v1 and d4v2; three knowledge compilers which all can transform Boolean functions into a d-DNNF representation. Comparing their performances and output qualities allows to determine the most effective compiler. This can benefit future research as it can use that one to compile and use bigger datasets efficiently.

4.1.1 Performance Evaluation

To evaluate the compiler performances, we compiled our formulas with each of the evalued knowledge compilers and measured the required time for each instance.

Instances

In order to accurately compare the different compilers' performances, we need a big data set containing many real-world instances of CNF formatted Boolean functions. This data

set is provided by the benchmark database [18].

We have used all satisfying formulas contained in the benchmark database that are not synthetic or random and contain between 100 and 1000 variables. Unsatifying formulas have been excluded as the entire approach of training a machine learning model on model counts requires the formulas to be satisfying; it would not make sense to compile unsatisfying ones. Random and synthetic formulas have been excluded as we want to train a machine learning model on the structre of, and the patterns contained in, real-world Boolean functions of different applications; including them reduce the quality of the data set. Functions with more than 1000 variables have been excluded for practical reasons: They generally took too long to compile for this evaluation. And functions with less than 100 variables have been excluded as the data set should not have a too wide range of variable amounts as the patterns for such different functions might differ strongly, which makes it harder for a machine learning model to learn them.

Environment

This evaluation has been performed on a Spack server, which runs on Rocky Linux 9.5. The used server contains 64 CPU cores, each of which is capable of running 2 threads in parallel (resulting in a total of 128 possible parallel threads). Each CPU core is part of an AMD EPYC 7702P 64-Core Processor. It has a total of 1,019,393 MB RAM.

GNU Parallel [23] has been used to run 32 compilation processes in parallel on this server. Runsolver [24] has been used to restrict the time for each compilation process to 5 hours and the memory of each to 30GB in order to avoid the processes to meaningfully influence each other.

The measurement that is used for this evaluation has been provided by runsolver. This is more consistent than using the custom imlpemented measurements provided by the different compilers, yet does not contain a significant measurement overhead.

Both versions of d4 (d4v1 and d4v2) are programmed in C++, while c2d is programmed in C.

Evaluation Results

Figure 4.1 displays the error rate of each compiler. This describes the share of formulas that could not be compiled due to a timeout, running out of memory or a compiler crash. It also displays the average time each compiler required to compile a formula. Only the average time for the formulas that all compilers could successfully compile has been used in order to guarantee the comparability. This average is an important metric as it correlates with the expected time to compile a certain set of formulas.

As expected, d4v1 and d4v2 both were able to compile more formulas than c2d and took less time on average. Surprisingly though, d4v1 performed better in both metrics than d4v2.

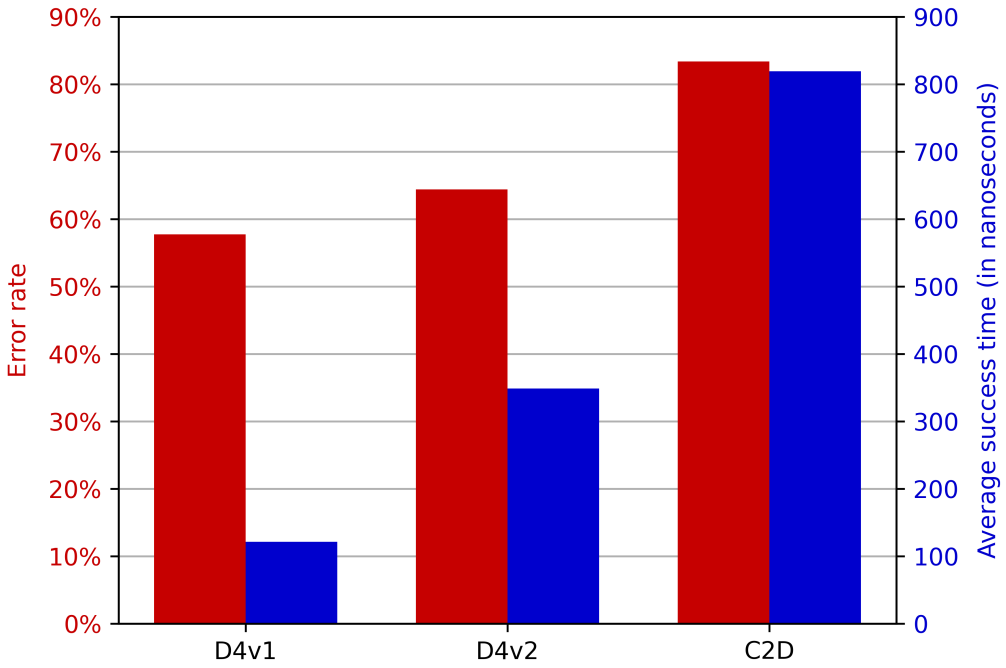

Figure 4.1: The error rate of each compiler and the average time each compiler required for a formula that has been successfully compiled by all compilers.

As the compilation time raises exponentially for more complex formulas, its average is mainly determinated by a very small subset of the most complex formulas and does not necessarily allow conclusions to be drawn on the performance on less complex functions. Therefore, Figure 4.2 displays a scatter plot with the time d4v2 took, in comparison to the time d4v1 took, to compile each given formula. It shows that d4v1 generally took less time than d4v2 to compile a formula, regardless of its complexity.

As Figure 4.3 shows, both d4v1 and d4v2 have both performed significantly better on almost all formulas than c2d.

4.1.2 Result Evaluation

While d4v1 has been shown to be more performant than its successor d4v2, we wanted to see whether this reflects a tradeoff for improved compilation results.

The core idea of knowledge compilation is to move the common expensive part of an algorithm, which is repeated for many different queries, into an initial compilation step to allow the repeated queries to be much faster. Thus, it would make sense to decrease the compilation performance in order to create "better" d-DNNF formula representations, on

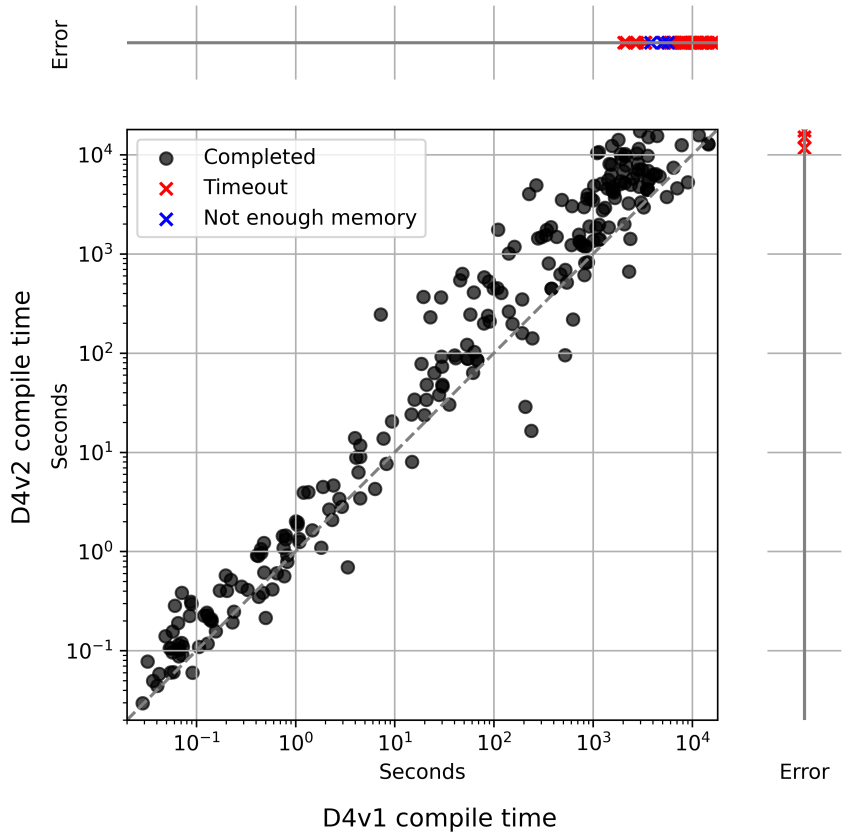

Figure 4.2: The time d4v1 and d4v2 took to compile each given formula.

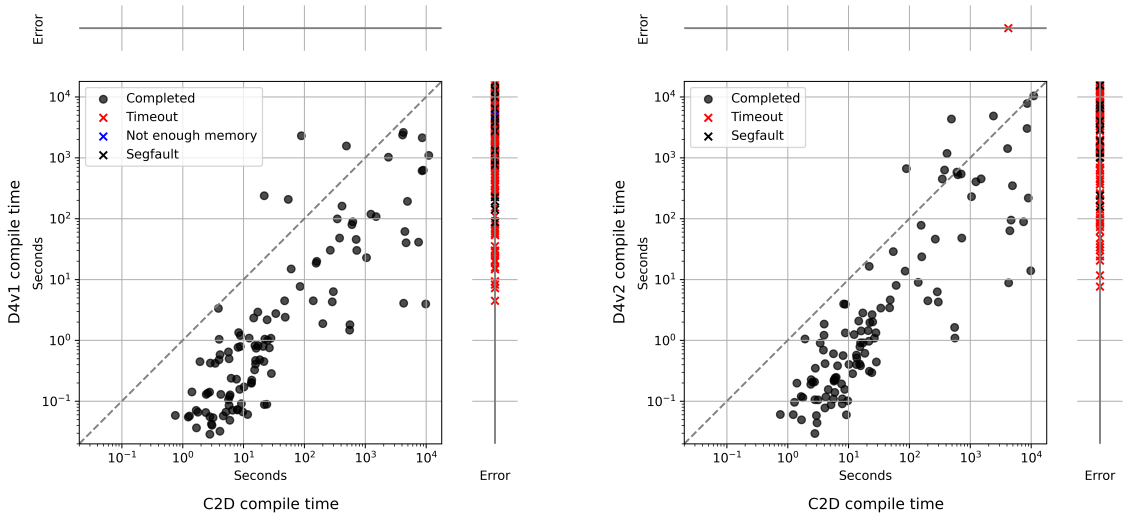

Figure 4.3: The time d4v1 and d4v2 took to compile each given formula in comparison to c2d.

which the model counting algorithm can operate even faster.

Therefore, we continued with evaluating the compilation results by measuring the time it took to parse and count each d-DNNF representation any compiler produced.

Instances

The tested instances are the ones that have been successfully compiled by any of the knowledge compilers in the course of the first evaluation.

Environment

As parsing and counting the models of a d-DNNF formula is much more performant than compiling it, the required time to do so is much lower. This means that just counting the time it takes to do this cannot guarantee an accurate measurement anymore as things like cache content, CPU throttling, benchmarking overhead and randomness can heavily influence such short measured times, resulting in a very large variance.

Therefore, we use the benchmarking suite Criterion in order to get accurate measurements of tasks that can take as little time as just a few nanoseconds. Criterion can produce highly precise benchmarks as it, among other things, repeats the benchmarked code for at least five seconds (potentially using millions of iterations) and only measures the time after at least three seconds of warming up to account for different initial cache contents [16]. Additionally, we only use a single process at a time to prevent multiple processes from influencing each other and affecting the measurements.

This evaluation has been performed on a Spack server, which runs on Rocky Linux 9.5. The used server contains 32 CPU cores, each of which is capable of running 2 threads in parallel (resulting in a total of 64 possible parallel threads). Each CPU core is part of an AMD EPYC 7551P 32-Core Processor. It has a total of 257,804 MB RAM.

Evaluation Results

Figure 4.4 depicts the time it took to parse respectively count each function that has been compiled by both d4v1 and d4v2 for both resulting d-DNNF representations.

There is no big visible difference, although the variance of the data points seems to grow with their complexity. However, it is difficult to judge the exact difference as the data points of many less complex formulas are very close together.

As the average is almost entirely driven by the most complex functions, and as those are more likely to be outliers, it is not very conclusive. Instead, Figure 4.5 presents the average ratio of time it takes to parse and count a formula (of each compiler) in comparison to the time it takes for the corresponding formul created by d4v1.

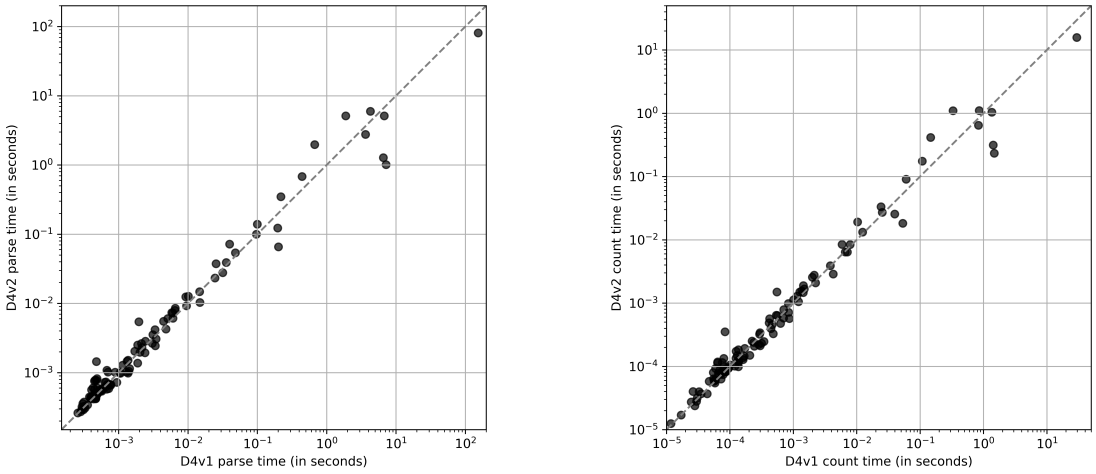

Figure 4.4: The time it took to parse resp. count each function that has been compiled by both d4v1 and d4v2.

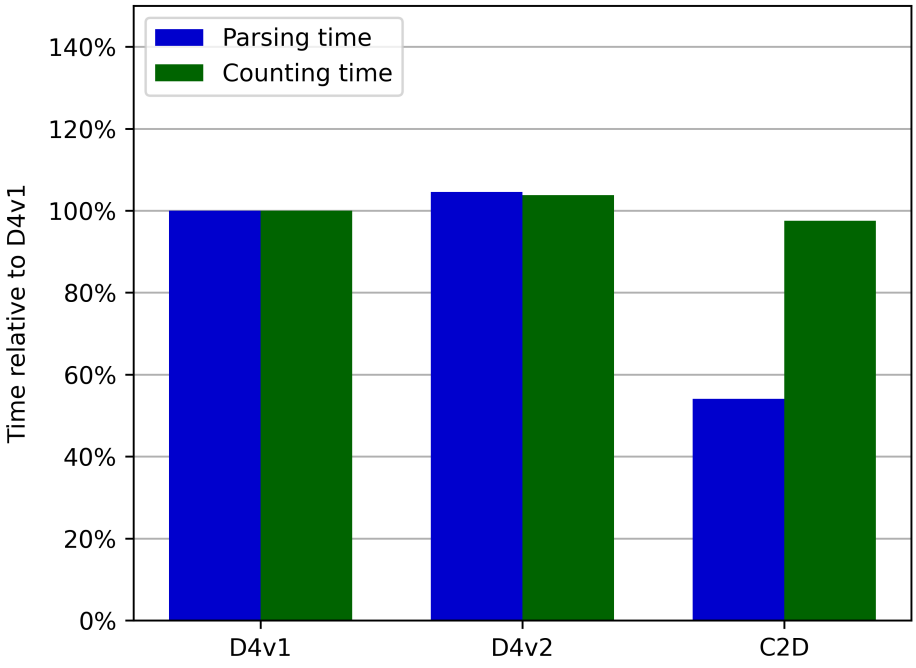

Figure 4.5: The average ratio of time to parse and count a formula for every compiler in comparison to d4v1.

While there is no significant difference between d4v1 and d4v2, the formulas compiled by c2d are surprisingly much faster to parse.

It should be noted though, that when taking all formulas that d4v1 and d4v2 were able to compile (disregarding c2d), as it has been done on Figure 4.4, their ratio difference grows to 118% for parsing and 116% for counting.

4.1.3 Conclusion

In conclusion, d4v1 has been able to compile the most Boolean functions within the given constraints and has generally been the fastest of the tested knowledge compilers. Additionally, the formulas it has produced have not systematically taken longer to parse or count than the ones of d4v2.

Therefore, we chose d4v1 as the used knowledge compiler for the following evaluations. While d4v2 may perform better on differently sized or otherwise specialized Boolean functions, we generally recommend the usage of d4v1 due to these evaluations.

4.2 Model Counting Evaluation

After selecting a knowledge compiler to use, the next step in the workflow is to create a data set to train a machine learning model on. The time it takes to calculate the input values is negligible, but the performance of the model counting, which is required to calculate the output values, is highly relevant, especially for more complex formulas.

Therefore, we evaluated the two available model counters: ddnnife and d-DNNF Model Counter. Again, comparing their performances allows to determine the more effective model counter that should be used in the future to research this approach (and other applications) most effectively.

4.2.1 Instances

As d4v1 has been chosen to be used as knowledge compiler, the used instances are all d-DNNF formulas that have been successfully compiled by d4v1 (from the respective original dataset, see Section 4.1.1).

4.2.2 Environment

Both ddnnife and d-DNNF Model Counter are programmed in Rust. This allows to use Criterion for both of them in a consistent way, which results in high-precision benchmarks to accurately compare these libraries.

We only use a single process at a time to prevent multiple processes from influencing each other and affecting the measurements.

This evaluation has been performed on a Spack server, which runs on Rocky Linux 9.5. The used server contains 32 CPU cores, each of which is capable of running 2 threads in parallel (resulting in a total of 64 possible parallel threads). Each CPU core is part of an AMD EPYC 7551P 32-Core Processor. It has a total of 257,804 MB RAM.

4.2.3 Evaluation Results

Figure 4.6 shows the average ratio of time it took to parse a formula with the d-DNNF Model Counter in comparison to the time it took to parse the same formula with ddnnife.

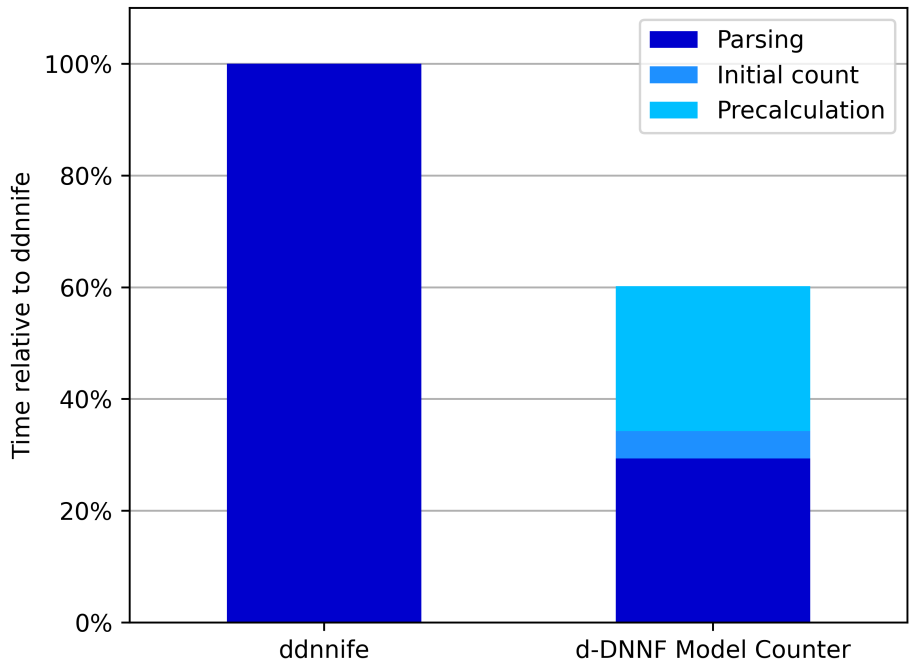

Figure 4.6: The average ratio of time to parse a formula with the d-DNNF Model Counter in comparison to ddnnife.

The d-DNNF Model Counter leaves the users the choice exactly which precalculations and optimizations to use. This allows for users to choose the configuration that is most performant for their use-case, as the cost of a precalculation required for an optimization techinque might not be worth it for a small number of requests.

This requires the user to explicitly request the precalculation (of core features and node variables) and caching of calculated model counts if that is wanted. To allow the user to

rely on the requested optimizations being used, the model counter returns an error if one of them cannot be used due to any missing prior calculation.

While ddnnife only uses a single parsing process, which precalculates everything required for the optimizations it uses to count the models, the d-DNNF Model Counter leaves the users the choice which exact precalculations and optimizations to use. This allows for users to choose the configuration that is most performant for their use-case, as the cost of a precalculation required for an optimization techinque might not be worth it for a small number of requests.

Therefore, the one-time cost the d-DNNF Model Counter requires to parse a formulas in order to repeatedly count its models under given constraints depends on the precalculations requested by the user. Figure 4.6 contains these different times it requires to just parse the formula without any precalculations, to parse it and perform an initial count to allow partial traversals with an initial graph traversal for each query, and to parse it and perform all possible precalculations to allow partial traversals without an initial graph traversal and using core features.

To just parse a formula without any precalculation, the d-DNNF Model Counter took (on average) 29% of the time ddnnife took. But even when using all precalculations, it still only required 60% of the time ddnnife does, while allowing for additional optimizations.

Interestingly, the average time the d-DNNF Model Counter took is *less than a percentage* than the one ddnnife took. This is due to their time difference increasing for more complex formulas. Figure 4.7 displays the time both ddnnife and the d-DNNF Model Counter took to parse each compiled formula, mapped to the amount of edges of that formula, which illustrates that the parse time depends on the amount of edges. Figure 4.8 confirms that this is also valid for the different precalculation tasks. However, after a certain threshold of around 10^6 edges, the parse times of ddnnife begins to increase at a significantly higher (constant) rate. But due to the rather small sample size of formulas at that scale, we cannot be certain that this is generally valid.

The actual model counting performance highly depends on a variety of factors. The size of the given partial assignment has a big impact as it can incrase the amount of nodes that have to be traversed when using partial traversal optimizations. At the same time, however, a bigger partial assignment increases the probability of an excluded core feature, which (using corresponding optimizations) results in a near-zero time.

As figure 4.9 shows, ddnnife was generally able to count the models more efficiently. Surprisingly, the full node optimization that prevents the d-DNNF Model Counter from traversing the graph multiple times seemed to have a negative impact for less complex formulas. This indicates a higher than expected overhead to compare the variables, which may be an opportunity for a bigger performance improvement in the future.

However, consistent with the parsing evaluation, it seems that the d-DNNF Model Counter can scale more efficiently as the performance difference heavily decreases with increasing formula complexity as well as for a higher amount of constraints. This is likely due to

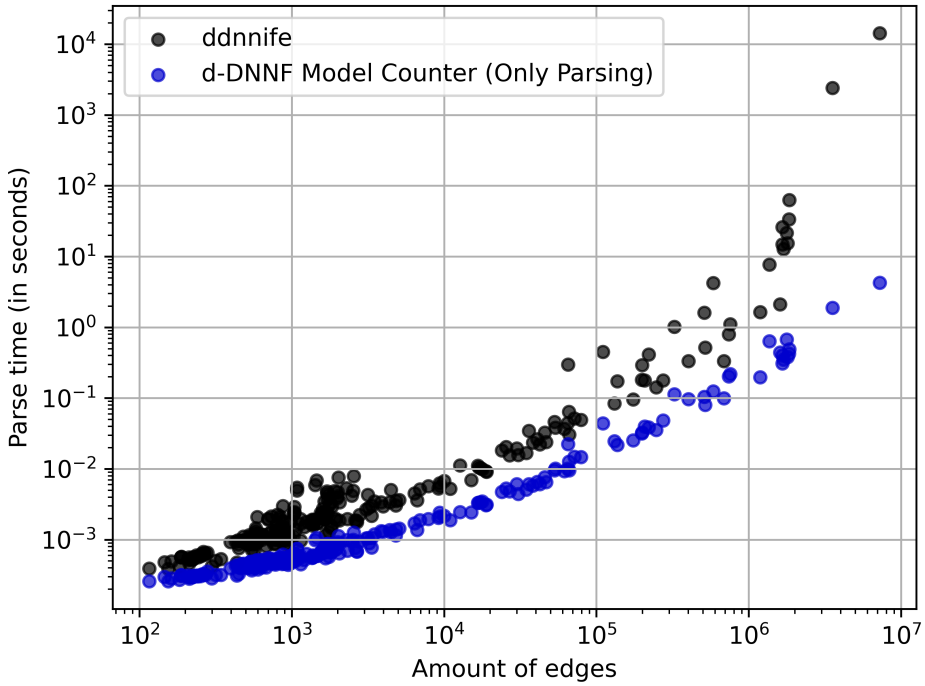

Figure 4.7: The time ddnnife and the d-DNNF Model Counter took to parse each compiled formula, mapped to its amount of edges.

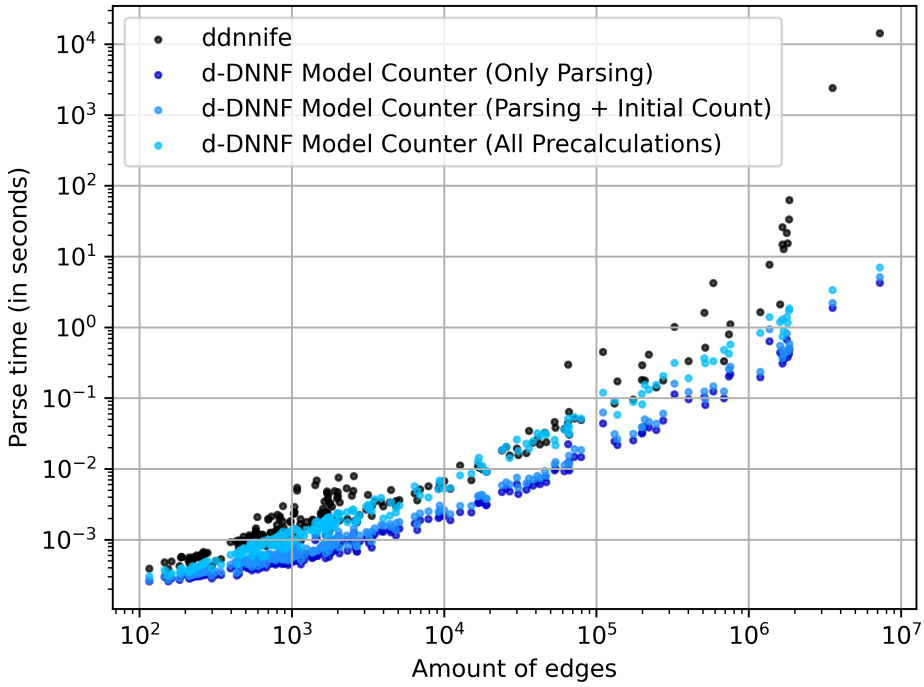

Figure 4.8: The average ratio of time to parse each formula with the d-DNNF Model Counter in comparison to ddnnife.

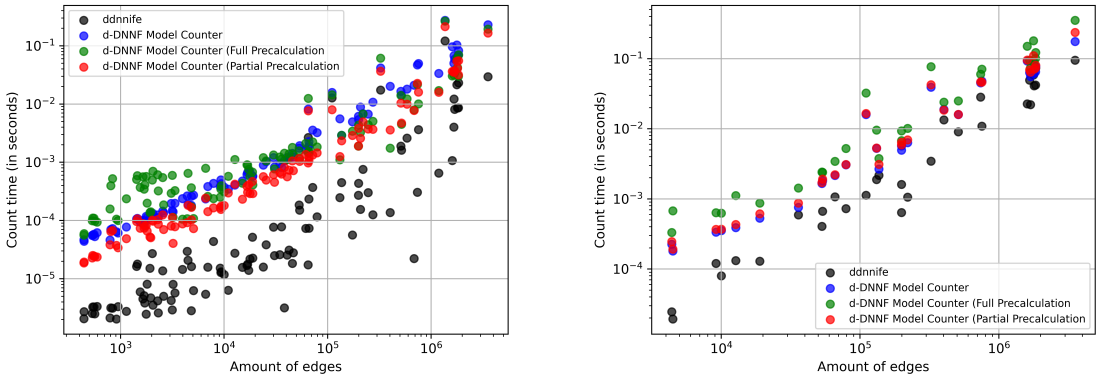

Figure 4.9: The time ddnnife and the d-DNNF Model Counter took to count the compiled formulas, mapped to their amount of edges, for the history sizes of 1 and 25.

the fact that ddnnife contains more advanced optimizations that especially benefigt smaller Boolean functions.

4.3 Machine Learning Evaluation

After selecting the more performant model counter to calculate a data set, the data set is then used to train a machine learning model. In order for the machine learning model to be able to improve the local search solver, it must perform as good as possible and accurately predict the model counts.

This is why we evaluated the accuracy of different machine learning models and their performance under different hyperparameters. That allows us to pick the most effective model and use it for the actual integration in a local search solver.

The usage of an explicit training and validation set has performed better results than cross-validation in early tests. This is likely due to the cross-validation implementation not taking the family information into account to perform equal distributions, resulting in some splits that make it hard to apply the learned patterns on the validation set. Therefore, we have used an explicit training and validation split for all machine learning evaluations.

4.3.1 Instances

In order to train a machine learning model, the data points are calculated by the SAT-Training framework (which uses the d-DNNF Model Counter and the CNF Analyzer). Similarly to the model counting evaluation, the formulas used to generate data points are all d-DNNF formulas that have been compiled by d4v1 (from the respective original dataset, see Section 4.1.1).

It should be noted that this may introduce a certain selection bias. As only the subset of models that have successfully been compiled could be used to generate a data set for the machine learning evluation, this might have distorted the distribution of patterns in the formulas and made them less representative.

4.3.2 Scoring Metrics

The default scoring metrics are the mean squared error and the r2 score. These use all output values to determine a score. However, the amount of variables in a Boolean function is variable; this is why we have included an "exists bit" in the input values, and fill up the input and output values with zeroes for the non-existing variables. Therefore, a scoring metric that uses all output values does not necessarily correlate with the true model performance. For example, a model that perfectly predicts the zeroes for non-existing variables but no other values could get better scores than a model that produces good predictions for

only the existing variables. This shows that the default scoring metrics are not very well suited for this application.

The default (scikit-learn) implementation of the r2 score also has an additional problem: It calculates the r2 score for each output feature individually and then returns the mean of these values. While this makes sense for other applications in which the output values all provide different data on different scales, the output features of this application contain one value for each variable, and the variable order is interchangeable. There is no structural difference between the output features (except for the one additional general value for models with an escape mechanism). Because of the way these data points are structured, the default implementation score strongly depends on the specific data set (and non-generalizing patterns about the variable order in it) and is not well suited as a metric for this application.

This is why we introduce our own so-called "valid part" scores. These valid part scores only calculate the mean squared error and the r2 score of the valid part of the output values (meaning of the variables that actually exist, but without taking the filled up values into account).

Implementation Details

The valid part scoring metrics depend on the amount of variables for each data point (which cannot be reconstructed from the output values, as it depends on the exists bit, which is an input value). And as we do not have control over the (cross-validation) splits that are performed by scikit-learn, we cannot know which data points are passed to the scoring functions.

We solve this by using a pandas DataFrame. This DataFrame saves the indices of each value and keeps them when splitting it up; this way, we can still access the given indices, use them to idenftify the data points and calculate the correct valid part score, even though it depends on the input values.

4.3.3 Random Forest Regressors

The machine learning models we initially used were random forest regressors as they usually provide good results in many applications and have rather few hyperparameters.

Environment

As we only want to evaluate the accuracy of the trained random forest regressor, we do not have to perform a benchmark. Therefore, the exact hardware information is irrelevant as it does not influence the outcome of this evaluation.

The evaluation has been performed by utilizing the SAT-Training framework, which is programmed in Python, and its hyperparameter search tasks. Thus, scikit-learn has been used for the creation and training of the different random forest regressors.

Tuning Parameters

While random forest regressors have less hyperparameters and a more fixed structure than many other machine learning models, there is still a range of parameters that can be turned in order to improve their accuracy.

Using the SAT-Training framework, we have concluded a random hyperparameter search, which created 2400 different random forest regressors and randomly set their hyperparameters within given ranges and values for each of them.

The exact hyperparameters and the ranges and values they were randomly sampled from are the following:

- The number of trees
  $10 - 4000$ (logarithmically sampled)
- The number of features to consider for each node's split
  30% probability: All, 15%: $\sqrt{\text{All}}$; 15%: $\log$ All; 30%: $0.01 - 1 * \text{All}$ (logarithmically sampled); 10%: $10 - 500$ (logarithmically sampled)
- The maximum depth of any tree
  30%: None; 70%: $4 - 40$ (logarithmically sampled)
- The minimum number of samples required to split an internal node
  80%: $2 - 32$ (logarithmically sampled); 20%: $0.02 - 0.2 * \text{All}$ (logarithmically sampled)
- The minimum number of samples required to be at a leaf node
  80%: $1 - 20$ (logarithmically sampled); 20%: $0.02 - 0.2 * \text{All}$ (logarithmically sampled)
- Whether bootstrap samples are used when building trees
  90%: True; 10%: False
- The minimum required impurity decrease to split a node 80%: 0; 20%: $0.0005 - 0.025$ (logarithmically sampled)
- The maximum number of leaves per tree
  75%: None; 25%: $75 - 750$ (logarithmically sampled)

Many of these ranges are a bit broader than usual in order to explore a wider parameter space. This was possible as we have used a high number of samples, ensuring that a large number of samples are directed towards the most promising hyperparameter combinations.

The most important hyperparameters are usually the amount of trees and the number of features to consider for each node's split[25].

Evaluation Results

Figure 4.10 gives an overview on the performance of a random forest regressor with the default hyperparameters. It can be seen that the regressor trained with an explicit training

and validation set significantly outperformed the one trained with cross-validation, which shows the importance of the data being distributed evenly.

Mean squared error	Training set	Validation set
Explicit sets	0.083	0.522
Cross-validation	0.080	0.677

R2 score	Training set	Validation set
Explicit sets	0.887	0.225
Cross-validation	0.890	0.047

Figure 4.10: Comparison of two tables.

Nevertheless, with the range of possible outputs being between -1 and 1, a mean squared error of 0.5 on the validation set is significant. The r2 score, which can be used to estimate the amount of variance that is explained by the machine learning model, also demonstrates the suboptimal prediction accuracy.

Because of these unsatisfying results, we have chosen to explicitly perform a random search in order to find a better combination of hyperparameters to improve the predictions.

Figure 4.11 shows the mean squared error and r2 score on both the training and the validation set of each random forest regressor that has been created with randomly sampled hyperparameters. The random forest regressor that has been created with the scikit-learn default hyperparameters is highlighted in red.

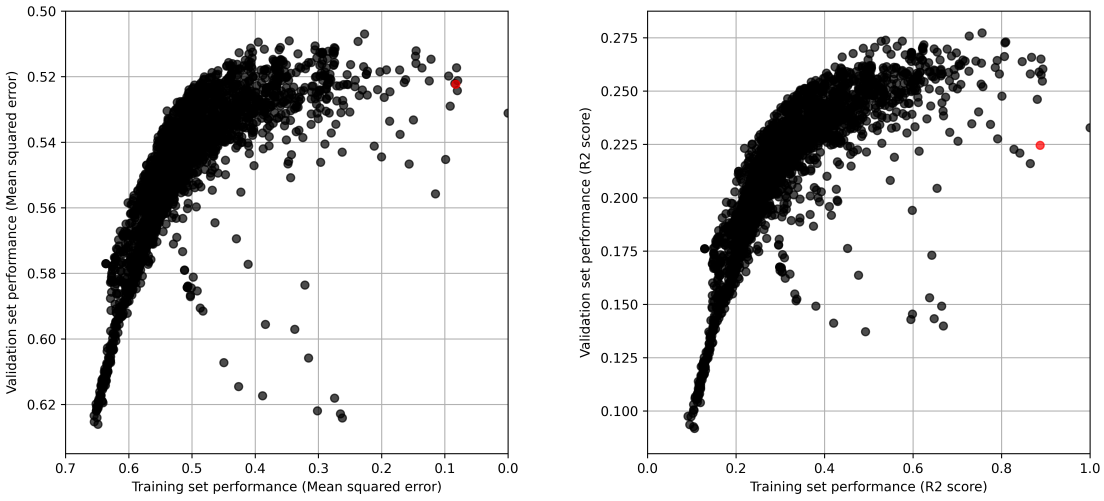

Figure 4.11: The performances on the training and validation set of all random forest regressors (the one with the default hyperparameter values being highlighted in red).

The effects on different amounts of capacity can easily be seen. Forests with too little capacity to learn the patterns perform poorly on both the training and the validation set and are displayed in the bottom left corner. Forests with the right amount of capacity perform the best on the validation set and good on the training set, and are therefore in the top center. And forests that have too much capacity overfit and learn non-generalizing patterns that only apply to the testing set, causing them to perform best on the training set but not that good on the validation set, and are located in the center right. Interestingly, the default hyperparameter values scikit-learn provide seem to slightly overfit.

However, it is important to pay attention to the scaling. While many forests achieved a mean squared error of less than 0.2 on the training set, the best ones have still produced an MSE of over 0.5 on the validation set. No forest has achieved an r2 score of 0.3 or higher on unseen data, which means that over 70% of variance in the data cannot be explained by any of the random forest regressors.

While the training shows that there is significant structure in the data that can be learned, those results fell short of the expectations. Therefore, we tried a different machine learning model that might be able to capture more of the patterns in the data.

4.3.4 Neural Network Advantages

Due to the poor performance of random forests, the decision has been made to also try deep neural networks. Those have several advantages: They are not only able to learn much more complicated relations and have more configuration options, but also have structural advantages over random forest regressors for this specific problem:

One key aspect of this problem is that the variable order is interchangeable and that there is no structural difference between the output features (except for the one additional general value for models with an escape mechanism). But a random forest cannot be aware of that and therefore wastes capacity by learning non-generalizing patterns about the variable order in the training set. In order to avoid that, we could randomly shuffle the variable order of each data point numerous times and use the newly created data set for training. But this is still not ideal as the random forest cannot take advantage of this structure and combine the data of each single variable to learn how specifically it implicates specific output values without requiring much capacity.

A neural network, however, can take advantage of this structure by using **shared weights**. As it gets the same structure of input values for each variable, we can create a sub-network that is used for each variable individually, extracting the important features out of the respective input values. This way, the neural network does not waste any capacity on learning patterns in the variable order, but instead just focuses on the variable data and combines the information it learns for each variable in a single network (which it can then apply to each variable).

Another way a neural network can take advantage of this structure is **aggregation**. Even when such a sub-network is used for each variable, when individually connecting their

output values to a second network, the order of the variables would still have an impact and could cause the model to learn information about patterns in the variable order of the training set. To avoid that, we can also use shared weights and biases to connect each variable network's n-th output value to an input neuron, but due to the distributive law, this is mathmatically the same as to calculate the sum (a kind of aggregation) of the n-th output value of each variable network and pass it to the input neurons. An additional advantage of this aggregation is that makes the network independent from the actual amount of variables in a formula. The sub-networks only have to be evaluated and aggregated for the existing variables, without the network having to take non-existing variables into account.

However, this aggregation has the consequence that actually useful information gets lost. For example, the model count might not only depend on the sum of a value for each variable, but also on the actual distribution of the values, e.g. its variance or whether there are clusters. To avoid losing this information, while still preventing the graph from learning non-generalizing patterns, it is also possible to instead sort the variable sub-networks based on one or multiple of their output values. A compromise could be to use multiple aggregations.

Using these approaches, we propose a specific neural network structure adapted to this problem (see Figure 4.12): It contains five subnetworks that are connected to each other and take advantage of the structure of the data points to not be aware of the variable order and learn the information for each variable combined.

- **Initial group network**
  The initial group network gets the input values of one variable as input values and extracts important information about the specific variable, that can be used in the following networks.
- **Initial general network**
  The initial general network gets the general input values (that don't describe a specific variable, but the entire formula) as input values and extracts important information about the general formula, that can be used in the following networks.
- **Secondary group network**
  The secondary group network gets the output values of the initial general network as well as the output values of the corresponding initial group network and extracts important information about the specific variable in the general state of the formula, that can be used in the following network.
- **Aggregation network**
  The aggregation network gets the output values of the initial general network as well as the aggregated input values of each initial group network (e.g. the sum and variance of all n-th output values of a single initial group network). This network then uses the general formula information and aggregated data of all variables to extract important information on the overall formula and the state of individual variables in it, that can be used in the following network.

- **Final group network**
  The final group network gets the output values of the aggregation network as well as the output values of the corresponding secondary group network and extracts the final information on the corresponding variable, that is used as prediction.

(When using a model with an escape mechanism, the aggregation network can output one additional value that is used as the general prediction of model amounts in the formula.)

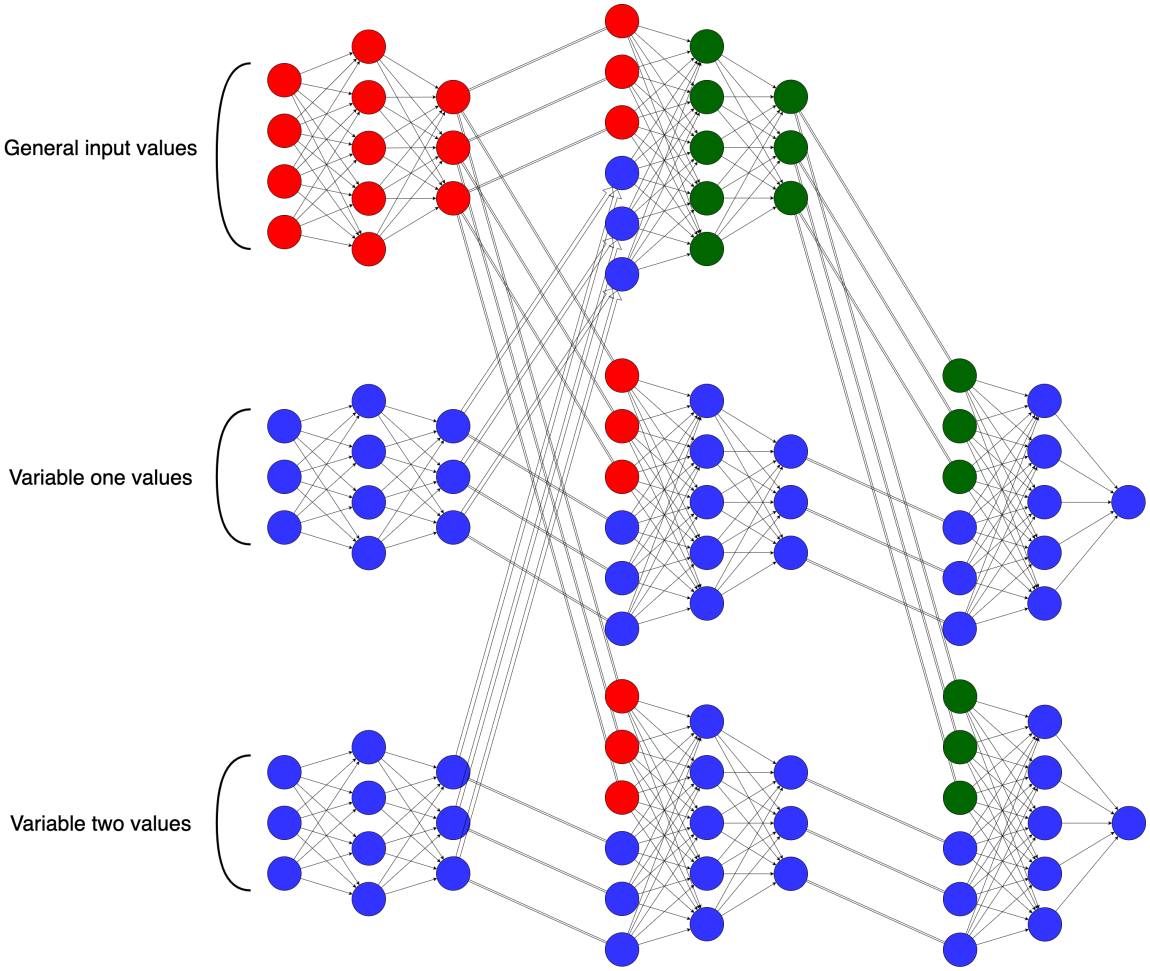

Figure 4.12: The structure of the proposed neural network.

4.3.5 Neural Networks

Environment

As with the evaluation of random forest regressors, we focused on the accuracy of the final trained model, and therefore didn't employ any benchmark.

Pytorch has been used in combination with the SAT-Training framework to create, train and test the neural networks.

Tuning Parameters

Neural networks are very flexible and therefore have numerous hyperparameters. While the specific architecture of the proposed neural networks already fixes some of them, there is still a substantial number of choices, including the depth of each sub-network, the amount of nodes per layers, the used activation function, the learning rate and the optimizer.

Due to the complex structure of the model and dependencies of such different hyperparameters on each other, conducting a systematic random search has not been feasible under the given conditions. Therefore, we instead applied a manual hyperparameter tuning scheme, testing different hyperparameter combinations that are known to be effective.

Evaluation Results

As established in Section 2.1.3, the increased flexibility and ability to learn more complex patterns of neural networks comes with the cost of a more challenging hyperparameter tuning process. As time constraints prevented a more comprehensive hyperparameter search, we prioritized more promising parts of this research.

We were therefore unable to yield useful predictions. In our experiments, the neural network consistently converged to predicting only zeroes.

5 Discussion

5.1 Conclusion

By implementing an entire framework specialized to utilizing knowledge compilation to effectively train machine learning models, this thesis lays the groundwork to make this approach more accessible.

Despite the challenging circumstances, having to create an entire data set from the ground up, we were able to achieve some remarkable results. By specializing an algorithm for this approach and leveraging new theoretical achievements, we could significantly improve the workflow performance to use especially complex Boolean functions for SAT solving improvement.

As the evaluations show, it presents a significant challenge to apply this workflow for real-world improvements. Nevertheless, we are confident that with more time and resources, future research will be able to build upon these findings as they hold the potential to deliver fundamental improvements.

Given the significant impact SAT solvers have on many algorithmic fields, this is a highly relevant area for future research.

5.2 Future Work

As this thesis opens up a new research field, it holds significant potential for future research and work.

We have identified several promising directions for further research, like a specialized proposed neural network structure, additional optimization approaches to compose valuable data even quicker and complementary techniques to help local search solvers to converge more quickly. While those could not be pursued within the scope of this work, they impose excellent research opportunities.

This also includes a more thorough evaluation of the developed approaches in order to utilize their full potential and find the most effective combinations. Another promising possibility is to apply these approaches on more specific subsets and SAT applications that deal with more comlex formulas, as the improvements introduced in this thesis make it possible to handle them especially well.

As the constraint of a fixed amount of variables imposed significant difficulties, it would also be a worthwile approach to explore other model types that could bypass this limitation, like a recurrent neural network that can take arbitrarily sized data as input.

Utilizing automated hyperparameter tuning could also help identifying ways to overcome the restrictions faced by some machine learning models in our study, and allow a more effective model training.

More future work is required to create a significantly larger data set. This is especially important as the entire approach heavily relies on the data set quality and size, in order to avoid selection bias and non-generalizing patterns. Spending more time to provide such an open data set could therefore provide substantial benefits to the general research community and increase both the accessibility and attractiveness of this research area.

Bibliography

[1] Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. CaDiCaL, Gimsatul, IsaSAT and Kissat entering the SAT Competition 2024. In Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2024 – Solver, Benchmark and Proof Checker Descriptions*, volume B-2024-1 of *Department of Computer Science Report Series B*, pages 8–10. University of Helsinki, 2024.

[2] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, July 1960.

[3] Sahel Alouneh, Sa'ed Abed, Mohammad H. Al Shayeji, and Raed Mesleh. A comprehensive study and analysis on sat-solvers: advances, usages and achievements. *Artificial Intelligence Review*, 52(4):2575–2601, 2019.

[4] Rehan Abdul Aziz, Geoffrey Chu, Christian Muise, and Peter Stuckey. #∃SAT: Projected model counting. In Marijn Heule and Sean Weaver, editors, *Theory and Applications of Satisfiability Testing – SAT 2015*, pages 121–137, Cham, 2015. Springer International Publishing.

[5] Md Solimul Chowdhury, Cayden R. Codel, and Marijn J. H. Heule. Tassat: Transfer and share sat. In Bernd Finkbeiner and Laura Kovács, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 34–42, Cham, 2024. Springer Nature Switzerland.

[6] Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17:229–264, 2002.

[7] Adnan Darwiche et al. New advances in compiling cnf to decomposable negation normal form. In *Proc. of ECAI*, pages 328–332. Citeseer, 2004.

[8] Jean-Marie Lagniez and Pierre Marquis. An improved decision-dnnf compiler. In *IJCAI*, volume 17, pages 667–673, 2017.

[9] Jean-Marie Lagniez and Pierre Marquis. About Caching in D4 2.0. In *Workshop on Counting and Sampling 2021*, Barcelona, Spain, July 2021.

[10] Chico Sundermann, Heiko Raab, Tobias Heß, Thomas Thüm, and Ina Schaefer. Reusing d-dnnfs for efficient feature-model counting. *ACM Trans. Softw. Eng. Methodol.*, 33(8), November 2024.

[11] Jia Liang. Machine learning for sat solvers. 2018.

[12] Jesse Michael Han. Enhancing sat solvers with glue variable predictions, 2020.

[13] Daniel Selsam and Nikolaj Bjørner. Guiding high-performance sat solvers with unsat-core predictions. In Mikoláš Janota and Inês Lynce, editors, *Theory and Applications of Satisfiability Testing – SAT 2019*, pages 336–353, Cham, 2019. Springer International Publishing.

[14] Haoze Wu. Improving sat-solving with machine learning. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '17, page 787–788, New York, NY, USA, 2017. Association for Computing Machinery.

[15] Jiongzhi Zheng, Kun He, Jianrong Zhou, Yan Jin, Chu-Min Li, and Felip Manyà. Integrating multi-armed bandit with local search for maxsat. *Artificial Intelligence*, 338:104242, 2025.

[16] Jorge Aparicio and Brook Heisler. Criterion: statistics-driven micro-benchmarking library, 2014–2025.

[17] Adnan Darwiche. The c2d compiler user manual. *Computer Science Department, UCLA, Tech. Rep. D-147*, 2005.

[18] Markus Iser and Carsten Sinz. A problem meta-data library for research in sat. *Proceedings of Pragmatics of SAT*, pages 144–152, 2015.

[19] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.

[20] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, CK Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Michael Suo, Phil Tillet, Eikan Wang, Xiaodong Wang, William Wen, Shunting Zhang, Xu Zhao, Keren Zhou, Richard Zou, Ajit Mathews, Gregory Chanan, Peng Wu, and Soumith Chintala. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*. ACM, April 2024.

[21] Todd Gamblin, Matthew LeGendre, Michael R. Collette, Gregory L. Lee, Adam Moody, Bronis R. de Supinski, and Scott Futral. The Spack Package Manager: Bringing Order to HPC Software Chaos. Supercomputing 2015 (SC'15), Austin, Texas, US, November 2015.

[22] Morris A. Jette and Tim Wickberg. Architecture of the slurm workload manager. In Dalibor Klusáček, Julita Corbalán, and Gonzalo P. Rodrigo, editors, *Job Scheduling*

Strategies for Parallel Processing, pages 3–23, Cham, 2023. Springer Nature Switzerland.

[23] Ole Tange. Gnu parallel 20240822 ('southport'), August 2024.

[24] Olivier Roussel. Controlling a solver execution: the runsolver tool. *JSAT*, 7:139–144, 11 2011.

[25] Scikit learn Developers. Random forest parameters, 2025. Accessed: 2025-04-14.