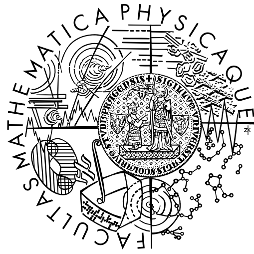

FACULTY OF MATHEMATICS AND PHYSICS
Charles University

MASTER THESIS

Marek Zelený

Effective implementation of DP elimination

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the master thesis: RNDr. Petr Kučera, Ph.D.
Study programme: Computer Science - Artificial Intelligence

Prague 2024

I declare that I carried out this master thesis on my own, and only with the cited sources, literature and other professional sources. I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In .... date ....

Author's signature

Title: Effective implementation of DP elimination

Author: Marek Zelený

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: RNDr. Petr Kučera, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract: We develop an efficient implementation of Davis-Putnam (DP) elimination, an algorithm for eliminating variables from a conjunctive normal form (CNF) formula. We use zero-suppressed binary decision diagram (ZBDD) for representing CNF formulas. Our focus is on evaluating the effect of minimising the formula during DP elimination by removing absorbed clauses. We also want to find a suitable heuristic for selecting the order of eliminated variables. Our motivation is compiling a CNF formula into a formula that is propagation-complete (PC). The formula can be encoded into decomposable negation normal form (DNNF), then back into CNF that contains auxiliary variables and implements domain consistency. Our program can be used to eliminate these auxiliary variables, thus obtaining a PC formula equivalent to the original formula.

Keywords: DP resolution, satisfiability, conjunctive normal form, ZBDD

Název práce: Efektivní implementace DP eliminace

Autor: Marek Zelený

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí bakalářské práce: RNDr. Petr Kučera, Ph.D., Katedra teoretické informatiky a matematické logiky

Abstrakt: Vyvinuli jsme efektivní implementaci Davis-Putnamovy (DP) eliminace, algoritmu, který eliminuje proměnné z formule v konjunktivní normální formě (KNF). Použili jsme tzv. zero-suppressed binární rozhodovací diagramy (ZBDD) pro reprezentaci KNF formulí. Zaměřili jsme se na zhodnocení efektu průběžné minimalizace formule odstraňováním absorbovaných klauzulí. Také jsme hledali vhodnou heuristiku pro pořadí, ve kterém se proměnné eliminují. Naší motivací je kompilace KNF formule do formy úplné vůči jednotkové propagaci, tzv. propagation-complete (PC). Formuli můžeme zakódovat do DNNF (decomposable negation normal form), poté zpět do KNF obsahující pomocné proměnné, která je tzv. doménově konzistentní. Náš program lze použít k eliminaci těchto pomocných proměnných, čímž získáme PC formuli ekvivalentní s původní formulí.

Klíčová slova: DP rezoluce, splnitelnost, konjunktivní normální forma, ZBDD

Contents

1 Introduction **6**
1.1 Preliminaries . . . 7
1.1.1 Logical Formulas . . . 7
1.1.2 Boolean Functions and Decision Diagrams . . . 8
1.1.3 Combination Sets . . . 9
1.1.4 DP Procedure . . . 10

2 ZBDDs **11**
2.1 Reduced Ordered BDDs . . . 11
2.2 Zero-suppressed BDDs . . . 12
2.2.1 Attributed Edges . . . 13
2.2.2 Basic Operations . . . 14
2.2.3 Additional Operations . . . 15

3 DP Elimination **19**
3.1 Using ZBDDs . . . 19
3.2 Variable Selection . . . 22
3.2.1 Ordering-based Methods . . . 22
3.2.2 Low-hanging Fruits . . . 22
3.2.3 Optimisation-based Methods . . . 24
3.2.4 Dynamic Methods . . . 26
3.3 Removing Tautologies . . . 26
3.4 Unit Propagation . . . 27
3.5 Formula Minimisation . . . 29
3.5.1 Absorbed Removal Over ZBDD . . . 30
3.5.2 Absorbed Removal Over Watched Literals . . . 31
3.5.3 Incremental Absorbed Removal . . . 32
3.5.4 More Efficient Absorption Detection . . . 33
3.6 Putting It All Together . . . 34

4 ZBDD Implementation **36**
4.1 Sylvan Library . . . 36
4.1.1 Node Sharing, Result Caching, and Garbage Collection . . . 37
4.1.2 Lace Framework . . . 38
4.2 Literal Mapping . . . 39
4.3 Algorithms . . . 40
4.4 Building ZBDDs . . . 42
4.4.1 Logarithmic Merging . . . 46

5 Programming Documentation **48**
5.1 Technical Decisions . . . 48
5.2 Project Structure . . . 48
5.3 External Libraries . . . 48
5.4 Software Architecture . . . 49
5.4.1 Data Structures . . . 49

5.4.2 Algorithms . . . 50
5.4.3 Metrics . . . 51
5.4.4 IO . . . 52
5.5 Experiment Execution and Result Processing . . . 52

6 User Documentation **54**
6.1 Prerequisites . . . 54
6.2 Compilation and Tests . . . 54
6.3 Program Stack Size Limit . . . 55
6.4 Parameters and Options . . . 55
6.4.1 Files . . . 56
6.4.2 Algorithm . . . 56
6.4.3 Complete Minimisation . . . 56
6.4.4 Partial Minimisation . . . 57
6.4.5 Incremental Absorption Removal . . . 58
6.4.6 Stop Conditions . . . 59
6.4.7 Sylvan . . . 60
6.5 Configuration Files . . . 60
6.6 File Formats . . . 61
6.6.1 DIMACS CNF . . . 61
6.6.2 Metrics Export . . . 62
6.7 Examples . . . 65

7 Experiments **67**
7.1 Inputs . . . 67
7.2 Setup . . . 68
7.3 Algorithm Breakdown . . . 69
7.4 Experiment 1: Absorption Removal . . . 73
7.5 Experiment 2: Variable Selection . . . 76
7.6 Experiment 3: Parallelisation . . . 77
7.7 Experiment 4: Comparison With PCCompile . . . 77

8 Conclusion **80**

Bibliography **82**

A Attachments **85**
A.1 Software and Data Attachments . . . 85

1 Introduction

The DP procedure, originally proposed in the 1960s by Davis and Putnam [1], is one of the first algorithms deciding satisfiability of a formula in propositional logic. It takes a CNF formula and eliminates its variables one by one, until it either derives an empty clause or ends up with an empty formula. While a single variable can be eliminated in polynomial time, the formula can grow exponentially before all its variables are eliminated. Because of that, it is not practical to use the DP procedure for deciding the SAT problem and more space-efficient methods have been developed instead. However, there might be some other applications for DP which only need to eliminate some of the formula's variables.

One such application is obtaining a propagation-complete (PC) formula, a concept introduced by Bordeaux and Marques-Silva [2]. This property can be advantageous for constraint programming solvers using SAT encodings as a consistency checking or propagation technique. Several algorithms were introduced for compiling a CNF formula into an equivalent PC formula, most of those based on iteratively adding new clauses, for example by Bordeaux and Marques-Silva [2] or Kučera [3]. The issue with these algorithms is that finding these new clauses is computationally expensive (NP-complete as shown by Babka et al. [4]).

Kučera and Savický [5] introduced a compilation algorithm of a formula in DNNF into a PC formula with polynomial time complexity. This offers a different approach for obtaining a PC formula — encode a CNF formula into DNNF (e.g. using encodings by Darwiche [6] or Lagniez and Marquis [7]), then compile the result back into CNF. Unfortunately, the intermediate DNNF formula can be significantly larger than the original CNF formula. Moreover, the compilation from DNNF to a PC CNF further grows the formula's size while also introducing auxiliary variables. These side-effects negatively impact the advantages of having a PC formula in the first place. In order to mitigate the formula growth, we could use an encoding from DNNF to CNF that implements domain consistency instead of propagation completeness, resulting in a formula that is only PC on the original variables, but not on the auxiliary ones. There are several such encodings, some of which are described by Abío et al. [8]. This is where the DP procedure steps in — we can use it to eliminate the auxiliary variables. Because DP elimination preserves domain consistency of the remaining variables, we will end up with a PC formula equivalent to the original CNF formula before it was encoded into DNNF.

The goal of our thesis is to efficiently implement DP elimination that can eliminate some specified range of variables from a given CNF formula. While encodings of DNNF implementing domain consistency are in general smaller than encodings implementing PC, the resulting CNF formula can still be significantly larger than the original one. In order to handle large formulas, we want to use efficient data structures for representing them. We were inspired by Chatalic and Simon [9], who use a data structure called ZBDD (introduced by Minato [10]) to represent CNF formulas and perform the DP procedure over them. To fight the bloating effect of DP elimination, we want to minimise the formula during the course of the algorithm. Most notably, we would like to try removing absorbed clauses (defined by Atserias et al. [11]). A necessary part of the thesis is developing

heuristics for selecting the order in which variables are eliminated.

As part of our thesis, we present results of experiments performed with our implementation of DP elimination on formulas obtained by applying the CNF $\to$ DNNF $\to$ CNF pipeline described earlier. We compare performance of various configurations and parameter settings of the developed program. The objective of these experiments is to determine efficiency of the implemented techniques, namely absorption removal and variable selection heuristics. The experiments also serve as a feasibility study of our approach with respect to the size of the input formulas.

In our thesis, after basic definitions and notation in Section 1.1, we introduce ZBDDs as a data structure for representing CNF formulas in Chapter 2. Then we analyse the DP elimination algorithm with focus on using ZBDDs, including possible optimisations and heuristics, in Chapter 3. After that, we move on to more technical details regarding ZBDDs and their efficient implementation in Chapter 4. We follow up with programming and user documentations of our program in chapters 5 and 6. Finally, we present and analyse results of experiments conducted with the program in Chapter 7.

1.1 Preliminaries

Before describing our approach in detail, we start by some general descriptions, basic definitions, and notation used in our thesis.

1.1.1 Logical Formulas

Let $\mathcal{X}$ be a set of propositional variables. Variable $x \in \mathcal{X}$ can occur in a propositional formula either as a *positive literal* x, or a *negative literal* $\neg x$. We denote $\mathrm{lit}(\mathcal{X})$ the set of literals over $\mathcal{X}$; a literal $l \in \mathrm{lit}(\mathcal{X})$ is either x or $\neg x$ for some $x \in \mathcal{X}$. A *(partial) assignment* $\mathfrak{A}$ is a (partial) mapping from a set of variables to Boolean values, i.e. $\mathfrak{A} : \mathcal{X} \dashrightarrow \{0, 1\}$; we sometimes denote $\bar{x} = \mathfrak{A}(x)$ where $\mathfrak{A}$ is obvious from (or unimportant for) the context.

In our thesis, we are mostly concerned with propositional formulas in *conjunctive normal form (CNF)*. A *clause* is a disjunction of literals, i.e. $C = (l_1 \vee \cdots \vee l_n)$. It is often practical to think of clauses as sets of literals $C = \{l_1, \ldots, l_n\}$, so that we can use notations such as $l \in C$, $C_1 \subset C_2$, etc. Note that in the set notation, the empty set $\emptyset$ denotes the empty clause, which represents a contradiction ($\bot$). A formula φ in CNF is a conjunction of clauses $\varphi = C_1 \wedge \cdots \wedge C_n$. Again, it is practical to consider CNF formulas being sets of clauses so that $C \in \varphi$ denotes a clause in formula φ. Here the empty set $\emptyset$ denotes the empty formula, representing a tautology ($\top$). We denote $\mathrm{Var}(\varphi)$ the set of variables occurring in formula φ.

We use $\varphi[\mathfrak{A}]$ to denote formula φ after applying assignment $\mathfrak{A}$ to variables in φ. Assignment $\mathfrak{A}$ is *complete* for formula φ if $\mathrm{Var}(\varphi) \subseteq \mathrm{dom}(\mathfrak{A})$. Assignment $\mathfrak{A}$ *satisfies* formula φ if $\varphi[\mathfrak{A}]$ evaluates to *true* (1). Formula φ is *satisfiable* if there exists an assignment that satisfies it; otherwise φ is *unsatisfiable* (a contradiction).

Formula g is an *implicate* of formula φ, denoted $\varphi \models g$, if every assignment satisfying φ also satisfies g. More often we use special cases like $\varphi \models C$ or $\varphi \models l$ for clause C and literal l. Clause C_2 is *subsumed* by clause C_1 if $C_1 \subseteq C_2$. Note that every subsumed clause is an implicate of its subsuming clause, i.e. if $C_1 \subseteq C_2$

then $C_1 \models C_2$. We use $\varphi \vdash_1 l$ to denote that literal l can be inferred by unit propagation on formula φ. Unit propagation is sound but not complete, i.e. $\varphi \vdash_1 l$ implies $\varphi \models l$, but not the other way around.

The concept of absorbed clauses was introduced by Atserias et al. [11], but we will use a more intuitive definition from Pipatsrisawat and Darwiche [12]. Let φ be a CNF formula, and $C = (l_1 \vee \cdots \vee l_n)$ be an implicate of φ. The clause C is *empowering* w.r.t. φ if one of its literals l_i, called *empowered literal*, is such that:

$$\varphi \wedge \bigwedge_{j \in 1..n,\, j \neq i} \neg l_j \quad \not\vdash_1 \quad \bot\,, \text{ and}$$
$$\varphi \wedge \bigwedge_{j \in 1..n,\, j \neq i} \neg l_j \quad \not\vdash_1 \quad l_i\,.$$

C is called *absorbed* by φ if it has no empowered literal. Note that Bordeaux and Marques-Silva [2] use a similar definition of empowerment and absorption, but they omit the first condition, i.e. unit derivation of a contradiction.

1.1.2 Boolean Functions and Decision Diagrams

Boolean function $f(\mathbf{y})$ on variables $\mathbf{y} = (y_1, \dots, y_n)$ is $f : \{0,1\}^n \to \{0,1\}$. There are many ways how to represent a Boolean function. Propositional formulas, even in CNF, are not a suitable representation in many applications — they are not unique (e.g. $f(\mathbf{y}) = 0$ is represented by any unsatisfiable formula), equivalence checking is hard, and in general they are very hard to interpret. Many representation languages form a subclass of *negation normal form (NNF)*:

- rooted, directed acyclic graph (DAG),
- leaves are labelled with literals y_i or $\neg y_i$,
- inner nodes are labelled with $\wedge$ or $\vee$.

There are no restrictions on node arity, i.e. nodes can have arbitrarily many children. For an inner node u we denote $\mathrm{Var}(u)$ the set of variables from $\mathbf{y}$ that are reachable from u by a directed path.

Evaluating a NNF formula given some assignment of $\mathbf{y}$ can be done by a simple recursive traversal of the DAG from its root, performing the Boolean operation in each node according to its label. However, NNF is still very general and various restrictions are often considered. A very common restriction is *decomposable NNF (DNNF)*: for each conjunction $(u_1 \wedge \cdots \wedge u_n)$, the conjuncts do not share variables (i.e. $\mathrm{Var}(u_i) \cap \mathrm{Var}(u_j) = \emptyset$ for $i \neq j$). This restriction is in fact very strong — checking satisfiability of a DNNF formula can be done in polynomial time, while the problem is NP-complete for general NNF.

Another possible representation of Boolean functions is a *binary decision diagram (BDD)*, reviewed by Drechsler and Sieling [13]. Typically when BDDs are used, they are in fact *ordered BDDs (OBDDs)* for some variable ordering. Let us assume the following ordering on $\mathbf{y}$: $y_1 < \cdots < y_n$. Same as NNF, an OBDD is a rooted DAG, but nodes have different properties:

- terminal nodes (leaves) are labelled 0 (false) and 1 (true),
- inner nodes are labelled with variables y_i,

- each inner node has two outgoing edges: 0-edge and 1-edge,
- for every (0- or 1-)edge (y_i, y_j) it holds that $y_i < y_j$.

The semantics of an OBDD are straightforward — a directed path from the root to the 0-terminal represents the function's evaluation to 0, a path to the 1-terminal evaluates to 1. On that path, taking the 0-edge from a node labelled y_i corresponds to $y_i = 0$, 1-edge corresponds to $y_i = 1$. If some variable y_i is not encountered on the path, then both assignments $y_i = 0$ and $y_i = 1$ lead to the same evaluation. Similarly to NNF, general OBDDs can be ambiguous, which is typically solved with some additional restrictions. We describe some of those restrictions in Chapter 2.

Despite the advantages these representations of Boolean functions bring, it is sometimes still desirable to encode the function as a CNF formula and solve it as a SAT problem. In these situations, the original representation of the function $f(\mathbf{y})$ can be converted into a CNF formula φ that represents $f(\mathbf{y})$ by using existentially quantified helper variables. Then φ is called a *CNF encoding* of $f(\mathbf{y})$ and the existentially quantified variables are called *auxiliary* variables. As a consequence, $\mathrm{Var}(\varphi) \supset \{y_1, \dots, y_n\}$. The encoding can have different properties; we are interested in two properties in particular (defined for example by Abío et al. [8]):

- encoding φ *implements domain consistency* when for each literal $l \in \mathrm{lit}(\mathbf{y})$ and every partial assignment $\mathfrak{A}$ to variables $\mathbf{y}$, if $f(\mathbf{y})[\mathfrak{A}] \models l$ then $\varphi[\mathfrak{A}] \vdash_1 l$,[1]
- encoding φ *implements propagation completeness* when for each literal $l \in \mathrm{lit}(\mathrm{Var}(\varphi))$ and every partial assignment $\mathfrak{A}$ to variables $\mathrm{Var}(\varphi)$, if $\varphi[\mathfrak{A}] \models l$ then $\varphi[\mathfrak{A}] \vdash_1 l$.

1.1.3 Combination Sets

Given a set of elements (a *domain*) $\mathcal{D} = \{a_1, \dots, a_n\}$, a *combination* over $\mathcal{D}$ is a subset of $\mathcal{D}$. A *combination set* S over $\mathcal{D}$ is a set of combinations over $\mathcal{D}$; essentially, it is a subset of the power set of $\mathcal{D}$ ($S \subseteq \mathbb{P}(\mathcal{D})$). A combination over $\mathcal{D}$ can be represented by an n-bit binary vector $\bar{b} = (b_1, \dots, b_n)$, each bit $b_i \in \{0, 1\}$ expressing whether the element a_i is included in the combination or not. Consequently, a combination set S over $\mathcal{D}$ can be represented by its *characteristic function* $\chi_S(\mathbf{y})$ — a Boolean function on $\mathbf{y} = (y_1, \dots, y_n)$[2]. For every n-bit vector $\bar{b}$, $\chi_S(\bar{b})$ determines whether the combination represented by the vector is included in the combination set or not. If $b_i = 1$, i.e. element a_i does belong to the combination, we say that the variable y_i is *active* for that combination.

Combination sets can be used as a representation of CNF formulas. Given a formula φ containing variables $\mathrm{Var}(\varphi)$, we can represent φ as a combination set over $\mathrm{lit}(\mathrm{Var}(\varphi))$, each combination corresponding to a clause in φ. We call

[1] We have not formally defined $\mathrm{lit}(\mathbf{y})$, $f(\mathbf{y})[\mathfrak{A}]$, and $f(\mathbf{y}) \models l$ for a Boolean function $f(\mathbf{y})$ and a vector of variables $\mathbf{y}$, but the definitions of these operators can be easily extended from those presented in Section 1.1.1.

[2] Note the distinction between the vectors $\bar{b}$ and $\mathbf{y}$ — b_i is a binary value, while y_i is a binary variable.

the characteristic function of such combination set the *characteristic function of formula* φ and denote it $\chi_\varphi(\mathbf{y})$. Note that we have overloaded the term *variable* here — variables of the characteristic function (corresponding to elements in the domain of a combination set) are different from variables in formula φ. More precisely, variables $\mathbf{y}$ of the characteristic function $\chi_\varphi(\mathbf{y})$ correspond to the *literals* in φ. To avoid confusion, we refer to variables of characteristic functions as *d-variables* (short for *domain variables*) in the rest of our thesis.

1.1.4 DP Procedure

The *Davis-Putnam (DP) procedure* was already mentioned in the introduction. Instead of the original description by Davis and Putnam [1], we will use a description by Chatalic and Simon [9]. Given propositional formula φ, expressed in CNF:

1. choose a variable $x \in \mathrm{Var}(\varphi)$,
2. replace all clauses in φ containing literals x or $\neg x$ with all binary resolvents on x (cut elimination of x),
3. remove all subsumed clauses from φ,
4. a. if φ is reduced to the empty clause, the original formula is unsatisfiable,

   b. if φ is empty, the original formula is satisfiable,

   c. otherwise, repeat steps 1. – 4. for the new φ.

For better understanding of the algorithm, we will illustrate its steps on an example. Let $\varphi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_2) \wedge (x_2 \vee \neg x_3) \wedge (x_3 \vee x_4)$.

1. We choose x_2 for elimination.
2. φ has one clause containing the positive literal x_2, let us denote this set $\varphi^+_{x_2} = \{x_2 \vee \neg x_3\}$, and two clauses containing the negative literal $\neg x_2$, denoted $\varphi^-_{x_2} = \{x_1 \vee \neg x_2 \vee \neg x_3, \neg x_2\}$. Binary resolvents $\varphi^+_{x_2}$ and $\varphi^-_{x_2}$ (i.e. resolvents of all pairs of clauses $C_1 \in \varphi^+_{x_2}$ and $C_2 \in \varphi^-_{x_2}$) are the following: $x_1 \vee \neg x_3$, $\neg x_3$. We replace $\varphi^+_{x_2}$ and $\varphi^-_{x_2}$ in φ with these resolvents and get a new $\varphi = (x_1 \vee \neg x_3) \wedge (\neg x_3) \wedge (x_3 \vee x_4)$.
3. In the new φ, the clause $x_1 \vee \neg x_3$ is subsumed by the clause $\neg x_3$, so it is removed, leaving us with $\varphi = (\neg x_3) \wedge (x_3 \vee x_4)$.
4. φ is neither empty, nor does it contain the empty clause, so we go back to Step 1.

2 ZBDDs

In this chapter, we will introduce zero-suppressed binary decision diagrams as a data structure for efficiently representing and manipulating CNF formulas. We will describe algorithms used for performing DP elimination on those formulas; their usage will be described in Chapter 3, implementation details of the data structure will be discussed in Chapter 4.

2.1 Reduced Ordered BDDs

In Section 1.1.2 we described a data structure called ordered binary decision diagram (OBDD) which can be used for representing Boolean functions. We noted that the general definition of an OBDD is too permissive, allowing different OBDDs to represent the same function. In order to prevent that, as well as to achieve some other advantages in terms of complexity of some queries, OBDDs are typically used in a reduced form called *reduced ordered BDD (ROBDD)*. This is achieved by applying two reduction rules:

1. eliminate all nodes with both edges pointing to the same node,
2. share all equivalent sub-graphs.

When neither of those two rules can be further applied to an OBDD, it is called a ROBDD. Consequently, ROBDDs are in this sense minimal, i.e. they cannot be further reduced while representing the same function. For a fixed ordering of variables, a Boolean function is uniquely represented by its ROBDD (see Drechsler and Sieling [13] for details). An example of a ROBDD is shown in Figure 2.1 — it represents a characteristic function of a combination set (see Section 1.1.3). We use dashed lines for 0-edges and full lines for 1-edges in our examples.

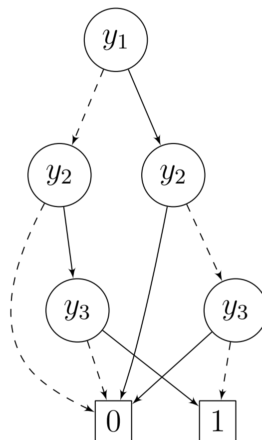

Figure 2.1 ROBDD representing combination set $\{\{a_1\},\{a_2,a_3\}\}$ over domain $\{a_1,a_2,a_3\}$[1].

[1]The reason for nodes in the ROBDD being labelled y_i instead of a_i originates in the definitions from Section 1.1.3 — a_i is an arbitrary element in a combination set (it does not have to be a variable); y_i is a d-variable corresponding to a_i.

2.2 Zero-suppressed BDDs

In our thesis, we want to use BDDs for representing logical formulas in CNF. We described in Section 1.1.3 that a CNF formula can be thought of as a combination set, and that such combination set can be represented by its characteristic function. ROBDDs are efficient for representing Boolean functions in general, but characteristic functions are a special case. Perhaps the largest inconvenience is that ROBDDs representing characteristic functions are domain-specific. The combination set $S = \{\{a_1\}, \{a_2, a_3\}\}$ shown in Figure 2.1 will have a different ROBDD representation over the domains $\mathcal{D}_3 = \{a_1, a_2, a_3\}$ and $\mathcal{D}_4 = \{a_1, a_2, a_3, a_4\}$. The reason is that if we used the same ROBDD in both cases, over $\mathcal{D}_4$ it would actually represent $S' = \{\{a_1, a_4\}, \{a_2, a_3, a_4\}\}$ instead of S, since a missing node in a ROBDD implies both edges from that node having the same destination (see Reduction rule 1 for ROBDDs), i.e. both $y_4 = 0$ and $y_4 = 1$ would yield the same result for function $f(\mathbf{y})$, $\mathbf{y} = (y_1, y_2, y_3, y_4)$. For comparison, the ROBDD representing S over $\mathcal{D}_4$ is shown in Figure 2.2. As a consequence, d-variables that are only active in very few combinations still need to have their node on almost every path of the ROBDD. This is particularly impactful for sets of sparse combinations over large domains, i.e. sets where most combinations are very small compared to the domain. Unfortunately, that is precisely the most common case with CNF formulas.

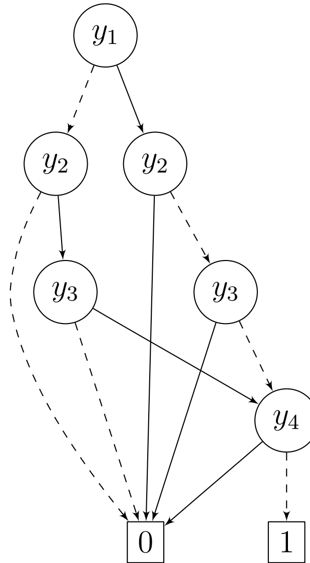

Figure 2.2 ROBDD representing combination set $\{\{a_1\}, \{a_2, a_3\}\}$ over domain $\{a_1, a_2, a_3, a_4\}$.

Zero-suppressed BDD (ZBDD), introduced by Minato [10] and summarised in more detail by Minato [14], is an alternative to the widely used ROBDD. It is also an OBDD, but it uses a different set of reduction rules:

1. eliminate all nodes with their 1-edge pointing to the 0-terminal, then connect their ingoing edges to the other (0-edge) sub-graph directly,

2. share all equivalent sub-graphs (same as for ROBDDs).

Notice that the first reduction rule is asymmetric — we do not eliminate nodes whose 0-edge points to a terminal node. This rule also changes how the diagram is evaluated, as it no longer holds that a missing variable y_i on a path means both $y_i = 0$ and $y_i = 1$ lead to the same evaluation, which was the case for OBDDs (see Section 1.1.2). Instead, the missing variable implies the path's evaluation to be 0 if $y_i = 1$ and unchanged if $y_i = 0$. Also note that we do not eliminate nodes with both edges pointing to the same node as we did with ROBDD. However, same as ROBDDs, for a fixed set of variables and their ordering, a ZBDD represents a Boolean function uniquely.

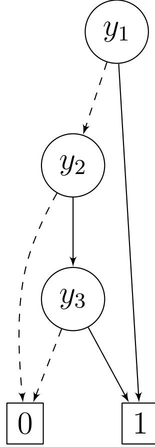

Figure 2.3 ZBDD representing combination set $\{\{a_1\}, \{a_2, a_3\}\}$.

Contrary to ROBDDs, when representing combination sets, ZBDDs do not depend on the combination domain. A ZBDD for $S = \{\{a_1\}, \{a_2, a_3\}\}$ is identical whether it is considered over the domain $\mathcal{D}_3$ or $\mathcal{D}_4$ — in both cases it looks as shown in Figure 2.3. This is a direct consequence of the fact that if a d-variable is not active for any of the combinations in a set, it does not appear in any node of the set's ZBDD representation. Minato [14] illustrates in Figure 6 how impactful this effect is by comparing the number of nodes needed for representing the same combination sets as ROBDDs and as ZBDDs. This is one of our main motivations for using ZBDDs as a representation of CNF formulas.

2.2.1 Attributed Edges

BDDs offer one more piece of optimization, and that is using attributes on edges. Minato et al. [15] describes usage of attributed edges on ROBDDs and how they improve manipulation efficiency. Minato [10] also proposes edge attributes for ZBDDs, although used in a slightly different way than with ROBDDs. In particular, it describes a *0-element edge* with the following semantics: The sub-graph pointed to by a 0-element edge has a path to the 1-terminal consisting only of 0-edges. In other words, such sub-graph represents a combination set that contains the empty combination ($\{\} \in S$).

In order to preserve uniqueness of ZBDDs, we need to place some constraints on the usage of 0-element edges:

1. Use the 0-terminal only (omit the 1-terminal),

2. Do not use the 0-element attribute on 0-edges (use on 1-edges only).

It is easy to show that the 1-terminal prohibited by Rule 1 can be replaced with a 0-element edge pointing to the 0-terminal. This also illustrates why the rule is necessary for ZBDDs uniqueness. However, it also poses a slight challenge — how do we represent a ZBDD consisting of the 1-terminal alone, i.e. with no incoming edge? This ZBDD represents the combination set $\{\{\}\}$ (it only contains the empty combination). The solution is giving each ZBDD an extra edge pointing to its root. This edge can carry the 0-element attribute with the usual semantics — the sub-graph (i.e. the whole ZBDD) has a 0-path to the 1-terminal. This workaround is just a technicality; we will say that the *root* of a ZBDD is attributed when we mean that the *edge pointing to the root* is attributed. The example combination set we have been using in this chapter is represented by a ZBDD with attributed edges shown in Figure 2.4.

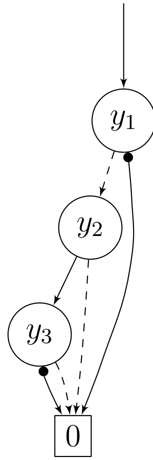

Figure 2.4 ZBDD with attributed edges representing combination set $\{\{a_1\}, \{a_2, a_3\}\}$.

Using 0-element edges improves efficiency of some operations over ZBDDs. The simplest example was already mentioned — checking if a ZBDD contains the empty combination $\{\}$ now only requires checking if the root is attributed by the 0-element or not. Without the attribute, we would have to traverse the 0-edge path from the root to determine whether it leads to the 1-terminal. This particular operation (or rather lookup) is performed quite often in our application, so being able to do it in constant time rather than linear in the depth of the ZBDD is a significant improvement. Impact on other operations will be further explored in Section 4.1.

2.2.2 Basic Operations

Basic operations over ZBDDs are described in the original paper by Minato [10], as well as algorithms implementing them. However, Minato [14] uses a slightly different notation which we like better, so we are going to use it instead of the original one. The basic operations that we need to use for DP elimination are:

- "$\mathbf{\emptyset}$" empty set $\{\}$ (0-terminal),
- "$\mathbf{1}$" set containing only the empty combination $\{\{\}\}$ (1-terminal),

- P.`top` d-variable at the root of P,
- P.`offset(`y`)` combinations in P that do not contain d-variable y,
- P.`onset(`y`)` combinations in P that contain d-variable y, excluding y from those combinations,
- P_0 P.`offset(`P.`top)` (subgraph of P under the 0-edge),
- P_1 P.`onset(`P.`top)` (subgraph of P under the 1-edge),
- $P \cup Q$ union of combinations in P and Q,
- $P \cap Q$ intersection of combinations in P and Q,
- $P - Q$ difference, i.e. combinations in P but not in Q,
- P.`count` number of combinations in P.

In addition, we can check if a ZBDD contains the empty combination, as shown in Section 2.2.1:

- $\{\} \in P$.

To clarify, P.`onset(`y`)` returns the subset of P such that $y = 1$, i.e. all combinations in P containing d-variable y, while removing y from these combinations. For example, $\{\{a_1, a_2, a_3\}, \{a_2\}\}$`.onset(`$a_1$`)` $= \{\{a_2, a_3\}\}$[2].

When implementing these operations, Minato [14] employs a helper procedure `Getnode(`y, P, Q`)`, which gets a node with d-variable y in the root, and subgraphs P under the 0-edge and Q under the 1-edge. The procedure either creates a new node with the required d-variable and children, or returns an existing node if it already exists. This is essential for efficient memory management of ZBDD implementations. In addition, procedures that are recursively implemented use caching of results, which in turn is essential for the data structure's efficiency. When we define our own operations over ZBDDs, we will use both of these features to stay consistent with the rest of the operations.

2.2.3 Additional Operations

Minato [14] also describes some additional operations related to *unate cube set algebra*; unate cube set is essentially what we call a combination set. Out of these operations, we need the product operation, $P * Q$, which generates all possible

[2] Formally, the operations are defined over ZBDDs, not combination sets they represent. For clarity and simplicity of notation, we substitute S for $ZBDD(S)$ (and `.onset(`a_i`)` for `.onset(`y_i`)`) in this example, as well as later ones.

unions of two combinations in respective combination sets. For example,

$$\begin{aligned}\{\{a_1,a_2\},\{a_2\},\{a_3\}\} * \{\{a_1,a_2\},\{\}\} &= \{\{a_1,a_2\}\cup\{a_1,a_2\}\}\cup\{\{a_1,a_2\}\cup\{\}\}\cup\\ &\quad \{\{a_2\}\cup\{a_1,a_2\}\}\cup\{\{a_2\}\cup\{\}\}\cup\\ &\quad \{\{a_3\}\cup\{a_1,a_2\}\}\cup\{\{a_3\}\cup\{\}\}\cup\\ &= \{\{a_1,a_2\}\}\cup\{\{a_1,a_2\}\}\cup\\ &\quad \{\{a_1,a_2\}\}\cup\{\{a_2\}\}\cup\\ &\quad \{\{a_1,a_2,a_3\}\}\cup\{\{a_3\}\}\\ &= \{\{a_1,a_2\},\{a_1,a_2,a_3\},\{a_2\},\{a_3\}\}.\end{aligned}$$

Looking at Step 3 of the DP procedure described in Section 1.1.4, we need an operation for removing subsumed combinations from a ZBDD. It is not obvious how to achieve that using standard operations over ZBDDs described so far. Chatalic and Simon [9] mention special ZBDD operators they implemented (clause-distribution, union) which remove subsumptions while computing the result of those operations. However, they do not provide any description of those operators, nor do they cite their source, and their *ZRes* software is no longer available. Fortunately, we were able to find their other publication, Chatalic and Simon [16], where the special operators are defined.

In total, Chatalic and Simon [16] describe three operators dealing with subsumptions:

- $P \mathbin{\tilde{\setminus}} Q$ subsumed difference,
- P.`noSub` subsumption elimination,
- $P \sqcup Q$ subsumption-free union.

$P \mathbin{\tilde{\setminus}} Q$ computes the set of combinations obtained by removing from P all combinations that are subsumed by some combination in Q. P.`noSub` removes from P all combinations that are subsumed by some other combination in P. Notice that P.`noSub` $\neq P \mathbin{\tilde{\setminus}} P$, because $P \mathbin{\tilde{\setminus}} P = \mathbf{\emptyset}$. $P \sqcup Q$ computes the union of P and Q while removing all subsumed combinations from the result. In other words, $P \sqcup Q$ yields the same result as $(P \cup Q)$.`noSub`, but computes it in one go. We provide algorithms for performing these operations in Algorithm 2.1, Algorithm 2.2, and Algorithm 2.3 respectively. They are based on the mathematical definitions of the operators by Chatalic and Simon [16], but we take a more programming-like approach, including caching as it is used by Minato [14]. Also note that Chatalic and Simon [16] make some assumptions about their ZBDDs that do not hold in general, so we had to adjust some base-cases from their recursive definitions. Specifically, they assume that ZBDDs containing the empty combination are reduced to $\mathbf{1}$, so in some cases instead of checking if $P = \mathbf{1}$, we need to check if $\{\} \in P$.

In the following chapters, we will introduce several algorithms utilising the operations over ZBDDs we have described so far. In addition to those, we might also sometimes want to manually traverse all combinations in a ZBDD. For that purpose, we introduce a convenience procedure in Algorithm 2.4. Whenever we use a for-each loop over a ZBDD in some pseudocode, as shown in an example Algorithm 2.5, we will understand it with semantics shown in Algorithm 2.6.

Algorithm 2.1: $P \tilde{\setminus} Q$ (subsumed difference)

Input: ZBDDs P and Q

1. **if** $P = \emptyset$ **then return** $\emptyset$
2. **if** $\{\} \in Q$ **then return** $\emptyset$
3. **if** $P = \mathbf{1}$ **then return 1**
4. **if** $Q = \emptyset$ **then return** P
5. **if** $P = Q$ **then return** $\emptyset$
6. $R \leftarrow \text{cache}[P \tilde{\setminus} Q]$
7. **if** R *exists* **then return** R
8. **if** $P.\texttt{top} > Q.\texttt{top}$ **then**
9. $R \leftarrow P \tilde{\setminus} Q_0$
10. **else if** $P.\texttt{top} < Q.\texttt{top}$ **then**
11. $R \leftarrow$ `Getnode(`P`.top`, $P_0 \tilde{\setminus} Q$, $P_1 \tilde{\setminus} Q$`)`
12. **else**
13. $R \leftarrow$ `Getnode(`P`.top`, $P_0 \tilde{\setminus} Q_0$, $(P_1 \tilde{\setminus} Q_1) \tilde{\setminus} Q_0$`)`
14. **end**
15. $\text{cache}[P \tilde{\setminus} Q] \leftarrow R$
16. **return** R

Algorithm 2.2: P.`noSub` (subsumption elimination)

Input: ZBDD P

1. **if** $P = \emptyset$ **then return** $\emptyset$
2. **if** $\{\} \in P$ **then return 1**
3. $R \leftarrow \text{cache}[P.\texttt{noSub}\ ]$
4. **if** R *exists* **then return** R
5. **if** $P_0 = P_1$ **then**
6. $R \leftarrow P_0.\texttt{noSub}$
7. **else**
8. $R \leftarrow$ `Getnode(`P`.top`, P_0`.noSub`, P_1`.noSub` $\tilde{\setminus}$ P_0`.noSub)`
9. **end**
10. $\text{cache}[P.\texttt{noSub}\ ] \leftarrow R$
11. **return** R

Algorithm 2.3: $P \sqcup Q$ (subsumption-free union)

Input: ZBDDs P and Q

1. **if** $\{\} \in P$ **then return 1**
2. **if** $\{\} \in Q$ **then return 1**
3. **if** $P = \emptyset$ **then return** Q.`noSub`
4. **if** $Q = \emptyset$ **then return** P.`noSub`
5. $R \leftarrow$ cache$[P \sqcup Q]$
6. **if** R *exists* **then return** R
7. **if** P.`top` $<$ Q.`top` **then**
8. | $R \leftarrow$ `Getnode(`P`.top`, $P_0 \sqcup Q$, P_1`.noSub` $\tilde{\setminus}(P_0 \sqcup Q)$`)`
9. **else if** P.`top` $>$ Q.`top` **then**
10. | $R \leftarrow$ `Getnode(`Q`.top`, $P \sqcup Q_0$, Q_1`.noSub` $\tilde{\setminus}(P \sqcup Q_0)$`)`
11. **else**
12. | $R \leftarrow$ `Getnode(`P`.top`, $P_0 \sqcup Q_0$, $(P_1 \sqcup Q_1) \tilde{\setminus} (P_0 \sqcup Q_0)$`)`
13. **end**
14. cache$[P \sqcup Q] \leftarrow R$
15. **return** R

Algorithm 2.4: `ForEachCombination(`P, S, `F)`

Input: ZBDD P, stack of d-variables S, function `F`

1. **if** $P = \mathbf{1}$ **then**
2. | $C \leftarrow$ convert stack of d-variables S to a combination
3. | `F(`C`)`
4. | **return**
5. **else if** $P = \emptyset$ **then**
6. | **return**
7. **end**
8. `ForEachCombination(`P_0, S, `F)`
9. S`.push(`P`.top)`
10. `ForEachCombination(`P_1, S, `F)`
11. S`.pop()`

Algorithm 2.5: Example of a for-each loop over a ZBDD

Input: ZBDD P

1. **foreach** $C \in P$ **do**
2. | do something with combination C
3. **end**

Algorithm 2.6: Semantics of a for-each loop over a ZBDD

Input: ZBDD P

1. $S \leftarrow$ empty stack
2. `F` $\leftarrow$ body of the **foreach** loop as a function
3. `ForEachCombination(`P, S, `F)`

3 DP Elimination

The DP procedure was already described in Section 1.1.4. In Chapter 1 we described how we want to use this algorithm for eliminating auxiliary variables from a formula in CNF that were introduced as part of the encoding of a DNNF formula. The advantage of using DP in this way is that it preserves propagation completeness of the remaining variables. The DP procedure in its basic form is a SAT decider — it eliminates all variables and based on the result decides whether the original formula was satisfiable or not. However, we only want to eliminate a certain subset of variables. Let φ be a CNF formula, and let $\mathcal{X}_p \subseteq \mathrm{Var}(\varphi)$ be a set of *protected* variables, i.e. variables that we do not want to eliminate. In the DP procedure described in Section 1.1.4 we replace steps 1 and 4b with the following:

1. choose a variable $x \in \mathrm{Var}(\varphi) \setminus \mathcal{X}_p$,

4. b. if φ only contains variables in $\mathcal{X}_p$, stop.

In order to distinguish this version of the algorithm from the original DP procedure, we will call it *DP elimination.* Note that if we indeed want to eliminate all variables in φ, we can define $\mathcal{X}_p = \emptyset$, which effectively unifies the new stop condition with the original one. In this chapter, we will analyse each step of DP elimination while discussing how to implement them efficiently.

3.1 Using ZBDDs

One of the crucial performance considerations regarding DP elimination (or any algorithm manipulating CNF formulas for that matter) is the representation of a formula. In Section 1.1.3 we outlined how a CNF formula φ can be represented as a combination set over the domain $\mathrm{lit}(\mathrm{Var}(\varphi))$, with its characteristic function $\chi_\varphi(\mathbf{y})$ (vector of d-variables $\mathbf{y}$ corresponds to the domain $\mathrm{lit}(\mathrm{Var}(\varphi))$, i.e. literals of φ). Subsequently in Chapter 2 we described a data structure called zero-suppressed binary decision diagram (ZBDD) that is very efficient for representing combination sets (through their characteristic functions). We were inspired by Chatalic and Simon [9] to use ZBDDs for representing CNF formulas in our implementation of DP elimination. As the authors point out, the size of a ZBDD is not directly related to the size of the formula it represents. In particular, the number of clauses in the formula is equal to the number of *paths* (not edges nor nodes) in the corresponding ZBDD from its root to the 1-terminal (see Minato [14]). Since the cost of set operations over ZBDDs only depends on the size of the ZBDD itself, this data structure looks promising for our use-case.

In Section 1.1.3 we already mentioned that there is a difference between propositional variables in formula φ and domain variables (d-variables) of its characteristic function $\chi_\varphi(\mathbf{y})$. The d-variables of $\chi_\varphi(\mathbf{y})$ correspond to literals in φ. Consequently, when using a ZBDD P to represent φ, P needs two d-variables for each variable in $\mathrm{Var}(\varphi)$. We will use the following notation: given variable $x_i \in \mathrm{Var}(\varphi)$, we denote y_i the d-variable corresponding to the positive literal x_i, and y_{-i} the d-variable corresponding to the negative literal $\neg x_i$. For our ZBDDs, we will assume d-variable ordering $y_1 < y_{-1} < \cdots < y_n < y_{-n}$. That way nodes

representing the positive and negative literal of the same variable will always be subsequent in the ZBDD. This might be an advantage for some algorithms that are aware of the semantics of our ZBDDs (i.e. that they represent a CNF formula), e.g. for removing tautologies. Figure 3.1 shows a ZBDD representing the characteristic function $\chi_\varphi(\mathbf{y})$ for formula $\varphi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_2) \wedge (x_2 \vee \neg x_3) \wedge (x_3 \vee x_4)$, assuming literals of variable x_i map to d-variables y_i and y_{-i}.

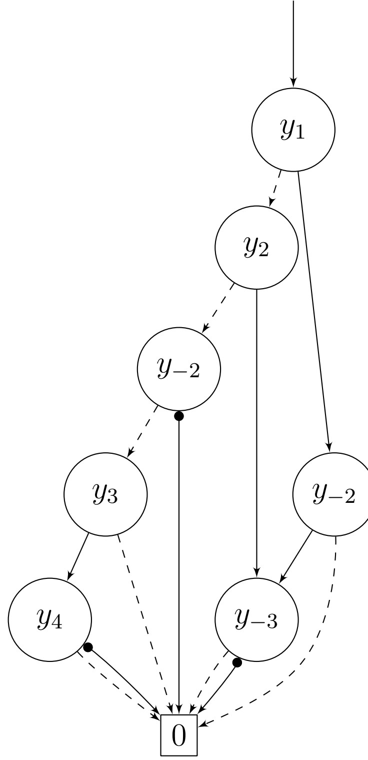

Figure 3.1 ZBDD representing formula $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_2) \wedge (x_2 \vee \neg x_3) \wedge (x_3 \vee x_4)$.

In order to execute DP elimination over a ZBDD, we need the following functionalities (steps refer to algorithm description in Section 1.1.4):

- selecting a non-protected variable for elimination (Step 1, modified in Chapter 3),
- cut elimination of a variable (Step 2),
- removal of subsumed clauses (Step 3),
- check for empty clause (Step 4a),
- comparison of remaining variables with $\mathcal{X}_p$ (step 4b, modified in Chapter 3).

Variable selection will be discussed in detail in Section 3.2. This step can also deal with checking whether there are any remaining variables outside of $\mathcal{X}_p$ — if there are none, variable selection fails. For removing subsumed clauses we have already defined ZBDD operations in Section 2.2.3, in particular the P.`noSub` operation described in Algorithm 2.2. Checking if a ZBDD contains the empty set is a basic ZBDD operation (see Section 2.2.2).

Cut elimination of a variable, which is the essential part of the algorithm, can mostly be broken down into basic ZBDD operations described in Section 2.2.2.

As noted by Chatalic and Simon [9], we can describe cut elimination of variable $x \in \mathrm{Var}(\varphi)$ as follows:

1. rewrite the formula φ as $(x \vee \varphi_x^+) \wedge (\neg x \vee \varphi_x^-) \wedge \varphi_x^0$, where
   - φ_x^+ (resp. φ_x^-) denotes the CNF of clauses in φ containing x (resp. $\neg x$) from which the literal x (resp. $\neg x$) was removed,
   - φ_x^0 denotes the CNF of clauses in φ containing neither x nor $\neg x$,
2. convert the formula $(\varphi_x^+ \vee \varphi_x^-) \wedge \varphi_x^0$ into CNF
   a. distribute clauses in φ_x^+ over clauses in φ_x^-,
   b. remove tautologies from the result,
   c. compute union of the resulting clauses with clauses in φ_x^0.

Now let us denote P the ZBDD representing formula φ, and assume that $x = x_i$. Obtaining φ_x^+ and φ_x^- directly corresponds to the ZBDD operations P.`onset`(y_i) and P.`onset`(y_{-i}), while φ_x^0 can be obtained by P.`offset`(y_i).`offset`(y_{-i}). Distributing clauses in formula φ over clauses in formula ψ (represented by ZBDDs P and Q respectively) corresponds to the product operation, i.e. $P * Q$, described in Section 2.2.3. Removing tautologies is not a standard ZBDD operation, as it is not a purely syntactic operation, but depends on the semantics of the underlying sets — it needs to know which d-variables correspond to complementary literals. As such, it will be described in Section 3.3. Finally, union of two sets of clauses naturally corresponds to the union of their ZBDDs.

To summarise, DP elimination over ZBDDs can be described by Algorithm 3.1. Note that when performing cut elimination after selecting a d-variable y_i, it does not matter whether y_i corresponds to a positive literal $x_{|i|}$ or a negative literal $\neg x_{|i|}$. The cut elimination steps are symmetrical for i being positive or negative (if $i = -1$, then $y_i = y_{-1}$ and $y_{-i} = y_1$). Also notice that the last operation of the cut elimination — union of two ZBDDs — is directly followed by subsumption removal. These two operations can be done in one go using the subsumption-free union operator described in Algorithm 2.3 of Section 2.2.3.

Algorithm 3.1: DP elimination over ZBDDs

Input: ZBDD P, set of protected variables $\mathcal{X}_p$

1. $y_i \leftarrow$ select d-variable from P such that $x_{|i|} \notin \mathcal{X}_p$
2. **while** $y_i \neq$ `None` **and** $\{\} \notin P$ **do**
3. $P^+ \leftarrow P$.`onset`(y_i)
4. $P^- \leftarrow P$.`onset`(y_{-i})
5. $P' \leftarrow P$.`offset`(y_i).`offset`(y_{-i})
6. $P_{new} \leftarrow P^+ * P^-$
7. remove tautologies from P_{new}
8. $P \leftarrow P' \sqcup P_{new}$ `/* substitutes` $P \leftarrow (P' \cup P_{new})$`.noSub */`
9. $y_i \leftarrow$ select d-variable from P such that $x_{|i|} \notin \mathcal{X}_p$
10. **end**

3.2 Variable Selection

In this section, we will address lines 1 and 9 of Algorithm 3.1, i.e. selecting a variable to be eliminated from φ. This step might have significant effect on the formula's behaviour, especially its size during the course of the algorithm. There are a couple of approaches to be considered, ranging from very simple ones based on the ZBDDs structure to heuristics based on the formula's statistics, or even dynamic behaviour of the formula over time. In this section, we will propose a couple of these variable selection methods and decide which ones are worth implementing. At first, we will ignore the constraint of some variables being protected from elimination ($\mathcal{X}_p$), and deal with this issue later. Note that because a ZBDD representing φ does not contain variables $\mathrm{Var}(\varphi)$, but rather d-variables of the characteristic function $\chi_\varphi(\mathbf{y})$, we actually want to select d-variables instead of variables, as shown in Algorithm 3.1.

3.2.1 Ordering-based Methods

Before we discuss non-trivial methods of d-variable selection, we need to set a baseline. One possibility is random selection, but we would rather avoid non-deterministic methods, as they would complicate reproducibility and predictability of results. Instead, we could utilise the assumed ordering of d-variables in ZBDDs $y_1 < y_{-1} < \cdots < y_n < y_{-n}$. This provides us with two selection methods — eliminate d-variables in **ascending** or **descending order**. Although we do not make any assumptions about the d-variable ordering, it is possible that in practice the ordering is not random. Especially in our context of working with CNF encodings of DNNF formulas, it is a reasonable guess that the ordering reflects the structure of the DNNF formula. Therefore, ascending and descending elimination order can exhibit different properties. Implementation of such selection methods is trivial, and so is dealing with protected variables $\mathcal{X}_p$ — they can be simply skipped.

3.2.2 Low-hanging Fruits

First, let us look more closely at the ascending order method from Section 3.2.1. If we ignore the protected variables $\mathcal{X}_p$, the smallest d-variable from the ordering $y_1 < y_{-1} < \cdots < y_n < y_{-n}$ is actually the **root d-variable** in the ZBDD (P.`top`). While this choice might be quite arbitrary with respect to DP elimination, taking the root d-variable can have an advantage in terms of performance — the P.`onset()` and P.`offset()` operations are trivial for the root d-variable, resulting in constant time complexity of decomposing P into P^+, P^-, and P' as described in Algorithm 3.1. However, there are no guarantees on the formula's growth in this case.

In order to limit how large the formula becomes after eliminating a variable, we need to take into account some properties of the formula and its clauses. A simple approach is finding a **unit literal** in the formula, i.e. a literal that appears in a unit clause in φ. Assuming φ has no subsumed clauses, if it contains a unit clause $\{l\}$, then the clause $\{l\}$ is the only occurrence of literal l in φ. Let us assume that l is a positive literal x. Then, φ_x^+ contains only the empty clause $\{\}$, i.e. it represents a contradiction ($\bot$). Distributing clauses in φ_x^+ over clauses in

φ_x^- results in φ_x^-, so the cut elimination x from φ becomes simply $\varphi_x^- \wedge \varphi_x^0$. This guarantees that the formula will not grow at all during this step. The situation is analogous if l is a negative literal $\neg x$. Finding a d-variable in a ZBDD that corresponds to a unit literal can be done using Algorithm 3.2.

Algorithm 3.2: `FindUnitLiteral(`P`)`

```
  Input: ZBDD P
1 while not (P = ∅ or P = 1) do
2 |  if {} ∈ P_1 then return P.top
3 |  else P ← P_0
4 end
5 return None
```

We could also find a **pure literal** — a variable that has either only positive (x), or only negative ($\neg x$) occurrences in φ. If that's the case, one of φ_x^+, φ_x^- is empty, i.e. always satisfied ($\top$), resulting in $\varphi_x^+ \vee \varphi_x^-$ being also $\top$, leading to the cut elimination of x in φ being just φ_x^0. In this case, we are guaranteed that the formula will actually shrink instead of growing. To find a d-variable that corresponds to a pure literal, we need to traverse the whole ZBDD and keep track of all d-variables we encounter, as shown in Algorithm 3.3. Although we could use the helper Algorithm 2.4, we do not actually need to traverse clauses (combinations), but only nodes in the ZBDD. Therefore, a custom algorithm is more efficient.

Algorithm 3.3: `FindPureLiteral(`P`)`

```
   Input: ZBDD P
 1 stack ← [P]
 2 visited ← {}
 3 literals ← {}
 4 while stack is not empty do
 5 |  Q ← stack.pop()
 6 |  if Q = ∅ or Q = 1 or Q ∈ visited then continue
 7 |  visited ← visited ∪ {Q}
 8 |  stack.push(Q_0)
 9 |  stack.push(Q_1)
10 |  literals ← literals ∪ {Q.top}
11 end
12 foreach y_i ∈ literals do
13 |  if y_{-i} ∉ literals then return y_i
14 end
15 return None
```

While the pure literal and unit literal approaches do offer some guarantees about limiting the formula's growth, their main disadvantage is their incompleteness — they only work if such literals are present in the formula. Otherwise, we must fall back to some other selection method. One possible approach is trying to find a pure literal, then a unit literal if unsuccessful, and using the root d-variable if none of the previous ones succeed, as it is shown in Algorithm 3.4.

Algorithm 3.4: Combined simple variable selection

Input: ZBDD P

1 $y_i \leftarrow$ `FindPureLiteral(`P`)`
2 **if** $y_i \neq$ None **then return** y_i
3 $y_i \leftarrow$ `FindUnitLiteral(`P`)`
4 **if** $y_i \neq$ None **then return** y_i
5 **return** P`.top`

Although this combination of simple methods does always find a d-variable to eliminate, it fails to provide any guarantees about how much the formula grows when the selected variable is eliminated (because of the fallback to P`.top`). Moreover, none of these methods are able to satisfy the constraint of protected variables $\mathcal{X}_p$, and modifying them to do so would introduce additional complexity. In order to solve both these issues, we need to find a more sophisticated way of selecting a suitable d-variable for elimination.

3.2.3 Optimisation-based Methods

A more involved approach to variable selection is trying to find an optimum of some metric based on formula statistics. The basic idea is that we want to select a variable that, when eliminated, results in the smallest possible formula. There might be multiple reasonable ways of defining the 'size of a formula', but let us now focus on the number of clauses in the formula's CNF. The only precise way to predict the number of clauses after eliminating a variable is to actually perform the elimination, due to tautologies, subsumption, etc. In order to find the optimal variable, we would need to try eliminating all (or at least a significant portion) of the variables and essentially backtrack. Since we are not performing a search, but rather need to eliminate all (or most of) the variables anyway, this approach seems unreasonable. Instead, we would like to use some heuristic for estimating the formula's growth when eliminating a variable.

When looking at the DP procedure described in Section 1.1.4, there are two steps that modify the number of clauses in the formula: cut elimination of a variable and removing subsumed clauses. We do not know how to predict the number of subsumed clauses other than actually performing the cut elimination and finding subsumptions in the result. However, the effect of cut elimination alone can be predicted reasonably well.

Let us revisit cut elimination as described in Section 3.1:

- $\varphi \equiv (x \vee \varphi_x^+) \wedge (\neg x \vee \varphi_x^-) \wedge \varphi_x^0$,
- $\varphi_{new} := \mathrm{CNF}(\varphi_x^+ \vee \varphi_x^-) \wedge \varphi_x^0$,
- $\mathrm{CNF}(\varphi_x^+ \vee \varphi_x^-)$:= "distribute φ_x^+ over φ_x^- and remove tautologies".

φ_x^0 is stable throughout the cut elimination. Assuming φ does not contain any tautologies, we can easily measure the size of the rest of the formula by counting the number of occurrences of x and $\neg x$ in φ, i.e. $|\varphi| - |\varphi_x^0| = |\varphi_x^+| + |\varphi_x^-|$. The size of $\mathrm{CNF}(\varphi_x^+ \vee \varphi_x^-)$ again cannot be known precisely without actually computing the formula (due to tautologies and identical clauses produced by distributing φ_x^+

over φ_x^-), but we can at least give it an upper bound. Since the clause distribution step essentially takes all clauses in φ_x^+ and unifies them with all clauses in φ_x^-, the result cannot be larger than $|\varphi_x^+| \cdot |\varphi_x^-|$. Consequently, we can deduce that:

$$|\varphi_{new}| \leq |\varphi| - (\left|\varphi_x^+\right| + \left|\varphi_x^-\right|) + (\left|\varphi_x^+\right| \cdot \left|\varphi_x^-\right|). \tag{3.1}$$

Our goal is to find a variable x in formula φ such that cut elimination of x from φ minimises the resulting formula's size. Based on the reasoning we have described so far, we will use the following heuristic:

$$h(x) = -(\left|\varphi_x^+\right| + \left|\varphi_x^-\right|) + \left|\varphi_x^+\right| \cdot \left|\varphi_x^-\right|. \tag{3.2}$$

We call this heuristic the **minimal bloat** heuristic, as it tries to minimise the bloat (growth) of the formula. In order to find $x = \arg\min_{x \in \mathrm{Var}(\varphi)} h(x)$, we need to compute $|\varphi_x^+|$ and $|\varphi_x^-|$ for each $x \in \mathrm{Var}(\varphi)$. For that, we can use Algorithm 3.5, which makes use of the for-each algorithm over ZBDDs (see Algorithm 2.4 in Section 2.2.3).

Algorithm 3.5: `CountLiterals(`P`)`

```
   Input: ZBDD P
 1 variableMap ← empty map
 2 foreach C ∈ P do
 3 |   foreach y_i ∈ C do
 4 |   |   if x_|i| ∉ variableMap then
 5 |   |   |   variableMap[x_|i|] ← (pos = 0, neg = 0)
 6 |   |   end
 7 |   |   if y_i represents positive literal then
 8 |   |   |   variableMap[x_|i|].pos ← variableMap[x_|i|].pos +1
 9 |   |   else
10 |   |   |   variableMap[x_|i|].neg ← variableMap[x_|i|].neg +1
11 |   |   end
12 |   end
13 end
14 return variableMap
```

Notice how finding an optimum of $h(x)$ is a generalisation of the combined selection methods described in Algorithm 3.4. If φ does contain a pure literal x (resp. $\neg x$), then $h(x) = -|\varphi_x^-|$ (resp. $h(x) = -|\varphi_x^+|$). If φ contains a unit literal x (resp. $\neg x$), then $h(x) = -1$. All other variables have $h(x) \geq 0$. The difference is that if φ does not contain any pure nor unit literal, then $\arg\min_{x \in \mathrm{Var}(\varphi)} h(x)$ still selects a sensible variable instead of the arbitrary choice of P.`top`.

Let us now address the requirement of only eliminating variables that are not in the protected set $\mathcal{X}_p$, i.e. the modified steps 1 and 4b of DP elimination presented at the beginning of Chapter 3. Using this heuristic-based variable selection, we can simply take $x = \arg\min_{x \in \mathrm{Var}(\varphi) \setminus \mathcal{X}_p} h(x)$. To achieve this algorithmically, we can iterate through the result of Algorithm 3.5 and skip variables that are in $\mathcal{X}_p$. If the set $\mathrm{Var}(\varphi) \setminus \mathcal{X}_p$ is empty, there are no more variables that can be eliminated and we stop the DP elimination. The full algorithm, including $\mathcal{X}_p$ filtering, is described in Algorithm 3.6.

Algorithm 3.6: `GetMinimalBloatLiteral`(P, $\mathcal{X}_p$)

```
   Input: ZBDD P, set of protected variables X_p
 1 varMap ← CountLiterals(P)
 2 h_best ← ∞
 3 y_best ← None
 4 foreach x_i ∈ varMap where x_i ∉ X_p do
 5 |   h_i ← −( varMap[x_i].pos + varMap[x_i].neg ) + ( varMap[x_i].pos ·
 |       varMap[x_i].neg )
 6 |   if h_i < h_best then
 7 |   |   h_best ← h_i
 8 |   |   y_best ← y_i
 9 |   end
10 end
11 return y_best
```

3.2.4 Dynamic Methods

In SAT solvers, it is common to use dynamic methods for variable selection. In fact, the optimisation method described in Section 3.2.3 can also be considered dynamic — instead of pre-computing the literal occurrences beforehand and using them statically during DP elimination, we count literals for the new formula after each elimination step. Some of these dynamic methods try to measure 'activity' of variables during the course of the algorithm (e.g. *VSIDS*, Liang et al. [17]) and prefer variables that are more active in some way. Such activity measures can be collected if the algorithm in question represents formulas in a way that makes it clear when a clause is added, removed, or altered in some way. However, in case of ZBDDs it would be very impractical to try to measure such activity of clauses or variables, because the formula is modified by ZBDD operations, which obscure changes to individual clauses or variables. Therefore, we did not try using any such methods.

3.3 Removing Tautologies

The remaining missing piece of Algorithm 3.1 is tautology removal. Tautology in general is a formula that always evaluates to true ($\top$). If a formula is in CNF — it is a conjunction of clauses — it can only be a tautology if it does not contain any clauses, or if all its clauses are tautologies on their own. A clause is a tautology if it contains both a positive and a negative literal of the same variable. Clearly, a tautological clause does not add any information to the CNF formula and can be safely removed without breaking equivalence.

Removing tautological clauses is not merely a minimisation technique, it is necessary for correctness of cut elimination as it is described in Section 3.1. Suppose formula φ contains a clause C with both x and $\neg x$. Then φ_x^+ contains $C \setminus \{x\}$, which still contains $\neg x$. Analogously, φ_x^- contains a clause that still contains x. When clauses in φ_x^+ are distributed over clauses in φ_x^-, the results contains literals of the variable x that was supposed to be eliminated. Therefore, we need to remove tautologies in each step of DP elimination.

Removing tautologies from a formula represented by a ZBDD cannot be done by a standard ZBDD operation. The reason is that ZBDDs represent combination sets, and being a tautology is not a property of a combination — it depends on how the combination is interpreted (a set of literals). In order to define a ZBDD operation for tautology removal, we need to utilise the semantics of ZBDDs in our context. In Section 3.1 we made an assumption about d-variable ordering in ZBDDs: d-variables corresponding to complementary literals always appear in succession in the ordering. As a consequence, if two nodes corresponding to complementary literals are on a directed path in a ZBDD, then one must be a child of the other. This is a basis of Algorithm 3.7. We recursively remove tautologies from both children of a ZBDD node. Then, if the 1-edge child corresponds to the node's complementary literal, we remove the whole 1-edge sub-graph of that child. Figure 3.2 illustrates the last step, assuming all children are already without tautologies. The result of the operation is a ZBDD representing an equivalent formula to the original one, but without any tautological clauses.

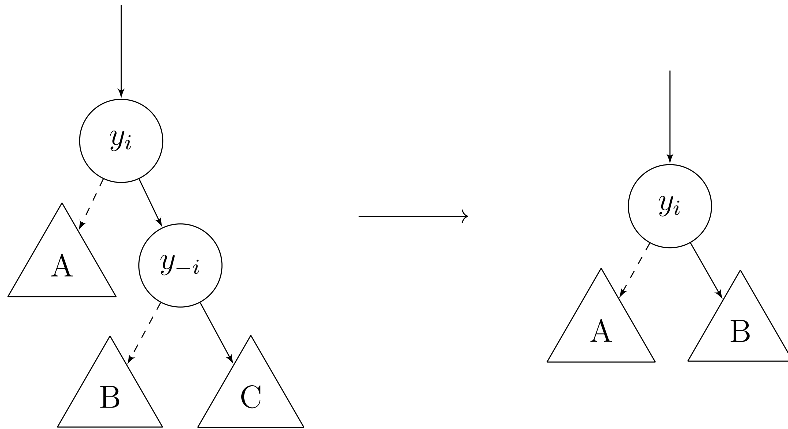

Figure 3.2 Illustration of recursive tautology removal step over ZBDD.

3.4 Unit Propagation

When discussing the effects of cut-eliminating a variable that appears in a unit clause (see Section 3.2.2), we argued that the advantage of such cut elimination is that the formula will not grow in size. The basis of our argument was that if $\{x\}$ is a unit clause in φ, then eliminating x from φ can be simplified to $\varphi_x^- \wedge \varphi_x^0$ (and analogously for unit clause $\{\neg x\}$). We can utilise this property of unit literals even more than just as a variable selection heuristic — we can skip a couple of steps done in the general-purpose cut elimination algorithm described in Section 3.1. We do not need to distribute clauses in φ_x^+ over φ_x^-; in fact, we do not need φ_x^+ (resp. φ_x^- for negative unit literal) at all.

Let us assume that we have found a unit literal l of variable x in formula φ and want to eliminate x from φ. This operation can be understood as assigning $\bar{x} = 1$ if l is positive, or $\bar{x} = 0$ if l is negative. We can do it repeatedly until there are no unit clauses in the result, essentially performing *unit propagation.* Because finding a unit literal in a ZBDD using Algorithm 3.2 is cheap (linear in the depth of the ZBDD), we can run unit propagation, implemented in algorithms 3.8 and 3.9, as a formula-simplification method after each iteration of DP elimination.

Algorithm 3.7: `RemoveTautologies(P)`

```
   Input: ZBDD P
 1 if P = ∅ or P = 1 then return P
 2 R ← cache[RemoveTautologies(P)]
 3 if R exists then return R
 4 P0 ← RemoveTautologies(P_0)
 5 P1 ← RemoveTautologies(P_1)
 6 y_i ← P.top
 7 y_j ← P1.top
 8 if P1 = ∅ or P1 = 1 then
 9 |  R ← Getnode(y_i, P0, P1)
10 if y_i = y_{-j} then
11 |  R ← Getnode(y_i, P0, P1_0)
12 else
13 |  R ← Getnode(y_i, P0, P1)
14 end
15 cache[RemoveTautologies(P)] ← R
16 return R
```

Algorithm 3.8: `AssignLiteral(P, y_i)`

```
   Input: ZBDD P, d-variable y_i
 1 P^- ← P.onset(y_{-i})
 2 P' ← P.offset(y_i).offset(y_{-i})
 3 return P^- ∪ P'
```

Algorithm 3.9: `UnitPropagation(P)`

```
   Input: ZBDD P
 1 y_i ← FindUnitLiteral(P)
 2 implied ← {}
 3 while y_i ≠ None and {} ∉ P do
 4 |  implied ← implied ∪ {y_i}
 5 |  P ← AssignLiteral(P, y_i)
 6 end
 7 return P, implied
```

Notice that the algorithm keeps track of d-variables corresponding to implied (unit) literals and returns them to the caller. This is done because we are not allowed to eliminate variables from a protected set $\mathcal{X}_p$, as explained at the beginning of Chapter 3. In order to comply with this requirement, we can use a trick. We will still perform unit propagation over all variables, but we will keep a list $\mathcal{Y}_u$ of d-variables that were implied this way. Once the DP elimination algorithm finishes, we will find all d-variables $y_i \in \mathcal{Y}_u$ such that $x_{|i|} \in \mathcal{X}_p$. Literals corresponding to these d-variables will be added to the resulting formula as unit clauses, ensuring that none of the protected variables are missing. Because all literals corresponding to $\mathcal{Y}_u$ were unit literals at some point during DP elimination, all other occurrences of such literals were subsumed by their unit clauses. This guarantees that after adding the unit clauses, the resulting CNF formula encodes the same Boolean function as the formula before DP elimination.

3.5 Formula Minimisation

We mentioned at the beginning of Chapter 1 that a formula running through DP elimination might grow exponentially. To reduce this effect, Chatalic and Simon [9] have included a step into the algorithm that removes subsumed clauses from the formula, as shown in the algorithm's outline in Section 1.1.4. While subsumption removal can significantly reduce the size of a CNF formula, we can do even better. Our goal is to efficiently remove from the formula as many clauses as possible while keeping it equivalent to the original form. A helpful concept in this regard might be the notion of absorbed clauses described in Section 1.1.1.

It follows directly from the definition that an absorbed clause is implied by the formula that absorbs it. Moreover, it is clear that such a clause does not add anything that benefits propagation, which is discussed by Bordeaux and Marques-Silva [2]. Therefore, similarly to subsumed clauses, we can safely remove any absorbed clause from the formula being processed to minimise its size. We show that every subsumed clause is also absorbed:

Lemma 1. *Let φ be a CNF formula, and C be an implicate of φ. Let also $C' = (l_1 \vee \cdots \vee l_n)$ be a clause in φ such that $C' \subset C$. Then C is absorbed by φ.*

Proof. Let l_i be a literal of C.

If $l_i \in C'$, then

$$\{C'\} \subseteq \varphi, \quad \{C'\} \wedge \bigwedge_{j \in 1..n,\, j \neq i} \neg l_j \quad \vdash_1 \quad l_i.$$

If $l_i \notin C'$, then

$$\{C'\} \subseteq \varphi, \quad \{C'\} \wedge \bigwedge_{j \in 1..n} \neg l_j \quad \vdash_1 \quad \bot.$$

□

Lemma 1 shows that the notion of absorbed clauses is at least as strong as the notion of subsumed clauses. Bordeaux and Marques-Silva [2] also provides

a simple example of an absorbed clause that is not subsumed: clause $(\neg x_1 \vee x_3)$ for formula $\varphi = (\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_3)$. Therefore, removing absorbed clauses is strictly stronger than removing subsumed clauses. We should note that Bordeaux and Marques-Silva [2] also considered an algorithm called '*minimize*' that iterates through all clauses of a formula and removes the absorbed ones.

The question now becomes how do we find absorbed clauses in a formula φ represented by a ZBDD. A direct approach is to simply follow the definition of an absorbed clause in Section 1.1.1, i.e. go through each literal l_i of a candidate clause $C = (l_1 \vee \cdots \vee l_n)$, run unit propagation while asserting the conjunction $\bigwedge_{j \in 1..n, j \neq i} \neg l_j$, and check if l_i is obtained as a unit implicate. For that, we need an algorithm for performing unit propagation. In our setting, two possible approaches come to mind; either we run unit propagation directly over the ZBDD at hand, or we convert the ZBDD into some standard data structure used for unit propagation. In the following sections, we will explore both of these possibilities. In our application, we will implement both and compare their performance empirically.

3.5.1 Absorbed Removal Over ZBDD

The more direct approach is finding absorbed clauses directly in ZBDDs. We have already discussed how to run unit propagation over ZBDDs in Section 3.4. Using algorithms 3.8 and 3.9 we can devise a function that determines whether a clause is absorbed by a ZBDD or not. We will take advantage of the fact that Algorithm 3.9 also returns unit literals that were found during the propagation. The resulting sub-routine, described in Algorithm 3.10, can be used in the algorithm that removes all absorbed clauses from a given ZBDD, described in Algorithm 3.11. Note that in the algorithm we use the *difference* operation $P - C$ of a ZBDD P and a clause C, which is technically not defined. We use it as a shortcut for creating a ZBDD containing a single clause C and subtracting that from P, i.e. $P - \text{ZBDD}(\{C\})$.

Algorithm 3.10: `IsClauseAbsorbedZBDD`(P, C)

```
   Input: ZBDD P, clause C
 1 if {} ∈ P then return True
 2 P, initImplied ← UnitPropagation(P)
 3 foreach y_i ∈ C where y_i ∉ initImplied do
 4 |   Q ← P
 5 |   implied ← initImplied
 6 |   foreach y_j ∈ C where y_j ≠ y_i, y_−j ∉ implied do
 7 |   |   if y_j ∈ implied then next y_i
 8 |   |   Q ← AssignLiteral(Q, y_−j)
 9 |   |   Q, newImplied ← UnitPropagation(Q)
10 |   |   if {} ∈ Q or y_i ∈ newImplied then next y_i
11 |   |   implied ← implied ∪ newImplied ∪ {y_−j}
12 |   end
13 |   return False
14 end
15 return True
```

Algorithm 3.11: `RemoveAbsorbedZBDD(`P`)`

Input: ZBDD P

1 **foreach** $C \in P$ **do**
2 | **if** `IsClauseAbsorbedZBDD(`P, C`)` **then** $P \leftarrow P - C$
3 **end**
4 **return** P

3.5.2 Absorbed Removal Over Watched Literals

While it is possible to perform unit propagation directly over ZBDDs, as demonstrated in Section 3.4, it is not the most efficient data structure for the task. Each variable assignment results in several recursive traversals of the ZBDD at hand, regardless of how many occurrences the assigned variable has in the formula. Because we need to perform many unit propagations when removing absorbed clauses from a formula, it might be more efficient to convert the ZBDD into a different data structure, remove absorbed clauses, and create a new ZBDD from the result.

The data structure for unit propagation widely used in SAT solvers and regarded as the state-of-the-art is *watched literals*, introduced by Moskewicz et al. [18]. It represents the formula as a list of clauses, each clause being a list of literals. On top of that, it uses some pointers and additional lists and sets for efficiently detecting unit clauses when some assignment changes. The most important feature of watched literals is its very efficient backtracking. This operation was not needed in our implementation of absorbed clause removal over ZBDDs directly (it was done implicitly by keeping the original ZBDD), but it is essential for a list-of-clauses representation, because such representation cannot be efficiently copied. For implementing absorbed clause detection, we will need the following operations over watched literals:

- W`.containsEmpty` checks if W contains an empty clause,
- W`.get(`l`)` assignment of literal l, either 0, 1, or ? (unassigned),
- W`.assign(`l`)` assigns $l = 1$,
- W`.backtrack(`n`)` backtracks to n-th level of assignment,
- W`.add(`C`)` add clause C,
- W`.remove(`C`)` remove clause C.

Note that `add` and `remove` operations might be more complex than it appears at first, because if W is in some non-trivial state (some level of assignment), it might change depending on the added or removed clause. However, we can ignore that complexity, because we only add or remove clauses at 0th assignment level. See algorithms 3.12 and 3.13 describing absorbed clause removal over watched literals. Also note that formally, clauses in a ZBDD do not contain literals of the represented formula, but d-variables corresponding to those literals. Because watched literals work directly with literals, there is a hidden assumption of some conversion happening between the two sets. In practice, this is likely not going

to be an issue, as the representations might coincide in code. Another implicit assumption is that we know how to build a ZBDD from a list of clauses. We have not discussed this topic so far, but it will be addressed in Section 4.4.

Algorithm 3.12: `IsClauseAbsorbedWatched`(W, C)

```
   Input: watched literals W, clause C
1  if W.containsEmpty then return True
2  foreach l ∈ C where W.get(l) ≠ 1 do
3      foreach m ∈ C where m ≠ l, W.get(¬m) ≠ 1 do
4          if W.get(¬m) = 0 then
5              W.backtrack(0)
6              next l
7          end
8          W.assign(¬m)
9          if W.containsEmpty or W.get(l) = 1 then
10             W.backtrack(0)
11             next l
12         end
13     end
14     W.backtrack(0)
15     return False
16 end
17 W.backtrack(0)
18 return True
```

Algorithm 3.13: `RemoveAbsorbedWatched`(P)

```
   Input: ZBDD P
1  W ← watched literals
2  foreach C ∈ P do
3      W.add(C)
4  end
5  foreach C ∈ W do
6      W.remove(C)
7      if not IsClauseAbsorbedWatched(W, C) then W.add(C)
8  end
9  return ZBDD from W
```

3.5.3 Incremental Absorbed Removal

It is clear that removing absorbed clauses from a formula is computationally expensive. When using watched literals, which should be the more efficient method, in the worst-case scenario we have to traverse the whole formula twice — once in the outer loop over clauses, once during unit propagation — resulting in quadratic time complexity w.r.t. formula length (sum of all clauses' lengths). This is a theoretical limit of the method rather than its implementation, hinted by the

definition of absorption itself. Although the worst-case scenario with respect to unit propagation is not likely to occur in practice, removing all absorbed clauses is still an expensive operation. Therefore, we might want to avoid performing full minimisation of the formula in each iteration of DP elimination. The obvious approach is to remove absorbed clauses conditionally based on the state of DP elimination and the formula, e.g. only when the formula grows significantly. We will explore this more in Section 3.6. In this section we would like to propose an alternative approach to full minimisation.

The inefficiency of absorbed clause removal stems from the need for checking each clause against the entire formula. Although this is the only complete algorithm, we could sacrifice completeness for efficiency while still achieving significant reduction of the formula's size. The idea is to check absorption *incrementally* — iterate through the formula's clauses, and only check if a new clause is absorbed by the clauses that were already checked. In case the clause is absorbed, we skip it; otherwise, we add it to the set of clauses to keep. Algorithm 3.14 implements this idea over watched literals, using directly ZBDDs would be very similar.

Algorithm 3.14: `IncrementalRemoveAbsorbed(`P`)`

```
   Input: ZBDD P
1  W ← watched literals
2  foreach C ∈ P do
3  |  if not IsClauseAbsorbedWatched(W, C) then W.add(C)
4  end
5  return ZBDD from W
```

We can apply this technique to our specific context of DP elimination. Looking at the stages of Algorithm 3.1, reduction of the formula's size (i.e. removing subsumed clauses) takes place while unifying P' with P_{new}. With incremental absorbed removal, we can take this a step further — instead of computing subsumption-free union, we can use the incremental algorithm for removing absorptions to compute the union of $P_{v'}$ and P_{new}. Described in Algorithm 3.15, which based on Algorithm 3.14, we take all clauses from the first given ZBDD and sequentially add to them clauses from the second ZBDD, but only if they are not absorbed by those already added. Despite us calling the algorithm '`AbsorptionFreeUnion()`', it does not guarantee that the result is absorption-free, because incremental absorption removal is not complete. However, it could reduce the resulting formula's size more than subsumption-free union, while not being as computationally expensive as complete absorption removal.

3.5.4 More Efficient Absorption Detection

Notice how both algorithms 3.10 and 3.12 naively follow the definition of an absorbed clause to detect one. They sequentially iterate through all literals in the candidate clause in two nested loops. Especially in the latter variant using watched literals, where backtracking is explicit, it might be visible that this straightforward implementation duplicates a lot of work. Once the outer loop is getting to the last literals, the earlier literals are assigned repeatedly in the inner loop after each backtrack to the 0-th assignment level.

Algorithm 3.15: `AbsorptionFreeUnion(`P, Q`)`

Input: ZBDDs P, Q
1 $W \leftarrow$ watched literals
2 **foreach** $C \in P$ **do**
3 | W`.add(`C`)`
4 **end**
5 **foreach** $C \in Q$ **do**
6 | **if not** `IsClauseAbsorbedWatched(`W, C`)` **then** W`.add(`C`)`
7 **end**
8 **return** *ZBDD from W*

It should be quite simple to improve the algorithm by only backtracking the necessary number of levels instead of going all the way back to 0. However, as already mentioned, this will only become efficient in the later iterations of the outer loop. We can do even better:

1. split the clause into two halves,
2. assign all literals from the first half; if it fails, skip to the other half,
3. perform the original algorithm on the second half; backtracking never goes lower than half of the clause,
4. if no empowering literal is found, backtrack to 0, swap the two halves and repeat; after that, the clause is absorbed.

We will not provide pseudocode for the full modified algorithm as it would get fairly complicated. The implementation can be found in Attachment A.1; for details, see Section 5.4.2.

3.6 Putting It All Together

Now that we have introduced all the necessary pieces, we can put them together into the final version of our DP elimination over ZBDDs, based on Algorithm 3.1. We use Algorithm 3.6 as variable selection heuristic. Removing tautologies is achieved with Algorithm 3.7. Algorithm 3.9 is used for unit propagation, and unit literals that belong to protected variables $\mathcal{X}_p$ are re-added when the DP elimination is finished. We have described two distinct methods of minimising a formula — subsumption removal (already included in the original algorithm) and absorption removal (Section 3.5), the latter being strictly stronger but more expensive. Since we want to measure their impact on efficiency of DP elimination, we offer quite detailed customisation of when each of these techniques is used. We proposed several variants of both subsumption removal and absorption removal; in the end, we use algorithms 2.1, 2.2, 2.3, 3.15, and 3.13. The decision to prefer absorption removal over watched literals rather than over ZBDDs was made during development — watched literals were much more efficient. The result of our efforts is Algorithm 3.16. Notice that when calling `AbsorptionFreeUnion()` on line 11, we first compute $(P_{new} \tilde{\setminus} P')$`.noSub`. This is a performance optimisation —

removing subsumptions can significantly reduce the number of clauses that need to be checked for absorption.

Algorithm 3.16: DP elimination over ZBDDs, final version

Input: ZBDD P, set of protected variables $\mathcal{X}_p$

1 P, $\mathcal{Y}_u \leftarrow$ `UnitPropagation(`P`)`
2 $y_i \leftarrow$ `GetMinimalBloatLiteral(`P, $\mathcal{X}_p$`)`
3 **while** $y_i \neq$ None **and** $\{\} \notin P$ **do**
4     $P^+ \leftarrow P$`.onset(`y_i`)`
5     $P^- \leftarrow P$`.onset(`y_{-i}`)`
6     $P' \leftarrow P$`.offset(`y_i`).offset(`y_{-i}`)`
7     $P_{new} \leftarrow P^+ * P^-$
8     $P_{new} \leftarrow$ `RemoveTautologies(`P_{new}`)`
9   **if** *condition for partial minimisation is met* **then**
10   **if** *condition for incremental absorption removal is met* **then**
11             $P \leftarrow$ `AbsorptionFreeUnion(`P', $(P_{new} \tilde{\setminus} P')$`.noSub)`
12   **else**
13             $P \leftarrow P' \sqcup P_{new}$
14   **end**
15   **else**
16         $P \leftarrow P' \cup P_{new}$
17   **end**
18   P, $\mathcal{Y} \leftarrow$ `UnitPropagation(`P`)`
19     $\mathcal{Y}_u \leftarrow \mathcal{Y}_u \cup \mathcal{Y}$
20   **if** *condition for complete minimisation is met* **then**
21         $P \leftarrow$ `RemoveAbsorbedWatched(`P`)`
22   **end**
23     $y_i \leftarrow$ `GetMinimalBloatLiteral(`P, $\mathcal{X}_p$`)`
24 **end**
25 **foreach** $y_i \in \mathcal{Y}_u$ **where** $x_i \in \mathcal{X}_p$ **do**
26     $P \leftarrow P \cup \{y_i\}$
27 **end**

4 ZBDD Implementation

In this chapter, we will discuss implementation details regarding ZBDDs, which were introduced in Chapter 2. Like all decision diagram implementations, efficiency of ZBDDs relies heavily on *node sharing* and *result caching*. Node sharing is a mechanism ensuring that if two nodes are identical, i.e. they represent the same function, they are unified into a single instance that is shared between all ZBDDs in the program. Consequently, each node is unique across all ZBDDs, and rather than each ZBDD being a single graph, all ZBDDs in the program form a global graph with multiple roots. This is achieved with the `Getnode()` function proposed by Minato [14] already mentioned in Section 2.2.2. Result caching is closely related to node sharing, as it relies on uniqueness of nodes. Whenever doing an operation over ZBDDs, the corresponding algorithm first checks if an identical operation (with the same arguments) was already performed earlier. If that is the case, the previously computed result is returned from the cache instead of computing it again from scratch.

4.1 Sylvan Library

Because our goal is to implement DP elimination over ZBDDs efficiently, a crucial requirement is to have an efficient ZBDD implementation. This limits the choice of programming language to high-performance, low-level languages like `C`, `C++`, or Rust. In order to efficiently implement node sharing and result caching in these unmanaged languages, it is necessary to utilise careful memory management, including custom *garbage collection (GC)*. This is not an easy task and seems unnecessary work for our thesis. Naturally, we decided to use some existing BDD library that would support ZBDDs. In addition to the already mentioned performance requirements, we also want the library to be open-source and reasonably easy to understand, as we will likely need to add custom ZBDD operations or modify existing ones.

Probably the most well-known and commonly used BDD library is *CUDD* (by Somenzi [19]). Unfortunately, the CUDD library is no longer maintained and we were unable to compile it with deterministic results on validation tests. After some research, we arrived at *Sylvan* (by van Dijk [20]), which is a BDD library implemented in `C` that uses work-stealing for parallel execution. Its implementation is very close to the original algorithms proposed by Minato [14], which convinced us to use it for our application. Note that in the Sylvan codebase, ZBDDs are called *ZDD*s.

A Sylvan ZBDD node carries the following information:

- d-variable (24-bit unsigned integer)
- high edge (1-edge)
- low edge (0-edge)
- 0-element attribute
- terminal node information (if it is a terminal node)

The library uses implicit ordering of d-variables from smallest to largest (starting at 0), so the smallest d-variable is always at the root of a ZBDD. If we wish to change the ordering of d-variables, we need to remap them manually. Note that due to the 24-bit d-variable representation, we are technically limited to formulas with at most $2^{23} \approx 8.3M$ variables.

Notice that the 0-element attribute is associated with a node instead of an edge as it was defined in Section 2.2.1. The semantics in Sylvan are that a node with the 0-element attribute denotes that the edge pointing to the node is attributed. Essentially, the attribute can be interpreted as 'ZBDD represented by this node contains the empty combination'. When ZBDDs in Sylvan are traversed, 0-element attributes are passed down from parents to children — 0-edge copies the attribute to the child, 1-edge scraps the attribute. Unfortunately, the 0-element attribute is not leveraged in any ZBDD operations in Sylvan. If we find this to be a performance bottleneck of our implementation, we could try improving some operations' efficiency by using the attribute.

4.1.1 Node Sharing, Result Caching, and Garbage Collection

A Sylvan ZBDD node does not carry all its information directly. A program variable of type `ZDD` only holds the 0-element attribute and an index into the so-called *unique table* — a pre-allocated table containing the remaining information about all ZBDD nodes currently existing in the application. This table alongside the `zdd_makenode(int var, ZDD low, ZDD high)` function (Sylvan's implementation of `Getnode()` introduced in Section 2.2.2) ensures node sharing. Whenever a new ZBDD node is being created, `zdd_makenode()` first checks if an identical node (same variable and children) already exists in the unique table. If so, it simply returns the unique table index of that existing node, otherwise it must first create a new entry in the table.

In order to be memory efficient, the unique table has to be cleaned up regularly from nodes that are no longer used. For that, Sylvan implements a custom GC algorithm, which is invoked when `zdd_makenode()` tries to add a new entry into the unique table that is already full. At that point, the whole table is traversed, marking nodes that are still in use. Once all nodes are marked, they are copied into a new table twice the original size, leaving out the unmarked nodes to be forgotten.

In order to preform GC, Sylvan needs to know which nodes are still in use and which can be deleted. For this purpose, the library uses a *node protection* mechanism. There are two basic ways the user can protect a ZBDD node from being garbage-collected:

- global protection: `zdd_protect(ZDD *ptr)`, `zdd_unprotect(ZDD *ptr)`,
- local stack-based protection: `zdd_refs_push(ZDD zdd)`, `zdd_refs_pop(int count)`, `zdd_refs_pushptr(ZDD *ptr)`, `zdd_refs_popptr(int count)`.

A protected node is guaranteed to survive GC, and so is the whole ZBDD rooted at this node. Global protection is designed to be used for persistent ZBDDs,

while local stack-based protection is recommended for ZBDDs stored as local variables in the code. As such, local protection stacks are specific for each Lace thread (see Section 4.1.2) and therefore cannot be used from a non-Lace thread. The two variants of local protection differ by what is actually protected — `zdd_refs_push(ZDD zdd)` protects the actual ZBDD passed as a parameter, while `zdd_refs_pushptr(ZDD *ptr)` protects the ZBDD stored under the given pointer (typically local variable), which is useful e.g. for accumulating results.

Because of the thread-specific ZBDD protection stacks, it is necessary that all threads cooperate on GC together. To ensure this cooperation, all Sylvan operations that can create new ZBDD nodes and employ some kind of looping (typically recursive) check whether some other thread has initiated GC. If that happens, the the working thread first cooperates on GC and only after that it continues with its task. In Sylvan codebase, this is achieved with the `sylvan_gc_test()` macro.

In order to ensure correctness of the program, it is essential to use the ZBDD protection mechanism correctly. Unfortunately, the library does not document very well when it is necessary to protect ZBDDs and when it is not. We had to go through a fair share of errors in our program caused by this mechanism, and needed to do some reverse-engineering to figure out the correct usage. In a nutshell, every function that either explicitly participates on GC in its implementation, or that might create a new ZBDD node, can perform GC. Therefore, before calling such functions, all ZBDDs in the program must be protected. Inside those functions, there holds a general assumption that any ZBDDs passed as arguments are already protected by the user, which also implies protection of their children. As a result, in Sylvan's codebase only newly created ZBDDs through other operations are being explicitly protected.

In addition to the unique node table, Sylvan has another global table that is used for caching results of operations. This *cache table* must be cleared during every GC, otherwise the cached results might point to ZBDD nodes that no longer exist in the unique table. Just as the unique table, the cache table doubles its size after it is cleared during GC. Both the initial size and maximum size of these tables are configurable through Sylvan's interface, which needs to be taken into consideration when dealing with large formulas in our DP elimination application.

4.1.2 Lace Framework

To parallelise ZBDD operations, Sylvan uses *Lace*, a work-stealing framework by Dijk and Pol [21]. In this section, we will briefly introduce the framework from a user's point of view. Lace uses a pool of special threads for parallel execution. These Lace threads are created when the framework is initialised by a dedicated function call. Once started, all threads are waiting for a Lace task to be created so that they can steal and execute it. Lace tasks are special functions defined by `C` preprocessor macros, e.g. `TASK_1(int, f, int, arg)`. Once defined, a task can be started in three different ways:

- `RUN(f, args)`: run task `f(args)` from the main (non-Lace) thread or from a Lace thread, wait until its execution is finished (i.e. is **blocking**),
- `CALL(f, args)`: same as `RUN(f, args)`, but can only be called from a Lace thread,

- `SPAWN(f, args)`: run task `f(args)` from a Lace thread **asynchronously**, the result is then obtained by `SYNC(f)` (blocking).

Note that in order to reduce verbosity of calling Lace tasks, it is a common practice in Sylvan to define helper macros for each defined task so that one can simply call `f(args)` instead of `RUN(f, args)`. Lace also offers some synchronisation primitives like *barriers* and *cooperative interruption*, which can be used e.g. for GC in Sylvan.

An important property of Lace is that its worker threads use busy-waiting when idle. While this may reduce latency of starting a task, it can severely hurt performance if used naively. The default behaviour of Lace's initialisation is to start as many threads as available in the system. Unfortunately, the main thread (from which Lace is started) is not taken into account. As a consequence, if one attempts to do some work in the main thread while Lace is already running, the main thread will compete with Lace threads for CPU compute time. There are two ways to avoid this problem: either initialise Lace with at least one thread fewer than available in the system, or use Lace's *suspend and resume* mechanism — suspend all Lace threads when not used, and resume them again before starting a new Lace task. The suspend and resume approach has its own caveats; if not done correctly, it can cause deadlocks and/or segmentation faults (when attempting to run a Lace task while Lace threads are suspended). However, even if used correctly, we still recommend reserving at least one CPU core for the main thread.

Unfortunately, during development of our application we found a race condition in Lace's suspend and resume mechanism (in version 1.3.1). If there's an attempt to suspend the threads right after being resumed (or started), it might happen that some (or even all) threads are not actually suspended. Consequently, a semaphore used in this mechanism becomes inconsistent, which might in some cases even lead to a deadlock. This situation is not very likely to arise during typical resume-suspend cycles, since some work is done between the two Lace calls. However, we observed it quite often when we suspended Lace threads right after initialising Lace. To mitigate the chance of this happening, we used system sleep for a short time, but there is no guarantee of correctness.

4.2 Literal Mapping

When working programmatically with CNF formulas, variables are typically represented simply as positive integers:

$$\begin{gathered} |\mathrm{Var}(\varphi)| = n \\ \upsilon : \mathrm{Var}(\varphi) \to \{1, \dots, n\} \\ \forall x_i, x_j \in \mathrm{Var}(\varphi),\, x_i \neq x_j : \quad \upsilon(x_i) \neq \upsilon(x_j). \end{gathered}$$

For each variable, its positive literal is represented by a positive integer, and its negative literal by the opposite (negative) integer:

$$\begin{aligned} \lambda : \mathrm{lit}(\mathrm{Var}(\varphi)) &\to \{-n, \dots, -1, 1, \dots, n\} \\ \forall x_i \in \mathrm{Var}(\varphi) : \quad & \lambda(x_i) = \upsilon(x_i), \\ & \lambda(\neg x_i) = -\upsilon(x_i). \end{aligned}$$

For obvious reasons, zero is left out in such representation. Unfortunately, we cannot use a direct (identity) mapping from literals to d-variables in ZBDDs for our application. There are two reasons, both of them mentioned in Section 4.1. One, Sylvan represents d-variables in ZBDD nodes as unsigned integers. Two, Sylvan implicitly uses ascending ordering of d-variables in ZBDDs. Therefore, we need to use some non-trivial mapping of literals to ZBDD d-variables.

The implicit d-variable ordering in Sylvan is actually very restrictive for us. In Section 3.1 we made an assumption that every d-variable corresponding to a negative literal will be a successor of the d-variable corresponding to its complementary (positive) literal. This leaves us with essentially only one possible mapping for d-variables:

$$\begin{aligned} \delta : \{y_1, y_{-1}, \ldots, y_n, y_{-n}\} &\to \{1, \ldots, 2n\} \\ \forall i \in \{1, \ldots, n\} : \quad \delta(y_i) &= 2\upsilon(x_i) - 1, \\ \delta(y_{-i}) &= 2\upsilon(x_i). \end{aligned}$$

There is one more thing to consider. In our application, we are likely going to need to transfer between variables/literals and d-variables very often. The conversion from literals to d-variables is straightforward, it is almost explicitly given by the mapping δ (up to the sign of the literal). However, the other direction is slightly cumbersome and can be simplified by shifting the mapping:

$$\begin{aligned} \delta' : \{y_1, y_{-1}, \ldots, y_n, y_{-n}\} &\to \{2, \ldots, 2n+1\} \\ \forall i \in \{1, \ldots, n\} : \quad \delta'(y_i) &= 2\upsilon(x_i), \\ \delta'(y_{-i}) &= 2\upsilon(x_i) + 1. \end{aligned}$$

Now getting a variable from a d-variable can be done by a single integer division:

$$\upsilon(x_i) = \left\lfloor \frac{\delta'(y_i)}{2} \right\rfloor.$$

4.3 Algorithms

In Section 2.2.2 we listed basic operations over ZBDDs that are needed for our implementation of DP elimination. Most of them are implemented in Sylvan; for clarity, we list how the ZBDD operations map to Sylvan's interface:

- "$\mathbf{\emptyset}$" `zdd_false`,
- "$\mathbf{1}$" `zdd_true`,
- P.top `zdd_getvar(`P`)`,
- P_0 `zdd_getlow(`P`)`,
- P_1 `zdd_gethigh(`P`)`,
- $P \cup Q$ `zdd_or(`P`, `Q`)`,
- $P \cap Q$ `zdd_and(`P`, `Q`)`,

- $P - Q$ `zdd_diff(`P`, `Q`)`,
- P`.count` `zdd_satcount(`P`)`,
- $\{\} \in P$ P `& zdd_complement != 0`, where P is a program variable of type `ZDD` and "`&`" is a bitwise AND operator[1].

Unfortunately, the library does not implement all operations that we need, so we will have to provide our own implementation. Namely, this concerns the following:

- P`.offset(`y`)`,
- P`.onset(`y`)`,
- $P * Q$.

Originally, we implemented these methods in our application on top of Sylvan. However, this had both performance and correctness implications. Performance-wise, we could not easily utilise parallelisation using Lace. Regarding correctness, we needed to implement our own result caching, which did not interact well with Sylvan's GC. In the end we decided to create our own fork of the Sylvan library and implement the operations directly in Sylvan's codebase.

For P`.offset(`x`)` and P`.onset(`x`)` operations, Sylvan has a function `zdd_eval(ZDD zdd, int variable, int value)`[2]. Using this function, P`.offset(`x`)` would correspond to `zdd_eval(`P`, `x`, 0)` while P`.onset(`x`)` would correspond to `zdd_eval(`P`, `x`, 1)`. Unfortunately, Sylvan's implementation of these operations only works for the root d-variable or earlier (smaller) d-variables. Therefore, we had to generalise the function — we provided a recursive implementation that works for any d-variable. Our implementation is based on the *onset* and *offset* algorithms described by Minato [14] and is outlined in Algorithm 4.1. In the algorithm, we illustrate one of the uses of ZBDD node protection in Sylvan, how to participate on GC, and how Lace can be used for parallel execution, as explained in sections 4.1.1 and 4.1.2 respectively.

For $P*Q$, we define our own function in Sylvan: `zdd_product(ZDD a, ZDD b)`. Its implementation is also based on Minato [14] and is outlined in Algorithm 4.2. It involves slightly more advanced usage of Lace parallel execution and Sylvan's GC protection. In pseudocodes for both algorithms we use the actual implementation's interface for recursive calls and Sylvan-specific functions, but for other ZBDD operations we use the notation outlined in Section 2.2 for better readability.

With these two algorithms added to our Sylvan fork, we can map the remaining three operations:

- P`.offset(`y`)` `zdd_eval(`P`, `y`, `0`)`,
- P`.onset(`y`)` `zdd_eval(`P`, `y`, `1`)`,

[1] This checks for the 0-element attribute. It is more of a hack than Sylvan's interface, but using the documented interface would result in transferring the whole 0-path from root to terminal, which is unnecessarily inefficient.

[2] In Sylvan, they naturally do not follow our convention of distinguishing between variables in formulas and d-variables in ZBDDs. When referring to specific function signatures, we prefer consistency with the codebase.

Algorithm 4.1: `zdd_eval`(P, var, b)

Input: ZBDD P, d-variable var, Boolean value b (0 or 1)

```
1  if P = ∅ or P = 1 or var < P.top then
2  |  if b = 1 then return ∅ else return P
3  else if var = P.top then
4  |  if b = 1 then return P_1 else return P_0
5  end
6  sylvan_gc_test()
7  R ← cache[zdd_eval(P, var, b)]
8  if R exists then return R
9  SPAWN(zdd_eval, P_0, var, b)
10 H ← zdd_eval(P_1, var, b)
11 zdd_refs_push(H)
12 L ← SYNC(zdd_eval)
13 zdd_refs_pop(1)
14 R ← Getnode(P.top, L, H)
15 cache[zdd_eval(P, var, b)] ← R
16 return R
```

- $P * Q$ `zdd_product(`P, Q`)`.

We have omitted operations concerning subsumption removal defined in Section 2.2.3. Naturally, they are not implemented in Sylvan and we had to add our own implementation as well. We will not provide detailed (pseudo)code for those operations; the algorithm descriptions we provide in Section 2.2.3 include all the necessary pieces. Adapting those algorithms to Sylvan follows the same principles we illustrated for `zdd_eval()` and `zdd_product()`.

Algorithm 2.4 implementing a for-each loop through clauses of a ZBDD is implemented outside of Sylvan's codebase, as it is a read-only operation and does not require GC, node protection, etc. Most of the algorithms defined in Chapter 3 do not work with any low-level details of ZBDDs, they are implemented by using ZBDD operations. As such, they are also implemented outside of Sylvan. The only exception is Algorithm 3.7 for removing tautologies. We included this operation in our Sylvan fork for better efficiency. However, similarly to subsumption removal operations, we will not provide any pseudocode other than the algorithm's description in Section 3.3.

4.4 Building ZBDDs

So far, we have assumed that before running DP elimination (Algorithm 3.16 in Section 3.6) we have somehow obtained a ZBDD representation of the CNF formula we want to process. However, it is very unlikely that the user would prepare their formula as a ZBDD to give it as an input to our application. In fact, typical representation of a CNF formula is a list of clauses, each clause being a list of literals. Moreover, in Section 3.5.2 when describing minimisation algorithms over the watched literals data structure, we made the assumption that we are able to build a ZBDD from a list of clauses. In this section, we will address this

Algorithm 4.2: `zdd_product`(P, Q)

```
    Input: ZBDDs P and Q
 1  if P = ∅ then return ∅
 2  if P = 1 then return Q
 3  if Q = ∅ then return ∅
 4  if Q = 1 then return P
 5  sylvan_gc_test()
 6  if P.top > Q.top then swap P and Q
 7  R ← cache[zdd_product(P, Q)]
 8  if R exists then return R
 9  if P.top = Q.top then
10  |   Q0 ← Q_0
11  |   Q1 ← Q_1
12  else
13  |   Q0 ← Q
14  |   Q1 ← ∅
15  end
16  SPAWN(zdd_product, P_0, Q0)
17  SPAWN(zdd_product, P_0, Q1)
18  SPAWN(zdd_product, P_1, Q0)
19
20  P1Q1 ← zdd_product(P_1, Q1)
21  zdd_refs_push(P1Q1)
22  P1Q0 ← SYNC(zdd_product)
23  zdd_refs_push(P1Q0)
24  T1 ← P1Q1 ∪ P1Q0
25  zdd_refs_pop(2)
26  zdd_refs_push(T1)
27
28  P0Q1 ← SYNC(zdd_product)
29  zdd_refs_push(P0Q1)
30  T1 ← T1 ∪ P0Q1
31  zdd_refs_pop(2)
32  zdd_refs_push(T2)
33
34  P0Q0 ← SYNC(zdd_product)
35  zdd_refs_pop(1)
36  R ← Getnode(P.top, P0Q0, T2)
37  cache[zdd_product(P, Q)] ← R
38  return R
```

missing piece in our design and propose an algorithm for building ZBDDs.

The basic principle of building a ZBDD from a list of clauses is quite straightforward:

- iterate through the list of clauses,
- for each clause C, create a ZBDD representing a set containing only C,
- accumulate these ZBDD clauses in a single set using the union operator.

Let us first address how to create a ZBDD representing a single clause. Sylvan offers us the function `zdd_ithvar(int var)` that creates a single node with d-variable var, its 0-edge going to $\emptyset$ and 1-edge going to $\mathbf{1}$. Essentially, it is a wrapper around $Getnode(var, \emptyset, \mathbf{1})$. However, we cannot combine these one-element combinations into larger combinations using basic ZBDD operations like union and intersection — they operate on the level of combination sets and do not alter individual combinations. Minato [10] defines a basic ZBDD operation P.`change`(y) which can be used to build combinations by adding d-variables one by one. Unfortunately, Sylvan does not implement this operation. We could provide an implementation in our Sylvan fork as we did with `zdd_eval()` and `zdd_product()`, but the operation is unnecessarily general for just building combinations. Instead, we implemented a function that creates a ZBDD combination directly from a list of d-variables: `zdd_combination_from_array(int *variables, int length)`. It uses the most straightforward approach possible — repeated calls to `Getnode()`. The function assumes that the supplied list of d-variables is sorted according to the ZBDDs d-variable ordering, i.e. ascending ordering in Sylvan, and traverses the list in reverse order so that smallest d-variable ends up in the root of the ZBDD. Because the function is implemented in `C`, the list needs to be supplied as a pointer and array length; in our pseudocode in Algorithm 4.3 we simplify these technicalities. Using this function, we can build a ZBDD representing a single clause using Algorithm 4.4.

Algorithm 4.3: `zbdd_combination_from_array`(V)

Input: sorted list of integers representing d-variables V

1 $P \leftarrow \mathbf{1}$
2 **foreach** $v \in V$ *in reverse order* **do**
3 | $P \leftarrow$ `Getnode`(v, $\emptyset$, P)
4 **end**
5 **return** P

Now that we can create a clause, we can build a ZBDD representing an entire formula. As already mentioned, the most straightforward approach is to accumulate the clauses one by one using the union operator, as shown in Algorithm 4.5. Notice the difference between usage of `zdd_refs_pushptr()` for the accumulator variable P and `zdd_refs_push()` for the new clause Q — the accumulator's value changes in each iteration, so it is protected indirectly through a pointer to the variable (denoted by the & before P on line 2).

Algorithm 4.4: `CreateClause`(C)

```
  Input: clause C
1 V ← empty list
2 foreach l_i ∈ C do
3 |  if λ(l_i) > 0 then d-var ← 2λ(l_i)
4 |  else d-var ← 2(−λ(l_i)) + 1
5 |  V.append(d-var)
6 end
7 V ← sort(V)
8 P ← zbdd_combination_from_array(V)
9 return P
```

Algorithm 4.5: `BuildFormula`(φ)

```
   Input: formula φ
 1 P ← ∅
 2 zdd_refs_pushptr(&P)
 3 foreach C ∈ φ do
 4 |  Q ← CreateClause(C)
 5 |  zdd_refs_push(Q)
 6 |  P ← P ∪ Q
 7 |  zdd_refs_pop(1)
 8 end
 9 zdd_refs_popptr(1)
10 return P
```

4.4.1 Logarithmic Merging

While the straightforward algorithm si certainly correct, it might not be very efficient. Once we have accumulated a significant part of the input formula, adding each new clause now requires traversing the whole ZBDD. In order to optimise this, we could borrow an idea from information retrieval used when building large document indexes called *logarithmic merging*. The basic principle is that instead of having a single accumulator, we can have several of them; let us denote them $A_0, A_1, \ldots$. We start the same way as previously, accumulating the resulting ZBDD into A_0. Once A_0 reaches a certain size N — this can be e.g. the number of added clauses — we merge A_0 into A_1. If A_1 was empty before the merging, this simply moves A_0 into A_1 and leaves A_0 empty. However, the next time this happens, A_1 is already full, which means it is now twice the size of N after merging with A_0. This triggers another merging, now of A_1 and A_2, and it continues until some A_k is empty and swallows A_{k-1}. Each accumulator has a size limit twice as large as the previous one — hence the name logarithmic merging. Once all clauses are added, the resulting ZBDD can be obtained by merging (unifying) all the accumulators.

While this technique does not reduce the number of unions that have to be computed (it actually requires some additional ones), the advantage is that most of the unions will happen on small ZBDDs (smaller than N). As the size of the accumulators grows, the number of operations performed decreases. Because the complexity of computing a union of ZBDDs coincides with the ZBDDs' size, the performance gains are very significant. Pseudocode using the logarithmic technique for building ZBDDs is shown in Algorithm 4.6. Note that we have omitted Sylvan's node protection for simplicity.

Algorithm 4.6: `BuildFormulaLogarithmic(`φ, N`)`

```
   Input: formula φ, accumulator size limit N
 1 A ← empty map of accumulators
 2 A[0] ← ∅
 3 foreach C ∈ φ do
 4     if A[0].count = N then
 5         i ← 1
 6         while i ∈ A and A[i] ≠ ∅ do
 7             A[0] ← A[0] ∪ A[i]
 8             A[i] ← ∅
 9             i ← i + 1
10         end
11         A[i] ← A[0]
12         A[0] ← ∅
13     end
14     Q ← CreateClause(C)
15     A[0] ← A[0] ∪ Q
16 end
17 foreach i ∈ A where i ≠ 0 do
18     A[0] ← A[0] ∪ A[i]
19 end
20 return A[0]
```

5 Programming Documentation

In this chapter, we will describe the programming aspects of our implementation of DP elimination. We will describe the project's structure, external libraries and their usage, and give an overview of the software architecture. We will also describe tools we have developed for running experiments conveniently and for processing their results. The whole codebase is provided as Attachment A.1 and all filesystem paths used in this chapter will be relative to the root directory of the attached archive.

5.1 Technical Decisions

Due to our requirement of developing an efficient implementation of DP elimination, we had to choose a programming language suitable for high-performance applications, allowing careful memory management. This disqualified garbage-collected languages like Python, Java, or `C#`. We decided to use `C++` — mostly due to the author's expertise in the language, but also because the Sylvan library (see Section 4.1) is implemented in `C`, which is natively integrated with `C++`. Rust would have been a suitable alternative, but we do not have that much experience with the language. For project management and building, we used the CMake build system. Contrary to the DP elimination application, experiment execution and result processing does not need low-level control, but rather flexibility and ease of use. Therefore, we wrote the convenience scripts in Python.

5.2 Project Structure

We decided to split the application into two parts:

- **dp_lib** (`lib/`) library with all data structures and algorithms,
- **dp** (`app/`) application providing command line interface (CLI).

The main motivation for this separation is that it allows seamless unit-testing of **dp_lib**, as a testing framework can link it as a library. However, it also makes it easier to use some of the library's functionalities in another application, or to provide a different (perhaps graphical) user interface. The `external/` directory contains all external libraries used in either **dp_lib** or **dp**. Unit tests are in the `tests/` directory, and the `experiments/` directory contains Python scripts for convenient experiment execution and results processing.

5.3 External Libraries

In total, we use five external libraries for our `C++` application, all of them are in the `external/` directory. **Sylvan** (Dijk [22]) was already introduced in Section 4.1, it provides the ZBDD data structure. As discussed in Section 4.3, we are using a custom fork of the library with some modifications and custom algorithms. For runtime logging we use **simple-logger** (Zelený [23]), which is a

small library written by the author of this thesis. Originally, it was developed for our application, but because the author found it useful for other projects as well, they decided to single it out as a stand-alone library. The **json** library (Lohmann [24]) is used for exporting application metrics in the JSON format. We use the **Catch2** framework (Hořeňovský [25]) for unit tests of **dp_lib**. Finally, **CLI11** (Schreiner [26]) is used for CLI argument and configuration parsing.

5.4 Software Architecture

The **dp_lib** library is divided into four parts:

- `lib/data_structures`,
- `lib/algorithms`,
- `lib/io`,
- `lib/metrics`.

In this section, we will give an overview of each of these parts. An outlier is the **`utils.hpp`** file which contains general-purpose utilities used throughout the library, typically related to templates. The whole library is encapsulated in the `dp` namespace.

The **dp** application uses **dp_lib** to load input files, run DP elimination, write the result, and export collected metrics. Other than that it contains mostly boilerplate code for parsing CLI arguments and for initialising libraries. We will not go into more detail of the application's source code; user interface and configuration will be described in the user documentation (Chapter 6).

5.4.1 Data Structures

`SylvanZddCnf` is a `C++` wrapper around Sylvan's ZDD. Apart from initialisation and clean-up, this class should be the only point of interaction with the Sylvan library. Its main responsibility is to ensure correct node protection (see Section 4.1.1) employing the *resource acquisition is initialisation (RAII)* pattern. It also provides translation between formula literals and ZBDD d-variables (see Section 4.2) making it transparent, i.e. the user does not know anything about d-variables and only works with literals. When delegating calls to Sylvan, it ensures Lace threads are correctly resumed and suspended (see Section 4.1.2), again employing the RAII pattern. On top of that, the class implements ZBDD algorithms that are not implemented in our Sylvan fork. This includes Algorithm 2.4 for iterating through clauses in the ZBDD (Section 2.2.3), algorithms 3.2, 3.3, and 3.5 extracting information for d-variable selection heuristics (Section 3.2), and algorithms 4.4 and 4.6 for building ZBDDs from CNF formulas (Section 4.4).

Originally, `SylvanZddCnf` also implemented algorithms that were later moved to our Sylvan fork. For those algorithms, it used the **`LruCache`** template for result caching, which implements a quite straightforward *least recently used (LRU)* cache. After moving all those algorithms to Sylvan, the cache is no longer used.

`WatchedLiterals` implements the watched literals data structure for unit propagation on CNF formulas, providing the interface described in Section 3.5.2. It works on top of an existing collection of clauses (can be a vector, set, or even a combination of multiple containers), but it does not copy their data. The class maintains the following information about each clause:

- reference to underlying clause (vector of literals),
- indices of two watched literals in the clause,
- activation flag,

and about each variable:

- assignment (positive, negative, unassigned),
- set of positive watches (clauses that are watching a positive literal of this variable),
- set of negative watches (watching a negative literal).

In addition, the class has an assignment stack (a vector of levels, each level being a vector of literals), a set of unit clauses, and a counter of empty clauses.

As discussed in Section 4.2, variables are represented as unsigned integers, literals are signed integers. Because the set of variables might grow when adding new clauses, information about each variable is kept in a vector indexed by the variables' representation. This is a performance optimisation which might result in additional memory consumption, especially if the variables are spread out (i.e. with many 'holes' between 0 and the highest variable). The assumption is that typically, variables in the represented formula are close together (and to 0).

Because removing clauses from the data structure could be problematic, instead we use an activation/deactivation mechanism — each clause can be deactivated, in which case it does not participate in unit propagation; later on it can be re-activated. Deactivated clauses are removed from all watches in order not to slow down the data structure. Note that activating or deactivating clauses cannot happen with assignment in place, as it could change the course of the already performed propagation. First, the data structure needs to backtrack, than it can activate/deactivate clauses, and assignment/propagation needs to be re-run after that.

`VectorCnf` is a CNF implementation based on vectors, i.e. a vector of clauses, each clause being a vector of literals. The class mirrors the interface of **`SylvanZddCnf`** and serves only for debugging and validation purposes. The implementation is not optimised in any way, and does not even support all features.

5.4.2 Algorithms

`unit_propagation.hpp` implements algorithms connected to unit propagation. The `dp::unit_propagation` namespace implements algorithms 3.8 and 3.9 (Section 3.4). Namespace `dp::absorbed_clause_detection` contains algorithms from Section 3.5 and is further divided into namespaces `without_conversion`

with algorithms 3.10 and 3.11, and `with_conversion` that uses the `Watched Literals` class in algorithms 3.12, 3.13, and 3.15. Note that Algorithm 3.12 is modified according to Section 3.5.4.

Absorption removal over ZBDDs (`without_conversion` namespace) is incomplete and not up-to-date. During development we found that watched literals were overwhelmingly more efficient and the overhead of conversions from/to ZBDDs is insignificant. Early implementation of algorithms 3.10 and 3.11 was kept in the codebase for reference in case someone is interested in testing them. However, they cannot be used through the CLI.

`heuristics.hpp` implements d-variable selection heuristics from Section 3.2 as functor classes. Ordering-based methods from Section 3.2.1 make sure that they select an existing d-variable — it is cheaper than executing the whole elimination step on a non-existent d-variable. Algorithms from Section 3.2.2 are also implemented in some form, but they were used only for development and debugging purposes. The main heuristic `MinimalScoreHeuristic` is a generalised version of Algorithm 3.6 which is templated by the minimised function, i.e. it can use also other functions than $h(x)$ from Equation 3.2.

`dp_elimination.hpp` puts all the pieces together and implements Algorithm 3.16. For easy customisation, the algorithm is parametrised by several functions (callable objects) either serving as predicates in control flow, or used for choosing between various implementations of some functionalities.

5.4.3 Metrics

`MetricsCollector` is a tool for collecting metrics throughout the library. We wanted to use some existing library, but did not find anything suitable for our use-case. The main focus of the tool is minimal overhead, and easy modification of the collected metrics. The collector supports four types of metrics:

- *counter* integer value, accumulated,
- *series* list of integers, appended,
- *duration series* list of durations, appended,
- *cumulative duration* duration value, accumulated.

Each metric entry has a unique identifier and is of one of those types. To make the individual entries easily modifiable, they are given as template arguments of the class. Integer metrics (counter and series) are collected through simple method calls. Duration metrics (series and cumulative) employ the RAII pattern through the `MetricsCollector::Timer` class, i.e. the timer starts when created and stops when destroyed. However, the interface also allows stopping the timer explicitly in order to allow seamless integration of timers in existing code without introducing unnecessary code blocks.

dp_metrics.hpp defines identifiers of metrics collected in the **dp_lib** library. It also creates an instance of `MetricsCollector` with those identifiers. These metrics are exported in JSON once the DP elimination finishes, allowing analysis and visualisation of the collected data in order to evaluate the algorithm. The format of exported data will be described in the user documentation (Section 6.6.2).

5.4.4 IO

Input/output operations of CNF formulas are handled by **CnfReader** and **CnfWriter**. They work with the *DIMACS CNF* format, which will be described in detail in the user documentation (Section 6.6.1). The format is very simple and easy to parse, so we wrote our own reader/writer to keep dependencies to a minimum. `CnfReader` is just a static class encapsulating a couple of related functions. It can either parse the input file into a vector of vectors, or the user can supply it with a custom callback function to process each clause individually (e.g. to create a ZBDD). `CnfWriter` is an ordinary class with internal state. It also offers out-of-the-box support for a vector of vectors, or it can be used manually to write clauses one by one.

5.5 Experiment Execution and Result Processing

Alongside the `C++` codebase implementing DP elimination, Attachment A.1 also includes several Python scripts in the `experiments/` directory for convenient execution of experiments and processing of their results. Note that these scripts are not intended to be a part of the user interface of our application; they served us as a tool when writing our thesis. The reason we include them in the attached codebase is to provide an example of utilising the metrics collected by the **dp** application. As such, the code quality of the Python scripts is not as high as of **dp** — it might require more extensive in-code adjustments when tweaking their behaviour.

The main functionality of these scripts is running **dp** over multiple input formulas under several configurations. That is, given a list of paths to input formulas and a list of configuration files, they are all combined — each formula with each configuration — and run through **dp** in a single script execution. Then, by another script invocation, metrics from all runs can be collected and summarised into various tables, plots, etc. We use `DataFrame`s from the **Pandas** library for processing the data and exporting tables. Currently supported export formats for tables are Markdown, LaTeX, CSV, and JSON; more can be easily added by adjusting `experiments.py`. For plotting we use the **matplotlib** library, and support all output formats supported by the library.

Aside from the Python scripts, the `experiments/` directory also includes a list of input formula paths `input_formulas.txt`, default configuration file `default_config.toml` for **dp**, and `environment.yml` which can be used to reproduce the Python environment necessary for running the scripts.

`experiments.py` is the main script providing CLI for the other scripts. It also generates all formula–configuration pairs from the given lists.

`run.py` implements execution of the **dp** application. The script is slightly overcomplicated due to its support for parallel execution of different runs. Later, it was adjusted to be suitable for execution — especially as an *array job* — under the *SLURM* workload manager (by Yoo et al. [27]) used on *high-performance computing (HPC)* clusters.

`tables.py` creates tables for each formula–configuration pair showing the data from exported metrics in a more human-readable format.

`plots.py` creates plots for each formula–configuration pair visualising how DP elimination proceeded over time.

`summary_table.py` creates a table summarising the whole experiment, extracting useful data to compare different configurations.

`summary_plots.py` creates plots from the summary table created by `summary_table.py`.

`grid_search.py` is a stand-alone script on the side — it is not executed through `experiments.py`, but provides its own CLI. It uses a separate list of input files and configurations for searching the parameter space of **dp** to find the optimal parameter setting.

`equivalence.py` is another stand-alone script. It takes two formulas as arguments and checks if they are equivalent; we used it for verifying the results of our program. Note that the script uses the **pysat** library, which is not included as a dependency listed in the `environment.yml` file — it has to be installed manually (pipy package called *python-sat*).

6 User Documentation

This chapter describes the **dp** application (see Section 5.2) from a user's point of view. We show how to compile the application, how to run it, what are its parameters and what outputs the application produces. We also provide examples of the usage.

6.1 Prerequisites

In order to compile and run **dp**, the following software is needed:

- Linux-based system,
- CMake (version 3.14 or later),
- `C++` compiler supporting `C++20` (GNU version 10 or later).

The application was only tested on Linux-based systems, support for other operating systems is not guaranteed. CMake is a tool for generating build files for systems like *GNU Make*, *Ninja*, etc. For compilation, we used the GNU compiler, but any compiler supporting `C++20` should work as well. Note that first-time compilation might also require internet connection in order to download external dependencies of the Sylvan library (see Section 4.1).

6.2 Compilation and Tests

We assume that the user has obtained the codebase in Attachment A.1 and that their terminal's current directory is the root directory of the attachment. In this chapter, we will refer to this directory as the *root directory*. We also assume that all prerequisites from Section 6.1 are met. In order to compile the **dp** application, run the following commands:

```
mkdir build
cd build
cmake ..
cmake --build . -j6
```

The `-j6` option tells the build system to use 6 threads for the compilation; this constant is quite arbitrary and should be adjusted according to the user's hardware. By default, CMake uses the GNU Make build system for building the application. While it should be available on any Linux-based operating system, there are more modern and more efficient tools that can be used instead. For example, to use Ninja instead of Make, the third command can be substituted for the following:

```
cmake -G Ninja ..
```

Also, the default build type is set to *Release*. For a *Debug* build, the user can add a different option to the third line (it can be combined with the Ninja option):

```
cmake -DCMAKE_BUILD_TYPE=Debug ..
```

After executing these commands, the newly created `build/` directory should contain an executable file called `dp`, which is the executable of the **dp** application. The directory should also contain the `tests` executable. By running

```
./tests
```

from the `build/` directory, unit tests of the **dp_lib** library (see Section 5.2) will be executed. This can be used to verify correctness of the compiled code.

6.3 Program Stack Size Limit

Many of the algorithms used in our codebase, and especially those in the Sylvan library, are implemented recursively. This results in unusually high usage of the program stack by the **dp** application. The required stack size depends on the input formula. On Linux-based systems, the corresponding user limit is typically set to 8 MB, which is insufficient for larger formulas. If **dp** hits this limit, it has no other option than to fail with segmentation fault. Therefore, it is important to increase the limit before running the application on large formulas. For inputs and configurations included in Attachment A.1 we managed to run successfully on stack size limit of 16 GB, although this is very generous and for reasonable configurations (non-experimental) much smaller stack should be enough.

On Linux, the user stack size limit can be managed with the `ulimit` command. Running

```
ulimit -s
```

displays the current limit in KB. In order to increase this limit to 16 GB, run

```
ulimit -s 16777216
```

before running the **dp** application. Note that setting the limit to `unlimited` might cause problems with initialising the Lace framework used by Sylvan.

6.4 Parameters and Options

The **dp** application provides a *command line interface (CLI)* with standard Unix syntax. After following compilation steps in Section 6.2, running

```
build/dp --help
```

from the root directory will print out the help message of the application, listing all the parameters and options. There is one positional parameter — path to the input formula in the DIMACS CNF format (see Section 5.4.4). Aside from that, there are many options divided into several categories modifying the application's behaviour.

6.4.1 Files

The **Files** category includes settings concerning output files of the application.

`--output-file` sets a path where the output formula after DP elimination is written.

The default path is `result.cnf`.

`--output-max-size` limits the size of the output formula by the maximum number of clauses allowed in the output. If the formula is larger, the output file is not created. This is a prevention mechanism to avoid cluttering the hard drive. As mentioned at the beginning of Chapter 1, the formula can grow exponentially during DP elimination. Due to efficient representation of the formula in-memory, the output file can also be exponentially larger than the in-memory representation, resulting in infeasibly large files.

The default value is the largest integer representable on the system (essentially unlimited).

`--metrics-file` sets a path for exporting the metrics file in JSON format (Section 5.4.3). The format of the metrics will be described in Section 6.6.2.

The default path is `metrics.json`.

`--log-file` sets a path for writing logs of the application. These logs can be used for debugging, or for detailed observation of the course of the algorithm.

The default path is `dp.log`.

6.4.2 Algorithm

The **Algorithm** category specifies which implementation of some part of DP elimination is used where multiple implementations are available.

`--heuristic` specifies how variables are selected for elimination (Section 3.2). These are the available arguments:

1. `ascending` (variables from smallest to largest),
2. `descending` (variables from largest to smallest),
3. `minimal_bloat` (Algorithm 3.6).

This is a required option without a default value.

6.4.3 Complete Minimisation

The **Complete minimisation** category customises when Algorithm 3.11 or 3.13 (depending on `--absorbed-removal-algorithm`) is executed.

--complete-minimization-condition decides what kind of condition is checked in each iteration of DP elimination to decide if complete minimisation should be performed. These are the available arguments:

1. `never`,
2. `interval` in regular intervals (after a certain number of iterations),
3. `relative_size` when the formula has grown by a certain ratio since last minimisation.

The default value is `relative_size`.

--complete-minimization-interval sets the number of iterations of DP elimination (number of eliminated variables) between two consecutive runs of complete minimisation.

Requires `--complete-minimization-condition` to be `interval`.

The default value is 1 (run always).

--complete-minimization-relative-size sets the ratio of growth of the formula before complete minimisation is run. For example, a ratio of 2 means that if the formula has N clauses after complete minimisation, the next complete minimisation will be run once the formula reaches at least $2N$ clauses.

Requires `--complete-minimization-condition` to be `relative_size`.

The default value is 1.5.

6.4.4 Partial Minimisation

The **Partial minimisation** category customises when the formula is partially minimized. Contrary to complete minimisation (Section 6.4.3), partial minimisation is not performed on a single formula, but when unifying two formulas. Looking at Algorithm 3.16, this union is of P' (clauses not containing the eliminated variable) and P_{new} (resolvents of clauses containing the eliminated variable). Partial minimisation either performs subsumption-free union (Algorithm 2.3), or absorption-free union (Algorithm 3.15). The choice depends on the setting of incremental absorption removal, which will be described in Section 6.4.5.

--partial-minimization-condition decides what kind of condition is checked in each iteration of DP elimination to decide if partial minimisation should be performed. These are the available arguments:

1. `never`,
2. `interval` in regular intervals (after a certain number of iterations),
3. `relative_size` when the size of P_{new} is at least a certain ratio of the size of P',
4. `absolute_size` when P_{new} has at least some constant number of clauses.

The default value is `relative_size`.

--partial-minimization-interval sets the number of iterations of DP elimination (number of eliminated variables) between two consecutive runs of partial minimisation.

Requires `--partial-minimization-condition` to be `interval`.

The default value is 1 (run always).

--partial-minimization-relative-size sets the minimum size ratio of P' and P_{new} in order to run partial minimisation. For example, a ratio of 0.2 means that if P' has N clauses, partial minimisation will be run if P_{new} has at least $0.2N$ clauses.

Requires `--partial-minimization-condition` to be `relative_size`.

The default value is 0.1.

--partial-minimization-absolute-size sets the minimum size of P_{new} in order to run partial minimisation.

Requires `--partial-minimization-condition` to be `absolute_size`.

The default value is 0 (run always).

6.4.5 Incremental Absorption Removal

The **Incremental absorption removal** category customises when absorption-free union (Algorithm 3.15) is executed. As noted in Section 6.4.4, incremental absorption removal is performed instead of subsumption-free union during partial minimisation. Therefore, even if the condition for incremental absorption removal is met, it will be only run if partial minimisation is also triggered (see Algorithm 3.16).

--incremental-absorption-removal-condition decides what kind of condition is checked in each iteration of DP elimination to decide if incremental absorption removal should be performed. These are the available arguments:

1. `never`,
2. `interval` in regular intervals (after a certain number of iterations),
3. `relative_size` when the size of P_{new} is at least a certain ratio of the size of P',
4. `absolute_size` when P_{new} has at least some constant number of clauses.

The default value is `relative_size`.

--incremental-absorption-removal-interval sets the number of iterations of DP elimination (number of eliminated variables) between two consecutive runs of incremental absorption removal.

Requires `--incremental-absorption-removal-condition` to be `interval`.

The default value is 1 (run always).

`--incremental-absorption-removal-relative-size` sets the minimum size ratio of P' and P_{new} in order to run incremental absorption removal. For example, a ratio of 0.2 means that if P' has N clauses, incremental absorption removal will be run if P_{new} has at least $0.2N$ clauses.

Requires `--incremental-absorption-removal-condition` to be `relative_size`.

The default value is 0.1.

`--incremental-absorption-removal-absolute-size` sets the minimum size of P_{new} in order to run incremental absorption removal.

Requires `--incremental-absorption-removal-condition` to be `absolute_size`.

The default value is 0 (run always).

6.4.6 Stop Conditions

The **Stop conditions** category sets the stop conditions for DP elimination. There are multiple conditions that are checked in conjunction — if one of them becomes true, the algorithm stops and outputs the results. If this happens before all auxiliary variables are eliminated, the remaining ones are still included in the resulting formula. In that case, they are guaranteed to have maintained the numbering from the input formula. There are also some implicit stop conditions — when an empty clause occurs in the formula, or when the formula becomes empty (all variables are eliminated).

`--max-iterations` sets the maximum number of iterations (eliminated variables) before stopping.

Unlimited if not set.

`--timeout` sets the maximum runtime duration of the algorithm in seconds. The measured runtime only concerns the DP elimination algorithm — it excludes initialisation, input parsing, etc. Note that this limit may be exceeded, as the algorithm waits for the on-going operation to finish. In most cases, this should not be more than a few seconds, but in some rare situations (excessively large formulas), this can be significant (even a couple of minutes, but theoretically without any limit).

Unlimited if not set.

`--max-formula-growth` sets the maximum size of the formula relative to the input. The size limit is expressed as a ratio — for example, a ratio of 2 means that once the formula becomes twice as large as the input formula, the algorithm stops. Note that the last iteration is not reversed in that case, i.e. the output formula *exceeds* the size limit.

Unlimited if not set.

`--var-range` defines the range of variables in the input formula to be eliminated. Effectively, variables outside of this range are protected from elimination (see

Chapter 3). Once the formula does not contain any more variables from this range, execution stops.

Accepts two integers separated by whitespace.

Unlimited if not set.

6.4.7 Sylvan

The **Sylvan** category provides settings for the Sylvan library (see Section 4.1).

`--sylvan-table-size` sets the default and maximum size of Sylvan's unique table. The size is defined in the number of entries (ZBDD nodes). A single entry takes up 24 bytes of memory. Because Sylvan only allows the size to be a power of 2, the values of this setting is expressed as a base-2 logarithm. The resulting memory consumption is therefore expressed as $2^k \cdot 24\text{B}$; for $k = 20$, this results in 24MB. If the program approaches the maximum size, it attempts to export collected data and terminates with an error; however, the situation might not be recoverable.

Accepts two integers separated by whitespace.

The default values are 20 (default), 25 (max).

`--sylvan-cache-size` sets the default and maximum size of Sylvan's cache table. The size is defined in the number of entries (cached operation results). A single entry takes up 36 bytes of memory. Because Sylvan only allows the size to be a power of 2, the values of this setting is expressed as a base-2 logarithm. The resulting memory consumption is therefore expressed as $2^k \cdot 36\text{B}$; for $k = 20$, this results in 36MB.

Accepts two integers separated by whitespace.

The default values are 20 (default), 25 (max).

`--lace-threads` sets the number of Lace threads (see Section 4.1.2). Because this number does not include the main thread, and because waiting Lace threads are busy-waiting, **we strongly recommend using fewer threads than the number of cores on the system**. Otherwise the user risks considerable slow-down of the application when Sylvan operations are interleaved with code executed in the main thread (e.g. when building a ZBDD). If the value is set to 0, Lace auto-detects available cores on the system. Therefore, **we do not recommend using 0**[1].

The default value is 1 (no parallelisation).

6.5 Configuration Files

Instead of supplying all the options listed in Section 6.4 directly to **dp**, the user can group them in a configuration file, which can be supplied using the **`--config`** option. The option names and values stay the same, only the names are stripped

[1] An easy experiment to observe the impact of this setting is to modify `tests/main.cpp` to use `lace_start(0, 0)` instead of the current values. Compare how long it takes to run unit tests (see Section 6.2) before and after the modification.

of the leading double-dash, i.e. `--option` becomes `option`. This approach makes it easier to use the same configuration for multiple runs of the application, or to use some automation for processing many input files. It is possible to supply multiple configuration files; in case of conflicting options, the later config overrides the former one. Similarly, if an option is given explicitly to the application, it overrides all previously supplied configuration files. The supported formats of configs are *INI* and *TOML*. An example of a configuration file in the TOML format can be the following:

```
output-file = "result.cnf"
log-file = "dp.log"
metrics-file = "metrics.json"
output-max-size = 20_000_000

heuristic = "minimal_bloat"

complete-minimization-condition = "relative_size"
complete-minimization-relative-size = 1.6

partial-minimization-condition = "relative_size"
partial-minimization-relative-size = 0.1

incremental-subsumption-removal-condition = "relative_size"
incremental-subsumption-removal-relative-size = 0.4

timeout = 3600 # seconds (1h)

sylvan-table-size = [20, 28] # 24 MB, 6 GB
sylvan-cache-size = [20, 28] # 36 MB, 9 GB
lace-threads = 3
```

6.6 File Formats

In this section, we describe file formats used by the **dp** application.

6.6.1 DIMACS CNF

The *DIMACS CNF* format, which is an industry standard for logical formulas in CNF for automated processing, is used for input and output formulas. It is not very precisely defined; there are several variants and extensions, and different parsers have various support and requirements. We use a definition that is simple enough for our use-case and compatible with most potential input files:

- a line beginning with the character c is a comment,
- the first non-comment line is the problem definition header with the following format:

  ```
  p cnf <variables> <clauses>
  ```

where <variables> indicates the number of variables and <clauses> indicates the number of clauses,

- after the header follow the clauses, typically one clause per line,
- each clause is a sequence of literals separated by white-space,
- positive/negative literals are represented by positive/negative integers,
- each clause ends with a 0 (which is never a literal).

For example, the formula $(\neg x_1 \lor x_2) \land (\neg x_2 \lor x_3)$ could be encoded as follows:

```
c This is an example DIMACS CNF file
p cnf 3 2
-1 2 0
-2 3 0
```

While our application does require the problem definition header, it does not fail if the number of variables or clauses does not match the declared values — it only issues a warning if such discrepancy occurs.

6.6.2 Metrics Export

The **dp** application collects various metrics while performing DP elimination. These metrics are then exported into a JSON file with the following structure:

```
{
  "counters": {
    ...
  },
  "cumulative_durations": {
    ...
  },
  "durations": {
    ...
  },
  "series": {
    ...
  }
}
```

Each of the top-level elements represents a category of metrics and contains nested elements with data. *Counters* are single integer values, while each *series* element contains a list of integer values. Duration-based metrics are measured in **microseconds**; *cumulative_durations* contain a single value, *durations* are similar to series — carrying a list of durations per element. Note that while all elements are sorted by their name within each category, we might not follow this ordering in this documentation. Also note that some metrics are only relevant with certain program options. In particular, some series or durations might contain empty lists in certain configurations.

Counters contain the following metrics:

- `MinVar` minimum variable allowed to be eliminated,
- `MaxVar` maximum variable allowed to be eliminated,
- `InitVars` number of variables in the input formula,
- `FinalVars` number of variables in the output formula,
- `EliminatedVars` number of eliminated variables,
- `RemoveAbsorbedClausesCallCount`
  how many times complete minimisation was performed,
- `AbsorbedClausesRemoved`
  total number of removed absorptions,
- `AbsorbedClausesNotAdded`
  total number of absorbed clauses detected during incremental minimisation,
- `UnitLiteralsRemoved`
  total number of variables eliminated by unit propagation,
- `WatchedLiterals_Assignments`
  total number of assignments performed by the watched literals data structure.

Note that in general

$$\texttt{FinalVars} \neq \texttt{InitVars} - (\texttt{EliminatedVars} + \texttt{UnitLiteralsRemoved}),$$

because some of the removed unit literals might have been added back to the results (see Section 3.4). Moreover, even though it is unlikely, it can happen that some variable disappears during some other variable's elimination (due to various minimisation methods).

Series contain the following metrics:

- `EliminatedLiterals`
  literals in their order of elimination,
- `ClauseCounts` number of clauses in each iteration,
- `NodeCounts` number of ZBDD nodes in each iteration,
- `HeuristicScores` score assigned to the literal selected for elimination,
- `ClauseCountDifference`
  difference between number of clauses before and after each elimination; includes incremental minimisation, excludes complete minimisation and unit propagation,
- `AbsorbedClausesRemoved`
  number of removed absorptions during each complete minimisation,

- `AbsorbedClausesNotAdded`
  number of absorbed clauses detected during each incremental minimisation,
- `UnitLiteralsRemoved`
  number of variables eliminated by unit propagation in each iteration.

Note that in general

$$\texttt{ClauseCountDifference}[i] \neq \texttt{ClauseCounts}[i+1] - \texttt{ClauseCounts}[i],$$

because `ClauseCounts` are measured after complete minimisation and unit propagation.

Cumulative durations contain the following metrics:

- `WatchedLiterals_Propagation`
  time spent by unit propagation in the watched literals data structure,
- `WatchedLiterals_Backtrack`
  time spent by backtracking in the watched literals data structure.

Durations contain the following metrics:

- `ReadInputFormula`
  total time spent by reading the input formula and building its ZBDD,
- `WriteOutputFormula`
  total time spent by writing the output formula into a file,
- `ReadFormula_AddClause`
  time spent by reading each clause of the input formula,
- `WriteFormula_PrintClause`
  time spent by writing each clause of the output formula,
- `AlgorithmTotal` total time spent by the DP elimination algorithm,
- `RemoveAbsorbedClauses_Serialize`
  time spent by serialising a ZBDD into watched literals during each complete minimisation,
- `RemoveAbsorbedClauses_Search`
  time spent by searching for absorptions during each complete minimisation,
- `RemoveAbsorbedClauses_Build`
  time spent by building a ZBDD from watched literals during each complete minimisation,
- `IncrementalAbsorbedRemoval_Serialize`
  time spent by serialising a ZBDD into watched literals during each incremental minimisation,

- `IncrementalAbsorbedRemoval_Search`
  time spent by searching for absorbed clauses during each incremental minimisation,
- `IncrementalAbsorbedRemoval_Build`
  time spent by building a ZBDD from watched literals during each incremental minimisation,
- `VarSelection` time spent by selecting a literal to be eliminated in each iteration,
- `EliminateVar_Total`
  time spent by eliminating each variable; excludes literal selection, full minimisation, and unit propagation,
- `EliminateVar_SubsetDecomposition`
  time spent by subset decomposition in each iteration,
- `EliminateVar_Resolution`
  time spent by clause resolution in each iteration,
- `EliminateVar_TautologiesRemoval`
  time spent by removing tautologies in each iteration,
- `EliminateVar_Unification`
  time spent by computing union in each iteration; includes subsumed removal and incremental minimisation.

6.7 Examples

We provide a couple of examples of using the **dp** application. Let us assume that we are in the root directory of our project and we have successfully compiled the application using instructions in Section 6.2 Additionally, we also have the configuration example from Section 6.5 saved as `config.toml`, and an input file in the DIMACS CNF format as `formula.cnf`. Example configuration files and formulas are included in Attachment A.1. Then by running

```
build/dp --config config.toml formula.cnf
```

we could get the following output:

```
Reading input formula from file formula.cnf...
Input formula has 21254 clauses
Eliminating variables...
Formula with 29782 clauses written to file result.cnf
Exporting metrics to metrics.json
```

Detailed information about the program's progress was logged into the `dp.log` file.

We might wish to give the program some extra options in addition to the configuration file:

```
build/dp --config config.toml --var-range 110 485 formula.cnf
```

This time, only variables starting from 110 ending with 485 will be eliminated.

We can also override some options set in the `config.toml` file:

```
build/dp --config config.toml --heuristic ascending formula.cnf
```

Now instead of the minimal bloat heuristic for variable selection we eliminate variables in ascending order.

It is also possible to use multiple configuration files at once:

```
build/dp --config config.toml --config another_config.toml formula.cnf
```

If there are conflicting options in the two configuration files, the latter (`another_config.toml`) overrides the former (`config.toml`).

7 Experiments

With our DP elimination application finished, we ran several experiments in order to evaluate the techniques we used. As outlined in Chapter 1, our main focus was to determine efficiency of absorption removal described in Section 3.5, and to compare variable selection heuristics from Section 3.2. In addition, we also tested the effect of parallelisation on ZBDD operations given its support in the Sylvan library (see Section 4.1). Finally, we looked at the whole pipeline described in Chapter 1 — encoding a CNF formula into DNNF and back to CNF, and then reducing it with our application to obtain a PC formula. We compare this approach with Kučera [3] on relevant formulas.

7.1 Inputs

As inputs for our experiments, we used a combination of formulas from various real combinatorial problems and some randomly generated formulas, all taken from the benchmark collection by Koriche et al. [28]. We chose formulas of such size so that a substantial amount of auxiliary variables (if not all of them) could be eliminated in a reasonable amount of time, which we decided to be one hour. All inputs are included in Attachment A.1 under the `experiments/inputs/` directory. Each formula has several associated files, we list them in their order of creation:

- `*.cnf` the original formula,
- `*.nnf` DNNF encoding of the formula,
- `*.dc.cnf` CNF encoding of the DNNF formula,
- `*.dc.min.cnf` minimised version of the CNF encoding (removed absorptions),
- `*.dc.min.toml` **dp** configuration file with formula-specific options.

For compilation into DNNF we used the modern **Bella** compiler by Illner and Kučera [29] instead of more common compilers by Darwiche [6] or Lagniez and Marquis [7]. Domain-consistent encoding of the DNNF formula back into CNF was computed with algorithm **FullNNF** by Abío et al. [8]. We used an implementation included in the tool **PCCompile** by Kučera [30]. It might be relevant that this implementation uses *postorder* numbering of DNNF nodes, i.e. nodes in a subgraph are assigned a lower number than the root of the subgraph. The minimised formulas were also obtained using the **PCCompile** tool. We use the minimised version of the CNF encoding (`*.dc.min.cnf`) as input of the **dp** program. The TOML configuration file is used for specifying which variables should be eliminated from each formula (the `--var-range` option, see Section 6.4.6).

Regarding size of the formulas, we filtered the benchmark collection in the following way:

- we selected only CNF formulas with < 400 variables,
- we compiled them with **Bella** with timeout of 300s and ignored the rest,
- we selected NNF formulas with < 10000 nodes in the header.

7.2 Setup

We ran all experiments on the Chimera cluster at Faculty of Mathematics and Physics, Charles University, Prague. The experiments reported in this chapter were run on homogenous hardware with the following specifications:

- CPU AMD EPYC 7543 (2.8 GHz),
- cores per task 4,
- RAM per task 32 GB.

Before running the experiments, it is necessary to increase the system stack size limit as explained in Section 6.3.

Regarding options for the **dp** program, we use a default configuration common for all experiments and adjust/supplement it with experiment-specific configurations. See sections 6.4 and 6.5 for details about program options and configuration files. Each experiment is conducted on all input formulas. The default configuration is as follows:

```
output-max-size = 20_000_000
sylvan-table-size = [20, 28]
sylvan-cache-size = [20, 28]
lace-threads = 3
```

Notice that one core is reserved for the main thread, as recommended in Section 6.4.7. Also, by default we use the `minimal_bloat` variable selection heuristic unless stated otherwise.

Apart from variable selection heuristics, which are one of the subjects of our experiments, **dp** has several other options that directly modify the behaviour of DP elimination. In particular, these are the options regarding complete minimisation (Section 6.4.3), partial minimisation (Section 6.4.4), and incremental absorption removal (Section 6.4.5). We conducted a parameter search on a limited subset of the input formulas to find optimal parameters for these settings (when they are activated). However, the results of the search were not very conclusive — different parameters were better for different formulas and we did not find any clear way of extrapolation. In the end, we gave more weight to formulas that were not randomly generated, but even among those there were very few clear trends. In general, we find the `relative_size` condition most reasonable (unless we choose `never`). If all three minimisation techniques are activated, we chose the following parameters:

```
complete-minimization-condition = "relative_size"
complete-minimization-relative-size = 1.5

partial-minimization-condition = "relative_size"
partial-minimization-relative-size = 0.1

incremental-absorption-removal-condition = "relative_size"
incremental-absorption-removal-relative-size = 1.0
```

7.3 Algorithm Breakdown

Before evaluating the experiments, we first want to give some insight into the course of the DP elimination algorithm. We will look at a run of the algorithm on a single input formula, using the default configuration shown in Section 7.2 with two modifications:

- incremental absorption removal is disabled,
- complete minimization relative size is set to 1.1.

This configuration results in more illustrative plots. The formula of choice is `instancesCompilation/Handmade/ais/ais6`. Table 7.1 shows a summary of the run. Some metrics in the table can be directly mapped to metrics described in Section 6.6.2, others are derived from them.

initial variables	261	read duration	72.0 ms
final variables	61	write duration	5.9 ms
eliminated variables	199	DP algorithm duration	23.7 s
initial clauses	1497	variable selection	40.3 ms
final clauses	2118	elimination	315.2 ms
removed unit literals	3	absorption removal	23.3 s
removed absorbed clauses	52196	unit propagations per second	622274
heuristic correlation	0.956	backtrack-to-propagation ratio	0.0021

Table 7.1 Run summary.

The first thing worth pointing out is that absorption removal (i.e. complete minimisation) takes up almost the whole runtime duration of the algorithm (23.3 out of 23.6 seconds). Such dominance of absorption removal is a trend we see on almost all input formulas. We can also see that parsing the input file (which includes building a ZBDD) and writing the resulting formula takes negligible time in comparison to actual DP elimination. If we compare variable selection to the elimination stage, it is quite significant. On the other hand, the heuristic predicting the formula's growth seems to be very accurate, judging by its high correlation with observed formula growth.

In Figure 7.1 we see how the size of the formula develops throughout the algorithm. There are two curves in the plot — one for the number of clauses (i.e. formula size), one for ZBDD nodes. Similarly to this run, in most cases the number of ZBDD nodes is ≈ 1.5 times larger than the size of the formula. Looking at the curves, we see that for roughly the first half of eliminated variables the formula size does not change much (or declines slightly). This illustrates how the algorithm first eliminates variables with low heuristic score (see Equation 3.2). Then between 100 and 150 eliminated variables the formula starts growing as no more 'cheap variables' are available, and the second half of the plot shows spikes of growth and decline with increasing frequency. These spikes mark where some minimisation happens, suddenly reducing the formula's size. It is not immediately clear what behaviour should be attributed to complete minimisation (absorption removal) and what behaviour to partial minimisation (subsumption removal), sometimes the effect is also combined.

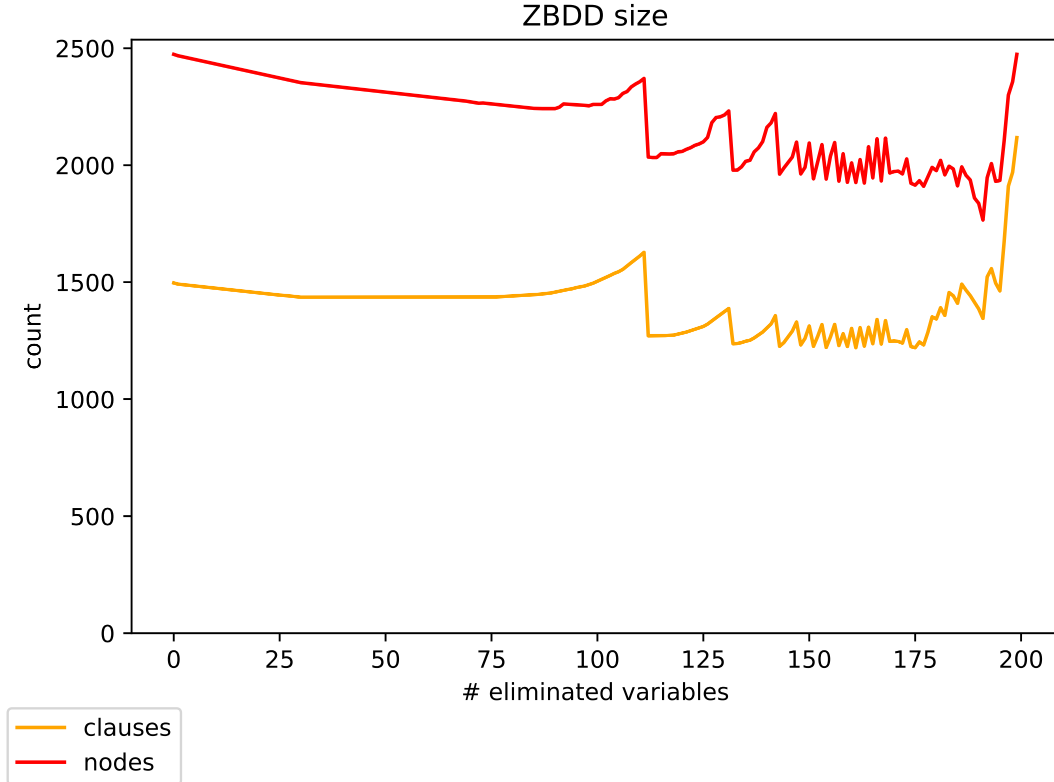

Figure 7.1 Formula size.

Let us look in detail at variable elimination. In Figure 7.2 we see the main part of the algorithm broken down into all its phases, the y-axis showing their duration. There are several interesting trends shown by the plot. Firstly, notice how at the beginning of the algorithm, the total duration follows certain cycles — first it linearly grows, then it suddenly drops down. Looking at the application log, we noticed that this behaviour strongly correlates with the eliminated variable and its position in the ZBDD d-variable ordering. Due to the implementation of the minimal bloat heuristic (see Algorithm 3.6), when there are multiple variables with the same heuristic score, we eliminate the smallest variable (earliest in the d-variable ordering). The first ≈ 50 eliminated variables all had heuristic score 0, the next ≈ 20 had score 1, then again ≈ 50 variables with score 3, etc. Each time the minimal heuristic score increases, the position of the eliminated variable drops to the beginning of the ZBDD variable ordering, which results in much shorter *subset decomposition* stage. This is not that surprising — subset decomposition is achieved using the `offset()` and `onset()` operations, which are more efficient for d-variables closer to the ZBDD root.

The second observation we can make based on Figure 7.2 concerns the phases of variable elimination. It is clear that most of the time, the dominant phase is subset decomposition. However, in the last part of DP elimination, the *unification* phase takes over and significantly slows down the whole algorithm. The reason is quite clear — due to the configuration we use, the algorithm does not perform any partial minimisation unless P_{new}.`count` is at least 10 % of P'.`count` (see Section 6.4.4). As a result, partial minimisation (in our case only subsumption-free union, see Algorithm 2.3) only contributes once the formula starts growing

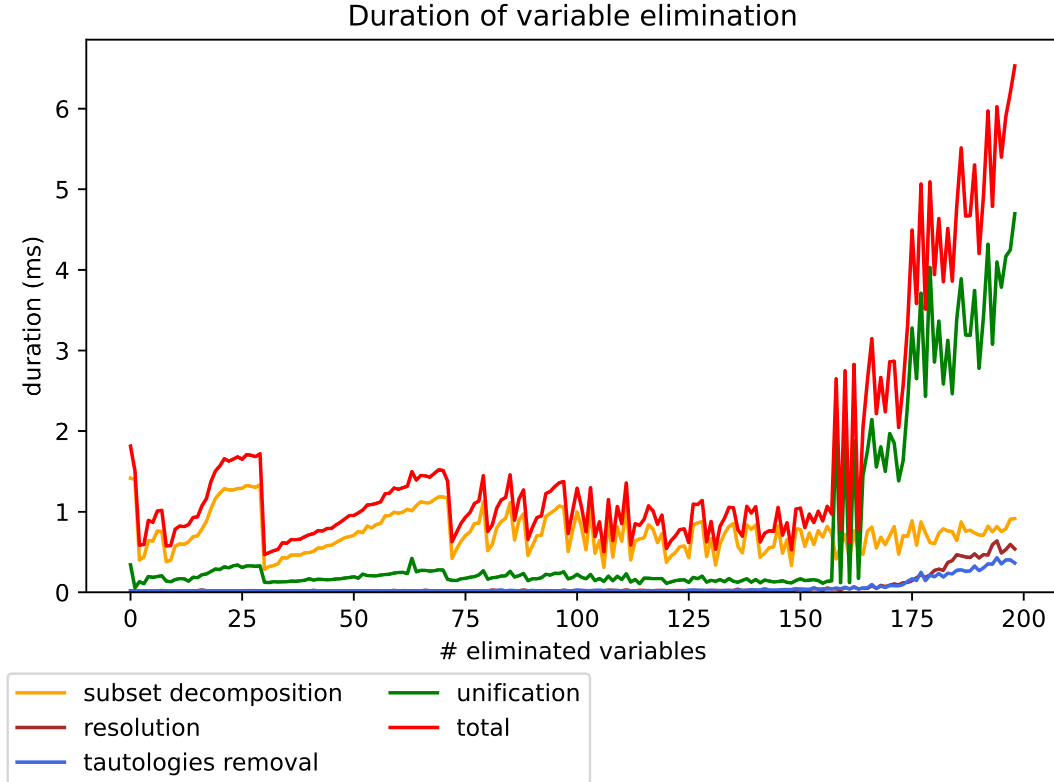

Figure 7.2 Variable elimination.

significantly. This is consistent with Figure 7.3 showing the heuristic score as well as actual difference in formula size. Note that the *clause count difference* curve excludes complete minimisation, but includes partial minimisation (it would be impossible to filter out its effect); this causes the inaccuracy of the heuristic once partial minimisation starts taking part.

Regarding the *resolution* and *tautology removal* phases in Figure 7.2, they also slow down for the last few elimination cycles similarly to *unification*, but they are still contributing significantly to the total duration.

If we compare figures 7.1 and 7.2, it is clear that partial minimisation alone cannot explain the sudden drops in formula size, because the two phenomenons do not start at the same time. To explain also the earlier formula size drops, we need to look more closely at complete minimisation. Figure 7.4 shows all invocations of Algorithm 3.13. The bars illustrate how many absorbed clauses were detected and removed (with y-axis on the left), the line plots break down how much runtime was spent in different phases of the algorithm (y-axis on the right):

1. serialising a ZBDD into a vector of clauses,
2. searching for absorptions with watched literals,
3. building a ZBDD from the vector of clauses.

Note that the x-axis does not represent time nor the number of eliminated variables by DP elimination, it simply tracks the number of invocations of complete minimisation. This is important to keep in mind, as minimisation is conducted with increasing frequency as more variables are eliminated.

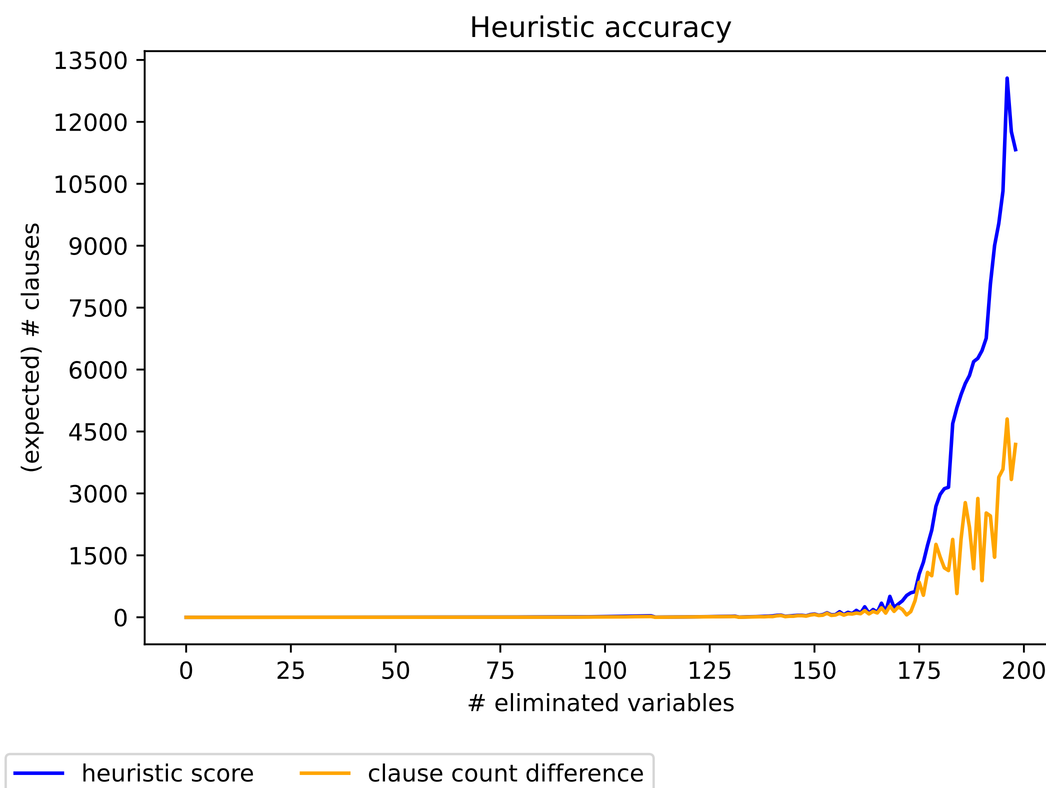

Figure 7.3 Heuristic prediction and actual clause difference.

We can see that there is a clear trend of more absorbed clauses being removed later on in DP elimination. From Figure 7.1 we see that the size of the formula does not change as drastically, which indicates that eliminating variables in the last phase of DP elimination creates a lot of absorbed clauses. Another observation is that the duration of the *search* part of complete minimisation strongly correlates with the number of removed absorptions. We can also see how it happens that absorption removal takes up most of DP elimination's runtime as shown by Table 7.1 — a single minimisation can run a few seconds. Finally, it is clear that using watched literals is a good choice, because both ZBDD serialisation and building are insignificant compared to the actual absorption search.

The final note we want to make with regards to the DP elimination breakdown concerns unit propagation. In Table 7.1 we see that in this particular formula, 44 unit literals were removed in total. The interesting part, which is visible from the collected metrics, is that all of those 44 literals were removed during the first run of unit propagation. In fact, we have not seen a single experiment where some unit clause would appear in the middle of DP elimination. To our understanding, it should not be impossible, but empirically it is very rare to say the least. Fortunately, as explained in Section 3.4, checking for a unit literal in a ZBDD is very fast and even if unsuccessful, it does not slow down the DP elimination algorithm.

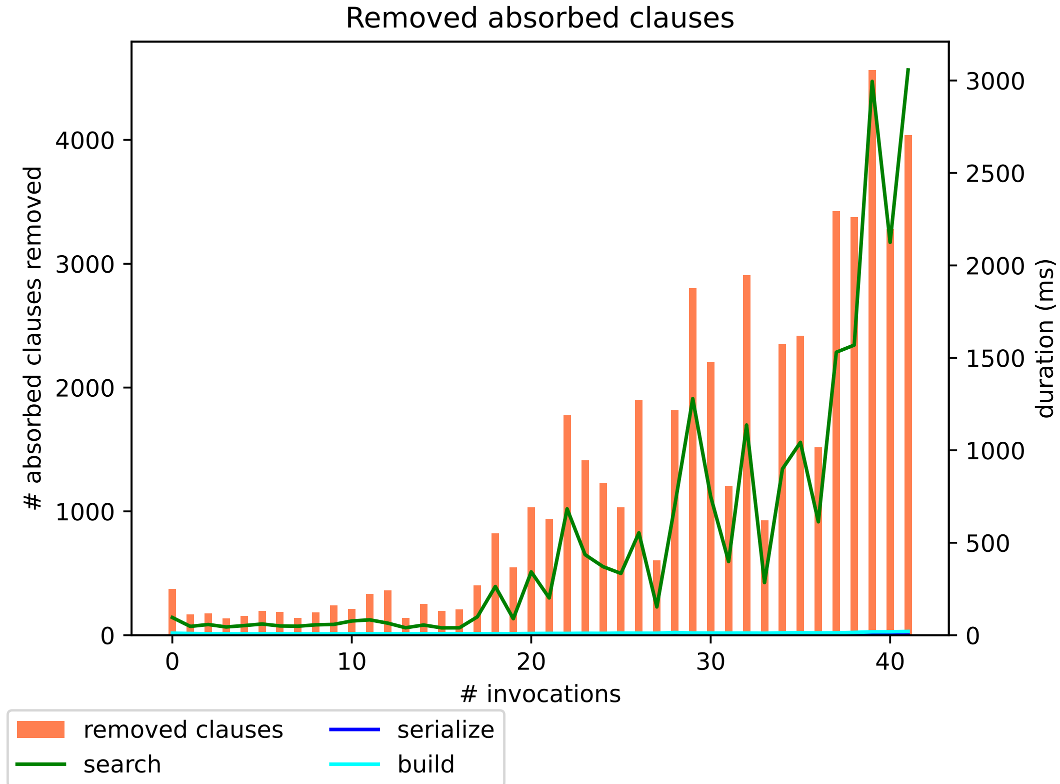

Figure 7.4 Complete minimisation.

7.4 Experiment 1: Absorption Removal

Our first experiment looks at the effect of removing absorbed clauses. We use three different configurations:

1. all minimisation enabled,

```
complete-minimization-condition = "relative_size"
complete-minimization-relative-size = 1.5

partial-minimization-condition = "relative_size"
partial-minimization-relative-size = 0.1

incremental-absorption-removal-condition = "relative_size"
incremental-absorption-removal-relative-size = 1.0
```

2. only complete minimisation (and subsumption removal) — disable incremental absorption removal,

```
complete-minimization-condition = "relative_size"
complete-minimization-relative-size = 1.1

partial-minimization-condition = "relative_size"
partial-minimization-relative-size = 0.1

incremental-absorption-removal-condition = "never"
```

3. no absorption removal — disable also complete minimisation.

```
complete-minimization-condition = "never"

partial-minimization-condition = "relative_size"
partial-minimization-relative-size = 0.1

incremental-absorption-removal-condition = "never"
```

The results of this experiment are captured in three plots, each plot showing a different metric. For every input formula there are three bars in each plot, one per configuration. Data used for creating the plots is included in Attachment A.1 as `experiments/summary_data.csv`. The file also contains data collected from other experiments and includes some additional information about each input formula. For more details about extracting the data from **dp** metrics and processing them see Section 5.5.

We should note that for 6 inputs, without minimisation the formulas grew so large that they crashed the program by completely filling Sylvan's unique table, which with our configuration fits $2^{28} \approx 2.68 * 10^8$ nodes. Unfortunately, **dp** is not able to recover from such situation and crashes without any output data. In order to avoid misleading plots, we tried to read the missing from log files and added them manually. The resulting file is also attached in Attachment A.1 as `experiments/summary_data_manual.csv`. This is the actual source for our summary plots. We were not able to derive from the logs how many variables were eliminated before crashing, so we filled in that no variables were eliminated.

Figure 7.5 shows the runtime duration. Notice the cut-off at ≈ 3600 seconds, which marks the 1-hour timeout.

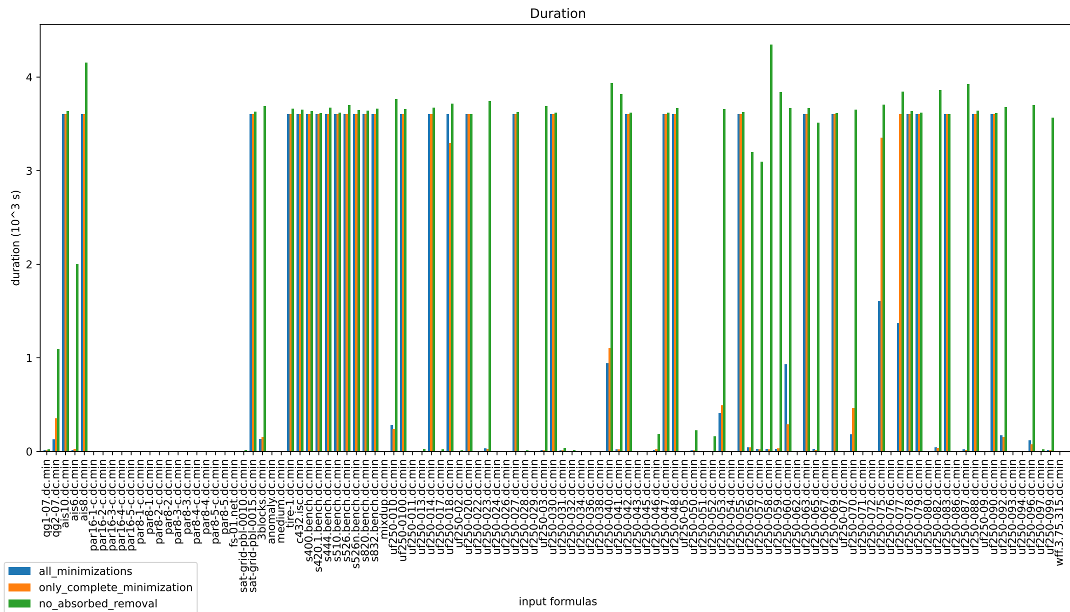

Figure 7.5 Experiment 1: duration of DP elimination.

Figure 7.6 is in some sense complementary to Figure 7.5. It shows how many auxiliary variables are left to be eliminated from the formula. If the program

finished before timing out, there are no auxiliary variables left. In other words, only if a bar reaches the 3600s bar in Figure 7.5 should the same bar have a non-zero value in Figure 7.6. The values in the plot are normalised — the y-axis is a percentage of the total number of auxiliary variables in the input formula. This figure also allows us to pinpoint which formulas crashed due to full Sylvan unique table, as discussed before — it shows that none of the variables were eliminated.

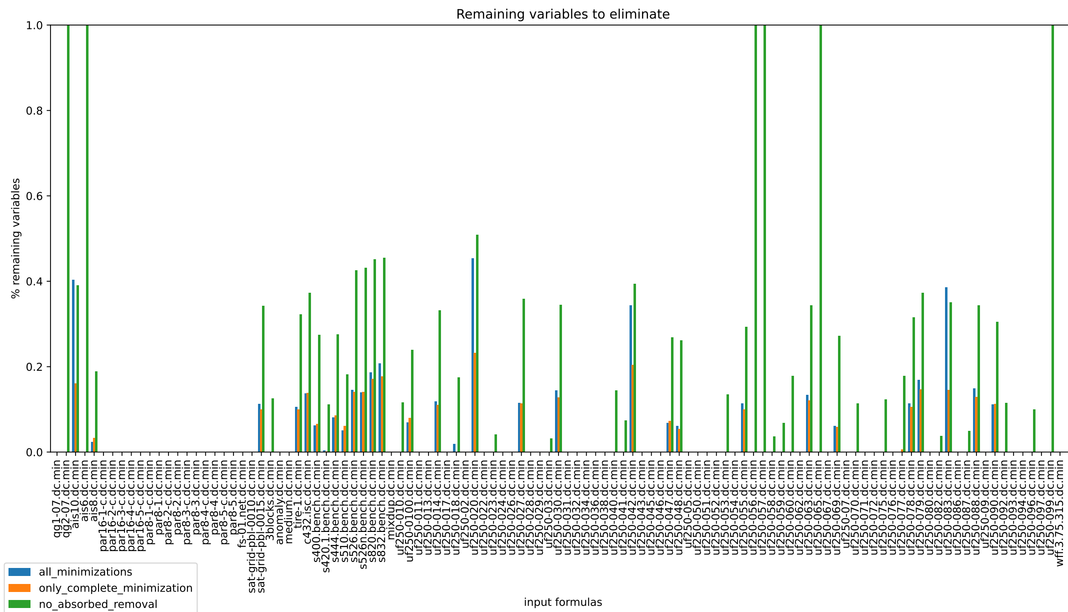

Figure 7.6 Experiment 1: auxiliary variables left to eliminate.

Figure 7.7 shows the maximum size of the processed formula during DP elimination relative to the input formula's size. Note that the y-axis is scaled logarithmically due to large disparity of values between different inputs.

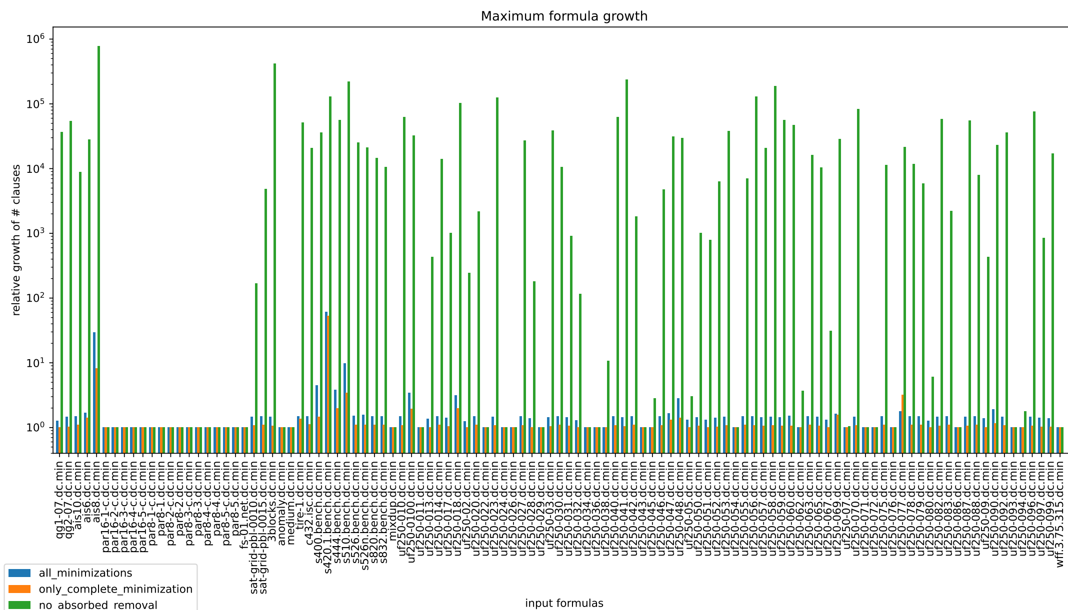

Figure 7.7 Experiment 1: maximum growth of formulas.

The effect of removing absorbed clauses during DP elimination is immediately

visible in Figure 7.7 — the formulas remain several orders of magnitudes smaller than without absorption removal for almost all inputs. While to some extent this also translates into DP elimination's efficiency, the effect is much less pronounced in figures 7.5 and 7.6. This illustrates two things: one, while absorption removal greatly reduces the formulas' size, it is expensive, as discussed in Section 7.3; two, ZBDDs are quite efficient even when dealing with very large formulas. Nevertheless, it is clear that the overall effect of absorption removal is very positive in terms of performance — none of the input formulas shows significantly better results with absorption removal turned off, and in most cases the results are considerably worse. Regarding the comparison of allowing or not allowing incremental absorption removal, the results are not conclusive. In most cases, the performance is comparable, sometimes going one way or the other.

7.5 Experiment 2: Variable Selection

The second experiment compares different variable selection methods described in Section 3.2. For all setups, we use the default configuration from Section 7.2, the only difference is in the `--heuristic` option (see Section 6.4.2):

1. minimal bloat,
2. ascending,
3. descending.

We show the same plots as we did in Experiment 1 (Section 7.4), but only this time we omit the plot showing formula growth — the differences rarely even reach one order of magnitude and do not seem to show any trend.

In figures 7.8 and 7.9 we can see that more often than not, the *minimal bloat* heuristic performs better than the ordering-based heuristics. The difference is especially visible with 'hard' formulas that DP elimination was not able to solve in the 1-hour limit. In a lot of cases the order-based heuristics manage to eliminate only a small fraction of variables where minimal bloat eliminates the majority of variables. However, there are some outliers, specifically within the randomly generated formulas (prefixed by `uf250*`) — sometimes the *ascending* heuristic beats minimal bloat by managing to finish in time where minimal bloat fails to do so. Although this is an interesting phenomenon, it is quite rare and given that it only happens on random formulas, we attribute it to pure luck — sometimes a (seemingly) random elimination sequence finds an optimum by chance.

Comparing the two ordering-based methods with each other, ascending order seems to be *almost always* better than descending (to varying degree). We attribute this to the fact also discussed in Section 7.3 that variables closer to the root of the ZBDD are cheaper to eliminate, giving ascending elimination order an advantage. There are two or three counter-examples which could perhaps be attributed to coincidental structure of the specific formulas, but we do not offer any conclusive insight. In any case, we can safely confirm that the minimal bloat heuristic works reasonably well and is the best choice in general.

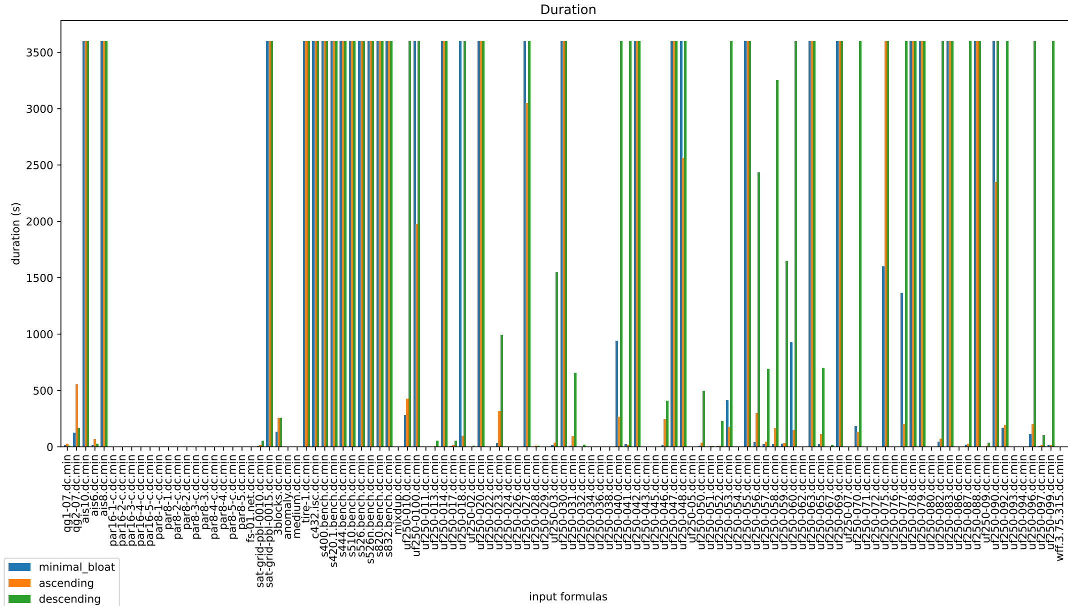

Figure 7.8 Experiment 2: duration of DP elimination.

7.6 Experiment 3: Parallelisation

We also conducted an experiment to see how Sylvan's parallelisation contributes to DP elimination's efficiency. In total, we run three setups:

1. 1 Lace thread (serial solution),
2. 3 Lace threads (default),
3. 7 Lace threads.

Perhaps surprisingly, the results show absolutely no difference between the three setups. This further illustrates that our implementation is limited by absorption removal, which dominates the runtime, as we already discussed in Section 7.3.

7.7 Experiment 4: Comparison With PCCompile

In Chapter 1 we cited some existing approaches of obtaining a propagation complete (PC) formula. In the final experiment, we wanted to compare them with our approach. Kučera [3] implemented a learning-based compilation algorithm, as well as an iteration-based one proposed by Bordeaux and Marques-Silva [2]. We ran their **PCCompile** program against our **dp** on formulas that are common in our datasets:

- ais6,
- sat-grid-pbl-0010,
- sat-grid-pbl-0015.

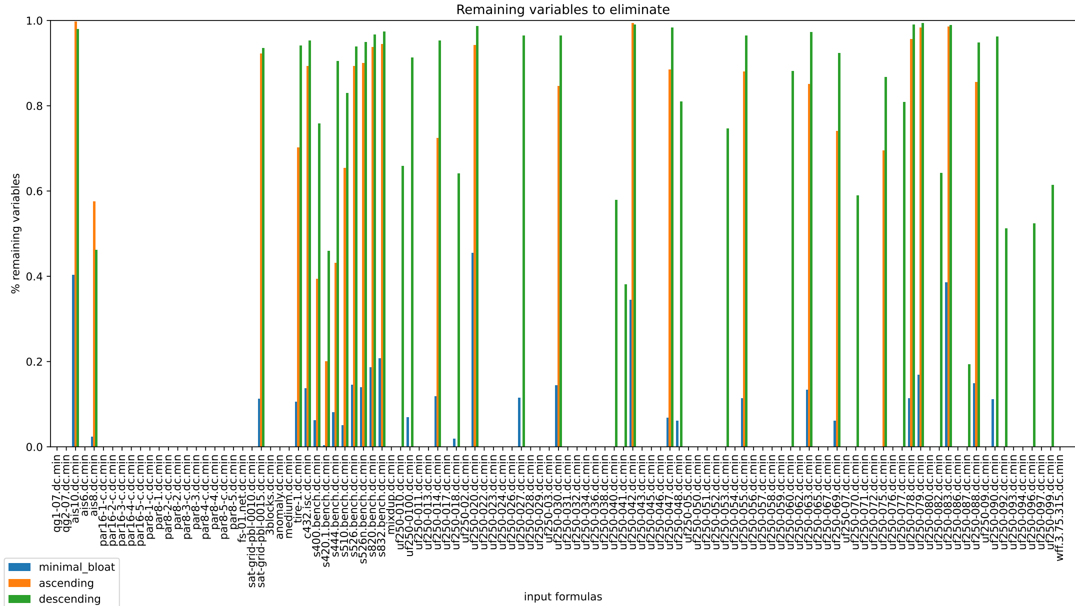

Figure 7.9 Experiment 2: auxiliary variables left to eliminate.

Based on the results of Kučera [3], we only used the *learning* algorithm of **PCCompile**, as it produced better results on all these formulas. As the underlying SAT solver we used *CaDiCaL* (by Biere [31]) instead of *Glucose* (by Audemard [32]), otherwise we followed the recommended settings for each formula. We have also verified that the resulting formulas from **dp** are equivalent to the original CNF formulas and that they are PC.

The results of these experiments are shown in Table 7.2. Note that this direct comparison is not completely fair. **PCCompile** takes a CNF formula as an input and outputs a PC formula, while **dp** needs preprocessing done on the original CNF formula, as described in Section 7.1. However, this preprocessing is insignificant — at most units of percent of the total execution time of **PCCompile**.

			PCCompile		dp	
formula	vars	size	time	size	time	size
ais6	61	581	3128.98 s	1962	15.31 s	2506
sat-grid-pbl-0010	110	191	3.28 s	297	5.95 s	522
sat-grid-pbl-0015	240	436	14418.30 s	15984	N/A	N/A

Table 7.2 Experiment 4: results comparing **dp** with **PCCompile**.

We can see that for `ais6`, the **dp** approach is much faster, although resulting in a larger formula. Here we should note that the size could be reduced, because **dp** does not explicitly minimise the formula when it finishes, while **PCCompile** does. This might give another unfair advantage regarding runtime to **dp**, but for this formula size it is not significant. For `sat-grid-pbl-0010`, **PCCompile** beats **dp** both in terms of performance and result size, but the difference in compilation time is much smaller than with `ais6`. Formula `sat-grid-pbl-0015` has proven to be hard for both compilation approaches, but significantly harder for **dp** — it has not managed to eliminate all variables even after 6 hours of computation.

Formulas `sat-grid-pbl-0010` and `sat-grid-pbl-0015` are similar to each

other in terms of structure, the only difference being that the latter is larger. However, there are some obvious differences compared to `ais6`. Firstly, `ais6` has much smaller variable-to-size ratio than the other two, which could result in a more favourable encoding for DP elimination. Secondly, it is missing a lot of clauses in order to become PC, which makes it expensive for the iterative approach to find them. We would need to conduct more experiments in order to find out which properties are most important to make one approach substantially better than the other. In any case, the results show that the CNF $\rightarrow$ DNNF $\rightarrow$ CNF approach is a viable alternative to iterative approaches for PC compilation, for some formulas even significantly better.

8 Conclusion

The goal of our thesis was to implement an efficient DP elimination algorithm able to eliminate a range of variables in a CNF formula. We had a specific application in mind — removing auxiliary variables from a domain-consistent CNF encoding of a DNNF formula in order to obtain a propagation-complete CNF formula. Our inspiration was the work by Chatalic and Simon [9], who performed the DP procedure on CNF formulas represented by ZBDDs. We wanted to improve their approach by applying additional techniques for formula minimisation during the algorithm — namely removing absorbed clauses. In addition, we had to find a suitable heuristic for elimination order of variables.

The result of our efforts is a program called **dp**. It uses ZBDDs implemented by the Sylvan library as described in chapters 2 and 4. We also implemented some custom algorithms over the data structure that were either missing in the library, or were too specific for our use-case. Chapter 3 explains how to utilise ZBDDs for DP elimination, and also describes various improvements to the basic algorithm, including absorption removal. The **dp** program's codebase is documented in Chapter 5, including unit tests and additional scripts for conducting experiments, and user documentation is available in Chapter 6. Finally, Chapter 7 describes and evaluates experiments conducted with our implementation.

Let us first note that our implementation is complete and correct. When conducting the experiments, we have not encountered any undefined behaviour nor unresponsiveness during the algorithm. In very few cases, the program hangs during clean-up of the Sylvan library, which we suspect to be connected to the race condition in the Lace framework documented in Section 4.1.2. However, this happens only after the algorithm has finished and all data have been exported. Regarding correctness, we have developed extensive unit tests to verify all sub-algorithms, and have also checked that if all auxiliary variables are eliminated from an input formula, the result is equivalent to the original CNF formula before it was encoded into DNNF.

In terms of performance, our experiments conclusively show that absorption removal is very beneficial to the efficiency of DP elimination, if not necessary. Without removing absorbed clauses during the algorithm, the formulas grow very large, in some instances to unfeasible proportions. However, the experiments also show that absorption removal is an overwhelming bottleneck for the program. This is not completely surprising — the algorithm we use for detecting absorptions has quadratic time complexity in the length of the formula, which is very expensive even if only performed occasionally. Unfortunately, this situation overshadows the otherwise very efficient implementation of DP elimination over ZBDDs.

Another conclusion arising from the experiments is that our heuristic for selecting eliminated variables, which tries to minimise growth of the formula, works reasonably well compared to naive heuristics based on variable ordering. On the other hand, these experiments hinted on an aspect that we did not explore — the ordering of d-variables in a ZBDD can impact efficiency of DP elimination. There are two reasons why we avoided this topic. One, the potential improvements seemed small given that most of the runtime is spent by removing absorptions rather than by ZBDD operations. Two, the Sylvan library does not offer any

support for determining a good variable ordering for ZBDDs. However, this is one of the possible directions for future work, perhaps also in connection with trying a different ZBDD implementation — for example CUDD (which we were not able to run) even supports dynamic reordering.

Finally, we would like to address the big picture involving our thesis — obtaining a PC CNF formula. Current approaches pioneered by Bordeaux and Marques-Silva [2], which are based on iteratively adding empowering clauses, work reasonably well with formulas that are already close to propagation completeness. However, in the opposite situation, PC compilation takes a very long time even on small formulas. Experiments comparing **dp** with **PCCompile** (by Kučera [30]) show that on some of these hard instances, **dp** can be very efficient. In conclusion, our approach — encoding the source formula into DNNF, then back to CNF, and finally eliminating auxiliary variables with **dp** — provides an interesting alternative which seems complementary to iterative approaches of PC compilation.

Bibliography

1. DAVIS, Martin; PUTNAM, Hilary. A Computing Procedure for Quantification Theory. *J. ACM*. 1960, vol. 7, no. 3, pp. 201–215. ISSN 0004-5411. Available from DOI: 10.1145/321033.321034.
2. BORDEAUX, Lucas; MARQUES-SILVA, Joao. Knowledge Compilation with Empowerment. In: BIELIKOVÁ, Mária; FRIEDRICH, Gerhard; GOTTLOB, Georg; KATZENBEISSER, Stefan; TURÁN, György (eds.). *SOFSEM 2012: Theory and Practice of Computer Science*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 612–624. ISBN 978-3-642-27660-6.
3. KUČERA, Petr. Learning a Propagation Complete Formula. In: *Integration of Constraint Programming, Artificial Intelligence, and Operations Research: 19th International Conference, CPAIOR 2022, Los Angeles, CA, USA, June 20-23, 2022, Proceedings*. Los Angeles, CA, USA: Springer-Verlag, 2022, pp. 214–231. ISBN 978-3-031-08010-4. Available from DOI: 10.1007/978-3-031-08011-1_15.
4. BABKA, Martin; BALYO, Tomáš; ČEPEK, Ondřej; GURSKÝ, Štefan; KUČERA, Petr; VLČEK, Václav. Complexity issues related to propagation completeness. *Artificial Intelligence*. 2013, vol. 203, pp. 19–34. ISSN 0004-3702. Available from DOI: https://doi.org/10.1016/j.artint.2013.07.006.
5. KUČERA, Petr; SAVICKÝ, Petr. Propagation complete encodings of smooth DNNF theories. *Constraints*. 2022, vol. 27, no. 3, pp. 327–359. ISSN 1572-9354. Available from DOI: 10.1007/s10601-022-09331-2.
6. DARWICHE, Adnan. New Advances in Compiling CNF into Decomposable Negation Normal Form. In: 2004, pp. 328–332.
7. LAGNIEZ, Jean-Marie; MARQUIS, Pierre. An improved decision-DNNF compiler. In: *Proceedings of the 26th International Joint Conference on Artificial Intelligence*. Melbourne, Australia: AAAI Press, 2017, pp. 667–673. IJCAI'17. ISBN 9780999241103.
8. ABÍO, Ignasi; GANGE, Graeme; MAYER-EICHBERGER, Valentin; STUCKEY, Peter J. On CNF Encodings of Decision Diagrams. In: QUIMPER, Claude-Guy (ed.). *Integration of AI and OR Techniques in Constraint Programming*. Cham: Springer International Publishing, 2016, pp. 1–17. ISBN 978-3-319-33954-2.
9. CHATALIC, Philippe; SIMON, Laurent. ZRes: The Old Davis–Putnam Procedure Meets ZBDD. In: MCALLESTER, David (ed.). *Automated Deduction - CADE-17*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 449–454. ISBN 978-3-540-45101-3.
10. MINATO, Shin-ichi. Zero-suppressed BDDs for set manipulation in combinatorial problems. In: *Proceedings of the 30th International Design Automation Conference*. Dallas, Texas, USA: Association for Computing Machinery, 1993, pp. 272–277. DAC '93. ISBN 0897915771. Available from DOI: 10.1145/157485.164890.

11. Atserias, Albert; Fichte, Johannes Klaus; Thurley, Marc. Clause-Learning Algorithms with Many Restarts and Bounded-Width Resolution. In: Kullmann, Oliver (ed.). *Theory and Applications of Satisfiability Testing - SAT 2009*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 114–127. isbn 978-3-642-02777-2.

12. Pipatsrisawat, Knot; Darwiche, Adnan. On the power of clause-learning SAT solvers as resolution engines. *Artif. Intell.* 2011, vol. 175, pp. 512–525. Available also from: `https://api.semanticscholar.org/CorpusID:909230`.

13. Drechsler, Rolf; Sieling, Detlef. Binary decision diagrams in theory and practice. *International Journal on Software Tools for Technology Transfer*. 2001, vol. 3, pp. 112–136. Available also from: `https://api.semanticscholar.org/CorpusID:1517733`.

14. Minato, Shin-ichi. Zero-suppressed BDDs and their applications. *International Journal on Software Tools for Technology Transfer*. 2001, vol. 3, no. 2, pp. 156–170. issn 1433-2779. Available from doi: `10.1007/s100090100038`.

15. Minato, S.; Ishiura, N.; Yajima, S. Shared binary decision diagram with attributed edges for efficient Boolean function manipulation. In: *27th ACM/IEEE Design Automation Conference*. 1990, pp. 52–57. issn 0738-100X. Available from doi: `10.1109/DAC.1990.114828`.

16. Chatalic, P.; Simon, L. Multi-resolution on compressed sets of clauses. In: *Proceedings 12th IEEE Internationals Conference on Tools with Artificial Intelligence. ICTAI 2000*. 2000, pp. 2–10. Available from doi: `10.1109/TAI.2000.889839`.

17. Liang, Jia Hui; Ganesh, Vijay; Zulkoski, Ed; Zaman, Atulan; Czarnecki, Krzysztof. *Understanding VSIDS Branching Heuristics in Conflict-Driven Clause-Learning SAT Solvers*. 2015. Available from arXiv: `1506.08905 [cs.LO]`.

18. Moskewicz, M. W.; Madigan, C. F.; Zhao, Y.; Zhang, L.; Malik, S. Chaff: engineering an efficient SAT solver. In: *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*. 2001, pp. 530–535. Available from doi: `10.1145/378239.379017`.

19. Somenzi, Fabio. *CUDD: CU Decision Diagram Package* [`https://web.archive.org/web/20150216083245/http://vlsi.colorado.edu/~fabio/CUDD`]. [N.d.]. [Software].

20. van Dijk, Tom. *Sylvan: multi-core decision diagrams*. Netherlands: University of Twente, 2016. isbn 978-90-365-4160-2. Available from doi: `10.3990/1.9789036541602`. PhD Thesis. University of Twente.

21. Dijk, Tom van; Pol, Jaco C. van de. Lace: Non-blocking Split Deque for Work-Stealing. In: Lopes, Luís; Žilinskas, Julius; Costan, Alexandru; Cascella, Roberto G.; Kecskemeti, Gabor; Jeannot, Emmanuel; Cannataro, Mario; Ricci, Laura; Benkner, Siegfried; Petit, Salvador; Scarano, Vittorio; Gracia, José; Hunold, Sascha; Scott, Stephen L.; Lankes, Stefan; Lengauer, Christian; Carretero, Jesús; Breitbart,

Jens; Alexander, Michael (eds.). *Euro-Par 2014: Parallel Processing Workshops*. Cham: Springer International Publishing, 2014, pp. 206–217. isbn 978-3-319-14313-2.

22. Dijk, Tom van. *Sylvan*. [N.d.]. Available also from: `https://github.com/trolando/sylvan`. [Software].

23. Zelený, Marek. *simple-logger*. [N.d.]. Available also from: `https://github.com/marek-zeleny/simple-logger`. [Software].

24. Lohmann, Niels. *json*. [N.d.]. Available also from: `https://github.com/nlohmann/json`. [Software].

25. Hořeňovský, Martin. *Catch2*. [N.d.]. Available also from: `https://github.com/catchorg/Catch2`. [Software].

26. Schreiner, Henry. *CLI11*. [N.d.]. Available also from: `https://github.com/CLIUtils/CLI11`. [Software].

27. Yoo, Andy B.; Jette, Morris A.; Grondona, Mark. SLURM: Simple Linux Utility for Resource Management. In: Feitelson, Dror; Rudolph, Larry; Schwiegelshohn, Uwe (eds.). *Job Scheduling Strategies for Parallel Processing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 44–60. isbn 978-3-540-39727-4.

28. Koriche, Frédéric; Lagniez, Jean-Marie; Lonca, Emmanuel; Marquis, Pierre; Mengel, Stefan. *Compile! Benchmarks*. [N.d.]. Available also from: `https://www.cril.univ-artois.fr/KC/benchmarks.html`. [Data].

29. Illner, Petr; Kučera, Petr. A Compiler for Weak Decomposable Negation Normal Form. *Proceedings of the AAAI Conference on Artificial Intelligence*. 2024, vol. 38, no. 9, pp. 10562–10570. Available from doi: `10.1609/aaai.v38i9.28926`.

30. Kučera, Petr. *PCCompile*. [N.d.]. Available also from: `https://ktiml.mff.cuni.cz/~kucerap/pccompile/`. [Software].

31. Biere, Armin. *CaDiCaL*. [N.d.]. Available also from: `https://github.com/arminbiere/cadical`. [Software].

32. Audemard, Gilles. *Glucose*. [N.d.]. Available also from: `https://github.com/audemard/glucose`. [Software].

A Attachments

A.1 Software and Data Attachments

This attachment is included as an archive in electronic format. The structure of the archive is shown below.

root/
- lib/ .... source code of the **dp_lib** library
- app/ .... source code of the **dp** application
- tests/ .... source code of unit tests
- external/ .... external libraries
- experiments/
  - experiments.py .... main script for running experiments
  - *.py .... other Python scripts and modules
  - inputs/ .... directory with input files
  - input_formulas.txt .... list of input formulas for experiments
  - setups/ ..... directory with configuration files for experimental setups
  - default_config.toml .... master configuration file
  - environment.yml .... definition of Python environment for running experiments
  - summary_data.csv .... summary data from experiments
  - summary_data_manual.csv .... includes manually added missing data
- CMakeLists.txt .... CMake build system file
- README.md .... basic compilation and usage instructions
- thesis.pdf .... this text