Exploiting d-DNNFs for Efficient Cardinality-Based Feature-Model Analyses

Heiko Raab

Institute of Software Engineering and Programming Languages
Heiko Raab:
Exploiting d-DNNFs for Efficient Cardinality-Based Feature-Model Analyses

Advisors:

Prof. Dr. Thomas Thüm
Department of Software Engineering and Programming Languages

M.Sc. Chico Sundermann
Department of Software Engineering and Programming Languages

Bachelor’s Thesis, University of Ulm, December 6, 2021.
Abstract

A product line is a family of related products. Feature models are a widely used formalism to describe product lines. As those feature models are too complex and extensive to analyze manually, tool support is required. In the last two decades, product-line analyses have been reduced to SAT and more recently to #SAT solving. Analyses based on #SAT often require numerous highly similar queries and #SAT solvers have to compute each of those queries individually. One approach to solve multiple queries is to use knowledge compilation. Therefore, we first invest time in an offline phase to transform the CNF into a corresponding representation in the target language to compute the cardinalities more efficiently in the following online phase. Deterministic decomposable negation normal forms (d-DNNFs) are a suitable target language because they allow us to compute #SAT in polynomial time. In this thesis, we present our tool dknife which exploits d-DNNFs to compute results for #SAT-based analyses. Moreover, we empirically evaluate dknife against state-of-the-art #SAT solvers. For 126 of the 127 evaluated models, at least one compiler can compute a d-DNNF within a few seconds. The fastest d-DNNF combination, consisting of dknife together with dsharp as compiler performs significantly better than the fastest #SAT solvers.
# Contents

List of Figures ........................................ vii

List of Tables ........................................... ix

List of Code Listings .................................... xi

1 Introduction ........................................... 1

2 Background ............................................. 3
   2.1 SAT ................................................... 3
      2.1.1 Decomposable Deterministic Negation Normal Form .... 4
      2.1.2 Solving SAT Problems .......................... 7
   2.2 #SAT ................................................. 8
      2.2.1 Solving #SAT Problems ....................... 8
   2.3 Feature Models .................................... 9
   2.4 Feature-Model Analyses ........................... 11

3 Feature-Model Analyses with d-DNNFs ................ 13
   3.1 Transforming CNF into d-DNNF .................... 13
   3.2 Cardinality of a Feature Model .................. 14
   3.3 Cardinality of Features ........................... 15
   3.4 Cardinality of Partial Configurations ............ 16
   3.5 Core and Dead Features ........................... 18
   3.6 Summary ........................................... 21

4 Implementation ......................................... 23
   4.1 Parsing ............................................. 23
   4.2 Data Structure ..................................... 24
   4.3 Algorithm Implementations ....................... 26
      4.3.1 Cardinality of Node .......................... 26
      4.3.2 Cardinality of One Feature ................... 28
      4.3.3 Cardinality of Several Features .............. 31
      4.3.4 Cardinality of Partial Configurations ....... 33
      4.3.5 Core and Dead Features ....................... 33
   4.4 Summary ........................................... 35

5 Evaluation ............................................. 37
   5.1 Research Questions ............................... 37
5.2 Subject Systems .................................................. 38
5.3 Experiment Design ............................................... 40
5.4 Results ............................................................... 42
  5.4.1 Experiment 1: Compilation ............................. 42
  5.4.2 Experiment 2: Cardinality of Feature Model ...... 44
  5.4.3 Experiment 3: Cardinality of Partial Configurations 45
  5.4.4 Experiment 4: Cardinality of Features ............. 48
5.5 Discussion .......................................................... 52
5.6 Threats to Validity ............................................... 54
5.7 Summary ............................................................ 55

6 Related Work ......................................................... 57

7 Conclusion and Future Work ........................................... 61

Bibliography ............................................................ 63
List of Figures

2.1 Propositional Formula Being Transformed into NNF .................................. 4
2.2 Propositional Formula in CNF as DAG ...................................................... 4
2.3 Transformation into a DNNF ................................................................. 5
2.4 Transformation into a d-DNNF ............................................................. 5
2.5 Add Smoothness to the d-DNNF .......................................................... 6
2.6 Generic Decision Node ........................................................................... 7
2.7 The Feature Model of a Sandwich ........................................................ 10
3.1 A d-DNNF as DAG and as Adjacency List ............................................. 14
4.1 UML Diagram for the d-DNNF Data Structure ....................................... 25
5.1 Comparison of CNF to d-DNNF Compilers ............................................. 43
5.2 Comparison of CNF to d-DNNF Compilers Regarding CDL Models ....... 43
5.3 Cardinality of a Feature Model Including Compile Time ......................... 45
5.4 Runtimes for the Cardinality of 250 Satisfiable Partial Configurations ....... 46
5.5 Runtimes for the Cardinality of 250 Unsatisfiable Partial Configurations .... 46
5.6 Break-Even Point between d-DNNFs and #SAT Solvers for Partial Configurations ................................................................. 48
5.7 Cardinality of Features .......................................................................... 49
5.8 Cardinality of Features: d-DNNF Reasoners Without Compilation Time ......................................................................................... 50
5.9 Break-Even Point between d-DNNFs and #SAT Solvers for Several Features ......................................................................................... 51
List of Tables

5.1 Overview of Industrial Feature Models ................................................. 39
5.2 Number of Queries Needed for dknife + dsharp to Overtake #SAT Solvers for the Cardinality of Partial Configurations .......................... 48
5.3 Number of Queries Needed for dknife + dsharp to Overtake sharp-SAT for the Cardinality of Features ................................................. 51
List of Code Listings

4.1  Cardinality of a Node (Naive) ........................................ 27
4.2  Cardinality of a Node When Using the Marking Algorithm .......... 28
4.3  Cardinality of Feature with Marking ................................. 29
4.4  The Algorithm for Upwards Propagation and Marks .................. 29
4.5  Core Computations for Cardinality-Based Analyses Using Marking  30
4.6  Cardinality of Features with Marking ................................ 32
4.7  Cardinality of Partial Configurations Using Marking ................. 33
4.8  Determine All Core Features of the d-DNNF ........................... 34
4.9  Check Whether a Query Is Satisfiable .................................. 35
4.10 Remove All Unnecessary Features From the Query .................... 35
1. Introduction

A product line is a family of related products that are composed into smaller units called features [SAKS14]. Many features can be combined to a product configuration and furthermore, we can reuse a feature for different product configurations [BSRC10]. Systems are often limited by certain constraints that, for instance, require one feature and exclude another one. Feature models are widely used to specify the valid configurations of a product line [Bat05].

Typically, feature models, as we use them in the real world, are not manually analyzable because they can contain up-to thousands of features and constraints [STS20]. Therefore, it is a tedious and error-prone task to analyze a feature model by hand [BSRC10]. Consequently, automated support for feature models [Bat05] is needed. We can transform each feature model into propositional formulas [MWC09]. Over time, analyses of product lines have been reduced to SAT [PLP11] and recently to #SAT [JP00, Thuo6, SNB’21]. A #SAT solver computes the number of valid solutions for a given formula which is often a Conjunctive Normal Form (CNF), while a SAT solver just determines whether there is at least one solution. Although #SAT solvers improved performance-wise over time, there are still some complex industrial systems that we cannot analyze with reasonable time effort, such as the operating system Linux [Thü20]. Additionally, many analyses depend on the cardinalities of several features or partial configurations which describe the number of valid configurations (1) containing a specific feature and (2) including some and excluding other features, respectively. Those analyses need multiple, potentially expensive calls [STS20]. Those calls are often highly similar because the majority of the formula for the #SAT solver stays the same.

To reuse knowledge over multiple queries, we use deterministic decomposable negation normal forms (d-DNNFs) [Dar01a] which represent a feature model as a rooted and directed graph. Existing work has shown that we can transform CNF formulas into d-DNNFs [Dar01a, SGRM18, Dar04]. The transformation itself is computationally expensive but we can utilize the resulting d-DNNF to increase the performance of cardinality-based analyses by providing a rich set
of polynomial-time logical operations, including computing the number of all valid configurations for the analyses in linear time [Dar01a]. Exploiting the structure of d-DNNFs with appropriate algorithms can save time for a large variety of applications [SNB+21, Sun20].

Furthermore, there exist numerous tools for building d-DNNFs from CNFs [Dar04, MMBH10, LM17] which can also solve #SAT problems but the research on the exploitation of d-DNNFs is limited. There is only one tool\(^1\) that is able to perform a few different types of queries given a d-DNNF but that d-DNNF reasoner is not optimized for cardinality-based feature-model analyses.

**Goal of this Thesis**

The goal of the thesis is to develop analyses and corresponding algorithms which form a reasoning engine. Specifically, we target the cardinality of features and partial configurations and we intend to decrease the overall expenditure of time for both problem settings by using d-DNNFs.

First, we determine the state-of-the-art solutions for our analyses. Second, we create a parser for the output of the d-DNNF compilers and a data structure to store the formula. Third, we create algorithms that can compute the cardinality of features and partial configurations and we base them upon our structure. Finally, we evaluate the performance of the algorithms with an empirical evaluation of industrial feature models, including a comparison to state-of-the-art tools.

**Structure of the Thesis**

In Chapter 2, we provide the needed background to understand the following work. In Chapter 3, we give a brief survey of cardinality-based feature-model analyses and provide our optimizations regarding a data structure and improved algorithms. In Chapter 4, we explain the parsing procedure and the implementation of our data structure that represents the d-DNNF as well as our algorithms to exploit d-DNNFs. In Chapter 5, we evaluate our algorithms in different experiments to determine the advantages of our implementation. Additionally, we aim to answer our research questions and discuss potential threats to validity. We present the related work in Chapter 6, provide a conclusion, and discuss possible future work in Chapter 7.

\(^1\)http://www.cril.univ-artois.fr/kc/d-DNNF-reasoner.html
2. Background

In this chapter, we provide the required information to understand the following chapters. First, we explain SAT and the propositional formulas that are needed for the computation of cardinality-based analyses. Second, we differentiate between SAT and \#SAT. Third, we describe feature models. Finally, we give an overview of some analyses regarding feature models.

2.1 SAT

The boolean satisfiability problem (SAT) refers to the satisfiability of propositional formulas. Every propositional formula $F$ holds a finite countable set of atomic propositional variables $\text{vars}(F)$. Those variables are a propositional formula on their own. We represent variables with capitalized letters of the alphabet or as $\{X_i | i \in \mathbb{N}\}$. In propositional formulas, each separate occurrence of a variable is called a literal. They can be connected with the logical operators \text{negation}($\neg$), \text{conjunction}($\land$), \text{disjunction}($\lor$) and form a propositional formula [BSRC]. There exist further logical operators but they can all be transformed into a combination of the named three operators. Besides that, parentheses are used to adjust binding strength between the different operators. The binding strength from strongest to lowest is parentheses, negation, conjunction, and disjunction. We omit parentheses that do not change the semantics. Each variable can be assigned one of the truth values $\{\top, \bot\}$ with $\top$ as true and $\bot$ as false [KZK]. Furthermore, each propositional formula can have different assignments $\mathcal{A}$. $\mathcal{A}(F)$ maps the truth values $\{\top, \bot\}$ to a (sub)set of the propositional variables. Some variables might be undefined in the assignment $\mathcal{A}$ and therefore, can still obtain both truth values. If the mentioned subset is actually the whole set of variables $\text{vars}(F)$, then it is called a full assignment and otherwise a partial assignment [KZK]. The number of assigned variables in a partial assignment is $|\mathcal{A}|$. If the formula on the one hand is always true $\forall$ assignments $\mathcal{A} \models \mathcal{A}(F) = \top$ then $F$ is called a tautology. On the other hand, if $\forall$ assignments $\mathcal{A} \models \mathcal{A}(F) = \bot$ then the formula is called a contradiction.
There are lots of different formats for propositional formulas in the literature [DMo2, Dar10a] and we present the important formats, that are required for our use case, in the following sections.

**Negation Normal Form**

A propositional formula in negation normal form (NNF) is a rooted, directed acyclic graph (DAG). The root as well as all inner nodes are either a conjunction or a disjunction and have at least one child. Negation is only allowed right before a leaf node. The leaf nodes are either \( \top \), \( \bot \), \( A \) or \( \neg A \) [DMo2]. All propositional formulas can be translated to an NNF with the de Morgan Rules: 
\[
\neg (A \land B) = \neg A \lor \neg B \quad \text{and} \quad \neg (A \lor B) = \neg A \land \neg B
\]
[HR04]. Figure 2.1 shows the DAG on the left side for \( \neg (\neg A \land B) \land (C \lor \neg B) \) which is not in NNF because of the first negation (red border). On the right side, the transformed formula \( (A \lor \neg B) \land (C \lor \neg B) \) is in NNF and semantically equivalent to the formula on the left. Here, we also used the double negation \( \neg \neg A \equiv A \).

![Figure 2.1: Propositional Formula Being Transformed into NNF](image)

**Conjunctive Normal Form**

The Conjunctive Normal Form (CNF) is a conjunction of disjunctions and also fulfills the requirements for the NNF as presented in Figure 2.2. The root node has to be a conjunction and all children \( C_1 \ldots C_n \) have to be disjunctions. Moreover, the children \( D_1 \ldots D_n \) have to be literals over the set of variables.

![Figure 2.2: Propositional Formula in CNF as DAG](image)

**2.1.1 Decomposable Deterministic Negation Normal Form**

The decomposable deterministic negation normal form (d-DNNF) is based on the NNF with the additional requirements decomposable and deterministic [DMo2].
A formula in $d$-DNNF is a highly traceable representation that allows many operations in polynomial and linear time [Dar01a]. In theory, every propositional formula can be transformed into a $d$-DNNF [LM17, MMBH10, Dar01a].

**Definition 2.1. Decomposability:** An NNF is decomposable iff the children $C_1, \ldots, C_n$ of each conjunction in the propositional formula do not share variables i.e. $\forall i, j, i \neq j | C_i \cap C_j = \emptyset$ [Dar01a, DM02].

**Figure 2.3:** Transformation into a DNNF

\[
(A \lor \neg B) \land (C \lor \neg B) \\
(Distributiv) \equiv ((A \lor \neg B) \land C) \lor ((A \lor \neg B) \land \neg B) \\
(Absorption) \equiv (A \lor \neg B) \land C \lor \neg B
\]  

**Equation 2.1**

**Figure 2.3** shows our NNF on the left side and the semantically equivalent NNF on the right that is decomposable. The two subtrees of the conjunction, which are also their children, hold the variable $B$ which the definition 2.1. **Equation 2.1** shows one possibility to follow the requirement for decomposability.

**Definition 2.2. Determinism:** An NNF is deterministic iff the children $D_1, \ldots, D_n$ of each disjunction in the propositional formula do not share solutions i.e. $\forall i, j \cap i \neq j | D_i \land D_j \equiv \bot$ [DM02].

**Figure 2.4:** Transformation into a $d$-DNNF
\[ C \land (A \lor \neg B) \lor \neg B \]
\[ (\text{Distributiv}) \equiv C \land A \lor C \land \neg B \lor \neg B \]
\[ (\text{Absorption}) \equiv C \land A \lor \neg B \]
\[ (\text{Identity}) \equiv C \land A \lor \top \lor \neg B \]
\[ (\text{Idempotent}) \equiv C \land A \land (B \lor \neg B) \land \neg B \]
\[ (\text{Distributiv, Commutativ}) \equiv A \land B \land C \lor A \land \neg B \land C \lor \neg B \]
\[ (\text{Absorption}) \equiv A \land B \land C \lor \neg B \]

Our NNF on the left side of Figure 2.4 is already decomposable and on the right side we transform the formula to add the deterministic property. Here we have to change both disjunctions in the NNF because \( A \land \neg B \equiv 1 \) for \( A = 1, B = 0 \) and \( C \land (A \lor \neg B) \land \neg B \equiv 1 \) for \( A = 1, B = 0, C = 1 \). We describe the transformation in Equation 2.2 and we are able to enforce determinism for both disjunctions with one transformation. Further, we easily see that the formula is still decomposable.

**Definition 2.3.** Smoothness: An NNF is smooth iff the children \( D_1, \ldots, D_n \) of each disjunction in the propositional formula contain the same variables i.e. \( \forall i, j \) \( \text{vars}(D_i) = \text{vars}(D_j) \) [DM02].

![Logical Equivalences](image)

**Figure 2.5:** Add Smoothness to the d-DNNF

\[ A \land B \land C \lor \neg B \]
\[ (\text{Identity}) \equiv A \land B \land C \lor \neg B \land \top \land \top \]
\[ (\text{Idempotent}) \equiv A \land B \land C \lor \neg B \land (A \lor \neg A) \land (C \lor \neg C) \]

In Figure 2.5, we transform the formula into a smooth d-DNNF. The left subtree of the root disjunction holds the propositional variables \( A \) and \( C \) which do not occur in the right subtree. Therefore, we have to add those to the right side and still maintain the established properties. Equation 2.3 shows the steps of the transformation.
Definition 2.4. Decision nodes: A decision node is a true, false or an or node with the form in Figure 2.6 and α, β as other decision nodes. B is called the decision variable. Moreover, decision nodes are deterministic by definition [DMo2, Daro4].

There is no decision node in our smooth d-DNNF in Figure 2.5. The root node misses the further decision nodes α, β, and the other disjunctions do not even hold conjunctions as children. Regardless of that, decision nodes occur in larger d-DNNFs that we create with appropriate compiler [Dar04, MMBH10].

2.1.2 Solving SAT Problems

The CNF is the most common format for SAT solvers [MWC09, TBW04]. Solvers often use the Davis-Putnam-Logemann-Loveland (DPLL) algorithm for SAT solving [TBW04, Liboo]. Algorithm 1 describes a simplified version of the DPLL algorithm. It starts off with a partial assignment that has no value assigned. It takes one unassigned variable, assigns it to ⊤ and checks $A(F) = ⊤$. If that is not the case, it continues with the other variables till $A(F) = ⊤$ or $A(F) = ⊥$. If the assignment evaluates to ⊥ then the algorithm backtracks and assigns the last variable $X_{last} = ⊥$. This procedure goes on until either an assignment $A(F) = ⊤$ is found or there are no more possible backtracking steps. Hence, the algorithm terminates [HDo5].

Algorithm 1 DPLL for SAT Solving. Adapted From [HDo5]

Require: Propositional formula $F$ and an assignment $A$.

1: function satisfiable($F$, $A$)
2:   if $A(F) = ⊤$ then
3:     return ⊤
4:   end if
5:   if $A(F) = ⊥$ then
6:     return ⊥
7:   end if
8:   $X_{next} = \text{next\_unassigned\_variable}()$
9:   return satisfiable($F$, $A | X_{next} = ⊤$) $\lor$ satisfiable($F$, $A | X_{next} = ⊥$)
10: end function
Complexity

SAT is an NP-complete Problem [GPFW96]. Hence, a solution for SAT can be verified in polynomial time and there is no deterministic algorithm that finds a solution in polynomial time under the assumption that $N \neq NP$ [Joh92, Wel82]. Nonetheless, there are heuristics and ways to solve large formulas [GPFW96].

2.2 \#SAT

In contrast to SAT solving which checks whether there is an assignment $A(F) = \top$, \#SAT computes the cardinality of full assignments that evaluate to $\top$ [KZK10, GSS06]. For a propositional formula $F$, all possible assignments are called $A_F$ and the cardinality of $F$ is: $\#F = |\{A \in A_F | A(F) = \top\}|$.

2.2.1 Solving \#SAT Problems

One option to solve \#SAT problems is to use an adaptation of Algorithm 1 which we describe in Algorithm 2. Instead of stopping at the first viable assignment $A \in A_F$, all $2^{\text{vars}(F)}$ assignments can be tested. After the first branch that the algorithm finds with $A(F) = \top$, we can add $2^{\text{vars}(F) - |A|}$ to the total count because each unassigned variable can be assigned $\top$ or $\bot$. On account of the backtracking of the formula and the distinction of assignments, all existing branches can be checked. We sum up all counts and finally get the solution for $\#F$. An adapted version of DPLL is often used for \#SAT solvers [SBB+04, Thu06, BSB15] because they perform best among the available \#SAT solvers [Sun20].

Algorithm 2 DPLL for \#SAT Solving [Sun20]

Require: Propositional formula $F$ and an assignment $A$.
1: function count_dpll($F$, $A$)
2: if $A(F) = \top$ then
3:   return $2^{\text{vars}(F) - |A|}$
4: end if
5: if $A(F) = \bot$ then
6:   return 0
7: end if
8: $X_{next} = \text{next_unassigned_variable}()$
9: return count_dpll($F$, $A | X_{next} = \top$) + count_d pll($F$, $A | X_{next} = \bot$)
10: end function

Another approach, the one we focus on in this thesis, is using a d-DNNF to compute the cardinality of satisfying assignments in polynomial time, exploiting the earlier defined properties determinism, decomposability, and smoothness [DMO02]. The combination of those properties allows us to recursively compute values for each node in the d-DNNF and the root node contains the result of the cardinality [BBH+09]. Every variable gets assigned the value 1. By exploiting determinism and decomposability, the count of a disjunction $D$ is the sum over the counts of its children $D_1, \ldots, D_n$, i.e $\#D = \sum_i \#D_i$ [BBH+09]. Similarly, the count of a conjunction $C$ is the product over the counts of its children.
Feature Models

Feature models are used to specify configurations of a product line. Typically, a member consists of multiple features that can be reused. A feature diagram is a tree structure of features and additional cross-tree constraints and can represent a feature model [Bato]. Figure 2.7 shows an example of a feature model. The model we use is adapted from one example of FeatureIDE [MTS17] which is a tool for feature-oriented software development.

Definition 2.5. Feature Model: A feature model \( FM = (FEATS, CONST) \) is a tuple with \( FEATS \) as the set of features that the model contains and a set of cross-tree constraints \( CONST \) [SNB21]. The features are in hierarchical tree order.

In the given example, we can see that each sandwich needs exactly one kind of bread. Additionally, we have the optional additions Cheese, Meat and Vegetables. For instance, if we choose to add Vegetables to our sandwich we have once again the choice between Cucumber, Tomatoes or neither of them. Lettuce is a mandatory choice. There are the following types of hierarchical relations/constraints in a feature model.

- **Mandatory Feature**: A feature \( f \) has to be selected if its parent is already selected [BSR10]. In our example, Bread is a mandatory feature.
Optional Feature: A feature $f$ can be selected if its parent is already selected [BSRC10]. Vegetables are an optional feature in a sandwich.

And Group: Every child of an and group is either mandatory or optional.

Or Group: At least one children of an or group has to be selected if their parent is already selected [BSRC10]. For instance, we have to choose at least one of the different meats Salami and Ham if we select Meat.

Alternative Group: Exactly one child of an alternative group has to be selected if their parent is already selected [BSRC10]. For our Bread we have the choice between Full Grain, Toast and Flatbread.

Besides the hierarchical relations, the feature model also contains cross-tree constraints. In our case, we have the constraints Full Grain $\Rightarrow$ Gouda and Flatbread $\Rightarrow$ Cucumber. For example, every sandwich with Full Grain also requires Gouda. The combination of feature relations and cross-tree constraints specifies the set of valid configurations.

Definition 2.6. Configuration: A configuration is a three tuple $C = (FM, I, E)$ with $I$ as the set of included features and $E$ as the set of excluded features. A feature can not be included and excluded, i.e. $I \cap E = \emptyset$. Furthermore, if each feature $f \in I \lor f \in E$ then $C$ is called total and otherwise is called partial [BSRC10].
Definition 2.7. Valid Configuration: A total configuration \( C = (FM, I, E) \) is valid if it fulfills the following properties [BSRC10].

- The root of the feature model is always included in a configuration.
- If a feature is included, then its parent also has to be included.
- The mandatory children of a feature \( f \) have to be included in a configuration iff \( f \) is included. For an and group that are all children, for an or group at least one child and for an alternative group exactly one child.
- No cross-tree constraints are violated.

For large feature models with a high number of cross-tree constraints, checking whether a configuration is valid, is time-consuming and error-prone if performed manually [Bat05]. Hence, automated support is needed. The common approach is to translate the feature model into the corresponding propositional formula. From there on, the formula can be transformed into a d-DNNF and subsequently used for different analyses [DMO2].

2.4 Feature-Model Analyses

In this section, we present the common analyses of feature models. In the following, we use cardinalities as a synonym for the number of valid configurations. Therefore, cardinality-based analyses are \#SAT problems. Analyses that assign properties are SAT problems.

Definition 2.8. Void Feature Model: A feature model \( FM \) is void iff no valid configuration can be derived [BSRC10].

A void feature model is often the result of wrongly used cross-tree constraints [BSRC10]. The presented feature model is not void considering that a sandwich that only contains Toast and Gouda is an example that fulfills all properties mentioned in definition 2.7 for a valid configuration.

Definition 2.9. Core Feature: A feature \( f \in FEATS \) is a core feature iff it has to be included in every valid configuration \( C \) [BSRC10].

For our sandwich we have to include Bread because the root has to be included in every valid configuration and Bread is a mandatory child.

Definition 2.10. False-Optional Feature: A feature \( f \in FEATS \) is false optional iff it has to be included in every valid configuration its parent is included, although it is not labeled as mandatory [BSRC10].

False-optional features occur due to cross-tree constraints. If a mandatory feature implies an optional one, then the optional feature becomes mandatory. Hence, the feature is a false-optional one. In our example, Gouda becomes a false-optional feature because Bread is a core feature and implies Gouda. Gouda is also a core feature because of the cross-tree constraint and Bread being a core feature.
**Definition 2.11.** Dead Feature: A feature \( f \in \text{FEATS} \) is a dead feature iff it can not be included in any valid configuration \( C \) [BSRC10].

Dead features can be the result of the wrong usage of cross-tree constraints. If a mandatory feature implies a feature of an alternative group, then all the other features in the group are dead. For instance, due to the alternative group for Cheese and the fact that Gouda is a false-optional feature, we can not include Cheddar without violating the rules for an alternative group. Consequently, Cheddar is a dead feature.

**Definition 2.12.** Atomic Set: Two features \( f, g \) are part of an atomic set iff \( f \) being included in a configuration implies the inclusion of \( g \) and vice versa [BSRC10].

If a parent node has mandatory children, then they form an atomic set. After finding an atomic set, it can be viewed as a unit for other analyses. For our Sandwich, Bread and Gouda are part of an atomic set as a consequence of being core features. That directly results in the fact that there can not be any valid configuration that includes one feature and excludes the other.

**Definition 2.13.** Cardinality of Feature Model: The cardinality of a feature model \( \#FM \) is the cardinality of the set of configurations that fulfill all properties defined in definition 2.7.

**Definition 2.14.** Cardinality of Feature: The Cardinality of feature \( \#FM_f \) with \( f \) as our selected feature is the number of valid configurations that include \( f \).

**Definition 2.15.** Commonality: Commonality of a feature \( f \) is the relative share of valid configurations that include the feature \( f \) [FAHCC14, BRCT05, TBC06]. Hence, the commonality of \( f \) describes the probability of \( f \) appearing in a valid configuration \( C \) [BRCT05].

Equation 2.4 describes the formula to calculate the commonality of a feature \( f \). \( \#FM \) can be equal to 0 (i.e. \( FM \) is void). In this case, we consider the commonality to be 0 for every feature.

\[
\text{Commonality}(FM, f) = \begin{cases} 
\frac{\#FM_f}{\#FM}, & \text{if } \#FM \neq 0 \\
0, & \text{if } \#FM = 0 
\end{cases} \quad (2.4)
\]

Moreover, we can identify core and dead features using commonality using Equation 2.5 respectively Equation 2.6.

**Equation 2.5**

\[
\text{Commonality}(FM, f) = 1 \iff f \text{ is a core feature} \quad (2.5)
\]

**Equation 2.6**

\[
\text{Commonality}(FM, f) = 0 \iff f \text{ is a dead feature} \quad (2.6)
\]

**Definition 2.16.** Cardinality of Partial Configuration: The Cardinality of a partial configuration \( \#FM_C \) describes the number of valid configurations that include all features \( i \in \text{Include} \) and exclude all features \( e \in \text{Exclude} \) [BSRC10].

Those definitions are examples of basic analyses that can be performed on feature models. We discuss further analyses that are based on cardinalities in Chapter 3.
3. Feature-Model Analyses with d-DNNFs

In this chapter, we discuss the different cardinality-based analyses and give solutions with linear time complexity using d-DNNFs. Research has shown that there are many applications of #SAT for feature models and all are based on only three different types of queries, namely cardinality of feature model, features, and partial configurations [Sun20, SNB+21, BSRC10, PM16]. We propose solutions for all three of them in the following.

First, we discuss the transformation of CNFs to d-DNNFs and explain how to depict d-DNNFs as adjacency lists. Second, we provide an optimized algorithm to compute each of the cardinalities. Third, we discuss further optimizations that are based on core and dead features.

3.1 Transforming CNF into d-DNNF

For simplicity reasons, the approaches we present are using a smooth d-DNNF. With some smaller changes, the algorithms can be adjusted to also work on non-smooth d-DNNFs. We describe the transformation from a feature model into a semantically equivalent smooth d-DNNF in Algorithm 4. We start by converting the feature model into the corresponding CNF and we then use one of the different off-the-shelf compilers to create a smooth d-DNNF out of our CNF [Dar04, MMBH10].

Algorithm 4 Transformation from a Feature Model into a Smooth d-DNNF

Require: The feature model FM.

1: function fm_into_d-dnnf(FM)
2:     CNF_{FM} = convert_to_cnf(FM)
3:     d-DNNF_{FM} = convert_to_smooth_d-dnnf(CNF_{FM})
4:     return d-DNNF_{FM}
5: end function
Instead of the representation of a d-DNNF as a graph that holds different nodes which hold references to other nodes, we use a special adjacency list that serves as a way to depict the d-DNNF. Figure 3.1 shows the corresponding adjacency list for our smooth d-DNNF example from Figure 2.5 in Chapter 2. A node itself contains all the necessary information for the algorithms, like for instance the kind of the node. Moreover, each node holds two lists, one for the children and one for the parents of the node. Additionally, we require the nodes to be sorted so that child nodes are listed before their parent nodes. For Instance, a post-order traversal of that directed acyclic graph like the numbering in our example would fulfill this requirement. Besides that, the last node in the list is our root node. We explain the construction of the graph as our adjacency list in Chapter 4.

This precondition allows an iterative computation for all the following use cases which is faster than a recursive solution. With a recursive approach, we have to look at nodes multiple times due to the graph structure. We aim to compute a node only once and therefore we have to mark the node after the computation, check beforehand if it is already computed, and also remove the markers for following queries after the computation of the whole graph. For example, we can visit node 2 in Figure 3.1 with different parent nodes and because of that, we have to visit node 2 one time for each parent in a single traversal. The resulting overhead may seem negligible, considering our toy example but it has a significant impact on industrial-sized feature models. An iterative approach with our adjacency list avoids that overhead.

### 3.2 Cardinality of a Feature Model

The basic query for cardinality-based analyses of a feature model is to compute the cardinality of a feature model with no further restrictions. We refer to the
number of valid configurations of a feature model as $\#FM$. $\#SAT$ solver can compute the $\#FM$ without any adaptions using the corresponding CNF. We also present a recursive way to compute $\#FM$ with d-DNNFs in Algorithm 3.

Algorithm 5 computes the cardinality of one node. We use that algorithm to set the focus on the specific aspects of the analysis and to reduce the size of the following algorithms. We simply distinguish between the different types of nodes for the computation of their count. For AND nodes, it is the product and for OR nodes it is the sum of the counts of their children. We assign both kinds of Literal and True nodes 1 as their count and False nodes 0 [BBH+09].

Algorithm 5 Cardinality of a Node

Require: A node of the d-DNNF with its children, feature number if existing for that type.

1: function node_count(node)
2:  if node.node_type == AND then
3:       node.count = $\prod_{child \in node.children} child.count$
4:  else if node.node_type == OR then
5:       node.count = $\sum_{child \in node.children} child.count$
6:  else if node.node_type == PositiveLiteral
7:  or node.node_type == NegativeLiteral
8:  or node.node_type == True then
9:       node.count = 1
10:  else if node.node_type == False then
11:       node.count = 0
12:  end if
13: end function

For $\#FM$ we just have to go through all of our nodes in the adjacency list and compute the node count without further adaptations using Algorithm 5. We describe that procedure in Algorithm 6. We store the final result in the root node, which is also the last node in our adjacency list. Besides that, we cache all count values for our nodes so we can use them again in the other algorithms.

Algorithm 6 Cardinality of a Feature Model

Require: The nodes of the d-DNNF as our adjacency list.

1: function model_count(nodes)
2:   for all node ∈ nodes do
3:     node_count(node)
4:   end for
5:   return nodes.get_last().count
6: end function

3.3 Cardinality of Features

The Cardinality of feature $\#FM_f$ with $f$ as our selected feature, is the number of valid configurations that include $f$. Algorithm 7 describes our naive method to
compute \( \#FM_f \). We go through our list and compute the node count for each node like in Algorithm 6 except for the negative occurrences of \( f \). The count of negative literals with the feature number of \( f \) is set to 0 because there are no valid configurations that include the negated form of the feature we include. Although we call it the naive method, this approach already contains all the benefits the adjacency list provides and, as a result of that, is an improvement compared to a recursive solution.

**Algorithm 7** Cardinality of Feature (Naive)

<table>
<thead>
<tr>
<th>Require: The d-DNNF as adjacency list, the number of the feature we include.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: function card_of_feature(nodes, include_f)</td>
</tr>
<tr>
<td>2:   for all node ∈ nodes do</td>
</tr>
<tr>
<td>3:     if node.node_type == NegativeLiteral And node.feature == include_f then</td>
</tr>
<tr>
<td>4:       node.count = 0</td>
</tr>
<tr>
<td>5:   else</td>
</tr>
<tr>
<td>6:     node.count = node_count(node)</td>
</tr>
<tr>
<td>7:   end if</td>
</tr>
<tr>
<td>8: end for</td>
</tr>
<tr>
<td>9: return nodes.get_last().count</td>
</tr>
<tr>
<td>10: end function</td>
</tr>
</tbody>
</table>

We can further improve the naive algorithm by identifying the nodes that change compared to \( \#FM \) by marking them. Therefore, we only have to compute the count of the marked nodes because we store the other values, which stay the same, from a previous call of Algorithm 6. We change the count of the negative literals corresponding to our selected feature \( f \) to 0 like in our naive version. Instead of calculating the count for each node again, although most stay the same, we just recursively mark the parent nodes of the negative literal containing the included feature number, until we reach the root in the advanced Algorithm 8. Hence, we can reduce the effort and improve performance-wise by only computing the nodes which are on at least one of the relevant paths. For the cardinality of features, we now just have to call Algorithm 8 for each feature \( f \) in \( FEATS \).

**Commonality**

We can compute the commonality \( \text{Commonality}(FM, f) \) of a feature \( f \) by computing the cardinality of our feature model \( \#FM \), the cardinality of feature and using Equation 2.4 which we present in Chapter 2. We describe the procedure in Algorithm 9. First, we need to compute \( \#FM \) so we can get our cached values. Second, we compute \( \#FM_f \). Finally, we simply insert the values in the formula and return the commonality.

**3.4 Cardinality of Partial Configurations**

The Cardinality of a feature is limited to one feature that is included, but for some use cases, it is necessary to consider a sub-set of a configuration
Algorithm 8 Cardinality of Feature (Upwards Propagation)

Require: The d-DNNF as adjacency list, the number of the feature we include.
1: function card_of_feature_up(nodes, include_f)
2:   mark(nodes.get_negative_literal(include_f))
3:   card_of_feature_up_h(nodes, include_f)
4: end function

5: function mark(node)
6:   node.marked = true
7:   for all parent ∈ node.parents do
8:     mark(parent)
9:   end for
10: end function

11: function card_of_feature_up_h(nodes, include_f)
12:   for all node ∈ nodes do
13:     if node.marked then
14:       if node.node_type == NegativeLiteral
15:         And node.feature == include_f then
16:           node.count = 0
17:         else
18:           node_count(node)
19:         end if
20:     end if
21:   end for
22: return nodes.get_last().count
23: end function

Algorithm 9 Commonality

Require: The d-DNNF as adjacency list, the feature to compute the commonality for.
1: function commonality(nodes, feature)
2:   #FM = model_count(nodes)
3:   #FM_f = card_of_feature_up(nodes, feature)
4:   if #FM ≠ 0 then
5:     #FM_f
6:     #FM
7: else if #FM = 0 then
8:     return 0
9: end if
10: end function
$C = (FM, I, E)$ [BSRC10]. $\#FM_C$ describes the number of valid configurations that include all features $i \in Include$ and exclude all features $e \in Exclude$. For computing the cardinality of partial configurations, we extend Algorithm 7. We set feature numbers we include and occur as NegativeLiteral, as well as feature numbers we exclude and occur as PositiveLiteral, to 0. The result of that adaptation is shown in Algorithm 10.

**Algorithm 10** Cardinality of Partial Configurations That Includes and Excludes Features (Naive)

**Require:** The d-DNNF as adjacency list, the numbers of the features that we include and those that we exclude.

1. function card_of_part_conf(nodes, include_fs, exclude_fs)
2. for all node ∈ nodes do
3. if node.node_type == NegativeLiteral
4. And include_fs.contains(node.feature) then
5. node.count = 0
6. else if node.node_type == PositiveLiteral
7. And exclude_fs.contains(node.feature) then
8. node.count = 0
9. else
10. node_count(node)
11. end if
12. end for
13. return nodes.get_last().count
14. end function

Besides that naive version of that algorithm, we can adapt the upwards propagation of Algorithm 8 for partial configurations. We describe that procedure in Algorithm 11. We now mark the paths of all NegativeLiterals that correspond to the features we include and all PositiveLiterals corresponding to the features we exclude. After that, we compute the different node counts, as described earlier in our naive version, for all the marked nodes.

When using Algorithm 11 for queries with lots of features that are included and excluded, the added effort for pathfinding and the fact that there is a higher coverage of the graph may cause a computational cost that exceeds the naive solution. Because of that, we need to find a threshold, depending on the number of included/excluded features, for the switch between the naive and the upwards propagation algorithm.

### 3.5 Core and Dead Features

We can use core and dead features not only for different applications, we can also avoid unnecessary computations in the previous cardinality-based analyses [Sun20]. In Chapter 2, we present that if a feature $f$ is core then the cardinality of feature $f$ is equal to the number of valid configurations (i.e., $\#FM_f = \#FM$). If a feature $f$ is dead then the cardinality of feature is equal to zero.
Algorithm 11 Cardinality of Features (Upwards Propagation)

**Require:** The d-DNNF as adjacency list, the numbers of the features that we include and those that we exclude.

```plaintext
1: function card_of_features_up(nodes, include_fs, exclude_fs)
2:     for all feature ∈ include_fs do
3:         mark(nodes.get_negative_literal(feature))
4:     end for
5:     for all feature ∈ exclude_fs do
6:         mark(nodes.get_positive_literal(feature))
7:     end for
8:     card_of_feature_up_h(nodes, include_fs, exclude_fs)
9: end function

10: function mark(node)
11:     node.marked = true
12:     for all parent ∈ node.parents do
13:         mark(parent)
14:     end for
15: end function

16: function card_of_feature_up_h(nodes, include_fs, exclude_fs)
17:     for all node ∈ nodes do
18:         if node.marked then
19:             if node.node_type == NegativeLiteral
20:                 And include_fs.contains(node.feature) then
21:                 node.count = 0
22:             else if node.node_type == PositiveLiteral
23:                 And exclude_fs.contains(node.feature) then
24:                 node.count = 0
25:             else
26:                 node_count(node)
27:             end if
28:         end if
29:     end for
30:     return nodes.get_last().count
31: end function
```
(i.e., $\#FM_f = \emptyset$). This conclusion is valid for both directions and based on Equation 2.5 as well as Equation 2.6.

Consequently, if we know beforehand that a feature $f$ is core, respectively dead, then we do not have to compute the cardinality of a feature for $f$. Additionally, every included core feature respectively excluded dead feature can be ignored in a partial configuration or set to $\#FM$ if all included features are core and all excluded features are dead. Further, if we include at least one dead feature or exclude at least one core feature or both, we can conclude that $\#FM_C = 0$ applies. This is a consequence of our definitions of core and dead features in Chapter 2. Algorithm 12 shows that fact in an algorithmic environment for the cardinality of partial configurations.

**Algorithm 12** Cardinality of Partial Configurations with Reduction

**Require:** The d-DNNF as adjacency list, the features we include and exclude.

1. **function** card_of_part_conf_reduced(nodes, include_fs, exclude_fs)
2.  
3.   **for all** $i \in$ include_fs **do**
4.   
5.     **if** is_dead(nodes, $i$) **then**
6.     
7.       **return** 0
8.     
9.     **else if** is_core(nodes, $i$) **then**
10.    
11.       include_fs.remove($i$)
12.    
13.   **end if**
14.  
15. **end for**
16.  
17. **for all** $e \in$ exclude_fs **do**
18.  
19.     **if** is_core(nodes, $e$) **then**
20.    
21.       **return** 0
22.    
23.     **else if** is_dead(nodes, $e$) **then**
24.    
25.       exclude_fs.remove($e$)
26.    
27.   **end if**
28.  
29. **end for**
30.  
31. **if** include_fs.is_empty() AND exclude_fs.is_empty() **then**
32.  
33.   **return** $\#FM$
34.  
35. **end if**
36.  
37. **return** card_of_part_conf(nodes, include_fs, exclude_fs)
38. **end function**

If we do not have the information about the features beforehand, we can compute them using d-DNNFs without much effort. Algorithm 13 depicts how to identify a feature as core and Algorithm 14 how to identify a feature as dead. We can simply classify features as core if only positive literals, respectively as dead if only negative literals corresponding to the feature appear in the smooth d-DNNF [Sun20]. Along with that, a feature is core if the corresponding negative literal, respectively dead if the positive literal corresponding to the feature is conjuncted with a False node and because of that, can not change the cardinality. CNF to d-DNNF compiler may use those contradictions to make the d-DNNF smooth. We can check whether there exists a literal or not and where to find it if
it exists with the help of some form of a lookup table. With our two approaches, we can find all core and dead features.

Algorithm 13 Check whether the Feature is Core

Require: The d-DNNF as adjacency list, the feature.

1: function is_core(nodes, feature)
2:     if nodes.contains_negative_literal(feature) then
3:         return false
4:     end if
5:     parent = nodes.get_negative_literal(feature).parents
6:     if parent.node_type == And then
7:         for all child ∈ parent.children do
8:             if child.node_type == False then
9:                 return true
10:         end for
11:     end if
12:     return false
13: end function

Algorithm 14 Check whether the Feature is Dead

Require: The d-DNNF as adjacency list, the feature.

1: function is_dead(nodes, feature)
2:     if nodes.contains_positive_literal(feature) then
3:         return false
4:     end if
5:     parent = nodes.get_positive_literal(feature).parents
6:     if parent.node_type == And then
7:         for all child ∈ parent.children do
8:             if child.node_type == False then
9:                 return true
10:         end for
11:     end if
12:     return false
13: end function

3.6 Summary

In this chapter, we considered three types of cardinalities: cardinality of a feature model, features, and partial configurations. We provided the representation of d-DNNFs as an adjacency list and described its advantages regarding the computation of the three types of cardinalities. For the cardinality of partial configurations and features, we propose a marking algorithm that aims to reduce the number of nodes we have to traverse. Additionally, we extended the
efficient identification of core and dead features as well as how to exploit that knowledge to further improve the runtime for the computation of cardinality-based feature model analyses. The described algorithms build the base for our implementation in Chapter 4 and empirical evaluation in Chapter 5.
4. Implementation

In this chapter, we explain the implementation for the algorithms, we describe in Chapter 3. We use Rust [MK14] as the programming language for the implementation. We provide an implementation for the cardinality of a feature model, features, and partial configurations. Additionally, we also give a possible implementation for the optimization based on core and dead features. We combine all those implementations in our d-DNNF reasoning engine dknife. That chapter enables the reader to get a deeper understanding of design choices and allows other people to re-implement our algorithms more easily.

First, we explain the parsing rules and how we implement them. Second, we describe our data structure and our design choices. Finally, we show the code implementation for the cardinality-based algorithms and give code-specific optimizations in Rust.

4.1 Parsing

Darwiche et al. introduced a format, similar to DIMACS, to store d-DNNFs in the c2d compiler project1 [Dar04]. This format is also used by dsharp [MMBH10] and d4 [LM17]. Hence, we can use the output of all three compilers without further adaptions. We present the format in the following.

The first line of the d-DNNF output contains the metadata of the file: \texttt{nnf v e n} where \textit{v} is the number of nodes, \textit{e} the number of edges, and \textit{n} the number of variables in the d-DNNF. After the header, all the different nodes follow in topological order, with children appearing before the parent node. There are no duplicates of nodes in the file. Each node can be either an and node, an or node, or a literal. Moreover, each node has an index that depends on the position in the file, starting with the first node at index 0 and ending with the root node at index \textit{v} – 1.

\footnote{http://reasoning.cs.ucla.edu/c2d/}
• An and node has the format $A \ c \ i_1 \ i_2 \cdots \ i_c$ with $A$ as identifier, $c$ as the number of children and $i_1 \ i_2 \cdots \ i_c$ are the indices of these children. An and node with $c = 0$ ($A\ 0$) represents true.

• An or node has the format $O \ j \ c \ i_1 \ i_2 \cdots \ i_c$ with $O$ as identifier, $c$ as the number of children which is either 2 or 0. If $c = 2$ then there are two child nodes at index $i_1, i_2$ that conflict on $j$ and the node is called a decision node. An or node with $c = 0$ and $j = 0$ ($O \ 0 \ 0$) represents false.

• A literal has the format $L \ [-]j$ with $[-]j$ as the literal corresponding to the node.

We parse the header separately from the nodes and assign the fields number_of_variables to $n$ and number_of_nodes to $v$. We do not use the number of edges.

We create one Node for each of the following lines in the d-DNNF file. For each of the nodes, we assign marker to false and temp to 0. We assign the positions of the child nodes for and, or nodes as given by the d-DNNF. In addition to that, we also add the parent positions to parents for all children. We save the signed variable number of a literal node and the decision variable of or nodes in var_number. Furthermore, we add each literal to our variables hashmap with the var_number as key and the position in the adjacency list as corresponding value. Besides that, we compute the count of each node according to Algorithm 5 while parsing. We use the cached values which we store in count for some of the following listings. Additionally, we compute the cardinalities for all analyses with the help of temp so we can reuse the cached values in count multiple times.

4.2 Data Structure

In this section, we discuss our data structure which stores the d-DNNF, one of the compilers generated. The struct Ddnnf, shown in Figure 4.1, represents the entire d-DNNF with all the nodes, the number of variables, the number of nodes, a HashMap as a lookup table for the node positions of literals, and a vector for temporary storage which is needed later for some algorithm implementations. The vector of nodes is a special adjacency list which is described in Chapter 3 in Figure 3.1.

A node holds different fields that give us the ability to compute the results for the different analyses. The count field stores the count of the node for the cardinality of a feature model. Hence, this field caches the values that we use in the different algorithms. We use the fields temp, marker, and parents to optimize the algorithms for the other cardinality-based analyses. We need the parent connections for the upwards propagation, the marker flag to determine which nodes have to be computed and the temp field to store intermediate results. Furthermore, each node can be one of the NodeType: And, Or, Literal, True, or False. And and Or are the only types of nodes that contain a vector with child nodes. Literals have a signed integer as their variable number that corresponds to a feature. A positive value indicates the inclusion of a feature, a negative
### 4.2. Data Structure

#### Ddnnf

| + nodes: Vec<Node> |
| + variables: HashMap<i32, usize> |
| + number_of_variables: u32 |
| + number_of_nodes: usize |
| + marked_nodes: Vec<usize> |

+ card_of_feature_m(&mut self, feature: &usize): Integer
+ card_of_partial_config(&mut self, features: &[usize]): Integer
+ card_of_partial_config_m(&mut self, features: &[usize]): Integer
+ card_of_features_m(&mut self, file_path_out: &str): void

#### Node

| + marker: bool |
| + count: Integer |
| + temp: Integer |
| + node_type: NodeType |
| + var_number: Option<i32> |
| + parents: Vec<usize> |
| + children: Option<Vec<usize>> |

---

Figure 4.1: UML Diagram for the d-DNNF Data Structure
value the exclusion. Or nodes, which are decision nodes in our case, also hold their decisive variable in the var_number field. That fields are a direct result of the file format.

We choose only one node struct over multiple structs with more individual fields to avoid downcasting and an overhead that would occur by boxing the different structs into one vector by exploiting the trait implementation or by using a wrapper enum. We accept the drawback with slightly higher memory costs due to the unused fields var_number and children by some NodeTypes for the benefit of a small performance increase.

The Ddnnf struct has all the relevant methods we apply for the cardinality-based analyses. We store the cardinality of a feature model in the count field of our root node. The cardinality of feature can be computed with card_of_feature in a naive way or with card_of_feature_m as our alternative with marking and upwards propagation. We can compute the cardinality of partial configurations with card_of_partial_config in a naive way or with card_of_partial_config_m, again with marking and upwards propagation. card_of_features_m can be used to compute the cardinality of features with the marking algorithm. We do not provide an alternative solution, using the naive approach because that version would always deliver worse results. In contrast to the previous algorithms, card_of_features_m does not return the result directly and instead writes the results in a file in .csv format. Each row consists of the selected feature followed by a comma and the cardinality of that feature. We use card_of_file_queries to compute multiple queries at once from a file and write the solutions in another file.

The input file consists of multiple lines with each line being a configuration. A partial configuration consists out of one or multiple feature numbers separated by spaces, with unsigned numbers corresponding to including features and signed numbers to excluded. The output file extends the input file with a semicolon followed by the cardinality of that partial configuration. We depict the implementation of that algorithms in the following sections as listings.

4.3 Algorithm Implementations

In this section, we discuss the implementation of the presented algorithms from Chapter 3 in detail, using Rust. Additionally, we give insights into the actual implementation and optimizations in the programming language we use. We do not cover the cardinality of a feature model, because that is already covered in the parsing process. The root node of our graph, which is the last node in the adjacency list already holds the result for $\#FM$ in count.

4.3.1 Cardinality of Node

Similar to Chapter 3, we first realize the computation of the cardinality of one node and then use that algorithm to compute the different types of queries.
Listing 4.1: Cardinality of a Node (Naive)

Listing 4.1 depicts our implementation of Algorithm 5. That algorithm does not use the cached values and only uses the temp field of the nodes.

In Line 3, we get the node with its position in the adjacency list and match for the type. If it is and node, then we assign temp to the product of the temp values of its child nodes in Lines 4-8. If it is an or node, then we assign temp to the sum of the two child nodes in Lines 9-15. We do not have to compute anything for all the other types of nodes and can simply assign false nodes to 0 in Line 16, literals, true nodes to 1 in Line 17.
We can now improve our implementation by using the cached values which we save in `count`. Listing 4.2 shows our implementation for the cardinality of a node when using one of the algorithms that are based on upwards propagation and marking. This listing only differs from the naive implementation in Line 7-11 and Lines 16-21. In Line 7 respectively Line 16, we check whether the children of the node, we compute the cardinality for, are marked or not. Marked nodes are on at least one of our upwards paths towards the root node and therefore, are already computed because we can make sure to compute the child nodes before their parents. Hence, we choose the `temp` value of the child node in Line 8 respectively Line 17. In contrast, child nodes that are not marked as part of any path do not have to be computed. Consequently, we can just use the cached `count` value from the parsing process.

### 4.3.2 Cardinality of One Feature

We use the upwards propagation algorithm with marking as described in Algorithm 8 for the computation of the cardinality of a feature. We depict the implementation in Listing 4.3. The parameter `feature` in Line 2 shows that the feature number can be positive which is the inclusion of a feature or negative to exclude that feature. We know from Chapter 3 that an included core feature and an excluded dead feature result in $\#FM_f = \#FM$. We check that cases in Line 3 and accordingly return $\#FM$ in Line 4. Furthermore, excluded core and included dead feature result in the cardinality of a feature is equal to zero which is shown in Lines 5-7. If neither of that two cases can be applied we continue in Line 9-13 with the check for the existence of the sign changed feature number. If the node exists, we can proceed with the actual marking algorithm in Line 10. If there is no node corresponding to the feature number with a changed sign, then we can simply return zero in Line 13. Later is an edge case that can only be reached with the wrong user input.
# [inline]

```rust
pub fn card_of_feature_m(&mut self, feature: &i32) -> Integer{
    if self.core.contains(feature) || self.dead.contains(&-feature) {
        self.nodes[self.number_of_nodes-1].count.clone()
    } else if self.dead.contains(feature)
           || self.core.contains(&-feature) {
        Integer::from(0)
    } else {
        match self.literals.get(&-feature).cloned() {
            Some(i) => self.compute_count_marker(vec![i]),
            // there is no literal corresponding to the feature
            // number and we can return zero
            None => Integer::from(0)
        }
    }
}
```

Listing 4.3: Cardinality of Feature with Marking

Before discussing the computation of the cardinality of a feature, we first explain the marking which is used later. We display the marking in Listing 4.4. We start off by marking the current node in Line 3 and adding the position to the list of marked nodes in Line 4. After that in Lines 8-14, we go through the parent nodes of the current node and call this function for each node that is not already marked.

# [inline]

```rust
fn mark_nodes(&mut self, i: usize) {
    self.nodes[i].marker = true;
    self.md.push(i);

    // check for parent nodes and adjust their count resulting
    // of the changes to their children
    for j in self.nodes[i].parents.clone() {
        // only mark those nodes which aren't already marked to
        // avoid marking nodes near the root multiple times
        if !self.nodes[j].marker {
            self.mark_nodes(j);
        }
    }
}
```

Listing 4.4: The Algorithm for Upwards Propagation and Marks

Each analysis that has an implementation of the marking algorithm uses Listing 4.5 for the actual computation after the query was reduced and information
of dead and core feature is applied. We assign the \texttt{temp} value of each node to 0 in Line 5 and start the marking of the nodes in Line 7 which we describe earlier in this subsection. In Line 11, we sort the list of marked nodes, to ensure to compute the child nodes before their parents. This is more efficient than checking all nodes for their marking status due to the rather small amount of marked nodes and the fact that the list is almost sorted because we mark child nodes before the parents. Nevertheless, sorting is necessary as a consequence of the graph structure. After that, we can simply iterate over the marked nodes and compute their \texttt{temp} value with Listing 4.2 in Lines 15-19. Next, we set the markings to false and remove all entries in the list of marked nodes in Lines 22-25 and return the result in Line 29.

```rust
#[inline]
fn compute_count_m(&mut self, index: Vec<usize>) -> Integer{
    for i in index.clone() {
        // change the value of the node
        self.nodes[i].temp.assign(0);
        // go through the path til the root node is marked
        self.mark_nodes(i);
    }
    // sort the marked nodes so that we make sure to first
    // compute the child nodes
    self.md.sort_unstable();

    // compute the count for all marked nodes, respectively
    // all nodes that matter
    for j in 0..self.md.len() {
        if !index.contains(&self.md[j]) {
            self.compute_count_marked_node(self.md[j]);
        }
    }

    // reset everything
    for j in 0..self.md.len() {
        self.nodes[self.md[j]].marker = false
    }
    self.md.clear();

    // the result is propagated through the whole graph
    // up to the root
    self.nodes[self.number_of_nodes - 1].temp.clone()
}
```

Listing 4.5: Core Computations for Cardinality-Based Analyses Using Marking
### 4.3.3 Cardinality of Several Features

To compute the cardinality of features, we actually compute the cardinality for every feature of the feature model $FM$. For optimal performance, we use a thread pool with a queue that holds all the features we want to compute. In the thread pool are four threads that take a feature number, compute the cardinality of a feature, and take the next feature number until the queue is empty. Then, we write our results in a .csv file. Moreover, we use a synchronization flag to enable the possibility that the threads can check whether the queue is empty and if more work gets added. Listing 4.6 shows the implementation of that idea without the actual implementation of the sync flag and the work queue for simplicity reasons.

In Listing 4.6, we begin with initializing the work queue in Line 4, the multiple producer, single consumer channel (MPSCC) in Lines 5-6, the sync flag in Line 10, the thread pool in Line 11 and the CSV writer in Line 14. After that, we fill the queue with our workload in Lines 16-21. Subsequently, we create the threads in Lines 23-47. First, we clone the necessary information for each thread, namely the queue, the MPSCC, the sync flag, and the d-DNNF in Lines 24-27. Second, we spawn the thread in Line 29. Third, the thread checks whether there is another feature to compute, compute the cardinality, and sends the result over the channel to the consumer in Lines 33-42 or explicitly tells the operating system that it can run other threads in Line 43 if there are no more computations in the queue. Finally, we add the thread to the pool in Line 46.

```rust
pub fn card_of_features_m(&mut self, file_path_out: &str)
    -> Result<(), Box<dyn Error>>{
    let queue = WorkQueue::new();
    use std::sync::mpsc::channel;
    let (results_tx, results_rx) = channel();

    // Create a SyncFlag to share whether or not there are more
    // jobs to be done.
    let (mut more_jobs_tx, more_jobs_rx) = new_syncflag(true);
    let mut threads = Vec::new();

    // start the csv writer with the file_path
    let mut wtr = csv::Writer::from_path(file_path_out)?;

    let mut jobs_remaining = 0;
    let mut jobs_total = self.number_of_variables;

    for l in 1..&self.number_of_variables+1 {
        jobs_remaining = queue.add_work(l as i32);
    }

    for thread_num in 0..MAX_WORKER {
```
```rust
let thread_queue = queue.clone();
let thread_results_tx = results_tx.clone();
let thread_more_jobs_rx = more_jobs_rx.clone();
let mut ddnnf: Ddnnf = self.clone();

let handle = thread::spawn(move || {
    let mut work_done = 0;
    while thread_more_jobs_rx.get().unwrap() {
        // If work is available, do that work.
        if let Some(work) = thread_queue.get_work() {
            let result = ddnnf.card_of_feature_m(&work);
            work_done += 1;

            match thread_results_tx.send((work, result)) {
                Ok(_) => (),
                Err(_) => { break; },
            }
        }
        std::thread::yield_now();
    }
    threads.push(handle);
})

while jobs_total > 0 {
    match results_rx.recv() {
        Ok((feature, cardinality)) => {
            wtr.write_record(vec![feature.to_string(), cardinality.to_string()])?
                jobs_total -= 1;
        },
        Err(_) => { panic!("All workers died unexpectedly."); }
    }
    more_jobs_tx.set(false).unwrap();
    for handle in threads {
        handle.join().unwrap();
    }
    wtr.flush()?;
    Ok()
}
```

Listing 4.6: Cardinality of Features with Marking

Lines 49-58 are responsible for the processing of the results which arrive while the queue gets emptied. Here, we just write the feature we computed, together
with the result, as a line into the .csv file. If there is no more work left, we inform the workers and close the threads in Line 59-62. In the end, we make sure to empty the buffer of the CSV writer in Line 64 and return without any errors in Line 65.

### 4.3.4 Cardinality of Partial Configurations

The cardinality of partial configurations is implemented similarly to Listing 4.3 for the cardinality of features. Listing 4.7 depicts the procedure for computing the cardinality. We start off with the check for satisfiability in Lines 4-5 and return 0 if the query is not satisfiable. If it is satisfiable we reduce the query as much as possible in Line 7 by applying our information about dead and core features. The remaining lines are basically the same as in Listing 4.3.

```rust
#[inline]
pub fn card_of_partial_config_m(&mut self, features: &Vec<i32>) -> Integer{
    if self.query_is_not_sat(features) {
        Integer::from(0)
    } else {
        let features: Vec<i32> = self.reduce_query(features);
        let mut index: Vec<usize> = Vec::new();
        for number in features {
            // we add the negative occurrences to be set
            // to 0 for the features we want to include/exclude
            if let Some(i) = self.literals.get(&-number).cloned() {
                index.push(i)
            }
        }
        if index.is_empty() {
            Integer::from(0)
        } else {
            self.compute_count_marker(index)
        }
    }
}
```

Listing 4.7: Cardinality of Partial Configurations Using Marking

### 4.3.5 Core and Dead Features

The identification of core and dead features allows us to reduce the effort of some queries to a minimum. Therefore, we present the identification and the simplification of queries in the following. We already applied the results of the recognition in earlier subsections.

We start with Listing 4.8 which discovers all core features. In Lines 2-3 we create a list, containing all feature numbers of the d-DNNF and start the process of
filtering the core features. On the one hand, if there is no negative literal for a feature number then the corresponding feature is a core feature. We check that in Line 5 and 21. On the other hand if that negative literal is conjuncted with a false node then it has no impact on the results of the different analyses and the feature is also a core feature. That procedure is shown in Lines 10-19. Additionally, we can identify dead features analog by only changing Line 5 and searching for existing positive literals.

```
fn get_core(&mut self) {
    self.core = (1..self.number_of_variables as i32+1).filter(
        |f|
        // match self.literals.get(f) { for dead features
        match self.literals.get(&-f) {
            Some(x) => {
                let par = self.nodes[*x].parents.clone();
                let mut res = false;
                for p in par {
                    if self.nodes[p].node_type == And {
                        let chi = self.nodes[p].children.clone();
                        for c in chi.unwrap(){
                            if self.nodes[c].node_type == False {
                                res = true;
                            }
                        }
                    }
                }
            },
            None => true
        }
        ).collect::<HashSet<i32>>()
}
```

Listing 4.8: Determine All Core Features of the d-DNNF

For partial configurations, we have to check multiple features that can be included or excluded. As a consequence, we have to evaluate every selected feature and determine its influence on the query. Listing 4.9 checks whether a query is satisfiable. We iterate over all selected features in Lines 3-4 and if there is any included dead feature (Line 6) or excluded core feature (Line 8), then we can conclude that the partial configuration can not be satisfiable.
If a query is satisfiable, we still might be able to reduce the size of the query and therefore, reduce the effort when computing a result. We accomplish that by removing selected features that do not change the outcome of the query. Listing 4.10 displays the implementation of that plan. We iterate again over all selected features in Lines 2-3, filter out included core features in Line 6 and excluded dead features in Line 8. We collect the remaining included and excluded features as a list in Line 10. This method results in fewer nodes that have to be marked. Hence, the computation of the different cardinality-based algorithms increases. Overall, the implementations we present improve the performance of the algorithms for the cardinalities even further.

4.4 Summary

In this chapter, we presented the d-DNNF format that CNF to d-DNNF compilers are using and how to parse it. Additionally, we depicted the implementation of the data structure as an adjacency list and of the algorithms in Chapter 3 using the programming language Rust. Moreover, we presented a multi-threaded approach to compute the cardinalities for multiple queries to further improve the performance. This is only possible for the cardinality of features and partial configurations.
5. Evaluation

In this chapter, we examine the scalability of our implementation from Chapter 4 and compare it to alternative publicly available approaches. First, we present our research questions. Second, we give a brief introduction to each tool of the evaluation and give more details about the industrial feature models. Furthermore, we specify the technical setup. Third, we describe the design of our four experiments. Fourth, we present the results of our empirical evaluation. Fifth, we answer our research questions using our results. Finally, we discuss potential threats to the validity. For our evaluation, we consider three different #SAT solvers [BSB15, Thu06, SRSM19], three different CNF to d-DNNF compilers [Dar04, DM02, MMBH10, LM17] in combination with dknife and another existing d-DNNF reasoner. Furthermore, we use 127 different industrial feature models and their corresponding CNFs for our evaluation.

For each of the three analyses: cardinality of feature model, partial configurations, and features, we identify the best performing tool respectively tool combination. We determine that the #SAT solver sharpSAT performs best for the cardinality of a feature model, but dknife performs significantly better for the remaining cardinalities with dsharp as d-DNNF compiler. Additionally, we give estimates for which circumstances we can apply the results and recommend when to use which approach.

5.1 Research Questions

In this section, we present the three research questions that we answer with our empirical evaluation.

- **RQ1**: For the three types of analyses, can d-DNNF reasoners, and especially dknife, outperform state-of-the-art #SAT solvers for industrial feature models? The considered applications on feature models can be reduced to the three types of queries: cardinality of a feature model, partial

\[\text{http://www.cril.univ-artois.fr/kc/d-DNNF-reasoner.html}\]
configurations, and features [SNB+21]. Hence, we divide RQ1 into three sub-questions that each address one of the three types. For each of the sub-questions, the d-DNNF based approach consists of the runtime of a compiler together with either dknife or query-dnnf.

- **RQ1.1**: Do #SAT solvers or d-DNNFs perform better regarding runtime for the cardinality of a feature model?
- **RQ1.2**: Do #SAT solvers or d-DNNFs scale better for the cardinality of partial configurations? Moreover, we differentiate between satisfiable and unsatisfiable partial configurations.
- **RQ1.3**: Do #SAT solvers or d-DNNFs scale better for the cardinality of features?

- **RQ2**: Which combination of a d-DNNF compiler and a reasoner is the fastest? We identify the fastest tool combination of a d-DNNF compiler together with a fitting d-DNNF reasoner. Furthermore, we aim to examine if one compiler produces d-DNNFs that lower the runtime of a reasoner, despite being not the overall fastest combination. Consequently, those d-DNNFs might be able to scale for a larger number of queries than we consider.

- **RQ3**: Is there a break-even point for a number of queries at which #SAT solvers have the same runtime as the combination of compilers and reasoners? One of the approaches may be faster for only a few queries and the other approach for a higher number of queries. Therefore, we aim to recommend an approach for a given number of queries.

## 5.2 Subject Systems

We only consider industrial models in contrast to synthesized ones because we consider the scalability in an industrial context as vital for the usability of an approach. We use product lines from databases, financial services, the operating system domain, and the automotive domain. We list those product lines in Table 5.1 sorted by the number of features. We group the 116 different CDL models for clarity reasons and because they are similar in terms of the number of features, constraints, and overall size of the corresponding CNF [STS20].

Knüppel et al. [KTM+18] provide the CDL, KConfig models as well as the Automotive02 model and can be found here². Additionally, BusyBox is also specified in KConfig and can be found here³. Component Definition Language (CDL) was developed for the eCOS component architecture [VD11]. KConfig is developed under Linux and can be used to manage configurable systems [OGB+19]. BusyBox unites different common Unix commands in one program. The other models, namely BerkeleyDB, FinancialServices, and Automotive01 can be found as example models in FeatureIDE⁴.

---

²[https://github.com/AlexanderKnueppel/is-there-a-mismatch](https://github.com/AlexanderKnueppel/is-there-a-mismatch)

³[https://github.com/PettTo/Measuring-Stability-of-Configuration-Sampling](https://github.com/PettTo/Measuring-Stability-of-Configuration-Sampling)

⁴[https://github.com/FeatureIDE/FeatureIDE/tree/develop/plugins/de.ovgu.featureide.examples/featureide_examples](https://github.com/FeatureIDE/FeatureIDE/tree/develop/plugins/de.ovgu.featureide.examples/featureide_examples)
In the following, we discuss the different #SAT solvers and combinations of CNF to d-DNNF compilers with d-DNNF reasoner.

- **countAntom [BSB15]** is a DPLL based #SAT solver. countAntom applies multi-threading and component caching which allows reusing results of sub-problems which occur multiple times. Therefore, it uses a special caching technique to ensure that racing threads do not lead to wrong results. countAntom is a #SAT solver that has shown to perform very well on CNFs that correspond to industrial feature models [Sun20]. For our evaluation, we use version v1.0\(^5\) with four threads.

- **sharpSAT [Thu06]** is a DPLL based #SAT solver. It utilizes component decomposition and caching. Furthermore, Thurley et al. [Thu06] introduce a cache management system that has a reduced cache size compared to other solvers. Additionally, sharpSAT is the other #SAT solver that has performed very well compared to other solvers in previous experiments [Sun20]. We use version v13.02.\(^6\)

- **ganak [SRSM19]** is a hash-framework based #SAT solver. It uses probabilistic component caching and universal hashing for exact model counting. Additionally, ganak develops new heuristics to further improve its performance. ganak shows great results in the model counting competition 2020 [FHH20] and as a consequence, we also consider ganak in our empirical evaluation. We use version v1.0.\(^7\)

- **c2d [Dar04, DM02]** is a compiler that takes a CNF and transforms it into a d-DNNF. c2d creates a binary decomposition tree with the clauses of the CNF as leave nodes. From there on, if two child clauses share variables they are eliminated from the clauses with the help of a case analysis. This continues

\(^5\)https://projects.informatik.uni-freiburg.de/projects/countantom
\(^6\)https://github.com/marcthurley/sharpSAT
\(^7\)https://github.com/meelgroup/ganak
until there are no more two child nodes that share common variables. The resulting d-DNNF has the format described in Chapter 4. Additionally, c2d is able to provide the smooth property which we need for dknife. For the empirical evaluation, we use version v2.20.\(^8\)

- **dsharp** [MMBH10] compiles a formula in CNF into the corresponding d-DNNF. Like c2d, dsharp also decomposes CNFs in smaller components but dsharp does this procedure dynamically. Moreover, dsharp also supports the smooth property and uses the same format as c2d. We use dsharp with the commit tag `b8b2526`.\(^9\)

- **d4** [LM17] compiles a formula in CNF into the corresponding d-DNNF. d4 uses the same dynamic decomposition approach as dsharp. Furthermore, d4 also uses the format that c2d defines but it does not support smoothing. Therefore, we can not use d4 in combination with dknife. We use version v1.0.\(^10\)

- **query-dnnf** is a d-DNNF reasoner that requires a d-DNNF from one of the compilers. query-dnnf exploits d-DNNFs like dknife with differences in the actual implementation. In addition to that, query-dnnf can handle d-DNNFs that do not fulfill the smooth property. We use version v0.2\(^11\) of query-dnnf.

The #SAT solvers do not need any additional processing and can directly use the corresponding CNFs of the feature models. In contrast to that, we have to combine CNF to d-DNNF compiler with reasoners. The combinations are: query-dnnf + c2d, query-dnnf + d4, dknife + c2d, and dknife + dsharp. The current version of dknife needs smooth d-DNNFs which d4 can not produce. Hence, this combination is not viable. Furthermore, query-dnnf gives only 0 as result for all queries, using the d-DNNFs from dsharp. As a consequence, we also do not combine those tools. However, dsharp on its own as #SAT solver and in combination with dknife does provide correct results. We test all other possible combinations, because that allows better comparisons between compilers and reasoners. For instance, we can compare query-dnnf and dknife independently if we use the same compiler output of c2d for both reasoners.

### 5.3 Experiment Design

In this section, we specify the design of the empirical evaluation. We differentiate between the five following experiments.

**EX1** Compilation of CNFs to d-DNNFs: We gather insights for the research question RQ2. We measure the runtime of the compilers c2d, dsharp, and d4 for

\(^8\)https://reasoning.cs.ucla.edu/c2d/
\(^9\)https://github.com/QuMuLab/dsharp
\(^10\)https://www.cril.univ-artois.fr/KC/d4.html
\(^11\)http://www.cril.univ-artois.fr/kc/d-DNNF-reasoner.html
all 127 models five times. Furthermore, we distinguish between d-DNNFs with and without the smooth property for c2d as well as dsharp. Each compiler gets 5 min until the timeout is reached. We use the median time of each model for the five runs. Moreover, we also consider the runtimes in the subsequent experiments to take the sum of compilation time and reasoner into account.

EX2 Cardinality of feature models: We cover the research questions RQ1.1 and RQ2. We measure the runtimes of computing the cardinality of a feature model for all 127 models five times with a timeout of 5 min for each of the runs. We test the following tools and tool combinations: countAn- tom, sharpSAT, ganak, query-dnnf + c2d, query-dnnf + d4, dknife + c2d, and dknife + dsharp.

EX3 Cardinality of partial configurations: We gather insights for the research questions RQ1.2, RQ2, and RQ3. We split the experiment into two parts: one considers only satisfiable and the other unsatisfiable partial configurations. Besides the timeout, which is 15 min for the satisfiable and 10 min for unsatisfiable partial configurations, the part experiments are identical in their structure. We set the timeout for unsatisfiable partial configurations lower because all considered #SAT solvers and d-DNNF reasoners include optimizations for that edge case. We test all the models which did not timeout for all runs in EX1 and EX2 because it is highly unlikely to solve 250 queries in 10/15 minutes after the tool respectively tool combination was not able to solve one query in five minutes. Consequently, we do not waste time and energy consumption for obvious results. This part of the evaluation takes 50 partial configurations for each of the lengths: 2, 5, 10, 20, 50. All 250 partial configurations are randomly created using Pythons standard seed-based pseudo-random number generator. We simply add one feature after another to the configuration until we reached the desired length while discarding each feature which leads to an unwanted satisfiability. We use the same tools as in EX2 with the addition of the single-threaded version of our reasoner in combination with c2d and dsharp.

EX4 Cardinality of features: We examine the research questions RQ1.3, RQ2, and RQ3. We test all the models which do not timeout for all tools and all runs in experiments 1 to 3. We run this part of the evaluation just one time with a timeout of 30 min due to the fact that we have to compute on average 1407 queries for each model. Additionally, we use the same tools respectively the same tool combinations as in EX3.

We aim to maximize the number of runs and the timeouts. Therefore, we eliminate feature models which almost certainly hit the timeout in a later part of the evaluation because they already hit the timeout for all tools in EX1 or EX2. Additionally, we do not want to spend unnecessary amounts of energy and time. We conclude that each experiment should result in conclusive results that allow us to analyze and compare the performance of tools. Because of that, we choose to run the last part, which has a higher timeout and more queries, only
once. Additionally, the high amount of queries leads to just as many measure-
ments which we expect to even out possible inaccuracies. The total runtime for
all experiments is in the worst case $\approx 125$ days when all timeouts are hit and we
would not remove any model. We expect that to be a tradeoff that still provides
us with appropriate results.

**Technical Setup**

In this section, we describe the technical setup of the system, we used for our
benchmark. We run the experiments on a 64-bit architecture on the operating
system Ubuntu 20.04 (Focal Fossa). The memory usage is limited to eight giga-
bytes and we use the i7-8700k CPU with a base clock frequency of 3.7GHz and six
cores. The benchmarking framework is written in Python v3.8.10 and we use the
timeit module\(^\text{12}\) for the measurements. Besides that, we make sure to only in-
clude the runtime of the different tools. So, we specifically exclude the time for
manipulating CNFs which is needed for the \#SAT solvers. To accomplish that, we
only measure the runtime of the executable \#SAT solver for each query and sum
those times up. Additionally, we exclude the time for creating the input string
that \texttt{query-dnnf} needs. Hence, we make sure that our benchmark program does
not distort our results.

### 5.4 Results

In this section, we present the results of our empirical evaluation, separating
the different experiments that we describe in the previous section.

#### 5.4.1 Experiment 1: Compilation

Figure 5.1 shows the runtimes for the generation of a corresponding d-DNNF
from the CNF for the three compilers: c2d, dsharp and d4. c2d and dsharp gen-
erate smooth d-DNNFs which is not possible using d4. Each point on the x-axis
corresponds to the indices in Table 5.1 and we sort by the number of features
from few to many. The y-axis has a logarithmic scale and shows the runtime in
seconds. The red line indicates the timeout of 300 s for the experiment. None of
the solvers was able to produce a d-DNNF for \texttt{linux-2.6.33-3}. Additionally, c2d
timed out four out of five times while generating a corresponding d-DNNF for
\texttt{Automotive02}. It needed 143 s for the one time it was able to produce a d-DNNF.
Nonetheless, the median hit the timeout, and consequently, we raise the results
in the following experiments to the timeout.

The fastest compiler is dsharp with a total time of 7.79 s for the eleven models
(excluding \texttt{linux}). In direct comparison for each model dsharp also is the fastest
compiler for eight out of the eleven models. c2d is the slowest compiler with
a total time of 187 s, including the time for the one successful run with the Au-
tomotive02 model. Despite that, c2d is the fastest compiler for the median of
the CDL models. d4 was able to generate all d-DNNFs in 22.9 s. Additionally, d4

\(^{12}\)\url{https://docs.python.org/3/library/timeit.html}
5.4. Results

Figure 5.1: Comparison of CNF to d-DNNF Compilers

Figure 5.2: Comparison of CNF to d-DNNF Compilers Regarding CDL Models
was the fastest compiler for FinancialServices and Automotive01. Overall, c2d and dsharp were slightly faster without adding the smooth property with a time save of 0.28 s for dsharp and 0.94 s for c2d.

The CDL models are structurally similar and we get similar results as depicted in Figure 5.2. The axis description and the timeout are the same as in Figure 5.1. For the median times after 5 runs of the CDL models, c2d has an average of 4.44 s with a standard deviation of 0.65, dsharp an average of 6.17 s with a standard deviation of 3.34, and d4 has an average of 6.44 s with a standard deviation of 2.91. Because of that, we summarize all CDL models and only use the median time of the median times after 5 runs in Figure 5.1. Furthermore, we use the same approach for all following experiments.

Besides that, the produced d-DNNFs differ in the number of nodes. We can not compare d4 with dsharp and c2d because d4 is not able to smooth the d-DNNFs which is required for dknife. Nonetheless, dsharp and c2d have differences in size for smooth d-DNNFs. c2d produces on median smaller d-DNNFs for the CDL models (i.e. 5.2 MB with c2d, 21.7 MB with dsharp) and dsharp generates the smaller d-DNNF for Automotive02 (i.e. 27.5 MB with c2d, 4.8 MB with dsharp).

5.4.2 Experiment 2: Cardinality of Feature Model

Figure 5.3 depicts the runtimes for computing the cardinalities of each feature model. We measure the runtime for all combinations of test subjects, presented in Section 5.3. The runtimes for using one of the reasoners also include the runtime for the compiler. Each point on the x-axis corresponds to the indices in Table 5.1 and we sort by the number of features in ascending order. The y-axis has a logarithmic scale and shows the runtime in seconds. The red line indicates the timeout of 300 s for the experiment. Furthermore, we summarize all CDL models and only use the median. Similar to the compilers, no #SAT solver was able to compute the cardinality for linux-2.6.33-3. As a consequence of that, we do not evaluate linux-2.6.33-3 in any of the following experiments. Hence, the indices from Table 5.1 change and Automotive02 has now the index eleven, while the other indices stay the same.

The fastest tool was sharpSAT with a total of 1.62 s, followed by ganak with 2.17 s. In contrast to that, the combinations of query-dnnf and dknife together with c2d were the slowest with a total of 199 s respectively 187 s for eleven models including Automotive02. Despite the high differences between the fastest #SAT solver and the slowest reasoners, the runtimes for the reasoners excluding the compilation were a total of 0.88 s for dknife and 12.2 s for query-dnnf, using d-DNNFs from c2d. The fastest combination of compiler and reasoner was dsharp with dknife and a total of 8.68 s, 0.89 s excluding the compile time. Hence, the fastest #SAT solver is 5.36 times faster than dknife with dsharp. Despite that, dknife together with dsharp was still 3.17 times faster than query-dnnf with its fastest compiler d4.
5.4. Results

5.4.3 Experiment 3: Cardinality of Partial Configurations

The runtimes for computing the cardinalities of satisfiable partial configurations for each feature model are displayed in Figure 5.4 and for unsatisfiable partial configurations in Figure 5.5. For both plots, each point on the x-axis represents one of the remaining eleven models with model index eight being the median of the 116 CDL models. The y-axis depicts the runtime in seconds with a logarithmic scale. The timeout that the red line represents is set to 900 s for satisfiable partial configurations and 600 s for unsatisfiable ones. In contrast to EX1 and 2, we have multiple queries. This allows us to differentiate dknife in one measurement for the standard multi-threaded version and one data point with a single-threaded version for each compatible compiler. The data points for the single-threaded version have an added s. for this experiment and EX4.

Regarding satisfiable partial configurations, the overall fastest tool combination was the multi threaded version of dknife together with dsharp at a total runtime of 15.2 s for all eleven models. The slowest tool was countAntom with a total time of 700 s. The fastest #SAT solver was sharpSAT with a total runtime of 162 s. Nevertheless, sharpSAT required 10.6 times more runtime than dknife with dsharp. Additionally, dknife together with dsharp was 7.24 times faster than query-dnnf with d4. For ten out of eleven models, one version of dknife in combination with dsharp performed best. For the median of the CDL models dknife together with c2d was the fastest.

Considering the unsatisfiable partial configurations, our single threaded reasoner using dsharp with 13.6 s was the fastest regarding the runtime, closely followed by our multi threaded version with 14.2 s. The fastest #SAT solver was
5 Evaluation

Figure 5.4: Runtimes for the Cardinality of 250 Satisfiable Partial Configurations

Figure 5.5: Runtimes for the Cardinality of 250 Unsatisfiable Partial Configurations
ganak with 19.7 s. Hence, ganak required 1.44 times longer than our single threaded tool in combination with dsharp. query-dnnf with c2d was the slowest tool/tool combination with a total runtime of 660 s for all models and also reaches the timeout for Automotive02. Moreover, dsharp together with dknife and query-dnnf with d4 each performed best for five out of the eleven models. Nevertheless our single-threaded tool in combination with dsharp was 2.21 times faster than query-dnnf with d4. countAntom is the fastest tool respectively tool combination for the median of the CDL models.

Figure 5.6 depicts the progress of the fastest #SAT solvers and the fastest tool combination of a d-DNNF compiler with a reasoner for the cardinality of partial configurations. For both plots, the logarithmic x-axis shows the runtime in seconds. The y-axis represents the number of computed satisfiable partial configurations for the left plot and of unsatisfiable ones for the right plot. Figure 5.6 shows that dknife together with dsharp starts with a higher runtime for both kinds of partial configurations and all models. Despite that, the gradient of the #SAT solvers are higher, except for the unsatisfiable partial configurations of the median of the CDL models. Moreover, the runtime of the #SAT solvers surpasses the runtime of the d-DNNF based approach for all models, excluding unsatisfiable partial configuration on the CDL models.

Table 5.2 summarizes the break-even points by providing the number of computed partial configurations till dknife together with dsharp surpasses the fastest #SAT solver. Moreover, Table 5.2 shows the percentage progress of the experiment until the break-even point. On average, the break-even point is at 47.1 queries with a standard deviation of 78.2 for unsatisfiable partial configurations, excluding the median of the CDL models. Additionally, for satisfiable partial configurations, the break-even point is at 11 queries with a standard deviation of 16.9.
Figure 5.6: Break-Even Point between d-DNNFs and \#SAT Solvers for Partial Configurations

Table 5.2: Number of Queries Needed for dknife + dsharp to Overtake \#SAT Solvers for the Cardinality of Partial Configurations

5.4.4 Experiment 4: Cardinality of Features

Figure 5.7 displays the runtimes for the computation of the cardinality of features. Each point on the x-axis represents one of the remaining eleven models with model index eight being the median of the 116 CDL models. The y-axis depicts the runtime in seconds with a logarithmic scale. The red line indicates the timeout which is 30 min for this experiment.
5.4. Results

The tool combination of dsharp with our multi-threaded tool was the fastest with a total runtime of 20.1 s. dsharp in combinations with our single and multi-threaded tool were the only ones to compute the cardinality of features for Automotive02 within the time limit. The fastest tool respectively tool combination, that does not include dknife, was query-dnnf with d4 and a total time of 41.1 min while only computing ten out of the eleven models. Hence, dknife with dsharp is 123 times faster than query-dnnf with d4. Furthermore, the fastest #SAT solver was sharpSAT with 50.9 min and therefore, was 152 times slower than the fastest tool combination. Additionally, the combination of dknife and dsharp was the fastest for ten out of the eleven models. c2d with dknife performed best for the median over all CDL models. The slowest tool was ganak with a total of 56.5 min.

Some tools timeouted on multiple occasions. Regarding the CDL models, ganak reached the timeout for 30 models, sharpSAT for 19 models, and query-dnnf with d4 for 2 models. Additionally, countAntom, ganak, sharpSAT, query-dnnf with c2d, and query-dnnf with d4 hit the timeout for Automotive02. Besides that, we can measure the progress and compute the average progress for the three #SAT solvers. Concerning them, Automotive02 was the model with the lowest progress for all #SAT solvers. countAntom only completed 4.42%, ganak 26.46%, and sharpSAT 29.10% of the computation.

In Figure 5.8, we compare the quality of the d-DNNFs by comparing the results of the same reasoner for different input data. Concerning dknife, the d-DNNFs that are created with c2d on average are 1.38 times faster than dsharp. On average, for query-dnnf the models of c2d are 2.74 times faster than d4, excluding the timeout for Automotive02 because both combinations did timeout for Au-
tomotive02 and we cannot measure their progresses. Regarding dknife, c2d is faster for seven out of the eleven models and for query-dnnf is d4 faster than c2d for six out of the ten computed models.

Figure 5.9 depicts the runtime over the computation progress for the fastest #SAT solver which is sharpSAT compared to the fastest d-DNNF based approach with dknife and dsharp. The x-axis shows the computed queries for the cardinalities of features on a logarithmic scale. The y-axis depicts the runtime in seconds which is also scaled logarithmically. The plot shows, that dknife in combination with dsharp starts at a higher runtime than sharpSAT for all models. In contrast to that, the gradient of sharpSAT is increasing faster with a higher gradient than dknife with dsharp. Consequently, the runtime of sharpSAT surpasses the runtime of dknife in combination with dsharp.

The break-even point is further illustrated by Table 5.3. The table shows the point at which dknife with dsharp breaks even with sharpSAT. Moreover, the second column computes the progress of the computation by dividing the break-even point by the number of features for the model. Overall, dknife in combination with dsharp needs on average 7.90 queries to break even with a standard deviation of 12.6. That is on average 1.50% of the whole computation with a standard deviation of 1.76.
Figure 5.9: Break-Even Point between d-DNNFs and #SAT Solvers for Several Features

Table 5.3: Number of Queries Needed for dknife + dsharp to Overtake sharpSAT for the Cardinality of Features
5.5 Discussion

In this section, we discuss the research questions from Section 5.1, using the results of our five experiments which we introduce in the previous section. We divide the answer to the research questions into smaller parts in which we take a closer look at all experiments that influence our answer and regard different aspects of the question.

RQ1

In this subsection, we aim to answer RQ1: For the three types of analyses, can d-DNNF reasoners, and especially dknife, outperform state-of-the-art #SAT solvers for industrial feature models? In Chapter 3, we introduced the three important types of cardinality-based analyses, namely cardinality of feature model, partial configurations, and features. The sub-research questions RQ1.1, RQ1.2, and RQ1.3 correspond each to one of those analyses.

The results of the first two experiments show that the runtime of the CNF to d-DNNF compilers is generally higher than one #SAT call, leading to #SAT solvers outperforming the combination of compiler and reasoner. Thus, we can answer RQ1.1: Do #SAT solvers or d-DNNF reasoners perform better regarding runtime for the cardinality of a feature model? #SAT solvers, namely ganak and sharpSAT perform better for the cardinality of feature model. Nevertheless, the fastest combination of compiler and reasoner which is dknife with dsharp is only 5.36 times slower than the fastest #SAT solver. Despite that, dknife in combination dsharp as well as query-dnnf with d4 were able to outperform countAntom more often than not.

The results of EX3 show that d-DNNFs can surpass #SAT solvers for the cardinality of 250 satisfiable partial configurations as well as 250 unsatisfiable ones, concerning the runtime. Hence, we can answer RQ1.2: Do #SAT solvers or d-DNNF reasoners scale better for the cardinality of partial configurations? d-DNNF reasoner scale better for partial configurations regardless if the partial configurations are satisfiable or not. Regarding the total runtime for satisfiable partial configurations, dknife together with dsharp is faster than the fastest #SAT solver sharpSAT for all eleven models individually and 10.6 times faster. For unsatisfiable partial configurations, dknife with dsharp is 1.44 times faster than the fastest #SAT solver ganak. Moreover, the d-DNNFs based approach was faster for ten out of the eleven models. We argue that the smaller differences in runtime for unsatisfiable partial configurations occur on the one hand due to the relatively higher compile-time compared to the runtime of the reasoner. On the other hand, the runtime differences between reasoners and #SAT solvers to compute one unsatisfiable query is smaller than the difference for a satisfiable query.

The last experiment about the cardinality of features strongly indicates that d-DNNFs scale better than state-of-the-art #SAT solvers. Thus, we can answer RQ1.3: Do #SAT solvers or d-DNNF reasoners scale better for the cardinality of features? d-DNNF reasoner scale better for the cardinality of features by multiple orders of magnitude. If we include the estimated time for sharpSAT based on
the solved percentage of Automotive02 at timeout, then dkknife together with dsharp is 370 times faster concerning the runtime of the cardinality of features. Despite that, sharpSAT is still the fastest #SAT solver with a total runtime of 124 min. Furthermore, the other d-DNNF reasoners with different compilers also outperform sharpSAT. Consequently, d-DNNF reasoners in combination with CNF to d-DNNF compilers outperform #SAT solvers by a noticeable margin.

Overall, we can summarize those d-DNNF reasoners together with compilers scale better than #SAT solvers for the cardinality of partial configurations and features but #SAT solvers are slightly faster for the cardinality of a feature model. Additionally, dkknife performs better in all experiments than the existing reasoner query-dnnf.

RQ2

In this subsection, we aim to answer RQ2: Which combination of a d-DNNF compiler and a reasoner is the fastest?

The results of experiment one show that concerning compile time itself dsharp is the fastest, followed by d4. c2d is significantly slower and was not able to consistently generate a corresponding d-DNNF for Automotive02. Besides that, we also have to take into account that d4 is the only compiler that can not fulfill the smooth property, which is an additional effort for c2d and dsharp. Being the fastest compiler can have a huge impact in the following experiments especially when the reasoners are fast compared to the compilers. As a consequence of that, dsharp is the fastest compiler in combination with a reasoner, followed by d4.

Despite the previous findings, dkknife with dsharp does not perform best for all considered models. Especially, for the median of all CDL models, c2d can generate smaller d-DNNFs resulting in a lower required runtime than using dsharp, despite dsharp being faster for the generation of the d-DNNF. Considering the other three experiments, dkknife did always perform better with the d-DNNF from c2d. Moreover, query-dnnf is also faster for the CDL models for computing the cardinality of satisfiable partial configurations and features.

In the direct comparison of dkknife and query-dnnf with the best performing compiler for each tool, dkknife was 3.17 times faster for the cardinality of a feature model, 2.21 times for unsatisfiable partial configurations, 7.24 times for satisfiable ones, and 123 times faster for the cardinality of features.

Overall, we can conclude that dsharp, followed by d4 perform best for most models and should be the first choice for an unknown model. Furthermore, query-dnnf performed better using d4 over c2d. Nevertheless, c2d can be an option, especially for a high number of queries. In that case, the possibly smaller d-DNNFs can be beneficial. Concerning the combination of reasoners and compilers, dkknife with dsharp was the fastest by a significant margin and therefore we recommend dkknife over query-dnnf for each cardinality-based analysis.
RQ3

In this subsection, we aim to answer RQ3: Is there a break-even point for a number of queries at which #SAT solvers have the same runtime as the combination of compilers and reasoners?

We can see for the cardinality of partial configurations and features the combination of CNF to d-DNNF compilers with a reasoner surpassed #SAT solvers. Hence, there is a break-even point at which the d-DNNF reasoners can compensate for the runtime of the compilers. For the following results, we only compare the fastest #SAT solver with the fastest combination of a CNF to d-DNNF compiler with reasoner. For unsatisfiable partial configurations ganak is the superior #SAT solver and sharpSAT performs best for satisfiable partial configurations as well as the cardinality of features. The fastest tool combination using d-DNNFs is dknife with dsharp.

For unsatisfiable partial configurations, d-DNNFs are faster for ten of the eleven models. For those ten models, the break-even point is reached on average after 47.1 queries with a standard deviation of 78.2. Additionally, d-DNNFs are faster for all models for satisfiable partial configurations. The average break-even point is at 11 queries with a standard deviation of 16.9. Furthermore, dsharp in combination with dknife surpasses sharpSAT for all models regarding the cardinality of features. On average, the break-even point is at 7.90 queries with a standard deviation of 12.6. Additionally, the combination of dknife with dsharp only has to compute 1.50% of features with a standard deviation of 1.76 to break even with sharpSAT which is the #SAT solver, requiring the least runtime.

5.6 Threats to Validity

In this section, we list potential threats and discuss their impact concerning the validity of our empirical evaluation.

- **Translation to Feature Model** We do not cover the translation of a configurable system in a feature model in our thesis, but it may be a threat to the validity. An incorrect feature model can lead to incorrect results. Knüppel et al. [KTM+18] which provide the majority of our models argue that some cross-tree constraints could be removed with their translation technique. Nonetheless, the data records are used in research and its minor errors are accepted [KTM+18].

- **Translation to CNF** The performance of #SAT solvers among other things depends on the translation of the feature model into a corresponding propositional formula [OGB+19]. We only use FeatureIDE and its translation to CNF. We estimate that the d-DNNFs are similarly affected by different translation techniques and because of that d-DNNF reasoners also profit or suffer from a specific technique. Furthermore, we do not expect huge differences from other translation techniques.
• **Computational Bias** In our empirical evaluation, we run EX4 only once. Despite that, we still argue that the high number of queries, which is equal to the number of features in a model, and the corresponding higher runtime reduces a potential bias. Hence, hundreds of queries and for some models, even multiple thousands of queries should be enough, so we can probably neglect deviations. We simply were limited by time and resources to run the experiment four more times. The additional runs may result in small runtime deviations but our conclusion should still be the same.

The c2d compiler results vary by a reasonable margin regarding the compilation time of the same model for each model. Additionally, for the Automotive02 model, c2d was able to generate a d-DNNF in one of the five runs. This issue cannot be circumvented. To counteract this a bit, we run the experiment five times and took the median as our results. We argue that this method reduces deviations, although we cannot completely exclude them.

• **Limited Number of Models** We have access to 127 different feature models but the 116 CDL models are highly similar and because of that our test subjects are limited. Regardless of the rather lower number of models, we still have a variety of different domains like databases, operating systems, and automotive. Additionally, the feature models have a wide range of features and cross-tree constraints as depicted in Table 5.1. Moreover, there is only a limited amount of industrial feature models we have access to.

• **Tool Invocation** We limited ourselves to mostly default parameters for all tools. We did only set flags for memory limitations and smoothing as well as the generation of an output file for the d-DNNF compilers. Additional parameters may shift the results in favor of some tools. This would result in higher computational resources and is out of the scope of our thesis.

Variable ordering may affect the runtime and the results of dsharp. This problem exists, but we were not able to reproduce that issue with a wide range of manual test samples. Those samples include a few of the CDL models and all other models of our empirical evaluation. For those samples, we compared the results of dknife using dsharp with all the #SAT solvers and we were not able to find any differences. Despite it might have an influence, we have no direct influence on the functionality of other tools.

### 5.7 Summary

In this chapter, we presented the results of our empirical evaluation comparing three state-of-the-art #SAT solvers to the combinations of three CNF to d-DNNF compilers in combination with dknife and an additional d-DNNF reasoner on 127 industrial-sized feature models. We subdivided our evaluation into four experiments: (1) the runtime for the compilers to generate d-DNNFs out of CNFs, the runtime of the different tools respectively tool combinations for the cardinality of (2) a feature model, (3) satisfiable/unsatisfiable partial configurations, and (4) features. dknife together with dsharp was the only tool respectively tool
combination that was able to compute the cardinalities for all experiments on all feature models excluding Linux. Furthermore, this tool combination required the least runtime for all cardinalities except the cardinality of a feature model where sharpSAT performed best. Because of that, we conclude that a d-DNNF based approach using the algorithms which we presented in Chapter 3 scales better than all other considered tools.
6. Related Work

In this chapter, we discuss related work. First, we describe the usage of knowledge compilation in feature model analysis and present other analyses. Second, we discuss non-propositional approaches that also consider feature attributes. Third, we inspect the current tool support for d-DNNF exploitation and compare it to dknife.

Knowledge Compilation in Feature-Model Analysis

In this section, we discuss other approaches for feature-model analysis that are also based on knowledge compilation. Other works consider decomposable negation normal forms (DNNFs), binary decision diagrams (BDDs), sentential decision diagrams (SDDs), and modal implication graphs (MIGs).

Voronov et al. [VÅE11] use DNNFs which are a super-set of d-DNNFs that do not demand determinism. They use DNNFs for the enumeration of valid configurations. Besides that, they conclude that determinism is needed to compute the number of possible assignments in polynomial time. Consequently, the deterministic property is beneficial, especially for larger feature models.

Besides d-DNNFs, BDDs are widely used for the analysis of feature models [HPMFA+16, ACLF13, MWCC08, PLP11]. BDDs are used by Heradio et al. [HPMFA+16] for the identification of core and dead features. Moreover, Benavides et al. [BSTRC07] are able to compute the cardinality of partial configurations, using BDDs. Furthermore, Pohl et al. [PLP11] conclude that BDD solvers outperform constraint satisfaction problems (CSPs) and SAT solvers regarding computing cardinalities. They were not able to compare state-of-the-art #SAT solvers and query-dnnf simply since our considered tools in the empirical evaluation were not published yet. Besides that, Darwiche et al. [DM02] show, that d-DNNFs can outperform BDDs by a significant margin. Additionally, the scalability issues of BDDs are confirmed [BSRC10, OGB+19, STS20]. Hence, even though BDDs are applicable for cardinality-based feature model analyses, there is currently no tool that scales for large feature models.
In addition to that, SSDs are proven to be more succinct than BDDs [Bov16]. However, Sundermann et al. [Sun20] also showed that the SSD compiler minic2d\(^1\) does not scale as good as the state-of-the-art \#SAT solvers which we consider in our empirical evaluation, regarding cardinality-based feature model analysis.

Krieter et al. [KTS+18] use MIGs for precomputing intermediate values in order to improve the performance of decision propagation. Decision propagation allows a backtracking-free configurations process. We do not consider interactive configuration support and solely focus on predetermined partial configurations. Furthermore, MIGs are not used for cardinality-based analyses.

**Non-Propositional Approaches**

For the analysis of product lines, we only consider solutions and approaches that are based upon SAT but there are other methods [BSRC10, BRCT05, MOP+19]. Benavides et al. [BRCT05] use constraint programming (CP) for the computation of the cardinality of features and partial configurations. In addition to that, they take functional and extra-functional features among attributes like availability, reliability, cost, etc. into account. Despite that, the authors only consider small feature models with up to 23 features, containing no cross-tree constraints. That are way less features than we consider in our empirical evaluation and therefore, we expect CP to not scale for those.

Munzo et al. [MOP+19] compared satisfiable modulo theory (SMT), CP, and \#SAT solving for computing the number of valid configurations for extra-functional features. For that, they introduce bit-blasting to encode boolean and numerical constraints and create a propositional formula from an arithmetic one. Their empirical evaluation shows that \textsc{sharpSAT} with bit-blasting is much faster and more scalable than the SMT and CP approaches. Hence, the insights from their work might be useful if we extend d-DNNF reasoners for extra-functional attributes in the future.

**Further d-DNNF Exploitation**

We focus on cardinality-based analyses but d-DNNFs and their exploitations are not limited to the three types of cardinality-based queries, we present in Chapter 3. Uniform random sampling creates a sample in which valid configurations are uniformly distributed i.e. it allows us to sample from a set of valid configurations where drawing each element is equally probable. This is critical for quality assurance operations like testing [AZAB17, VAHT+18] and making statistical inferences [OGB+19]. Sharma et al. [SGRM18] provides an algorithm for uniform random sampling using d-DNNFs. We only consider the general use case of computing the cardinality of partial configurations which can be used for uniform random sampling [Sun20].

Kübler et al. [KZK10] utilizes \textsc{c2d} as a \#SAT solver to compute the cardinality of partial configurations. They do not exploit the possibility to generate d-DNNFs.

\[^1\]http://reasoning.cs.ucla.edu/minic2d/
Furthermore, Darwiche [Dar01b] proposed even more algorithms, concerning d-DNNFs. He also presents an idea that uses assumptions to compute the cardinality of satisfying assignments when changing a single literal without needing another traversal of the d-DNNF. Beforehand, this idea needs two traversals of the d-DNNF for each assumption to create partial derivations for a counting graph. This is particularly interesting for computing the cardinality of features. Nevertheless, the author does not provide an implementation or technical description. Hence, there also is no empirical evaluation to compare its performance with existing tools. Notwithstanding the foregoing, it might be beneficial to develop an implementation of Darwiche’s idea.

With query-dnnf, we considered the one existing d-DNNF reasoner besides our implementation. query-dnnf expands dknife by providing the option to compute weighted d-DNNFs and it also accepts non-smooth d-DNNFs as input. Lagniez et al. did not publish any explanation for the implementation of their reasoner. Despite that, the source code is publicly available here\(^2\). We use other techniques to compute the three types of cardinalities and perform better in our empirical evaluation in Chapter 5.

\(^2\)https://www.cril.univ-artois.fr/KC/d-DNNF-reasoner.html
7. Conclusion and Future Work

The exploitation of d-DNNFs for cardinality-based feature-model analysis results in significant performance benefits compared to other approaches. Overall, we present a new way to represent d-DNNFs in our data structure and fitting algorithms that utilize it. Our approach aims to reuse as much information from previous calls as possible to reduce the percentage of the d-DNNF that needs to be traversed.

In our empirical evaluation, we compare different state-of-the-art #SAT solvers (sharpSAT, countAntom, ganak), d-DNNF compilers (c2d, dsharp, d4) on their own and together with dknife as well as the alternative reasoner query-dnnf. Using the insights from our empirical evaluation, we conclude the following: (1) d-DNNFs scale to a vast majority of industrial feature models (i.e., 126 out of 127 models with linux as exception). (2) d-DNNF reasoners scale better than state-of-the-art #SAT solvers for analyses that require multiple #SAT queries (i.e., 1.4 times faster for the cardinality of unsatisfiable partial configurations, 11 times for satisfiable partial configurations, and 370 times for features). (3) our advances in exploiting d-DNNFs show better results than the existing tool query-dnnf for the feature models in our empirical evaluation (i.e., 3.17 times faster for the cardinality of a feature model, 2.13 times for unsatisfiable partial configurations, 7.24 for satisfiable partial configurations, and 123 times for features). (4) d-DNNFs overtake #SAT solver at some point and there is an average break-even point (i.e., on average after 47.1 queries for the cardinality of unsatisfiable partial configurations, 11 queries for satisfiable ones, and after 7.90 queries for the cardinality of features). Overall, we determine that dknife scales better than state-of-the-art #SAT solvers and the d-DNNF reasoner query-dnnf. Consequently, we argue that future work regarding the exploitation of d-DNNFs for feature model analysis with dknife is promising and we identified potential in the following aspects.

Smoothness

The algorithms that we present in Chapter 3, require smooth d-DNNFs. While simplifying and accelerating the computation, smoothness is not a necessary
property to efficiently compute the cardinalities of feature models. Furthermore, the CNF to d-DNNF compiler d4 does not support the generation of smooth d-DNNFs and therefore, dknife cannot use its d-DNNFs. Apart from that, d4 was the fastest compiler for FinancialServices and Automotive01. Hence, the extension for non-smooth d-DNNFs would allow us to extend our empirical evaluation and we might be able to further increase the performance.

**Hard Feature Models**

The results of our empirical evaluation show that neither a #SAT solver nor a d-DNNF compiler was able to compute the cardinalities respectively to generate a d-DNNF for Linux. This motivates the idea to approximate results because approximating should be faster than an exact computation. We consider simplifying the CNF before the compilation process. The compilers then might be able to generate a corresponding d-DNNF for the simplified CNF and afterward, our reasoner could compute different cardinalities.
Bibliography


