Construction of Decision Diagrams for Product Configuration

Popov, Maxim

2023

Popov, M, Balyo, T, Iser, M & Ostertag, T 2023, Construction of Decision Diagrams for Product Configuration. in J M Horcas, J Ángel Galindo, R Comploi-Taupe & L Fuentes (eds), Proceedings of the 25th International Workshop on Configuration (ConfWS 2023). CEUR Workshop Proceedings, vol. 3509, CEUR, pp. 108-117, International Workshop on Configuration, Malaga, Spain, 06/03/2023. < https://ceur-ws.org/Vol-3509/paper15.pdf >

http://hdl.handle.net/10138/567194

Construction of Decision Diagrams for Product Configuration

Maxim Popov[1,2], Tomáš Balyo[1], Markus Iser[2,3] and Tobias Ostertag[1,*]

[1]CAS Software AG, CAS-Weg 1 - 5, 76131 Karlsruhe, Germany
[2]Karlsruhe Institute of Technology (KIT), KIT-Department of Informatics, Karlsruhe, Germany
[3]University of Helsinki, Department of Computer Science / HIIT, Helsinki, Finland

Abstract

Knowledge compilation is a well-researched field focused on translating propositional logic formulas into efficient data structures that allow polynomial-time online queries related to the SAT problem. Knowledge compilation techniques can be used to partition product configuration tasks into two distinct phases: fast online processing and slow offline preprocessing. Binary Decision Diagrams (BDDs) are widely studied in this area and provide a graph representation of Boolean formulas. However, BDD construction can be time-consuming, particularly for large instances, as their size grows exponentially with the number of variables. This paper explores methods to improve BDD construction time, including optimizing variable ordering. The evaluation involves applying these techniques to formulas in Rich Conjunctive Normal Form, comparing the results with Sentential Decision Diagrams. The experiments use CAS Software AG benchmarks.

Keywords

Configuration, Knowledge Compilation, Decision Diagrams

1. Introduction

Propositional logic is a common form of representing real-life logical relations and rules in a way that can easily be used in computer. The following example demonstrates how Boolean formulas are used in the area of product configuration:

Example 1.1. Suppose a company selling bikes offers various configurations, where selecting one component (i.e., bike frame) can limit choices for other components (i.e., wheels) due to compatibility constraints. These constraints can be represented as Boolean formulas.

$$R1 = \neg F1 \vee ((W \vee B) \wedge \neg BL \wedge \neg G) \tag{1}$$

Each variable in Equation 1 is assigned a value of *true* if the option is chosen. *F1* is the variable representing the frame option, and *W*, *B*, *G*, *BL* are the variables representing the frame colors white, blue, green, and black respectively. The formula represents the rule that if a frame 1 is chosen, only the colors white and blue can be selected.

Configuration of complex products with many options may be a computationally hard problem that also can be formulated as a Boolean Satisfiability Problem (SAT). SAT involves determining if a Boolean formula has satisfying assignments. While the problem is NP-complete, it can often be solved online using SAT solvers. Another way is to use knowledge compilation methods, whereby the solutions first get prepared and stored in a data structure offline and then can efficiently be retrieved online. Binary Decision Diagrams (BDDs) and Ordered Binary Decision Diagrams (OBDDs) are well-known knowledge compilation methods that represent Boolean formulas as binary trees. They were used in formal verification and were proved to be efficient in analysing systems with large amount of states [1].

Given the OBDD representation of a Boolean formula, satisfiability can be checked in constant time, solutions can be found in linear time, and models can be counted in polynomial time [2]. However, the size of the BDD as well as its construction time can be exponential in the number of variables.

For the product configuration, it means that configuration rules can be efficiently verified in the runtime, but we have to consider potentially long preprocessing time. Therefore, the reduction of BDD is essential for improving performance and can be achieved by optimizing variable ordering.

In this paper, we overview existing approaches for minimizing BDD size, apply them to RCNF formulas and introduce modifications of existing approaches: variable frequency and M-FORCE constraint ordering heuristics and construction strategies. We evaluate existing as well as our heuristics using real-world configurations and compare them to existing approaches. Lastly, we briefly present our modification of ordering heuristics for Sen-

ConfWS'23: 25th International Workshop on Configuration, Sep 6–7, 2023, Málaga, Spain
*Corresponding author.
maxim.popov@campus.tu-berlin.de (M. Popov); markus.iser@kit.edu (M. Iser); tobias.ostertag@cas.de (T. Ostertag)
0000-0003-2904-232X (M. Iser); 0000-0003-3294-3807 (T. Ostertag)

tential Decision Diagrams (SDDs) construction, which is a recently developed type of decision diagram that is a superset of OBDDs.

Chapter 2 presents some basic definitions that will be used throughout the work. Chapter 3 provides some insights into related works. Chapter 4 presents our ordering heuristics. Chapter 5 presents the libraries used to implement our approach and which also serve to measure baseline performance. Finally, Chapter 6 provides evaluation results for the described methods.

2. Preliminaries

This chapter contains definitions and examples of main concepts used in this work.

2.1. Boolean Formulas

This section contains definitions and notions including canonical normal forms of Boolean formulas.

Definition 2.1. Conjunctive Normal Form (CNF) is a conjunction of clauses $\bigwedge_i c_i$, where each clause c_i is a disjunction of literals $\bigvee_j l_j$. A CNF clause is *satisfied* if at least one of its literals is satisfied. A CNF formula is *satisfied* if all of its clauses are satisfied.

Definition 2.2. Disjunctive Normal Form (DNF) is a disjunction of terms $\bigwedge_i c_i$, where each term c_i is a conjunction of literals $\bigvee_j l_j$. A DNF term is *satisfied* if all of its literals are satisfied. A DNF formula is *satisfied* if at least one of its terms is satisfied.

Definition 2.3. An At-Most-One (AMO) constraint is a Boolean formula that takes a set of literals as an input and outputs *true* (is *satisfied*) if and only if maximal one of the input literals is satisfied. For a set of literals $\{l_1, ..., l_n\}$, we use the following notation for the AMO constraint: $AMO(l_1, ..., l_n)$

Definition 2.4. A Rich Conjunctive Normal Form (RCNF) formula is a conjunction of constraints, where a constraint can be a DNF formula, a disjunction of literals (equal to the CNF clause) or an AMO constraint. A RCNF formula is *satisfied* if all of its constraints are satisfied. Basically, RCNF is an extension of a CNF that allows more types of constraints, and thus allows smaller representation of a complex configuration rules .

2.2. The Boolean Satisfiability Problem

Let $\{x_1, ..., x_k\}$ be a set of Boolean variables and q be a propositional logic formula in CNF that contains only literals of $\{x_1, ..., x_k\}$. Formula q is *satisfiable* if and only if there exists a set of variable assignments, so that q is true. The Boolean Satisfiability Problem (SAT) is solved if either the satisfying assignment of the formula is found or it is determined that the formula is not satisfiable.

2.3. Binary Decision Diagram

This section is based on the Handbook of Model Checking [3].

Definition 2.5. A Binary Decision Diagram (BDD) represents a Boolean function as an acyclic directed graph, with the nonterminal vertices labeled by Boolean variables and the leaf vertices labeled with the values 1 and 0. Each nonterminal vertex v has two outgoing edges: $hi(v)$, corresponding to the case where its variable has value 1, and $lo(v)$, corresponding to the case where its variable has value 0.

Definition 2.6. An Ordered Binary Decision Diagram (OBDD) is a BDD for which an additional ordering rule applies: for each nonterminal vertex v associated with variable x_i and a vertex $u \in \{lo(v), hi(v)\}$ associated with variable x_j, we must have $i < j$

An OBDD can be reduced by eliminating redundant nodes and merging terminal and duplicate nodes. The result of such reduction is the Reduced Ordered BDD (ROBDD) [3]. This reduction can be performed in the time linear in the size of the original graph [4].

ROBDDs serve as a canonical form for Boolean functions, meaning that, for a given variable ordering, every Boolean function has a unique representation as a ROBDD. The construction of a ROBDD is essential to keep the BDD as small as possible, as the complexity of most algorithms that utilize BDD is dependent on the number of nodes/length of paths in the tree. In this paper, every mention of BDD refers to ROBDD.

Given a BDD of a function, we can answer these and other questions related to a SAT problem for a given instance. The function is satisfiable, if it does have a terminal node labeled with value 1. We can find a random solution for the formula by traversing the diagram from root toward the "1" leaf. The complexity of such algorithm is $O(h)$, where h is the height of the BDD. We can count the number of solutions by traversing the BDD and counting the paths. The complexity is $O(n)$, where n is the number of nodes. [2, 3]

The common strategy for BDD frameworks is to divide an overall function into smaller functions and creating BDDs bottom-up. We start by creating BDDs for single literals, and then subsequently use the BDDs from previous steps to create new ones by applying operations like AND, OR, XOR. The generalization of these operations is called Apply algorithm. The algorithm creates a BDD that represents the given result of applying the operation between the formulas of input BDDs. The overall time complexity of an Apply operation is $O(N_u \times N_v)$, where

N_u and N_v are the number of vertices in BDDs where vertices u and v respectively are the root nodes of input trees.

A BDD requires a defined variable ordering that will be followed along all paths of a diagram. The size of a BDD depends heavily on the ordering of input variables. Some functions can rise in size from linear to exponential in the number of variables due to a bad ordering. However, the problem of finding an optimal variable ordering to construct a minimum-size BDD is proven to be NP-complete and some functions don't have an optimal ordering [5]. Thus, instead of computing an optimal variable ordering, is a common approach to use heuristics to generate a good ordering and use it during BDD construction.

2.4. Sentential Decision Diagram

This section is based on the work of Darwiche (2011) [6].

Sentential Decision Diagram (SDD) is a more recent technique of representing of propositional knowledge bases. SDDs are a strict superset of OBDDs and are inspired by two discoveries: structured decomposability and strongly deterministic decompositions.

To explain SDDs, we first define the decomposition that is used to construct this type of decision diagram and the we define the important notion of *vtrees*.

Definition 2.7. Consider a Boolean function $f(X, Y)$ with non-overlapping variables X and Y. If $f = (p_1(X) \wedge s_1(Y)) \vee ... \vee (p_n(X) \wedge s_n(Y))$ then $\{(p_1, s_1), ..., (p_n, s_n)\}$ is called a (X, Y)-*decomposition* of function f. We call each pair (p_i, s_i) an *element* of the decomposition, p_i a *prime* and a s_i *sub*. If $p_i \wedge p_j = false$ for $i \neq j$ the decomposition is called *strongly deterministic* on X.

Definition 2.8. A *vtree* for variables X is a full binary tree whose leaves are in one-to-one correspondence with the variables in X.

The vtree is used to recursively decompose a given Boolean function starting at the root of the tree. The left subtree of each node corresponds to the X variables, and the right subtree to Y variables of the (X, Y)-decomposition. The SDD representation is then based on a recursive application of the presented decomposition technique. The formal definition of this operation is as follows:

Definition 2.9. *Notation:* $\langle . \rangle$ denotes a mapping from SDDs into Boolean functions. α is an SDD that respects vtree v if:

- $\alpha = \bot$ or $\alpha = \top$
- $\alpha = X$ or $\alpha = \neg X$ and v is a leaf with variable X
- $\alpha = \{(p_1, s_1), ..., (p_n, s_n)\}$, v is an internal node, $p_1, ..., p_n$ are SDDs that respect the left subtrees of v, $s_1, ..., s_n$ are SDDs that respect the right subtrees of v, and $\langle p_1 \rangle, ..., \langle p_n \rangle$ is a partition.

An SDD that consists of a constant or a literal is called *terminal*. Otherwise, it is called *decomposition*. SDDs are canonical, which means that for a given vtree, every Boolean function has a unique representation of an SDD [6]. SDDs are a strict superset over BDDs. The variable ordering of a BDD will then correspond to the total order of the vtree, which is defined as a sequence of variables obtained from the left-right traversal of the vtree [6].

BDD-trees are twofold exponential in treewidth, whereas SDDs are just exponential. The SDDs are also as tractable as BDDs, but are more succinct both in theory and in practice [7]. There exist some Boolean functions that can be represented with at least exponential BDD size and only polynomial SDD size [7].

3. Related Work

This section provides some insights into existing researches of BDDs and SDDs.

3.1. Product Configuration Using BDDs

The work of Hadzic et al. [8] presents BDDs as an efficient solution to the configuration problem. The authors also describe how they applied this method practically in the commercial software product *Configit*. They highlight that BDDs can be efficiently applied in industry use cases, since they have several advantages over commonly used search-based configurators, including faster response times, better scalability, and improved rule quality [8]. However, the research just mentions variable ordering methods to optimize BDDs, but does not provide any examples of efficient heuristics.

3.2. Static Variable Ordering

Static variable ordering techniques attempt to determine a near-optimal variable ordering before constructing the BDD based on prior analysis of the input function [9].

Many algorithms that were proven to be efficient are described in the work of Rice and Kulhari (2022) [9]. It includes straightforward approaches like Dependent Count, Variable Appending, Sub-Graph Complexity etc., as well as different metric optimization heuristic techniques.

One example of static variable ordering techniques is the MINCE (Min Cut Etc.) heuristic proposed by Aloul et al. (2004) [10]. Its main idea is to partition the variables into groups with minimal functional correlation between variables in separate groups by translating it into *balanced min-cut hypergraph partitioning problem.*[9].

The authors of the MINCE heuristic conjecture that their heuristic captures structural properties of Boolean functions arising from real-world applications [10].

3.3. Dynamic Variable Ordering

In contrast to static variable ordering techniques, the dynamic ordering techniques attempt to adjust the ordering online during the construction of the decision diagram [9]. The idea was presented by Rudell (1993) [11] based on the observation that swapping two adjacent variables of a BDD can be implemented without major changes to the Boolean function library API [3]. One of such techniques is *sifting*. Variables are moved up and down in the ordering, until the algorithm finds a location that leads to an acceptable number of total vertices. Evaluation results show that sifting improves the memory performance, but it is also a time-consuming process [3].

3.4. Top-Down SDD Compiler

Most SDD constructing algorithms work analog to the BDD construction that we presented earlier in this chapter: create a decision diagram from smaller decision diagrams. This process is usually referred to as *bottom-up* compilation.

The work of Oztok and Darwiche (2015) [12] describes a *top-down* compiler constructing SDDs from CNF formulas. The top-down compiler produces a subset of SDDs called Decision-SDDs. The compiler utilizes techniques from SAT solvers and model counting algorithms to decompose a formula. Results presented in [12] show that the top-down compiler is consistently more performant than the bottom-up compiler.

The miniC2D software package created by the University of California includes code for SDD compilation based on the idea of a top-down compiler from [12]. The program doesn't produce the SDDs itself, but its output can be transformed to the SDD in linear time.

3.5. Multivalued Decision Diagram

Multivalued Decision Diagram (MDD) is another structure that is also proved to be efficient in product configuration area. MDDs can be seen as a generalization of BDDs, where a function can work with more than binary (true/false) values. Research by Andersen et al. [13] provides an analysis of MDD usage in an interactive cost calculation task. The research also highlights the importance of variable ordering for both MDDs and BDDs. The evaluations even show that MDDs can perform better than BDDs for presented tasks. Nevertheless, most variable ordering heuristics are generally considered to be appliable to both BDDs and MDDs [9], so we could also apply the heuristics that we overview in our research to MDDs.

4. Analysis and Approach

This chapter describes different methods of ruleset preprocessing, variable ordering and BDD construction strategies. We will discuss the usage of existing state-of-the-art methods like FORCE as well as suggest some new algorithms.

4.1. Variable Reordering

This section explores various variable reordering heuristics and their algorithms aimed at improving BDD construction speed. The algorithms prioritize low ordering time, manageable implementation complexity, and effective variable ordering specifically for RCNF formulas.

We implemented and experimented with the following two different variable ordering heuristics: Variable Frequency (VF) and FORCE (F).

4.1.1. Variable Frequency

We propose the variable frequency (VF) as an easy to implement and efficient heuristic that produces reasonable orderings. The VF heuristic evaluates the variable's influence on the function output using the frequency metric. This metric counts either overall appearances of each variable or the number of constraints containing this variable. Subsequently, the algorithm sorts the list of variables using these values in descending order. The heuristic can be seen as a modification of the Dependent Count heuristic described in [9], but used mainly for a more general type of decision diagram called Multi-Valued Decision Diagrams (MDDs).

The intuition behind this heuristic is that more constrained variables are placed on a level closer to the root of the tree, which allows them to shorten the paths to the terminal nodes. However, the frequency metric does not consider the semantics of the formula and can lead to a false conclusion about variable influence.

The frequency counting takes linear time in the number of constraints in a formula $\Theta(|C|)$, considering that we have information about each constraint's contained variables. Sorting takes $\Theta(|V|log|V|)$. Overall, the algorithm takes loglinear time in the number of variables $\Theta(|V|log|V|)$

4.1.2. FORCE Heuristic

The FORCE heuristic which is described in the paper by Aloul et al. (2003) [14]. FORCE is introduced as an alternative to MINCE, as described in Section 3.2, and comes with a simpler implementation and orders-of-magnitude increased speed, while providing competitive results with MINCE.

The algorithm is based on the same observation as the MINCE heuristic: Related variables in satisfiability

typically participate in the same CNF clauses [14], so the heuristic reorders Boolean variables to place "connected" variables close to each other. FORCE transforms the variable ordering problem into the linear placement problem. The vertices of a hypergraph correspond to variables and edges correspond to clauses. Since in our case, the RCNF is used, the clauses are replaced with constraints for all the definitions.

The FORCE algorithm uses the force-directed placement instead of a min-cut placement. The idea behind it is that interconnected objects (vertices of a hypergraph or variables in our case) experience forces analog to springs according to the Hooke's law. The algorithm computes these forces and displaces the vertices in the direction of the forces iteratively.

After an initial ordering is given, the center of gravity of each hyperedge e is defined the following way:

$$COG(e) = \left(\sum_{v \in e} l_v \right) / |e| \tag{2}$$

with l_v denoting the index of a vertex v in a current placement.

The new position l'_v is calculated with the following formula in which E_v is the set of all hyper-edges connected to the vertex v.

$$l_v = \left(\sum_{e \in E_v} COG(e) \right) / |E_v| \tag{3}$$

Thereafter, the vertices are sorted according to the newly calculated positions. These iterations continue until a given metric of ordering stops improving. As proposed in the paper, the total variable span metric is used and the iterations stop after the metric doesn't decrease after given number n of iterations. Additionally, the iterations number is bounded by $c \cdot log|V|$, where c is a constant.

The worst-case time of the algorithm is $O((|C| + |V|log|V|) \cdot log|V|)$ [14], where C is the set of constraints, and we assume that the average degree of hyperedges and the average degree of vertices are limited by a constant.

4.2. Constraint Reordering

Another approach to reduce BDD construction time is by manipulating the ordering of constraints in an RCNF formula. By strategically grouping certain constraints together, the time required for combining smaller BDDs during construction can be decreased. This not only results in a smaller BDD but also reduces the time for subsequent operations. Constraint reordering, particularly when combined with dynamic reordering, can be highly efficient as it minimizes the number of nodes in intermediate results, thereby accelerating the sifting operation.

We implemented the following two different constraint ordering heuristics: Variable Frequency (VF-C) and Modified FORCE (M-FORCE).

4.2.1. Variable Frequency

We propose a concept of variable frequency ordering for constraints. The idea is to sort constraints according to the variables that they contain (similar to the VF ordering for variables). Specifically, it evaluates which variables are most influential in the ruleset and places the constraints that contain such variables at the beginning of a ruleset. Analog to the variable ordering with the same name, it evaluates the influence of the variables using frequency metric.

The proposed algorithm works the following way: we use the results of variable frequency ordering (Section 4.1.1). Then we analyze which variable in each constraint is the most frequent in the whole formula. If there are several variables with the same frequency, the one with the lowest index is taken. The constraint is then associated with this variable, and we sort the constraints according to the frequency of their associated variables.

It should be noted that the method induces a partition over the set of constraints based on the associated variables. The partition is specified in Equation 4. Let C be the set of all constraints, and $MFV(c)$ the associated variable of constraint c, i.e., the most frequent one.

$$P = \left\{ [c] \subseteq C \mid c' \in [c] \iff MFV(c) = MFV(c') \right\} \tag{4}$$

So, in addition to the most frequent variables being added to the overall BDD in the first iterations, this approach also groups the constraints with the same variables.

4.2.2. Modified FORCE

We present the heuristic that utilizes the idea of the FORCE variable ordering heuristic by applying it to the constraint ordering. It uses the concept of interconnected objects and placement by measuring their forces, but uses constraints as objects and redefines the interconnected objects as constraints having the same variables.

Basically, the algorithm is a modification of the FORCE variable ordering, the difference is in the types of objects that it takes as input. We build the hypergraph by using constraints as nodes and edges as sets of constraints that obtain the same variable. For a set of variables $|V|$, a set of constraints C and hypergraph edges E definition looks like this:

$$E = \left\{ e_v \subseteq C \mid v \in V, \forall c \in e_v : v \in c \right\} \tag{5}$$

We then use Formulas 2 and 3 and run the same algorithm to find a constraint placement that minimizes

the total span of the hypergraph (C, E). The number of iterations is bounded by $c \cdot \log |C|$

As we can assume that every variable is used at least once, it applies $|E| = |V|$. So, the worst-case time of each iteration is $|V| + |C|$ (analog to FORCE, we assume that average degrees of vertices and hyperedges are bound by a constant). The sorting takes $\Theta(|C|log|C|)$, so the worst-case performance of the algorithm is $O((|V| + |C|log|C|)log|C|)$.

The hypergraph construction is not as trivial as in the case of the origial FORCE heuristic. With the usage of mapping from variables to their parent constraints and a mapping of constraints to the edges attached to them, the overall time complexity of the hypergraph construction is $O(|C| + |V|)$, assuming that the number of variables in a constraint and the number of constraints containing a certain variable are bound by a constant.

The algorithm can be modified by assigning different weights to each constraint based on their influence on the output. These weights can be used in equations, such as the center of gravity and position formulas, to prioritize the faster movement of more influential constraints.

4.3. Diagram Construction

Given a constraint ordering, there can be several construction strategies on how to use that ordering to construct a BDD. The Apply algorithm is used to recursively create BDD from smaller BDDs starting with just variables. The order in which the algorithm is applied affects the construction time of a BDD and can be changed either by constraint reordering, which we discussed in the previous section, or by construction strategies.

In this section, we will present two construction strategies for RCNF formulas. We mention the commonly used Depth-First strategy and present the Merge Strategy.

4.3.1. Depth-First Strategy

A common straightforward approach: we append a smaller BDD to the overall diagram as soon as it gets constructed. For a RCNF formula, the strategy creates a constraint, appends it to the overall BDD tree and moves on with construction of the next constraint.

Constraint construction time stays bound by constraint size, whereby the overall tree increases its size with each iteration, which slows down the appending of the next BDDs.

4.3.2. Merge Strategy

We present the Merge Strategy as an alternative to the Depth-First. This strategy tries to solve the problem by dividing this problem into smaller ones.

First, we merge the first two constraints of them together, then we merge the resulting BDD with another BDD of two constraints. We subsequently continue merging the BDDs containing the same amounts of constraints, until the overall BDD is built. The construction order can be represented as a binary tree (Figure 1).

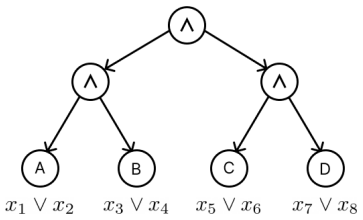

Figure 1: Merge construction tree

Example 4.1. Construction tree for a formula $(x_1 \vee x_2)(x_3 \vee x_4)(x_5 \vee x_6)(x_7 \vee x_8)$.

This way, constraint BDDs do not get appended in the exact order provided by this ordering, but global ordering is not influenced too much. For example, if we swap every two constraints (for instance, swap A and B, C and D in the example 4.1) the construction of their resulting tree will stay the same.

4.4. AMO Constraint Construction

An AMO constraint is not a binary operation, and its construction is not directly possible using frameworks like CUDD or libsdd that we will discuss later. Therefore, we have evaluated two ways on how to transform it into a form that uses Boolean operators.

The first way is to create a DNF representation of the AMO constraint, which is shown by Equation 6 and Equation 7 shows the whole formula f.

$$c_i = \Big(\bigwedge_{j=1}^{i-1} \neg l_j\Big) \wedge l_i \wedge \Big(\bigwedge_{j=i+1}^{n} \neg l_j\Big), \qquad i \in \{1, ..., n\} \tag{6}$$

$$f = \Big(\bigvee_{i=1}^{n} c_i\Big) \vee \Big(\bigwedge_{j=1}^{n} \neg l_i\Big) \tag{7}$$

In this case, the number of operations needed to build a BDD grows quadratically in the number of literals in the AMO formula.

Another way of presenting the AMO constraints is to use the XOR operation, which is also supported by the Apply algorithm. The formula constructed with XOR is shown by Equation 8:

$$\begin{gathered} f = (l_1 \oplus l_2 \oplus ... \oplus l_n \wedge c_{one}) \vee c_{zero} \\ c_{one} = \bigvee_{i=0}^{n} \neg l_i, \quad c_{zero} = \bigwedge_{i=0}^{n} \neg l_i \end{gathered} \tag{8}$$

The number of operations grows linear in the number of literals, which is makes it a more efficient method of building decision diagrams for AMO constraints.

4.4.1. SDD Vtrees and Variable Orders

As we discussed earlier, the SDDs variable ordering is more complex and is defined by vtrees instead of total variable ordering that is used in OBDDs.

Darwiche and Choi presented the following definition in [15]:

Definition 4.1. A vtree *dissects* a total variable order π if a left-right traversal of the vtree visits leaves (variables) in the same order as π.

In order to evaluate performance of the previously described BDD heuristics in the context of SDDs, we propose generating a total variable ordering, and then creating a vtree that dissects that ordering.

For one total variable ordering, there are many trees that can dissect it. Right-linear trees were discussed preciously in section 2. SDDs that respect right-linear vtrees correspond precisely to the OBDDs, and therefore they cannot lead to any enhanced performance. Another choice are left-linear trees and balanced trees. The balanced trees were used for evaluation in [15] and are also supported by the framework presented in this paper.

5. Implementation

In this chapter, we describe the frameworks that allow constructing BDDs and SDDs using methods described in Chapters 2 and 4.

5.1. CUDD

CUDD[1] (Colorado University Decision Diagram) is an open-source state-of-the-art package for BDD manipulation written in C [16]. Practically, the package allows presents an implementation of the Apply algorithm and all needed data structures like unique table and cache.

The package also contains implementations of dynamic ordering algorithms. Available algorithms include sifting, window permutations, group sifting and others. The chosen algorithm will be used every time the number of nodes has increased up to a given threshold, which is set automatically after each reordering.

5.2. libsdd

libsdd is an open-source library for SDD construction and performing queries on them [17]. The interface and functionality of this package are very similar to the CUDD, but with respect to the SDD specifics.

[1] https://davidkebo.com/cudd

We can create a vtree with a given total order and pass a parameter that specifies the type of the tree (right-linear, left-linear, balanced, vertical or random) that dissects a given total variable ordering. The library also allows automatic SDD minimization, which is similar to BDD dynamic ordering.

6. Evaluation

This chapter focuses on the evaluation of the algorithms described in the previous sections. We will take a look at the evaluated benchmarks and compare the results of different program configurations on these benchmarks.

For the evaluation, we use a GPU computer with 64GB RAM, Intel Core i9-9900K 3.60Ghz CPU and Nvidia GeForce RTX 2080 Ti GPU.

We used CUDD framework to implement the BDD construction and ordering and libsdd for SDD construction and ordering.

6.1. Benchmarks

One of the CAS Software applications is the Merlin CPQ configuration tool. It allows creating configuration rules using different complex relations between products and product parts. Fundamentally, the program has to solve the SAT problem for product configurations.

The evaluated benchmarks consist of real product configuration rules of different companies that use Merlin CPQ. Each benchmark corresponds to a company and contains rulesets that describe company products. Their rules were converted from the Merlin CPQ format into boolean formulas. Each ruleset is an independent RCNF formula saved as a DIMACS file. Table 1 shows the benchmarks and the number of variables and constraints in each of them.

6.2. Evaluation Goals

We evaluate the construction times as well as diagram sizes for the baseline approaches and different configurations of our approach. This includes several combinations of variants of variable ordering, constraint ordering, construction strategies, and an optional prior conversion of RCNF to CNF. We compare these configurations to the total construction time of SDDs using vtrees generated by the miniC2D top-down SDD compiler.

Configurations in the experiments are named using abbreviations. The construction strategy is specified only if it differs from the default depth-first strategy. All configurations utilize sifting dynamic ordering and XOR operations for AMO constraints, as shown in the Section 4.4.

Benchmark	#	Sum #vars	Median #vars	Sum #constraints	Median #constraints
vms	31	35258 (37985)	977 (1019)	15301 (27692)	295 (489)
campers	4	9713 (14543)	2430 (3546)	25592 (44386)	4754 (9121)
heating	9	24331 (42320)	2730 (4637)	73564 (237325)	7488 (26245)
forklifts	36	483226 (682785)	13637 (19166)	735319 (1972435)	19315 (53567)
printers	263	346017 (414854)	1309 (1459)	640915 (1488435)	1473 (2364)
boards	41	25600 (35857)	471 (630)	26596 (69165)	520 (1071)
trucks	50	1378761 (2466468)	27601 (46294)	6026775 (39474710)	118075 (736760)
plants	10	30638 (36922)	3589 (4398)	31119 (57871)	2592 (5266)
circuits	8	4486 (5194)	614 (694)	6349 (9869)	688 (1193)
Total	**452**	**2338030 (3736928)**	-	**7581530 (43381888)**	-

Table 1
Benchmark sizes. Numbers in brackets refer to the values of benchmarks converted to CNF

Abbreviations	Description
VF	Variable Frequency (variable ordering)
VF-C	Variable Frequency (constraint ordering)
FORCE (F)	FORCE (variable ordering)
M-FORCE (MF)	Modified-FORCE (constr. ordering)
MERGE (M)	"Merge" Construction Strategy
mC2D	miniC2D vtree Ordering
cnf	Input rulesets converted to CNF

Table 2
Abbreviations used for heuristics

6.3. Evaluation Methods

When constructing a decision diagram, choosing a suboptimal ordering and dealing with numerous constraints in a ruleset can result in construction times lasting several days. To manage this, we impose a time limit during the construction process and evaluate the algorithm's coverage. If the total construction time exceeds the limit, the process is stopped, and the ruleset is considered unconstructed. We assess the algorithms by comparing the number of constructed rulesets and analyzing various statistics for each algorithm. For example, we compare the ordering time for cases where ordering was completed within the time limit.

First, we evaluate different configurations with a 5-minute limit and then use the most performant ones for construction with a 1-hour limit.

6.4. Ordering Time

Figure 2 shows how many instances of the whole set of ruleset can be ordered in a time given by the y-axis. Here, we evaluate only individual heuristics (with an exception to M-FORCE/FORCE/MERGE/cnf), since the present variable and constraint ordering heuristics are independent, and configurations with both methods being used will just have an ordering time that equals the sum of variable and constraint orderings. In contrast, the CNF benchmark changes the ordering time, since the number of variables and clauses is higher (as can be seen from Table 1).

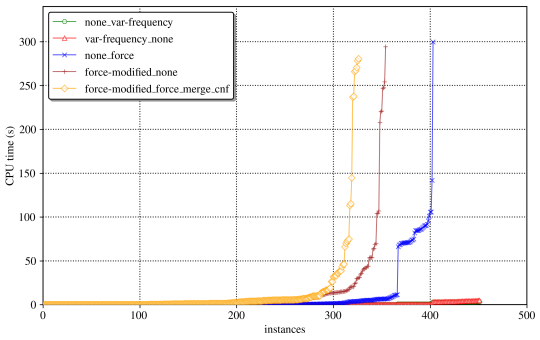

Figure 2: Time needed to find ordering using different configurations in 5 min limit

We can observe that the only configurations managing to order all instances under the limit are either VF or VF-C heuristics. The FORCE variable heuristic comes close to ordering all instances. M-FORCE constraint heuristic is even less performant, which can be explained with the fact that the number of constraints has more influence here and this number is bigger than the number of variables in the given benchmarks. The combination of M-FORCE and FORCE heuristics applied to the rulesets converted to CNF is the least performant of all, since the number of clauses in CNF rulesets is normally bigger than the number of constraints in original RCNF rulesets.

6.5. Coverage Statistics

In this section, we will evaluate the decision diagram construction with a time limit.

6.5.1. 5-Minute Limit

In Table 3 we can see the results for BDD construction with variable ordering, constraint ordering, and different construction strategies. `trucks`, `heating` and `forklifts` benchmarks did not result in successfully constructed rulesets and are therefore not present. Merge

Bench	R R	∅ ∅	VF-C VF	MF F M *cnf*	VF-C VF M	MF F	MF F M
vms	22	24	24	26	24	24	27
campers	2	2	2	2	2	2	2
printers	82	85	99	119	125	144	162
boards	15	34	25	32	34	33	36
plants	0	0	1	1	3	0	3
circuits	4	7	7	8	7	8	8
Total	**125**	**152**	**158**	**188**	**195**	**211**	**238**

Table 3
5-minute coverage results with different constraint orderings (R, ∅, VF-C, and MF), variable orderings (R, ∅, VF, and F), construction strategies (M), and optional conversion to cnf

Bench	SDD MF F M	BDD VF-C VF	SDD MF F	BDD VF-C VF M	BDD MF F	SDD mC2D *cnf*	BDD MF F M
plants	2	2	0	3	2	8	3
heating	0	0	3	0	0	0	0
circuits	7	7	0	8	8	7	8
trucks	0	0	8	0	0	0	0
vms	24	24	24	25	29	27	29
boards	26	31	29	35	36	33	36
printers	83	122	139	145	168	170	176
campers	2	2	2	2	2	2	2
Total	**144**	**188**	**205**	**218**	**245**	**247**	**254**

Table 5
Decision diagrams constructed in 1-hour limit

strategy consistently outperforms depth-first in every configuration. The best performance was achieved by the M-FORCE/FORCE/MERGE configuration, which also showed better results when the formula was given in RCNF rather than CNF.

Bench	∅ ∅ *cnf*	MF F M	MF F	∅ ∅	mC2D *cnf*
vms	11	23	22	22	27
campers	2	2	2	2	2
printers	72	72	96	113	168
boards	17	21	25	27	33
plants	0	1	3	0	7
circuits	3	7	8	6	7
Total	**105**	**126**	**156**	**170**	**244**

Table 4
5-minute coverage results for SDDs with different constraint orderings (∅, and MF), variable orderings (∅, and F), construction strategies (M), and optional conversion to cnf, including the baseline mC2D

In Table 4 we can see that the heuristics do not work as well for SDDs as for BDDs. Configurations using variable and constraint heuristics do not improve coverage, suggesting unsuitability for constructing efficient vtrees for SDDs. The best result is shown by the construction using vtrees created by miniC2D tool. We can see that M-FORCE/FORCE/MERGE BDD construction (Table 3) almost reaches the performance of best SDD configuration. `trucks`, `heating` and `forklifts` benchmarks again did not yield constructed rulesets.

6.5.2. 1-Hour Limit

In Table 5 we can see coverage results for the 1-hour limit. The table contains both results for SDDs and BDDs

The M-FORCE/FORCE/MERGE configuration manages to outperform SDDs constructed with vtrees from miniC2d. M-FORCE/FORCE also shows comparable results.

We can see that even the best configuration shows only 56,1% done rulesets. `Forklifts` did not yield any constructed rulesets and is not present in the table and `heating` and `trucks` yielded just a few. From Table 1 we can see that these benchmarks are the biggest in terms of both variable number and constraint number, which leads to long ordering time. Still, big rulesets from `heating` and `trucks` are only constructed by MF/F, which manages to create the most optimal ordering.

We also evaluated the numbers of nodes for the diagrams constructed with the 1-hour limit. Here, we take only the subset of rulesets that are covered by each evaluated configuration.

Bench	SDD MF F M	BDD VF-C VF	SDD MF F	BDD VF-C VF M	BDD MF F	SDD mC2D *cnf*	BDD MF F M
plants	63464	570814	5559	78494	1112080	**4745**	390098
printers	2155	7573	**1908**	6935	4668	3238	4559
vms	3937	4759	**2091**	5302	3280	3050	6170
circuits	**3050**	8141	3130	9946	12993	4975	29231
boards	5552	11306	**2879**	12200	6024	4611	6599
campers	3202	15079	**1899**	16447	2177	2015	7472

Table 6
Median values of node counts for each configuration

In Table 6 we can see that SDD M-FORCE/FORCE configuration has the lowest median node count on 4 benchmarks out of 6. SDD configurations generally have better scores than BDD. The best performance BDD configuration is M-FORCE/FORCE.

6.6. Evaluation Summary

M-FORCE/FORCE/MERGE configuration delivers the best results for overall BDD construction time and outperforms all SDD configurations in 1-hour limit. However, it provides results that have one of the highest nodes counts. The presented configurations also improve construction time for SDDs in 1 hour limit, but still are inferior to

some BDD configurations. In opposition to BDD, M-FORCE/FORCE results in the smallest number of nodes for SDD.

Some benchmarks (`trucks`, `heating`, `forklifts`) could not be constructed within the limit due to complexity that can be observed from number of variables and constraints. Such big instances need more time to be compiled or more complex ordering heuristics to be applied.

Overall, with presented ordering heuristics, BDDs are much more efficient in modelling the rulesets and show promising results for use cases of knowledge compilation.

7. Conclusion

In this paper, we examined methods to enhance the construction time and size of BDDs for RCNF formulas. We presented variable ordering and constraint ordering methods that utilize the ideas of commonly used variable ordering methods. Furthermore, we considered different tree construction strategies. Additionally, we discussed the application of all described methods for SDD construction using vtrees.

We evaluated heuristics on RCNF benchmarks, assessing coverage in different time limits and determining the best results for each configuration, and found that Modified-FORCE and FORCE can greatly improve the BDD construction time. Furthermore, we applied the variable ordering heuristics to construct balanced vtrees for SDD construction, and results showed that FORCE and Modified-FORCE result in the best decision diagram size among all configurations.

References

[1] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, L.-J. Hwang, Symbolic model checking: 1020 states and beyond, Information and computation 98 (1992) 142–170.

[2] D. E. Knuth, The Art of Computer Programming, Volume 4, Fascicle 1B: Binary Decision Diagrams, Addison-Wesley Professional, 2009.

[3] R. E. Bryant, Binary Decision Diagrams, Springer International Publishing, Cham, 2018, pp. 191–217. URL: https://doi.org/10.1007/978-3-319-10575-8_7. doi:10.1007/978-3-319-10575-8_7.

[4] D. Sieling, I. Wegener, Reduction of obdds in linear time, Inf. Process. Lett. 48 (1993) 139–144.

[5] Bryant, Graph-based algorithms for boolean function manipulation, IEEE Transactions on Computers C-35 (1986) 677–691. doi:10.1109/TC.1986.1676819.

[6] A. Darwiche, Sdd: A new canonical representation of propositional knowledge bases, in: IJCAI, 2011.

[7] S. Bova, Sdds are exponentially more succinct than obdds, CoRR abs/1601.00501 (2016). URL: http://arxiv.org/abs/1601.00501. arXiv:1601.00501.

[8] T. Hadzic, S. Subbarayan, R. M. Jensen, H. R. Andersen, J. Møller, H. Hulgaard, Fast backtrack-free product configuration using a precompiled solution space representation, in: Proceedings from the International Conference on Economic, Technical and Organisational aspects of Product Configuration Systems, Technical University of Denmark (DTU), 2004, pp. 133–140.

[9] M. Rice, S. Kulhari, A survey of static variable ordering heuristics for efficient bdd/mdd construction (2008) 13.

[10] F. Aloul, I. Markov, K. Sakallah, Mince: A static global variable-ordering heuristic for sat search and bdd manipulation, JOURNAL OF UNIVERSAL COMPUTER SCIENCE 10 (2004) 1562–1596.

[11] R. L. Rudell, Dynamic variable ordering for ordered binary decision diagrams, Proceedings of 1993 International Conference on Computer Aided Design (ICCAD) (1993) 42–47.

[12] U. Oztok, A. Darwiche, A top-down compiler for sentential decision diagrams, in: Proceedings of the 24th International Conference on Artificial Intelligence, IJCAI'15, AAAI Press, 2015, p. 3141–3148.

[13] H. R. Andersen, T. Hadzic, D. Pisinger, Interactive cost configuration over decision diagrams, Journal of Artificial Intelligence Research 37 (2010) 99–139.

[14] F. A. Aloul, I. L. Markov, K. A. Sakallah, Force: A fast and easy-to-implement variable-ordering heuristic, in: Proceedings of the 13th ACM Great Lakes Symposium on VLSI, GLSVLSI '03, Association for Computing Machinery, New York, NY, USA, 2003, p. 116–119. URL: https://doi.org/10.1145/764808.764839. doi:10.1145/764808.764839.

[15] A. Choi, A. Darwiche, Dynamic minimization of sentential decision diagrams, Proceedings of the AAAI Conference on Artificial Intelligence 27 (2013) 187–194. URL: https://ojs.aaai.org/index.php/AAAI/article/view/8690. doi:10.1609/aaai.v27i1.8690.

[16] F. Somenzi, CUDD User's Manual, 2005. URL: http://web.mit.edu/sage/export/tmp/y/usr/share/doc/polybori/cudd/node3.html.

[17] A. Choi, A. Darwiche, SDD Advanced-User Manual Version 2.0, Automated Reasoning Group Computer Science Department University of California, 2018. URL: http://reasoning.cs.ucla.edu/sdd/doc/sdd-advanced-manual.pdf.