Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Technische Universität Dresden
Fakultät Informatik

Towards Solving a System of Pseudo Boolean Constraints with Binary Decision Diagrams

Valentin Christian Johannes Kaspar Mayer-Eichberger

Master Thesis in
Computational Logic

Dissertação apresentada na Faculdade de Ciências e
Tecnologia da Universidade Nova de Lisboa para obtenção do
grau de Mestre em Lógica Computacional

Supervision:
Prof. Pedro Barahona

Lisboa
September 1, 2008

Acknowledgements

I owe thanks to Prof. Pedro Barahona for his patient supervision. My colleague Jean Christoph Jung contributed with productive discussions and corrections, and I likewise wish him success for his thesis. Last but not least, I want to thank the library of Faculade de Letras in Cidade Universitaria, where I spent countless hours to work on this thesis. It remains the only place where I escaped the disquietude of Lisbon.

Abstract

In this work we address the applicability of binary decision diagrams (BDDs) for solving combinatorial (NP complete) problems. Typically, these problems are represented in conjunctive normal form (CNF), but we analyze problems specified as a conjunction of pseudo Boolean constraints (PBC), which is a more compact representation.

We present a recently published algorithm that transforms a PBC into a BDD, and provide a different correctness proof. After this initial transformation all PBCs are encoded as BDDs. Then we consider two approaches to compute a solution to the complete problem.

The first adopts a monolithic conjunction of BDDs, that conjoins all constraints via BDD algorithms. In special cases this leads to a compact representation of the complete solution space, but often it is not applicable due to space limitations. The second is a search based approach. We treat the initial conjunction of BDDs as a pre-processing step, until they reach a certain size. These fewer large BDDs are considered as global constraints.

The main work of this thesis lies in the analysis of how propagation is possible with BDDs. We present different techniques and conclude by proposing a propagation directed BDD data structure, UPBDDs. Although not being able to compete in general with current solvers, we show with a selection of benchmark problems the potential of the monolithic and search based method.

Contents

1 Introduction

Reduced Ordered Binary Decision Diagrams (BDDs) are a common data structure to represent Boolean functions and are widely used for model checking and circuit synthesis. The general success of BDDs raised interest to apply them in different areas. Within this thesis we analyze how BDDs perform as a key mechanism to solve a system of Pseudo Boolean Constraints (PBCs).

1.1 Motivation

The motivation for this thesis came from our previous project where BDDs have shown to be a suitable structure to represent the Boolean function specified by a PBC. In that project an algorithm for the translation from BDDs to PBCs was implemented and applied on problems given as a conjunction of PBCs from the satisfiability section of the *Pseudo Boolean Evaluation*[1]. The instances were solved by further conjoining the BDDs representations, ending up with a BDD representing all solutions. As a result we were able to solve a small special class of problems, but in general this simple approach was not competitive. However, the drawback of our approach was not in the translation to BDDs, but lied in the sequential conjunction. The intermediate BDDs in the conjunction phase tended to explode in size, this observation was not contrary to our expectation, since the challenge naturally exists in the conjunction of the constraints.

To our knowledge, the other PBC solvers came from the SAT, CSP or MIP community, and none of them used BDDs directly as the core representation of constraints. Thus, together with the results from our earlier research, we were motivated to explore further this novel way for a logical representation of PBCs and, in particular, to extend this approach by a more enhanced *conjunction*, incorporating techniques from other paradigms to overcome the detected problems.

The general plan for the development of the BDD based Pseudo Boolean Constraint Solver was thought to be:

- preprocessing:
  - transform PBCs constraints into BDDs
  - cluster translated BDDs into several large *global constraints*
- search:
  - enumerate variable and update BDDs followed by
  - propagation and local consistencies
  - further clustering

[1] http://www.cril.univ-artois.fr/PB07/

– conflict analysis from failed search

This thesis shows the results obtained and the problems still to be overcome, when pursuing this general research plan.

1.2 Concepts

This thesis concerns solving problems *automatically* with the help of computers, in the sense that there is some common structure to these problems for which a method can be designed to provide a solution in a general way. Many typical problems in computer science belong to the class of NP-complete problems, namely knapsack problems, subset sum, vertex covering, graph coloring. These problems occur in various forms in the real world and pose a significant challenge for finding reasonable and scalable solvers.

In general there is no algorithm that can solve these problems in polynomial time (unless $P = NP$). The typical example of a representative problem for this class is the satisfiability problem of propositional logic, i.e. to find an assignment to variables in a Boolean formula such that it evaluates to true.

For computing a reasonably good solution, we could be interested in an approximation algorithm, or an algorithm that produces a solution in some special cases, but cannot guaranty soundness or completeness. However, instead of finding a special tuned algorithm for each problem class, there is considerable interest in constructing an algorithm to generally solve these NP problems as good as possible. Such a solver is useful, as the given problem might not belong to one defined class but still has a structure, that can be utilized in finding a solution efficiently, avoiding unnecessary computation.

1.2.1 SAT and CSP

The constraint satisfaction problem over finite domains and the satisfiability problem, respectively CSP and SAT, are two paradigms in problem solving. Given a problem description, both search for an assignment to their variables such that constraints are satisfied wrt. CSP and clauses wrt. SAT.

Definition 1.1. *Let f be a Boolean function in Conjunctive Normal Form (CNF). Then the* SAT *problem is to determine whether there exists an assignment to the variables of f such that f evaluates to true or to prove that no such assignment exists.*

SAT problems are normally not coded to CNF by hand, but rather generated or translated from a higher representation. This standard for SAT problems has advantages from a practical point of view (for example, solvers are easily comparable).

One of the most studied algorithm for SAT is the DPLL algorithm, [11]. The basic backtracking algorithm is enriched by further techniques, as no good learning, conflict analysis and variable and value ordering heuristics. A reason for its success are highly optimized data structures for storing clauses and performing the various techniques.

CSP allows for a richer modeling by e.g. variables with finite domains and special global constraints.

Definition 1.2. *A constraint satisfaction problem instance* $(\mathcal{V}, \mathcal{D}, \mathcal{C})$ *consists of a set of variables* $\mathcal{V}$*, their domains* $\mathcal{D}$ *and a set of constraints* $\mathcal{C}$*. A solution is an assignment to all the variables such that all constraints are satisfied.*

In the context of CSP problems, complete solvers interleave backtracking with constraint propagations as prune values from the variables domains. In particular, maintaining generalized arc consistency (GAC) over an n-ary constraint $C \in \mathcal{C}$ requires all domains to be pruned such that remaining values to any variable of C can be extended to a satisfying assignment of C. In the special case of binary domains, this corresponds to finding all variables that are implied by the constraints.

1.2.2 PBC

Definition 1.3. *A* linear pseudo Boolean *constraint (PBC) is an expression of the form:* $a_k \cdot L_k + a_{k+1} \cdot L_{k+1} + \ldots a_n \cdot L_n \geq t$*, where* $k, n \in \mathbb{N}$ *and* $t \in \mathbb{Z}$*,* $\forall i \in \mathbb{N}$ *with* $k \leq i \leq n$*:* $a_i \in \mathbb{Z}$ *and* L_i *is a literal of the form* x_i *or* $\bar{x}_i = x_i - 1$ *and* $x_i \in \{0, 1\}$*.* t *is called the threshold. Values 0 and 1 are taken by* x_i *if the Boolean variable is false or true respectively. A PBC in normal form contains only positive coefficients* a_i*.*

Every PBC can easily be transformed into normal form by changing the parity of those variables that have negative coefficients.

Example 1.4. *Given the PBC* $4x_3 - 4x_4 + 6x_5 \geq 5$*, we can rewrite it with positive coefficients, since:*

$$\begin{aligned} 4x_3 - 4x_4 + 6x_5 &\geq 5 \\ 4x_3 - 4(1 - \bar{x}_4) + 6x_5 &\geq 5 \\ 4x_3 - 4 + 4\bar{x}_4 + 6x_5 &\geq 5 \\ 4x_3 + 4\bar{x}_4 + 6x_5 &\geq 9 \end{aligned}$$

Problems formulated as a conjunction of PBC are more compact than the usual CNF representation and this reduction might lead to an advantage for the solving algorithm compared to the same problems formulated in CNF. Recently there is an evolving interest from the SAT community in PBCs encodings of satisfiability problems [25].

PBCs have connections to the CSP, SAT and 0-1ILP (Integer Optimization) formalisms. They extend CNF representation allowing for cardinality expression and even more complex formulas. A PBC can also be seen as a constraint of higher arity, in the terminology of CSP, though with domains restricted to 2 values. In fact, a system of PBCs is a system of linear inequalities of 0-1 variables, a paradigm extensively studied in Operations Research.

1.3 Related Work

In [13], BDDs are used as a mediator for the translation from a PBC to its CNF encoding. The union of all CNF encodings for the given PBCs, is given to their SAT solver. The

translation presented in this thesis is an improvement to their first part of the translation.

More close in terms of the translation from PBCs to BDDs, is the approach by [5] to solve 0-1 optimization problems. They introduce an algorithm, that is equivalent to the one presented here. They apply parallel conjunction on the translated BDDs. If this is successful the produced BDD contains all solutions and can be used to find an optimum in an optimization problem. Except from simple caching of calls, there is no use of proper search techniques from either SAT or CSP.

In [10] an algorithm for checking the satisfiability of a conjunction of BDDs is presented. The BDDs are created by clustering clauses from problem specifications in CNF. Further they perform propagation of implied variables and clause learning. The difference to our approach is that they work entirely on CNF encodings and their problems are non accessible benchmark problems. They neither go into detail how propagation works.

There are approaches to tightly integrate reasoning from BDDs into SAT. In [17], information about variables are channeled between the paradigms, as the BDD solver performs a powerful but different inference than the SAT solver. In [15] a SAT solver is ran until the problem size reaches a threshold for then be passed to the exhaustive BDD solver.

A successful application of BDDs within CSP is the use as a domain propagator in Set Constraint problems [16]. They show how the characteristic function of a set can be represented with BDDs and thus be used to represent set bounds. With the operations on BDDs, powerful propagators can be *computed* and included in a CSP solver. As in our approach they are confronted in the domain propagation process to efficiently compute the implications of a BDD. They introduce split domains, which seem to be a first step towards our deeper integration of propagation into BDDs.

Most of these approaches need a efficient way to compute the implications of a BDD. In CSP terms speaking, this corresponds to achieving generalized arc consistent on these global constraint represented by a BDD. In [9] an optimized top down algorithm is presented. Our approach to propagation is closely related but shows an alternative bottom up computation and a step towards integration into the data structure.

In the area of groundness analysis of logic programs, [3] analyzes how to compute ground variables in a BDD representation and introduces a tight integration of this process into the data structure, as we do. However, this is in context of finding exclusively positive occurrences of variables and our work goes a step further. Moreover, in their work there is no application to propagation in search, which is our central application.

1.3.1 Emphasis in this Thesis

The main topics in this thesis are

- transformation from PBCs to BDDS.
- several approaches for propagation techniques in BDDs. In particular a top down and bottom up algorithm.
- definition of a BDD related data structure that tightly integrates implication detection.

- implementation and application on satisfiability problems from the pseudo Boolean evaluation.

The thesis is structured as follows. In Section 2 we explain the basic concepts regarding BDDs in a way they are relevant to this thesis. Section 3 concerns the translation from pseudo Boolean constraints to BDDs and we present a recently published algorithm. Further we show a novel correctness proof and discuss the complexity of the translation. In Section 4, we analyze propagation of BDDs, and several techniques to achieve it. A prototypical implementation is then applied on a variety of benchmark problems in Section 5. Section 6 concludes with a discussion on the advantages, open issues and limits of the approach.

2 Binary Decision Diagrams

In this section we introduce BDDs in a general way and provide the algorithms relevant to this thesis. For a more comprehensive introduction we refer to the book [29], that exhaustively explores Decision Diagrams and their related data structures, as well as complexity results for different classes of Boolean functions and their algorithms.

2.1 Background

Graphical representation of abstract concepts help to understand and visualize their properties. For Boolean functions a natural choice for representation are branching programs or decision diagrams. In Computer Science we are interested in handling of Boolean functions from a more practical perspective.

In 1986 [7] presented elegant algorithms to treat a certain subclass of decision diagrams and showed its complexity. The idea of using a global ordering on the variables and only work with reduced diagrams lead to a clear and simple representation. Reduced Ordered Binary Decision Diagrams (ROBDD) then became famous and widely used in many areas of computer science, such as symbolic model checking or circuit synthesis [20].

There is obviously no perfect general representation for Boolean functions, and further research went into modification of BDDs, either in the graph structures themselves or to differences in their semantics. Some applications did not require the full definition and special algorithms for substructures were introduced to make use of such special cases. Others used different semantics for BDD graphs. In [18], for example, it is argued that *Zero-Suppressed BDDs* are better suited to handle large sets and set operations. Regarding implementation, optimizations of the BDD structure were investigated to make the handling of several large BDDs more efficient. (For example *Shared BDDs* with Attributed Edges [19]).

2.1.1 Boolean Functions and Expressions

Boolean variables can take at most two distinct values, *true* or *false*. These two truth values can also be treated as the two numbers 1 and 0, respectively, in which case they might be used in arithmetical expressions. This is also called the "pseudo" Boolean nature of these variables. Variables are denoted by x, y, z, with indexes if needed, e.g. $x_1, x_2, \ldots, x_n$, with the index set $I = \{1, 2, \ldots n\}$. The set $\{0, 1\}$ is denoted by $\mathbb{B}$. The complement of variable x, written as $\bar{x}$, represents $\bar{x} = 1 - x$.

An n-ary Boolean function maps from an n-tuple of Boolean variables to a Boolean, $f : \mathbb{B}^n \to \mathbb{B}$. One way to define Boolean functions is by means of Boolean expressions, built from connectives $\neg, \wedge, \vee, \rightarrow, \ldots$ and evaluated by their known semantics, e.g. $f = (x_1 \vee x_2) \rightarrow (x_1 \wedge x_3)$. Boolean expressions together with their semantics are the object of *Propositional Logic*.

A (partial) *truth assignment* is a set of assignments of variables to values, e.g. $\{x_1 = 1, x_2 = 0 \ldots\}$ assigning 1 to x_1 and 0 to x_2 and so forth. For compactness we allow to write such a set by abbreviating $x = 1$ to x and $x = 0$ to $\bar{x}$, e.g. $\{x_1 = 1, x_2 = 0 \ldots\}$ abbreviated to $\{x_1, \bar{x}_2 \ldots\}$. An assignment is *complete* if all variables are assigned a value, such that a function under this assignment evaluates either true or false. In general, we can list list all possible assignments to their variables and their corresponding result in a table, called the *truth table*. However, this table is only useful as a presentation for functions with small arity, as the table has 2^n entries.

The expression $f|_{x_i}$, equivalent to the function f where all occurrences of x_i are evaluated to 1, is called the *positive co-factor*. Likewise $f|_{\bar{x}_i}$ the negative *co-factor*, where all $x_i = 0$. The expression $f|_S$ means that function f is partially evaluated under the truth assignment S.

A Boolean function is called a *tautology* if it maps to 1 under all assignments, and a *contradiction* (or *unsatisfiable*) if under no assignment the function evaluates to 1. If there is at least one successful assignment, the function is called *satisfiable*.

2.2 Definition

A Binary Decision Diagram is a directed, acyclic, binary graph with two distinct sinks, 0 and 1. Every node is labeled by one variable and has two descendants (or successors) referred to as *positive* or *high* branch and *negative* or *low* branch. On every path through the graph there is at most one occurrence of every variable. Such graph represents a Boolean function in the following way. An assignment to all variables corresponds to a path in the graph from the root to one of the sinks, following the positive (negative) descendent of a node if its variable is assigned 1(0). The sink value is the result of the defined Boolean function for this assignment. Variables that do not occur on a path from root to sink may be regarded as *don't care* variables.

Given a Boolean function f as a Boolean expression, we can easily construct a BDD that represents it. Note, that this construction is not unique and depends on several factors. It is done recursively on the co-factors of f.

- We choose one remaining variable x_i in f and compute the two co-factors $f|_{\{x_i\}}$ and $f|_{\{\bar{x}_i\}}$. This will be represented by a new node labeled with x_i and the successors are the result of further computation. Then we continue by instantiation of further variables in f.

- when $f|_{\{x_i,\ldots,x_j\}}$ is completely instantiated - there are no variables left - resulting in the constant 0 or 1, we have reached one of the sinks.

By this expansion of f we have constructed a semantically equivalent decision diagram. In other words, we have transformed the Boolean expression into the $if-then-else$ normal form. However, this diagram still corresponds to a decision tree and further criteria are introduced for reduction.

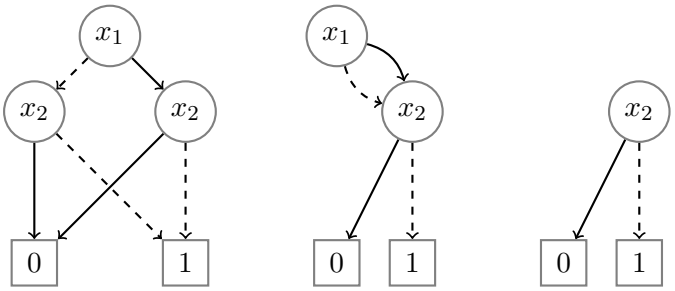

Figure 2.1: Three BDDs with the same semantics, but different reduction criteria. The left one is not reduced at all. Of the two nodes labeled with x_2 the two high branches point to the same node 0 and the two low branches to 1, thus both nodes encode the same Boolean function and can be merged (middle BDD). This BDD contains a redundant test of variable x_1 in the top node. The *reduced* version of both BDDs is on the right.

2.2.1 Reduced and Ordered BDDs

Since general decision diagrams are not adequate to handle for further processing, reduced ordered binary decision diagrams are defined through further restrictions additional to the definition:

- A BDD is *ordered* if there is a fixed total ordering $\prec$ on the variables and on every path the variables occur according to this order.
- A BDD is *reduced* if it full fills the following reduction conditions:
  1. There is no node in the diagram with its two successors directed to the same node.
  2. There are no two distinct nodes with the same variable and the same low and high successors.

It is easy to transform an ordered BDD into its reduced form. For all nodes that violate the first rule, we shortcut their incoming links with their successor. Further, to realize the second rule, we check if there are two nodes in the same level with the same descendants and merge them. This is done by removing one node and pointing all its incoming edges to the other. It can easily be seen that the semantics of its Boolean function is not changed by both of these reductions. Figure 2.1 shows three BDDs with the same semantics but reduced to different levels.

Reduced and ordered BDDs are usually abbreviated by ROBDD. However, in most of the literature, whenever BDD is written, ROBDD is actually meant. Subsequently, also our use of BDDs will refer to ROBDD. This does not cause much confusion, as reduction of a BDD is done during construction and does not have to be explicitly handled outside node management.

If not explicitly stated otherwise, variables are ordered top to bottom $x_1 \prec x_2 \prec x_3 \ldots$. So in that case the relation compares variables wrt. to their indices: $x_i \prec x_j$ iff $i < j$.

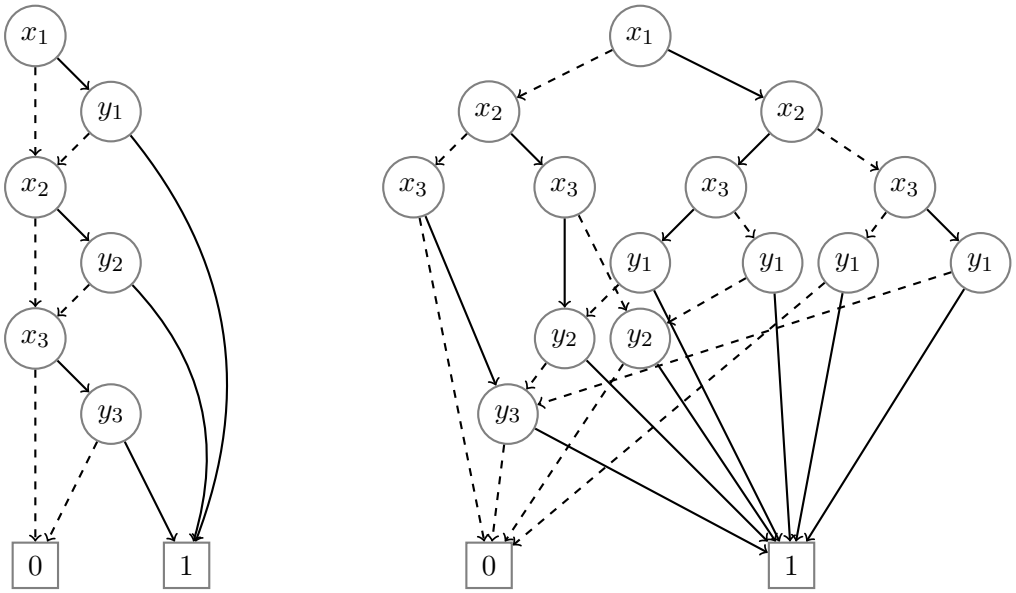

Figure 2.2: Two different orderings for the BDD of the formula $x_1y_1 \vee x_2y_2 \vee x_3y_3$. The BDD on the left with ordering $x_1, y_1, x_2, y_2, x_3, y_3$, and that on the right with ordering $x_1, x_2, x_3, y_1, y_2, y_3$.

2.2.2 Fixed Global Order

When handling several BDDs simultaneously there is normally a fixed global order for all BDDs. This results in the important concept of canonicity of Boolean functions wrt. an ordering. For any given two Boolean functions, their BDDs are equivalent if and only if they are semantically equivalent. This can be proven by induction on the arity of Boolean functions. As a consequence, the two functions 0 and 1 are unique BDDs and checking whether a BDD is unsatisfiable is just to check if it is equivalent to the 0 sink.

The *size* of a BDD is the number of its nodes. BDDs representing the same Boolean function with different variable orderings may vary significantly in size. When referring to classes of Boolean functions we are normally interested in lower and upper bounds of the sizes of their BDD representation wrt. free or fixed variable orderings. For certain classes the size is exponential wrt. to all variable orderings (e.g. multiplication of two n bit numbers) or polynomial wrt. all orderings (e.g. addition). However, for some classes, whether the representation is polynomial or exponential depends on the ordering. There is an interest in these theoretical bounds,namely to assess the applicability of this kind of representation for certain classes.

Another important line of BDD research goes into determining *good* orders. In general to find the best variable ordering for a Boolean function is NP-Hard, even to improve an ordering is NP-Complete [28]. Therefore, orderings normally are heuristically computed and dynamic reordering techniques are applied during computation of several BDDs.

2.3 Algorithms

2.3.1 Implicit Reduction

We have shown how to reduce a given BDD. However, explicit reduction is normally not required while constructing a BDD since both reduction laws are realized implicitly. Whenever a node is added, the implementation checks if its Boolean function is already in use and returns the existing identifier. To explain how this is done efficiently we introduce the data structures used for implementation of BDDs.

Implementations of BDDs use hash tables to store all nodes. Every node has a unique id *id*, a variable *var* and two references to its successors, *high* and *low*. A node entry thus consists of the following components.

$$(id, var, high, low)$$

For notational reasons, the two sinks 0 and 1 have id 0 and 1, respectively. A table, *unique* stores all nodes. A lookup for some component is denoted by the dot operation followed by the required information. E.g. $f.var$ returns the variable of node f. To ensure the reduction rules under construction and manipulation of BDDs, we introduce operation *insert* on table *unique*. This operation inserts a row in the table whenever necessary and returns the *id* of its node, see Algorithm 2.1.

Operation *insert* is called with a combination$(var, high, low)$ and returns its corresponding *id*. If we want to insert an entry with equal identifier *high* and *low*, *high* is returned (reduction rule 1). If the combination $(var, high, low)$ already exists, the identifier of this node is returned (reduction rule 2). With a good hashing function the two operations *lookup* and *insert* can be done in constant time.

Algorithm: insert(*var*,*high*,*low*)
Input: Level *var*, the two descendent BDDs
Output: Returns the id f of the new node
1 **if** $low = high$ **then return** *high*
2 **if** $f \leftarrow lookup(var, high, low)$ **then return** f
3 **else**
4 $\quad f \leftarrow put(var, high, low)$
5 $\quad$ **return** f

Algorithm 2.1: The insert algorithm in an UPBDD context.

Sharing of subfunctions is not only required in one single BDD. When several BDDs are considered they share all their common subgraphs (normally referred to as *Shared BDDs*, where one large BDD has several root nodes). All the BDDs obey the global ordering. In the implementation this is done by using the same table for all BDD nodes.

In Table 2.1 we list the basic algorithms negation, satisfiability check and equality, which come for free with reduced and ordered BDDs. However, to switch the two sinks would result in traversing the BDD and re-adding all nodes and have a linear complexity. To archive constant speed for the negation operation, complemented edges were introduced [19].

The underlying Boolean function of a BDD node is satisfiable if it is not the 0 sink.

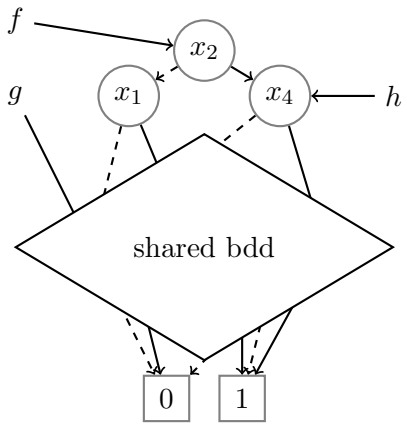

Figure 2.3: Schematic example of a shared BDD. f, g, h point to inner or top nodes.

name	idea	complexity
$sat(f)$	f is satisfiable if it is not the 0 sink	$O(1)$
$equal(f, g)$	test if the identifiers f and g are equal	$O(1)$
$neg(f)$	switch the two sinks	$O(1)$

Table 2.1: Basic algorithms for BDDs

Further to find a satisfiable assignment, we need one path from the root node to the 1 sink, and the variables that are not assigned by such a path, can be chosen freely, as they are *don't care*.

2.3.2 Binary Conjunction

Conjoining two BDDs under the same ordering, is done by a recursive algorithm that processes in each call two nodes. The algorithm is listed as Algorithm 2.2. To avoid recomputing common sub calls there is a table *computed* that stores the calls and their result. The operation *exist* checks if a combination of BDDs has been conjoined before and *get* returns the result.

The first four lines check the base cases, either if one of the BDDs is a sink, or if they are equal. Then we check the table if we are about to recompute a combination of BDDs. If so, we return its result. Otherwise we do a case analysis on the variables. We branch on the positive and negative co-factor of the BDD that is higher up in the global ordering to compute the new *low* and *high* successor. If both BDDs agree on the same variable, we split both of them up. Then we insert a new node to the *unique* table, memorize the (f, g, res) in *computed* and return the constructed node. The resulting BDD is reduced and ordered, as this is ensured by the *insert* method.

At most each node in one BDD is called with each node in the other. With the use of the *computed* table, the number of recursive calls is bound by $size(f) \cot size(g)$ in

Algorithm: and(f,g)
Input: two BDDs f, g
Output: $f \wedge g$

if $f = 0$ *or* $g = 0$ **then return** 0
else if $f = g$ **then return** f
else if $f = 1$ **then return** g
else if $g = 1$ **then return** f
else if *computed*.`exist`(f,g) **then return** *computed*.`get`(f,g)

$$high = \begin{cases} and(f.high, g)) & f.var \prec g.var \\ and(f, g.high)) & f.var \succ g.var \\ and(f.high, g.high)) & f.var = g.var \end{cases}$$

$$low = \begin{cases} and(f.low, g)) & f.var \prec g.var \\ and(f, g.low)) & f.var \succ g.var \\ and(f.low, g.low)) & f.var = g.var \end{cases}$$

$res \leftarrow$ *unique*.`insert`($min(f.var, g.var)$,*high*,*low*)
computed.`insert`(f,g,res)
return res

Algorithm 2.2: Recursive construction of the conjunction of two BDDs

the worst case and the size of the resulting BDD is also bound by the number of calls. However, the size is typically much less then this upper bound.

It should be noted that the conjunction algorithm has the prerequisite that both BDDs have the same variable ordering. If this is not the case, they have to be reordered, which can be very costly and BDDs may explode in size. The common order is also a prerequisite for using a shared BDD.

This conjunction is a special case of the *apply* algorithm to compute the result of any binary Boolean operator. *apply* has an additional argument, that takes the Boolean operator. This operator is used to define the base cases, that we have encoded in the algorithm for the conjunction explicitly. The recursive calls have the same structure. With the general apply algorithm we can build a BDD more elegant from a Boolean expression, by applying it for each connective.

The node table grows as more and more BDD nodes are added under construction and manipulation. However, many nodes become unused if their root BDD is consumed in some Boolean operation. To manage the reference and unreferenced nodes in the table and remove them, garbage collection techniques have to be applied.

2.3.3 Restriction

To instantiate a partial assignment S in a BDD is done by the Algorithm *restrict* in Algorithm 2.3. Here we traverse the BDD and shortcut those nodes that exist in S. The resulting BDD does not contain any nodes labeled by variables in S. Note, that this algorithm, in the worst case, reproduce the nodes above lowest level, that is to be instantiated. If the resulting BDD collapses to 0, the partial assignment S is not part in any model of the BDD.

Algorithm: restrict(f,S)
Input: ROBDD f and set of assingments S
Output: $f|_S$
1 **if** $f = 0$ *or* $f = 1$ **then return** f
2 **else if** *computed*.`exist(`f`)` **then return** *computed*.`get(`f`)`
3 **else if** $f.var \in S$ **then return** `restrict(`$f.high$`,`S`)`
4 **else if** $f.\bar{var} \in S$ **then return** `restrict(`$f.low$`,`S`)`
5 **else**
6 **return** *unique*.`insert(`$f.var$`,restrict(`$f.high$`,`S`),restrict(`$f.low$`,`S`))`

Algorithm 2.3: Algorithm to instantiate partial assignment in a BDD

3 A Transformation from PBCs to BDDs

In this section we address the transformation of a pseudo Boolean constraint into a BDD.

There are several approaches for the translation in literature. In [13] a BDD is used to perform a translation from a PBC into CNF. In that work, firstly a BDD is created from a PBC, and subsequently translated to CNF by the Tseitin transformation. The process of creating the temporary BDD is similar to our approach, however weaker in its predetection. More similar is the approach of [5], associating to every created BDD node a lower and upper bound, used to predetect identical nodes. This key idea is also used in the transformation presented in this chapter and we give a better formalization and a different proof.

This chapter is structured as follows: Firstly, we introduce the naive transformation of a PBC to a BDD, through a straight forward depth first construction of the BDD. In order to understand the optimizations of this basic algorithm, we then introduce an equivalence relation and the notion of a temporary sum set.

In the last part we discuss the size of the BDD representing a PBC. An important influence on the size is the chosen variable ordering and we give an example that has exponential different sizes wrt. different orderings. We also show that the variable ordering does not have a strong influence on the size if the coefficients are *small*.

3.1 The Naive Transformation

In the transformation we work only with PBCs in normal form, i.e. coefficients are positive. To transform a PBC into normal we need to introduce negated variables. The negation of a variable corresponds to switching all positive and negative descendants of nodes labeled with it. This can be done after the transformation.

Algorithm 3.1 introduces a simple transformation from a PBC to a BDD, which is constructed by recursively instantiating variables x_i from top to bottom.

The trace of the recursion corresponds to the unreduced BDD representing the PBC.

Example 3.1. *Consider the normalized PBC from example 1.4,* $4x_3 + 4\bar{x}_4 + 6x_5 \geq 9$. *In Figure 3.1 we show the unreduced BDD representing this PBC and the numbers on the arcs are the so called temporary sums.*

The temporary sums are denoted by p in the pseudo code and k indicates the current level (or variable x_k). Clearly if we reach the threshold with the temporary sum, we are at the sink 1. On the other hand if we have instantiated all variables and we are below the threshold, we return the 0 sink. The algorithm is a simple depth first search through this instantiation.

```
Algorithm: simpleBuildBDD(k,p)
Input: current level k and temporary sum p
Output: the BDD node
1 if p >= t then return 1
2 else if k = n + 1 then return 0
3 else
4     high ← simpleBuildBDD(k + 1,p + a_k)
5     low ← simpleBuildBDD(k + 1,p)
6     return insert(k, high, low)
```

Algorithm 3.1: Given a PBC $\sum_{i=1}^{n} a_i x_i \geq t$, the naive transformation algorithm.

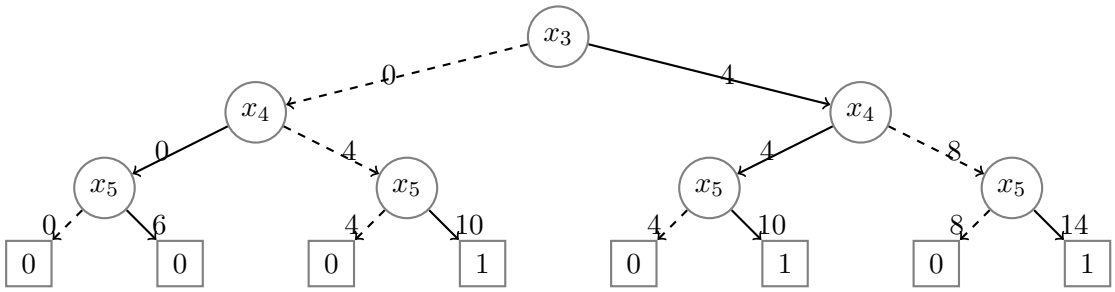

Figure 3.1: Naive computation of an unreduced BDD from the constraint $4x_3+4\bar{x}_4+6x_5 \geq 9$.

A usual enhancement of Algorithm 3.1 is the memorization of calls, to avoid recomputing subtrees called in the same level with the same temporary sum, as in [13]. By so doing, some (but not all) equivalent subtrees would be detected before a complete computation.

However, we can do better. In Figure 3.2 all equivalent subtrees of the BDD produced in Figure 1.4 are merged. The *temporary sum set* and the *value interval*, denoted by $\{,\}$ and $[,]$ respectively, will be used to ease the merging of nodes.

3.2 Value Equivalence Relation

The above notions are now formalized

Definition 3.2. *Let $\sum_{i=1}^{n} a_i x_i \geq t$ be PBC. For every k with $1 \leq k \leq n+1$ we define an equivalence relation $\sim_k$ on $\mathbb{Z}$ by*

$$p \sim_k q \text{ iff } \left(\forall x_k, \ldots x_n : (p + \sum_{i=k}^{n} a_i x_i \geq t) \iff (q + \sum_{i=k}^{n} a_i x_i \geq t) \right)$$

As usual from this equivalence relation $[p]_k = \{q \mid p \sim_k q\}$ denotes the equivalence class. So $[p]_k$ is a set of values for which the *remaining* PBC encodes the same logical function. This becomes clear in the next example:

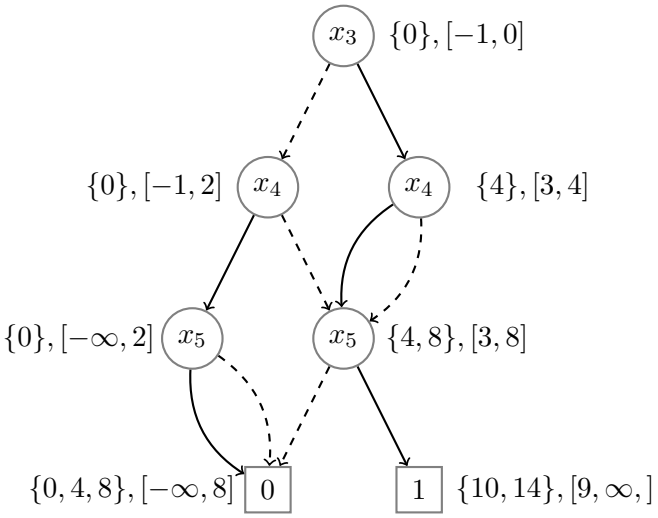

Figure 3.2: Merged subgraphs of BDD in Figure 3.1. Each node has attached the temporary sum set in curly brackets {}, and the value interval in brackets []

Example 3.3. *Let* $4x_1 + 3x_2 + 4x_3 + 4x_4 + 6x_5 \geq 9$ *be a PBC, then*

$$\begin{aligned}[6]_4 &= \{p' \mid \forall x_4, x_5 : \ \ 6 + 4x_4 + 6x_5 \geq 9 \text{ iff } p' + 4x_4 + 6x_5 \geq 9\} \\ &= \{5, 6, 7, 8\}\end{aligned}$$

This is the case because $4x_4 + 6x_5 \geq 9 - 8$, $4x_4 + 6x_5 \geq 9 - 7$, $4x_4 + 6x_5 \geq 9 - 6$ *and* $4x_4 + 6x_5 \geq 9 - 5$ *encode the same Boolean function, namely* $x_4 \vee x_5$.

Proposition 3.4. *Let PBC* $\sum_{i=1}^{n} a_i x_i \geq t$ *and* $1 \leq k \leq n + 1$. *Then*

1. *equivalence relation* $\sim_k$ *partitions* $\mathbb{Z}$.
2. *for all* $p \in \mathbb{Z}$, *the set* $[p]_k$ *is an interval.*

Proof.

1. Follows directly from $\sim_k$ being an equivalence class.
2. Let $q \in [p]_k$ and without loss of generality $p \leq s \leq q$ then $\forall x_k \ldots, x_n$

$$\sum_{i=k}^{n} a_i x_i \geq t - p \geq t - s \geq t - q$$

Every assignment to variables $x_k \ldots x_n$ that makes the left hand side of the PBC greater or equal than $t - p$ also makes it greater or equal than $t - s$. On the other hand assignment to variables $x_k \ldots x_n$ that makes the left hand smaller than $t - q$ also makes it smaller than $t - s$. Thus p, q, s determine the same logical function and $s \in [p]_k$. Consequently $[p]_k$ is an interval.

□

Example 3.5. *Let $4x_1 + 3x_2 + 4x_3 + 4x_4 + 6x_5 \geq 9$ be a PBC, then $k = 4$ partitions $\mathbb{Z}$ into 6 intervals. We choose p to be $-3, 1, 4, 6$ and 9, where each represents one partition of $\mathbb{Z}$.*

$$\begin{aligned}[-3]_4 &= [-\infty, -2] \\ [1]_4 &= [-1, 2] \\ [4]_4 &= [3, 4] \\ [6]_4 &= [5, 8] \\ [9]_4 &= [9, \infty]\end{aligned}$$

With $k = 4$ we are left with PBC $4x_4 + 6x_5 \geq 9 - p$, then we may chose representative values (e.g. 1 for the interval $[-1, 2]$).

- *For $p = -3$ the PBC $4x_4 + 6x_5 \geq 9 + 3 = 12$ is equivalent to the Boolean function 0.*
- *For $p = 1$ the PBC $4x_4 + 6x_5 \geq 9 - 1 = 8$ is equivalent to $x_4 \wedge x_5$*
- *For $p = 4$ the PBC $4x_4 + 6x_5 \geq 9 - 4 = 5$ is equivalent to x_5*
- *For $p = 6$ the PBC $4x_4 + 6x_5 \geq 9 - 6 = 3$ is equivalent to $x_4 \vee x_5$*
- *For $p = 9$ the PBC $4x_4 + 6x_5 \geq 9 - 9 = 0$ is equivalent to 1*

In fact, there is no need to guess the bounds of the partitions, given the following theorem.

Theorem 3.6. *Let PBC $\sum_{i=1}^{n} a_i x_i \geq t$ and $1 \leq k \leq n+1$ and $p \in \mathbb{Z}$, then the equivalence classes can be computed recursively:*

$$[p]_k = \begin{cases} [-\infty, t-1] & k = n+1, p < t \\ [t, \infty] & k = n+1, p \geq t \\ [p]_{k+1} \cap \{q - a_k \mid q \in [p + a_k]_{k+1}\} & \textit{else} \end{cases}$$

Proof. We prove by an inductive argument on k:

The cases for $k = n + 1$ are clear, as they follow from $p + \sum_{i=n+1}^{n} a_i x_i = p \geq t$.

Now we know that it holds for $k + 1$ and we want to prove it for k. First we use the definition, then we split the Boolean function into the two co factors and with adjusting

the sums we end up with the recursive formula.

$$
\begin{aligned}
[p]_k =& \{q \mid \forall x_k, x_{k+1}, \dots x_n : \ \ (p + \sum_{i=k}^{n} a_i x_i \geq t) \text{ iff } (q + \sum_{i=k}^{n} a_i x_i \geq t)\} \\
=& \{q \mid \forall x_k, x_{k+1}, \dots x_n : \ \ (p + a_k x_k + \sum_{i=k+1}^{n} a_i x_i \geq t) \text{ iff } (q + a_k x_k + \sum_{i=k+1}^{n} a_i x_i \geq t)\} \\
=& \{q \mid \forall x_{k+1}, \dots x_n : \left((p + 0 \cdot a_k + \sum_{i=k+1}^{n} a_i x_i \geq t) \text{ iff } (q + 0 \cdot a_k \sum_{i=k+1}^{n} a_i x_i \geq t) \right) \\
& \wedge \left((p + 1 \cdot a_k + \sum_{i=k+1}^{n} a_i x_i \geq t) \text{ iff } (q + 1 \cdot a_k + \sum_{i=k+1}^{n} a_i x_i \geq t) \right)\} \\
=& \{q \mid \forall x_{k+1}, \dots x_n : (p + \sum_{i=k+1}^{n} a_i x_i \geq t) \text{ iff } (q + \sum_{i=k+1}^{n} a_i x_i \geq t)\} \\
& \cap \{q \mid \forall x_{k+1}, \dots x_n : (p + a_k + \sum_{i=k+1}^{n} a_i x_i \geq t) \text{ iff } (q + a_k + \sum_{i=k+1}^{n} a_i x_i \geq t)\} \\
=& [p]_{k+1} \cap \{q - a_k \mid q \in [p + a_k]_{k+1}\}
\end{aligned}
$$

□

3.3 The Temporary Sum Set

Now we define the notion of a *temporary sum set*, as in [23]. Note that all coefficients are positive, $a_i \in \mathbb{N}$

Definition 3.7. *Let f be an arbitrary node in a BDD representing a PBC:*

1. *The* temporary sum *of a node f on a path p is the sum of all coefficients of those variables that occurred positive on the path from the root to node f*
2. *The temporary sum set of a node f, $TS(f)$, is the set of all temporary sums from the top node to node f.*

$$
\begin{aligned}
TS(f) = \{p \mid & \exists \text{ path with positive } x_{i_1}, x_{i_2}, \dots x_{i_s} \\
& \text{from the root to node } f \text{ with } p = \sum_{j=i_1}^{i_s} a_j\}
\end{aligned}
$$

Note that on a path with *don't care* variables we have two temporary sums, one with and one without its coefficient.

Example 3.8. *Let $4x_1 + 3x_2 + 4x_3 + 4x_4 + 6x_5 \geq 9$ be a PBC and Figure 3.3 be the BDD representing it. Let f be the right most node in level 3, then $S(f) = \{7\}$ since there is only one path from the root. $x_1 = 1, x_2 = 1$.*

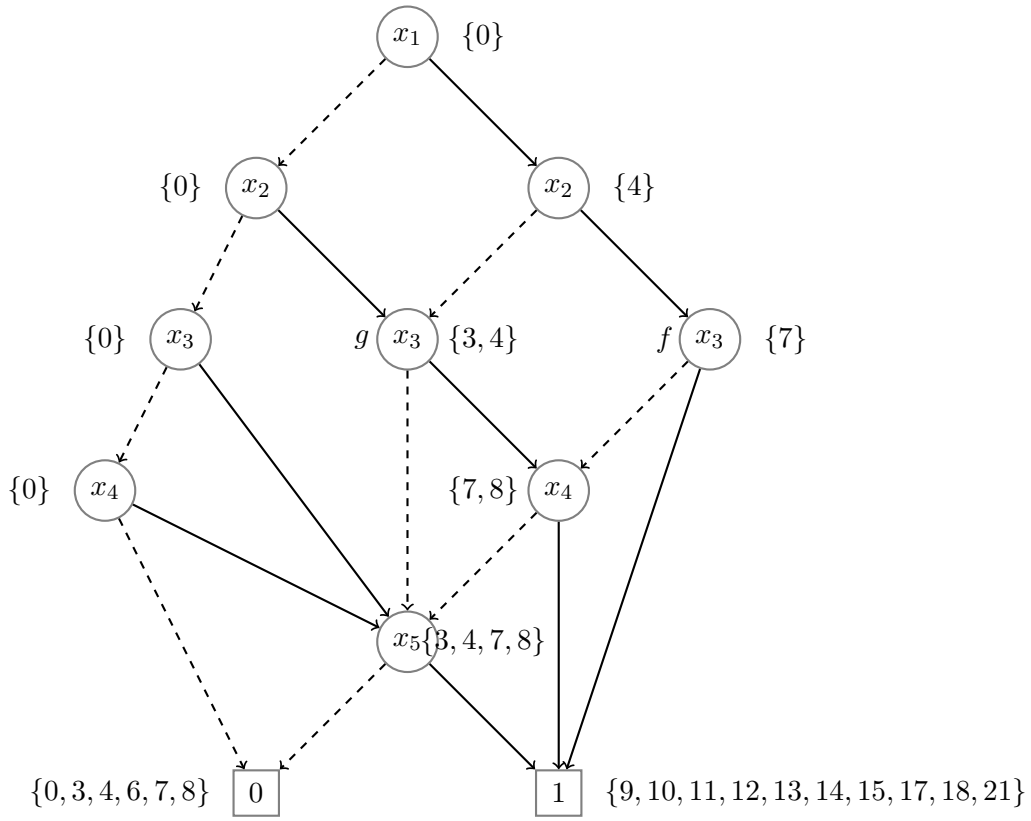

Figure 3.3: Temporary sum sets of the PBC $4x_1 + 3x_2 + 4x_3 + 4x_4 + 6x_5 \geq 9$

For the middle node in level 3, however it is $TS(g) = \{3, 4\}$*, because we can reach it by path* $x_1 = 1, x_2 = 0$ *and* $x_1 = 0, x_2 = 1$*.*

Not surprisingly, we relate the temporary sum sets and $\sim_k$:

Proposition 3.9. *Let* f *be a node in a BDD representing a PBC* $\sum_{i=1}^{n} a_i, x_i \geq t$*,* $f.var = k$ *and* $p \in TS(f)$*. Then*

1. *the temporary sum set of* f *is contained in its corresponding equivalence class:*

$$TS(f) \subseteq [p]_k$$

2. *let* $g \neq f$*,* $g.var = k$*,* $q \in TS(g)$*,*$p < q$ *then*

$$max([p]_k) < min([q]_k)$$

Proof.

1. Boolean function f represents PBC $p + \sum_{i=k}^{n} a_i, x_i \geq t$. Since all temporary sums that end in node f induce the same Boolean function, all of them belong to the equivalence class $[p]_k$.

2. Since g and f are different nodes, they encode two different Boolean functions. Thus we have two different intervals in the same level. Since the intervals of a level form a partition, they do not overlap.

□

From this Proposition follows (as in [23]):

Corollary. *In a BDD representing a PBC, the temporary sum sets of two distinct nodes* f *and* g *in the same level do not overlap:*

$$max(TS(f)) < min(TS(g))$$
$$or$$
$$max(TS(g)) < min(TS(f))$$

This result follows from Proposition 3.9. More interestingly, we use this proposition to obtain the least upper and greatest lower bound, $lb(), ub()$, for all temporary sum sets.

Definition 3.10. *Let* f *be a BDD node within a BDD representing a PBC* $\sum_{i=1}^{n} a_i, x_i \geq t$*,* $f.var = k$ *and* $p \in TS(f)$*. Then*

$$lb(f) = min([p]_k)$$
$$ub(f) = max([p]_k)$$

3.4 Algorithm

The above notions can now be used for an algorithm to obtain a BDD from PBC. The key is to detect equivalent nodes in advance, to merge them as soon as possible. In a simple approach, we recursively build the BDD by calculating the temporary sums until we reach the 1 sink, then we would merge identical subtrees, as it is done in the naive transformation Algorithm. However in creating a BDD from a threshold function we can reference common subtrees before multiple versions are created.

When recursively building the BDD and finishing the computation of a complete subtree, we compute and keep the lower and upper bounds of the node that is the root of this subtree. If, throughout the search, we reach the level of this node again, we can decide (before computing its subtree) whether this node has already been computed before.

This is done by checking if the temporary sum set of the node is within one of the lower and upper bounds of some node already computed in this level. The Algorithm to compute the BDD of a PBC uses this advance detection. For this, we have to save in addition to a node entry $(var, high, low)$, its lower and upper bound. So an entry in the table now contains $(var, high, low, lb(f), ub(f))$. Given this additional information, we need an efficient search that given a level and a temporary sum returns us the previous computed node. This can be done by an appropriate data structure, e.g. an AVL tree, in log(width of level) time.

Algorithm: buildBDD(k,p)
Input: current level k and temporary sum p
Output: the BDD node with bounds
1 **if** $p \geq t$ **then return** 1
2 **else if** $k = n + 1$ **then return** 0
3 **else if** *exists a node g in level k with* $lb(g) \leq p \leq ub(g)$ **then return** g
4 **else**
5 $\quad high \leftarrow$ buildBDD($k + 1$,$p + a_k$)
6 $\quad low \leftarrow$ buildBDD($k + 1$,p)
7 $\quad lb \leftarrow max(lb(low), lb(high) - a_k)$
8 $\quad ub \leftarrow min(ub(low), ub(high) - a_k)$
9 $\quad$ **return** insert($k, high, low, lb, ub$)

Algorithm 3.2: Given a PBC $\sum_{i=1}^{n} a_i x_i \geq t$, the optimized transformation algorithm

Algorithm 3.2 transforms a PBC into a BDD adopting the optimizations discussed. Initially the algorithm is called with a partial sum $p = 0$ and first variable $k = 1$. On each call to the recursive function we check for several base cases and then make two recursive calls with the next variable in the ordering. In the first call, the coefficient to the current variable is added to the partial sum - corresponding to the *one* branch in the BDD - and in the second call without adding it to partial sum - resulting in the *zero* branch. The BDD is thus built stepwise.

On each call we create (or reference in case of shared nodes) one BDD node with the

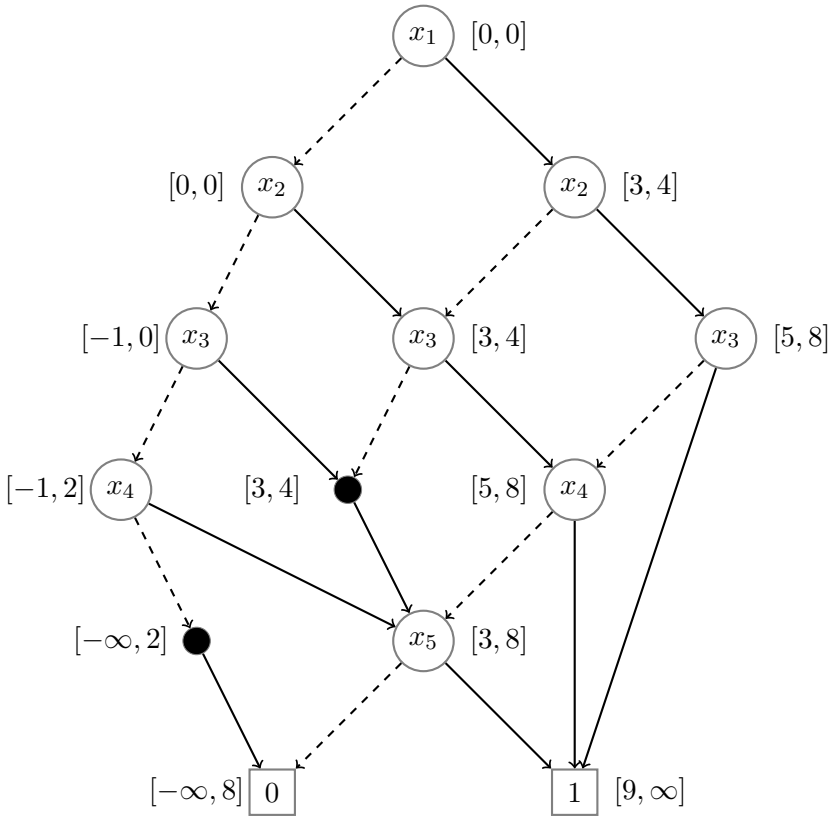

Figure 3.4: Translation of PBC $4x_1 + 3x_2 + 4x_3 + 4x_4 + 6x_5 \geq 9$ into a BDD

current variable. The base cases are as in the basic algorithm (line 1,2). If these do not apply, we check if the node has been computed before (line 3). This is done by searching in all computed nodes in that level, if the current partial sum is contained in one of the intervals. If so, we return this node, since after the previous reasoning, these nodes are identical.

If none of the previous cases applies, we recursively call *buildBDD* twice with the next variable in the ordering and updated partial sum (line 5,6). The new bounds of this node are computed (line 7,8) and inserted into the BDD table (line 9). The calculation of the bounds follow from taking the intersection of the value intervals from the two descendants.

Note that redundant nodes (nodes with both edges pointing to the same node) are kept as reference points with bounds throughout the transformation. We need these nodes for detection and computation of bounds. In the output BDD the redundant nodes are removed. For convenience, in Figure 3.4 we indicate these nodes as black dots. Long edges are allowed to the 1 sink, because if both descendants point to the 1 sink, we do not need to update the bounds in the intermediate levels, since $[t, \infty] \cap [t - a_i, \infty] = [t, \infty]$.

Another option would be to allow these nodes to be present explicitly and formally be working with so called *quasi* reduced ordered BDDs, as it is done in [5]. We chose to avoid this technical detail for convenience.

Example 3.11. *In Figure 3.4 we show the BDD built from the pseudo Boolean constraint*

$4x_1 + 3x_2 + 4x_3 + 4x_4 + 6x_5 \geq 9$. *On each node the lower and upper bound is indicated by* $[lb, ub]$.

For example, in the level of x_3 *there are three nodes with the intervals* $[-1, 0], [3, 4]$ *and* $[5, 8]$. *If we reach on one path level* x_3 *and have a partial sum that is included in one of these intervals, one link to the corresponding node concludes that run, and we backtrack.*

Let us take a closer look what happens around the central node of these three. There are two paths to the central x_3 *node, with bounds* $[3, 4]$. *The first path is* $x_1 = 1, x_2 = 0$, *then we have a partial sum of 4, since the coefficient is* 4 *of* x_1. *After the subsequent calls we compute the new lower bound by* $max(3, 4 - 4)$ *and new upper bounds* $min(8, 8 - 4)$, *for which a redundant node at level 4 (shown as a black dot) is used, resulting in the interval* $[3, 4]$. *On the second path* $x_1 = 0, x_2 = 1$ *with a partial sum of* 3, *we find that it is included in the interval* $[3, 4]$ *and we reference a shared node.*

Another example is the bounds calculation of the left most node labeled with x_4, *that involves bounds of another redundant node, not in the final BDD, and indicated by another black dot. This temporary node has bounds* $[-\infty, 2]$. *The interval of the left most node is thus calculated by* $[max(-\infty, 3 - 4), min(2, 8 - 4)] = [-1, 2]$.

Proposition 3.12. *The transformation algorithm is sound and complete.*

The transformation algorithm is an application of Theorem 3.6 enhancing the basic Algorithm 3.1. The predetection does not change the resulting reduced BDD but computes it more efficiently.

Proposition 3.13. *Let BDD* f *be the result of the algorithm that transforms a PBC* $\sum_{i=1}^{n} a_i x_i \geq t$ *into a BDD. The run-time complexity of the transformation algorithm is in* $O(log(width(f)) * size(f))$.

Proof. In each step of the algorithm we create one BDD node that stays unique, i.e. there will be no subtrees merged after the transformation. Thus we visit each node exactly once and consequently the run time of the algorithm is linear in the number of BDD nodes times the overhead done in each node computation. Since we have to check all computed nodes in the respective level and this can be done in *log* time, we get this upper bound. □

Note that if the size of the resulting BDD is polynomial or exponential in n then consequently the algorithm has a polynomial or exponential performance, respectively. In the next section we will discuss the size of BDDs representing PBCs.

3.5 Size and Variable Ordering of BDDs representing PBCs

The task of determining the best variable ordering for a PBC is not simple. Initially, it might seem that ordering the variables in a descending order of their coefficients is best, since then the variables with largest coefficient are tested first and this gives hope to reach the threshold as high in the BDD as possible. However, this rule does not work in general. We can, in fact, construct a counter example, as follows (taken from [23]).

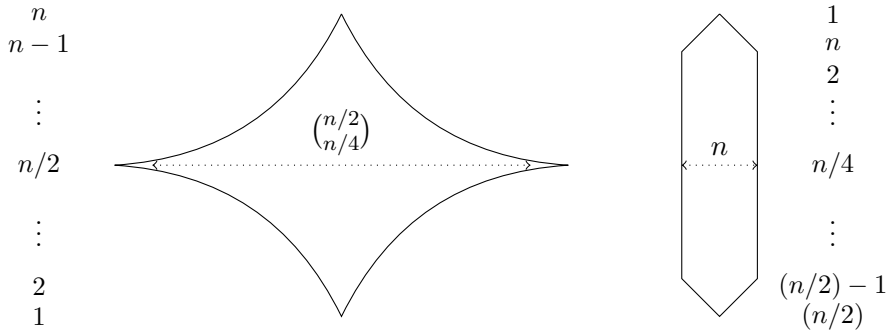

Figure 3.5: Schematic drawing of the BDDS with different orderings. On the left the decreasing ordering, that has an exponential width, and on the right one with linear width.

Proposition 3.14. *Let n be divisible by 4 and $\sum_{i=1}^{n} a_i x_i \geq t$ with a_i and t defined as follows:*

$$a_i = \begin{cases} 2^{i-1} & (1 \leq i \leq n/2) \\ 2^{n/2} - 2^{n-i} & (n/2+1 \leq i \leq n) \end{cases}$$

$$t = (n/4) * 2^{n/2} = \sum_{i=1}^{n} a_i/2$$

We consider two orderings

- *Ordering 1: $x_n, x_{n-1}, \ldots x_1$ (descending coefficients)*
- *Ordering 2: $x_n, x_1, x_{n-1}, x_2 \ldots x_{\frac{n}{2}-1}$ (interleaving)*

Then the size of the BDD is exponential in n wrt. ordering 1, and polynomial in n wrt. ordering 2.

Example 3.15. *For convenience we will state such a PBC for $n = 8$:*

$$1x_1 + 2x_2 + 4x_3 + 8x_4 + 8x_5 + 12x_6 + 14x_7 + 15x_8 \geq 32$$

Note that for $1 \leq i < \frac{n}{2}$, it holds $a_{n-i+1} + a_i = 2^{\frac{n}{2}}$. E.g. in this example, $x_2 + x_7 = 16$.

Proof. of Proposition 3.14
Considering the first case: the variable ordering is $n, n-1, n-2, \ldots 1$. In the middle level (when $n/2$ variables have already been considered), we have tested all the *larger* coefficients and need a distinct node for each of the following combination:

- more than $n/4$ variables are 1. Then the threshold is exceeded and we would have a link to the 1 sink.
- less than $n/4$ variables are 1. Then the threshold can not be exceeded anymore and we would have a link to the 0 sink.

- exactly $n/4$ variables are 1. Every combination of $n/4$ positive assignments to the $n/2$ larger coefficients, correspond to a different combination of $n/4$ of the $n/2$ lower variables, such that they add to $(n/2) \cdot 2^{n/2}$. There are $\binom{n/2}{n/4}$ such combinations.

Since the third item is $\binom{n/2}{n/4}$ and the central binomial coefficient is exponential, the width of the BDD is exponential.

On the other hand there exists an ordering such that the size of the BDD is polynomial. In the following we will call two variables *matching each other* if their sum is equal to $2^{n/2}$. If we order such that the matching variables directly follow each other, the width of the BDD is $O(n)$. Let the ordering be $1, n, 2, n-1, 3, n-3 \ldots n/2, n/2-1$, thus the smaller $n/2$ variables are tested right before their matching counter part. In every odd level of this BDD we find maximally n nodes storing the following information:

- $n/2$ nodes counting the number of matching variables, where both were positive
- $n/2$ nodes counting the number of matching variables, where only the bigger one is positive

Since there can be maximally n nodes per level, the size of this BDD is n^2. In Figure 3.5 we schematically present the different two orderings and the shape of the corresponding BDDs.

We have shown now that there exist two variable orderings that result in exponential different sized BDDs. □

For constructing this example we needed the majority of the coefficients differ exponentially (intuitively, for all a_i, a_j with $a_i > a_j$ the ratio is greater than some $r > 1$, $\frac{a_i}{a_j} \geq r$). This leads to the assumption that we cannot construct an exponentially large BDD representing a PBC with only polynomial coefficients.

Proposition 3.16. *Let f be a BDD representing a PBC $\sum_{i=1}^{n} a_i, x_i \geq t$, then*

$$size(f) \leq n \cdot \sum_{i=1}^{n} a_i$$

Proof. Let us fix a level k. The number of different equivalence relations in that level is maximally $\sum_{i=k}^{n} a_i$. This occurs if all equivalence classes contain only one element, (except from the left most and right most). The equivalence class with the lowest representant is $[0]_k$ and with the maximal is $[\sum_{i=1}^{k-1} a_i]_k$. Since this holds for each level and we have n levels, it follows the upper bound $n \cdot \sum_{i=1}^{n} a_i$. □

From this we can determine an upper bound on the size by examining the coefficients of a PBC. With polynomial coefficients we tend to have a small BDD representation. With exponential coefficients, it is possible to construct a PBC with exponential BDD size in all variable orderings (for an example see [23] or [29]).

We conclude:

- if coefficients differ polynomial in n, then the size of the BDD representation is polynomial in n regardless of the variable ordering (this can be explained by Proposition 3.16).
- if coefficients differ exponential in n, then determining the best order is NP hard (which might have exponential size difference in the BDD). For a proof see [5].

4 Propagation in BDDs

In this section we introduce the computation of implications of a BDD. Propagation in SAT solvers is based on the unit propagation mechanism. Determining the implications of a clause is simply checking when the clause has only one element left. This element is then implied and propagated. The challenge of this task lies in its implementation, and there has been great effort in finding efficient data structures to optimize this procedure, like counting schemes or watched literals (see [6] for an overview). The success of SAT solvers is partly based on an optimized unit propagation.

A single BDD can encode any Boolean function and computing implications is more involved. From a CSP perspective, detection of all implied variables corresponds to achieving GAC on the global constraint represented by a BDD.

Logical consistency is insured by a BDD since it collapses to 0 if instantiated with an assignment that leads to a contradiction. In a satisfiable BDD, there is at least one path to the 1 sink. However, on its own, a BDD does not inform about implied variables.

This chapter is organized as follows. Firstly, we introduce the notion of an implication set on arbitrary Boolean formulas. Then we analyze how to compute the implications of a BDD and present several algorithms. In the final part we propose a change in the BDD data structure that tightly includes the computation of implications. We present a first attempt on algorithms for this new data structure and compare their size with normal BDDs.

4.1 Implications

Definition 4.1. *Let L be the set of all literals.*

- *$x \in L$ is an implication of the Boolean formula ϕ iff*

$$\phi \models x$$

- *the implication set (or implications) impl of a Boolean formula ϕ contains all implications*

$$impl(\phi) = \{x \in L \mid \phi \models x\}$$

For the two constant Boolean functions it holds $impl(0) = L$ and $impl(1) = \emptyset$.

Example 4.2. *For the propositional formula $f = (x_1 \vee x_2 \vee x_6) \wedge x_3$ there is only one implied literal: $impl(f) = \{x_3\}$.*

In the following proposition we analyze properties of implications wrt. to logical operations:

Proposition 4.3. *Let ϕ_1, ϕ_2 be Boolean formulas. Then*

1. *Conjunction:*
$$impl(\phi_1 \wedge \phi_2) \supseteq impl(\phi_1) \cup impl(\phi_2)$$
2. *Disjoint Conjunction ($var(\phi_1) \cap var(\phi_2) = \emptyset$):*
$$impl(\phi_1 \wedge \phi_2) = impl(\phi_1) \cup impl(\phi_2)$$
3. *Disjunction:*
$$impl(\phi_1 \vee \phi_2) = impl(\phi_1) \cap impl(\phi_2)$$

Proof.

1. Let $x \in impl(\phi_1) \cup impl(\phi_2)$, then $\phi_1 \models x$ or $\phi_2 \models x$. Without loss of generality $\phi_1 \models x$. Then
$$\begin{aligned} \phi_1 \models x &\Rightarrow \models \phi_1 \rightarrow x \\ &\Rightarrow \models (\phi_1 \wedge \phi_2) \rightarrow x \\ &\Rightarrow \phi_1 \wedge \phi_2 \models x \end{aligned}$$
Thus $\subseteq$ holds. The two sides are not always equal, for example $\phi_1 = x_1 \rightarrow x_2$ and $\phi_2 = \bar{x_1} \rightarrow x_2$ are a counter example.
2. When ϕ_1, ϕ_2 do not share any variables, equality holds: If $x \in impl(\phi_1 \wedge \phi_2)$ and we know that x can only appear in either ϕ_1 or ϕ_2, then it will appear in one of the sets in the conjunction.
3. Similarly for the equality of the disjunction and the intersection:
Case $\supseteq$: surely if x is in both implications sets, then we can project it out in the disjunction.
Case $\supseteq$: Let $x \in impl(\phi_1 \vee \phi_2)$.Then the following reasoning applies:
$$\begin{aligned} (\phi_1 \vee \phi_2) \models x &\Rightarrow \models (\phi_1 \vee \phi_2) \rightarrow x \\ &\Rightarrow \models (\phi_1 \rightarrow x) \wedge (\phi_2 \rightarrow x) \\ &\Rightarrow \models (\phi_1 \rightarrow x) \text{ and } \models \phi_2 \rightarrow x) \\ &\Rightarrow x \in impl(\phi_1) \cap impl(\phi_2) \end{aligned}$$

□

4.2 Computing Implications in BDDs

From this general view on implications we want to focus on computing implications of a BDD. As we have seen, implications are all $x \in L$ for which $f \rightarrow x$ is a tautology. Now f is represented by a BDD. In all models of f, x is assigned the same value. For BDDs this means, that variable x is set to the same value on all paths from the root to the 1 sink.

For determining these implications algorithmically we consider three possibilities

- a top down approach;
- a bottom up approach;
- a tight integration into the BDD data structure

4.2.1 Top Down Approach

A top down algorithm is tightly related to the formulation of implications in BDDs:

Proposition 4.4. *A variable x (resp. $\bar{x}$) is implied in a BDD iff*

- *all shortcuts (branch surpassing nodes labeled with x) end directly in 0; and*
- *for all nodes labeled with x all 0-branches (resp. 1-branches) go directly to 0*

Proof. Let x be an implied variable of a BDD. Then it is true in all models of the BDD, thus it is set to the same value on all paths from the root node to sink 1. This is the case when there is no path to the 1 sink where x is don't care (shortcut) or where we pass a node labeled with x and the negative descendant is nonzero. The other direction is by the same reasoning. □

Example 4.5. *Consider the two BDDs f and g of Figure 4.1, where all nodes of f at level k have the zero children linked to 0, and all nodes of g at level k have the one children linked to 0. Then x_k is an implied variable of f and $\bar{x}_k$ of g.*

To determine fixed variables, the BDD must be traversed from top to bottom, checking the definition for each level. A slightly different way is to initially set all domains of the variables to $\emptyset$ and then traversing the BDD in a backtrack search. For each node, value 1(0) is added to the domain of its variable if the high (low) branch is not the 0 sink. After this search, each variable has an updated domain, and all variables with a single domain are implications.

From the point of view of CSP to maintain GAC on an *ad hoc* global constraint represented by a BDD we find in [9] a similar algorithm. Improvements to avoid unnecessary traversals are incorporated into the search. Caching techniques and a so called delta cut is adopted for pruning.

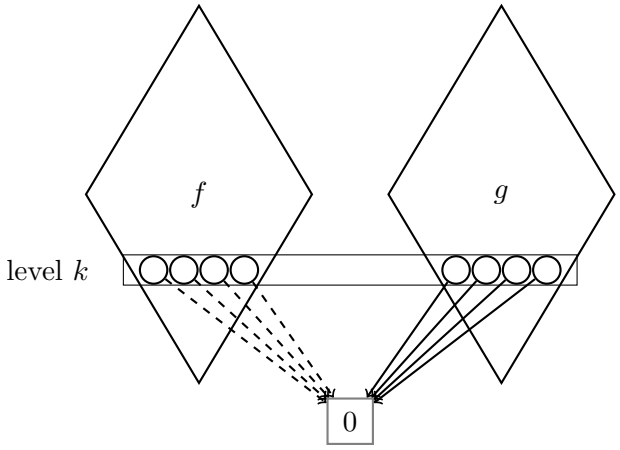

Figure 4.1: Two abstract BDDs f and g symbolized as a diamond. Level k is not shortcutted.

4.2.2 Bottom Up Approach

A possibly more elegant way to consider implications in BDDs is through recursion. The implications of a BDD are the intersection of the implications of its descendants or, when one of the descendants is 0, the variable (positively or negatively) is added. The 1 sink has no implications. Formally:

Proposition 4.6. *The implications of a nonzero BDD f labeled by variable v and the two children high and low, are given by the following recursive formula:*

$$impl(f) = \begin{cases} \emptyset & \text{if } f = 1 \\ impl(high) \cup \{v\} & \text{if } low = 0 \\ impl(low) \cup \{\overline{v}\} & \text{if } high = 0 \\ impl(low) \cap impl(high) & \text{otherwise} \end{cases}$$

Proof. By induction. Base cases: $f = 1$ is clear, there are no implications by the constant 1 function.

Induction step. For the case that one of the children is 0, let $low = 0$. From the semantics of a BDD node:

$$\begin{aligned} impl(f) &= impl((v \to high) \wedge (\overline{v} \to low)) \\ &= impl((v \to high) \wedge (\overline{v} \to 0)) \\ &= impl((v \to high) \wedge v) \\ &= impl(high \wedge v) \\ &= impl(high) \cup \{v\} \end{aligned}$$

Notice the equality in the last step, since v does not occur in $high$. The proof is similar

for the other case, $high = 0$.

To prove the final case, in addition to basic rewriting of propositional formulas, we use the identities from Proposition 4.3, and the fact that neither v nor $\bar{v}$ occur in the high or low subgraphs.

$$\begin{aligned}
impl(f) &= impl((v \to high) \wedge (\bar{v} \to low)) \\
&= impl((\bar{v} \vee high) \wedge (v \vee low)) \\
&= impl((v \wedge high) \vee (\bar{v} \wedge low) \vee (high \wedge low)) \\
&= impl((v \wedge high) \cap impl(\bar{v} \wedge low) \cap impl(high \wedge low)) \\
&= (\{v\} \cup impl(high)) \cap (\{\bar{v}\} \cup impl(low)) \cap impl(high \wedge low) \\
&= (impl(high) \cap impl(low)) \cap impl(high \wedge low) \\
&= impl(high) \cap impl(low)
\end{aligned}$$

□

Algorithm 4.1 is a direct implementation of Proposition 4.6:

Algorithm: basicImpl(f)
Input: BDD f
Output: Set of implications *impl*
1 **if** $f.low = 0$ **then** $impl \leftarrow basicImpl(f.high) \cup \{f.var\}$
2 **else if** $f.high = 0$ **then** $impl \leftarrow basicImpl(f.low) \cup \{\neg f.var\}$
3 **else** $impl \leftarrow basicImpl(f.low) \cap basicImpl(f.high)$
4 **return** *impl*

Algorithm 4.1: Basic Bottom-Up approach to compute the implications of a BDD

This algorithm can be optimized by noting that the implications of a BDD node f do not contain variables below x_i iff there exists a node in level i with an empty implication set. This is formalized by the following lemma:

Lemma 4.7. *Let f and g be BDD nodes such that there is a path from f to g (g occurs in f), and g is labeled with variable x_i, then*

$$impl(g) = \emptyset \quad \to \quad impl(f) \cap \{x_i, \bar{x}_i, x_{i+1}, \bar{x}_{i+1}, \ldots\} = \emptyset$$

Proof. Let $g_0, g_1 \ldots g_i$ be a path from f to g in reverse order, i.e. $g_0 = g$ and $g_i = f$. We proof by induction on n.

- base case: since $impl(g_0) = \emptyset$ it is given
- induction case: $impl(g_n)$ does not contain variables greater than i, case analysis:
  - if one of the descendants of g_{n+1} is 0, we add some variable x_{n+1} with $n < i$ to $impl(g_n)$

– if the other descendant has an implication set h with variables $\geq i$ they are lost, because of the intersection. $impl(g_{n+1}) = impl(g_n) \cap h$

□

Algorithm 4.2 improves Algorithm 4.1 by taking advantage of Lemma 1, by means of a delta cut Δ that stores the highest level with an empty implication. Moreover, it memorizes the calls, as usual in double recursive algorithms.

```
   Algorithm: compImpl(f)
   Input: BDD f
   Output: Set of implications impl
1  if Δ ⪯ f.var then return ∅
2  else if f ∈ visited then return visited.get(f)
3  else
4  |  if f.low = 0 then  impl ← compImpl(f.high) ∪ {f.var}
5  |  else if f.high = 0 then  impl ← compImpl(f.low) ∪ {¬f.var}
6  |  else  impl ← compImpl(f.low) ∩ compImpl(f.high)
7  |  if impl = ∅ ∧ f.var ≺ Δ then  Δ ← f.var
8  |  visited.put(f, impl)
9  |  return impl
```

Algorithm 4.2: Optimized Bottom-Up approach to compute the implications of a BDD

We suspect that our Δ corresponds to the Δ cut in [9]. The top down algorithm traverses the BDD and its complexity is linear in the number of nodes. The Bottom Up Algorithm is also linear in the number of nodes, but performs an intersection (linear in the number of variables) in each call. The number of variables is normally much smaller than the BDD size, but this corresponds still to some "overhead" in each call compared to the top down case.

The algorithms could be improved by some heuristic regarding which branch to look first. For example, it would be advantageous to try to reach the 1 sink as early as possible, because then the delta cut prunes many levels.

4.3 A Tight Integration: UPBDDs

The recursive characterization of implications in the previous section justifies a tight integration of the implication set into the BDD data structure. This section is centered around this basic idea.

We name the altered BDD data structure as *UPBDD*, where the prefix UP denotes that implication sets are pushed higher **UP** in the BDD, or from the original idea to include **U**nit **P**ropagation into BDDs.

For a first intuition, the flow of the implication set is analyzed, see Figure 4.2 on the left. The implications of node f are the intersection of the implications of g and h. Notice that the implications of node f correspond to the impled literals of the Boolean function f. Hence, they are the variables that take the same value in all models of f, or in BDD terms, on all paths from f to the sink 1 these variables are set to the same value.

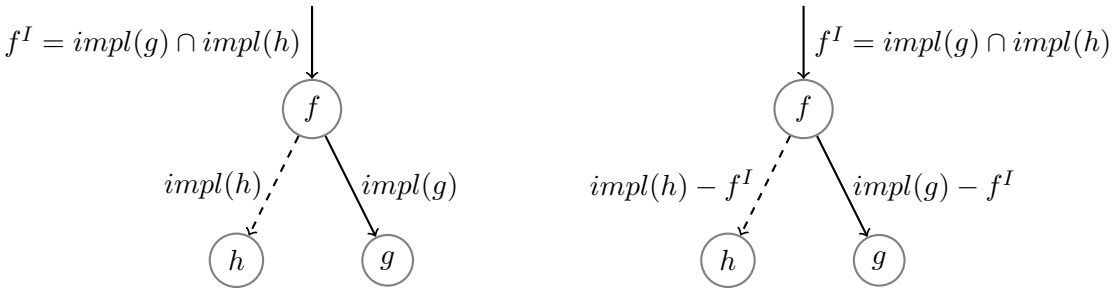

Figure 4.2: Basic idea from bottom up calculation to tight integration: BDDs to UPBDDs

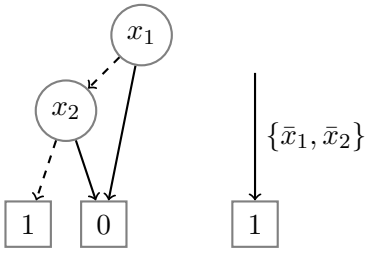

Figure 4.3: BDD size 2, UPBDD size 2

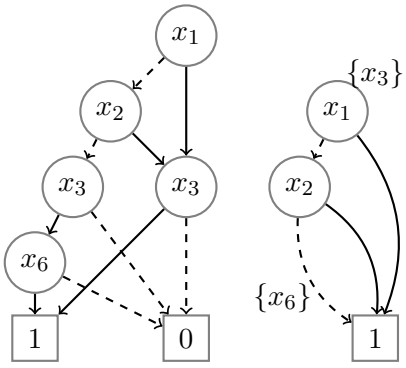

Figure 4.4: BDD size 5, UPBDD size 4

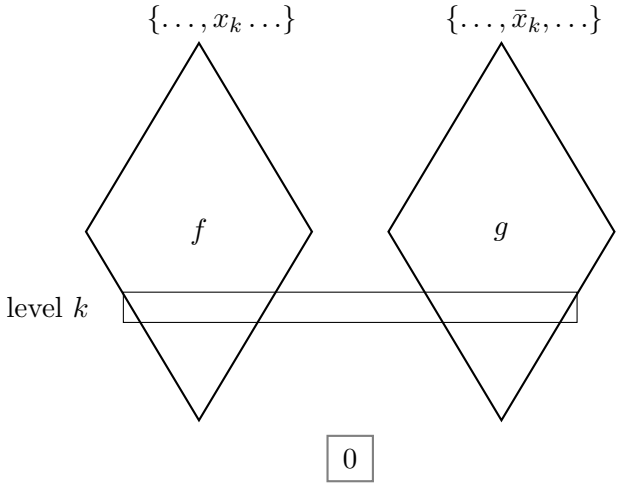

Figure 4.5: The UPBDD version of the two BDDs in Figure 4.1.

The key idea of UPBDDs is not only to keep these implied literals at node f (as if cached in the bottom up algorithm), but also remove the nodes labeled with them from the subgraphs of g and h, as seen in Figure 4.2 on the right. This is achieved by subtracting the common literals from the implication set of g and h.

An exhaustive application of this rule (intersection and subtraction) on a given BDD produces an UPBDD. See Figure 4.3 and 4.4 as small examples, that compare BDDs with the corresponding UPBDDs. The first is a trivial example showing that a BDD that contains only implications, corresponds to one single implication set, all nodes in the right BDD are removed and we are left with one set. It will be later explained why the set is at an edge to the 1 sink and not attached to the 1 sink. The second example shows a BDD where variable x_3 is implied at the root. The UPBDD version of this BDD contains the implication at the top node and nodes with x_3 are removed. Note that at the node label with variable x_2 we did not keep the implication x_3 as it got lifted up to the top node by exhaustive application of the intersection. The node with x_6 is removed from the BDD as well, as it is also an implication. However it does not take part in any intersection, thus is is only rewritten as a set with one element.

Figure 4.5 is a more abstract example of an UPBDD version of the two BDDs of Figure 4.1. In the UPBDD all nodes with variable x_k are removed and instead we have an additional implication on the root.

4.3.1 Implications Attached to the Nodes

By applying intersection and subtraction at every node (until no more change is possible, or if bottom up then only once for each node), we produce an UPBDD. Such implementation is based on the following property:

Proposition 4.8. *At every UPBDD node f with implication f^I it holds:*

$$var(f) \cap var(f^I) = \emptyset$$

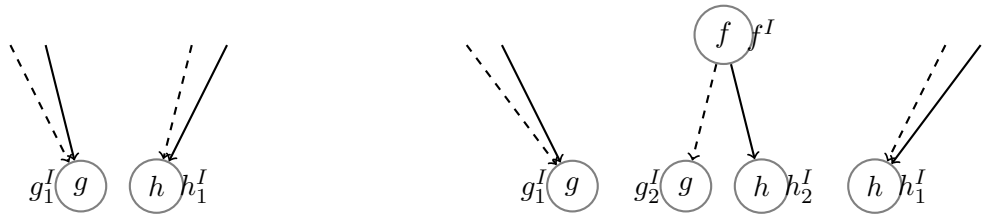

Figure 4.6: Left: UPBDD nodes with implication stored at the nodes. Right: We add node f with the two descendants g and h.

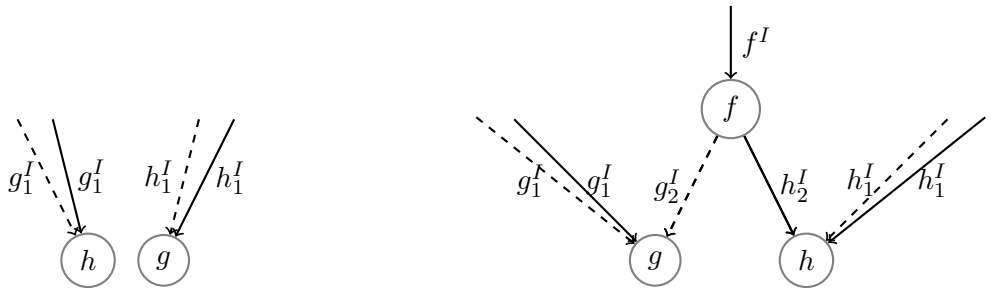

Figure 4.7: Left: Two UPBDDs with implications stored at the edges. Right: We add a node f with descendants $\langle g, g^I \rangle$ and $\langle h, h^I \rangle$. *addNode* returns $\langle f, f^I \rangle$.

We will now see how this can be easily integrated. The two reduction rules for BDDs (no redundant nodes and common subgraphs) are implemented within the node management. In a similar manner, the UPBDD property can be insured whenever we add a node. See Figure 4.6 for an illustration. On the left, UPBDD nodes g and h have attached implication set g_1^I and h_1^I. When a node with g and h as successors is added with some element in common, then the intersection is nonempty $f^I = g_1^I \cap h_1^I$. To ensure now the reduction property of UPBDDs we must duplicate nodes g and h (nodes with the same successors and variable) to which the updated implication sets $g_2^I = g_1^I \setminus f^I$ and $h_2^I = h_1^I \setminus f^I$, respectively, are attached. Note that we cannot reuse the initial nodes g, g^I and f, f^I since they are referenced by other nodes (incoming arcs from elsewhere). After this preprocessing we can safely add node f with its implication set f^I and successors g with g_2^I and h with h_2^I. Finally we have added node f and all nodes obey the UPBDD property. However, we potentially needed to create two additional nodes. This can be avoided by referencing the implication sets in the arcs.

4.3.2 Implications Attached to the Arcs

To avoid the creation of possibly two additional nodes whenever we insert UPBDD nodes (as seen in Figure 4.6), we attach the implication set at the incoming arcs instead of the nodes. Henceforth, an UPBDD node u with an implication set u^I attached to its incoming arc, is denoted by the tuple $\langle u, u^I \rangle$.

See Figure 4.7 as an illustration. We have two distinct nodes g and h where implications g_1^I and h_1^I are stored in all incoming edges. If we now add node f with descendants g and h

(and arcs with g^I and h^I, respectively), we perform the same steps as before, intersection and and subtraction. However, we only create one node f with the new two descendants $\langle g, g_2^I \rangle$ and $\langle h, h_2^I \rangle$.

It might seem strange to avoid duplication of nodes by duplication of implication sets (g_1^I and h_1^I occur here twice instead of once). We will see later that equal implication sets will only be stored once. This is done by externalizing the implication sets and having references at the arcs. See Section 4.4 for a discussion.

Now we can define the components of an UPBDD node:

Definition 4.9. *An UPBDD node f consists of five components: The level (or variable) var, the high and low branch, and the implication sets at the arcs to high and low. An entry in the node table is then*

$$(var, \langle high, high^I \rangle, \langle low, low^I \rangle)$$

As discussed before, the process of inserting a new node and keeping the UPBDD property can be seen as an extension of the *insert* operation for BDDs. See Algorithm 4.3. If one of the descendants is 0 we introduce a new implication to the implication set of the other descendant (line 1,2). We intersect (line 4) and subtract the implication sets in addition to the usual BDD redundancy test (line 3,5). Algorithm *addNode* can be seen as the *insert* algorithm for BDDs combined with one step in the bottom up computation of a BDD.

Algorithm: addNode(var,$\langle high, high^I \rangle$,$\langle low, low^I \rangle$)
Input: Level var, the two descendent UPBDDs with implications
Output: Returns the id f of the new node and the implication f^I
1 **if** $low = 0$ **then return** $\langle high, high^I \cup \{var\} \rangle$
2 **if** $high = 0$ **then return** $\langle low, low^I \cup \{\neg var\} \rangle$
3 **if** $low = high$ *and* $high^I = low^I$ **then return** $\langle high, high^I \rangle$
4 $f^I \leftarrow high^I \cap low^I$
5 **if** $f \leftarrow lookup(var, \langle high, high^I \setminus f^I \rangle, \langle low, low^I \setminus f^I \rangle)$ **then return** f
6 **else**
7 $f \leftarrow$`insert(`var, $\langle high, high^I \setminus f^I \rangle$,$\langle low, low^I \setminus f^I \rangle$`)`
8 **return** $\langle f, f^I \rangle$

Algorithm 4.3: The insert algorithm in an UPBDD context.

The intersection and two subtractions are linear in the size of the implication sets. If we want to add a new node and we know in advance that the two implication sets are disjoint, we can avoid the intersection and subtraction by providing a specialized *addNode* algorithm. We omit this here, since it would be equivalent to the normal *insert* for BDDs.

4.3.3 Algorithms

We have explained what an UPBDD is and how it can be transformed from a BDD. If nodes are added bottom up (as in BDDs), we can ensure reduction rules throughout building an maintaining of UPBDD graphs. The next step is to examine how we can build algorithms working on this data structure. In general, algorithms applicable to BDDs will

be adopted to UPBDDs (we conjecture that they will have similar or less complexity). In particular we are interested in instantiation, negation, binary Boolean operations, parallel Boolean operations, model counting etc.

For UPBDDs we propose the following guidelines. Algorithms should be defined recursively and new nodes added bottom up. The global ordering of variables should not be violated throughout the execution. Algorithms should produce correct UPBDDs.

To take advantage of UPBDDs, the implication sets should be used to identify base cases early in the recursion (e.g. contradiction in conjunction). The caching can be improved with UPBDDs when special hashing functions for UPBDDs are defined.

In this thesis we focus on conjunction and present a first simple approach.

4.3.4 Conjunction

First we present algorithm *propagate* that takes two UPBDDs and their implication sets and instantiates each other until a fixpoint is found. Then it returns all common implications and the input nodes with common implications removed, see Algorithm 4.4. This algorithm will be of help in the conjunction.

Algorithm: propagate($\langle f, f^I\rangle$,$\langle g, g^I\rangle$)
Input: two UPBDDs $\langle f, f^I\rangle$,$\langle g, g^I\rangle$
Output: each instantiated and the combined implication
1 $c_1^I \leftarrow f^I \cup g^I$
2 **if** $c_1^I = \emptyset$ **then return** $(f, g, \emptyset)$
3 **else if** $\{x, \bar{x}\} \subseteq c_1^I$ **then return** $(0, 0, \emptyset)$
4 **else**
5 $\quad \langle g, g^I\rangle \leftarrow$ `instantiate(`g,f^I`)`
6 $\quad \langle f, f^I\rangle \leftarrow$ `instantiate(`f,g^I`)`
7 $\quad (f, g, c_2^I) \leftarrow$ `propagate(`$\langle f, f^I\rangle$,$\langle g, g^I\rangle$`)`
8 $\quad$ **return** $(f, g, c_1^I \cup c_2^I)$

Algorithm 4.4: Propagation between two UPBDDs. The UPBDDs instantiate each other until no new implications are detected. Instantiation of an UPBDD is straight forward.

The conjunction is a simple extension of its corresponding BDD algorithm. This extension considers the implication sets in each recursion call, see Algorithm 4.5.

The conjunction algorithm is called with $\langle f, f^I\rangle$ and $\langle g, g^I\rangle$. First we check if the implication sets contradict each other (line 1). Then we check the base cases as in the BDD conjunction (line 2,3). Note that we never call *and* with the zero sink. Then we call *propagate* with both nodes and implications. Now we can be sure that the new f and g do not contain variables from c^I. Then we check if propagation lead to a contradiction (line 5). Otherwise we call the *and* recursive as in the usual BDD algorithm and add the new node with the new descendants. However, we return it by adding the implication set computed in *propagate*.

Algorithm: and($\langle f, f^I\rangle$,$\langle g, g^I\rangle$)
Input: two UPBDDs $\langle f, f^I\rangle, \langle g, g^I\rangle$
Output: the conjunction

1 **if** $\{x, \bar{x}\} \subseteq (f^I \cup g^I)$ **then return** $\langle 0, \emptyset\rangle$
2 **else if** $f = g$ *or* $g = 1$ **then return** $\langle f, f^I \cup g^I\rangle$
3 **else if** $f = 1$ **then return** $\langle g, g^I \cup f^I\rangle$
4 $(f, g, c^I) \leftarrow$`propagate(`$\langle f, f^I\rangle$,$\langle g, g^I\rangle$`)`
5 **if** $f = 0$ *or* $g = 0$ **then return** $\langle 0, \emptyset\rangle$
6

$$\langle high, high^I\rangle = \begin{cases} and(f.high, \langle g, \emptyset\rangle)) & f.var \prec g.var \\ and(\langle f, \emptyset\rangle, g.high)) & f.var \succ g.var \\ and(f.high, g.high)) & f.var = g.var \end{cases}$$

7

$$\langle low, low^I\rangle = \begin{cases} and(f.low, \langle g, \emptyset\rangle)) & f.var \prec g.var \\ and(\langle f, \emptyset\rangle, g.low)) & f.var \succ g.var \\ and(f.low, g.low)) & f.var = g.var \end{cases}$$

8 $\langle r, r^I\rangle \leftarrow$`addNode(`$min(f.var, g.var)$,$\langle high, high^I\rangle$,$\langle low, low^I\rangle$`)`
9 **return** $\langle r, r^I \cup c^I\rangle$

Algorithm 4.5: Modification of the binary conjunction for UPBDDs, with a complete propagation on both subtrees.

This conjunction can identify many contradictions early. However, a significant drawback is the additional traversal of each UPBDD in *propagate* in each call. This algorithm could serve as a first step towards a more efficient *and*. Possibly we could weaken the propagation and perform a more lazy instantiation, but to this point we have to refer to further work in that issue.

The motivation behind conjunction of UPBDDs is shown with the following examples:

Example 4.10. *Consider the two BDDs in Figure 4.1 and the UPBDDs in Figure 4.5. Obviously their conjunction leads to a contradiction. The BDD conjunction algorithm would detect the contradiction at latest in level k, after having done a huge traversal. The UPBDD conjunction algorithm would detect the contradiction from the implied literals of the roots.*

In many cases contradictions are detected earlier than with BDDs. However, with UPBDDs there is no guarantee that contradictions are detected by simply checking the impled sets at the roots. The following example shows how Algorithm 4.5 conjoins BDDs. Here we do not detect the contradiction in the first call:

Example 4.11. *Given f and g as*

$$f = (x_1 \vee x_2) \wedge ((x_3 \vee x_4) \rightarrow x_5)$$
$$g = (x_3 \vee x_4) \wedge ((x_1 \vee x_2) \rightarrow \bar{x}_5)$$

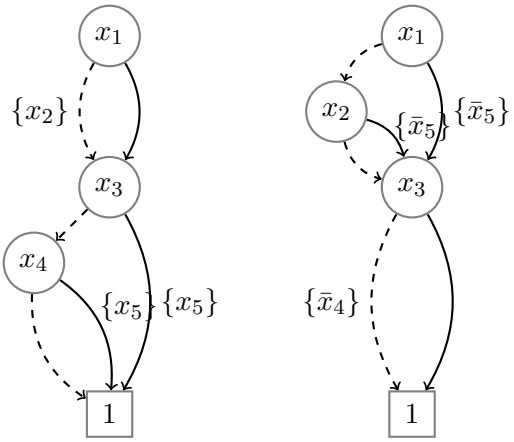

Figure 4.8: The two UPBDDs f and g of Example 4.11

The conjunction algorithm would detect the contradiction in the propagation phase after branching on the top variable:

1. *The positive descendant of x_1 in g has one implication $\bar{x}_5$. In the propagation this causes f to give back $\{\bar{x}_3, \bar{x}_4\}$, because both positive branches from x_3 and x_4 are cut off. This implication then lead to a contradiction in g. Note that this whole process takes place in the propagation phase after the first branching step.*

2. *The negative branch of x_1 gives implication x_2 in f, which leads to implication $\bar{x}_5$ in g. With this $\bar{x}_5$, we have the same situation as in the previous case.*

4.4 Comparing Size of BDDs vs. UPBDDs

To analyze the sizes of UPBDDs wrt. BDDs, we firstly state an intuitive result for BDDs. The algorithm *restrict* instantiates a BDD with a partial assignment, see Algorithm 2.3. In general, the size of a BDD decreases when instantiated by its implications. More precisely we state the following proposition:

Proposition 4.12. *Given a BDD f and its implication set $impl(f)$, then*

$$size(f) \geq size(restrict(f, impl(f)) + size(impl(f))$$

The size of a BDD is the number of its nodes, where shared nodes count only once. For UPBDDs we distinguish between the actual UPBDD nodes and the space needed for storing the implication sets. The size of implication sets depends on how they are implemented. We store the sets as BDDs - preserving the global order, thus having the possibility to share tails.

Definition 4.13. *The size of an UPBDD is the number of (distinct) nodes plus the representation of the implication sets, that are stored in separate BDDs with the same global ordering.*

This definition of UPBDD size should guarantee a decrease in number of nodes, transforming BDDs to UPBDDs. Surprisingly, we cannot extend the result from Proposition 4.12 to UPBDDs:

Proposition 4.14. *Let the BDD f and UPBDD g with the same ordering represent the same Boolean function. $size(f)$ and $size(g)$ are incomparable.*

Proof. We consider 2 examples: First the expected case, that the size is smaller (or equal), see Figure 4.4. However, we can construct a special BDD where the UPBDD version is larger, see Figure 4.10. □

There are several observations, that explain the increase and decrease of size from BDDs to UPBDDs:

- Through the intersection and subtraction we push implications up in the BDD and possibly decrease the size of representation. This can be seen in Figure 4.4, there variable x_3 is lifted and in total a reduction of one node is obtained. This was the initial motivation behind UPBDDs.
- More subfunction sharing: as in Figure 4.9, by lifting variable x_4 there is a sub function sharing that was not possible before.
- More subset sharing: by externalizing the implication set, we might encounter implication set sharing, that was not possible before. Savings can be substantial, more than linear on the number of variables. Example 4.15 is a good example.
- The increase in nodes from Figure 4.10 is due to loss of subset sharing, because by lifting some variables we might loose the suffix sharing feature. We expect this case to happen rarely.

Example 4.15. *Let $f_1, f_2 \dots f_n$ be distinct Boolean functions containing variables $x_{k+1}, x_{k+2} \dots x_n$ and with empty implications sets. We construct a function with the following meaning: Exactly one x_i is true and from each variable x_i follows function f_i:*

$$\bigvee_{1 \le i \le k} x_i \quad \wedge \quad \bigwedge_{1 \le i < j \le k} (\bar{x}_i \vee \bar{x}_j) \quad \wedge \quad \bigwedge_{1 \le i \le k} (x_i \to f_i)$$

The difference decrease in nodes wrt. BDDs due to sublist sharing is $\frac{(k-2)\cdot(k-1)}{2}$ (in the levels 0 to k). See Figure 4.11 for the BDD representation and 4.12 for the UPBDD representation.

We finish this section on UPBDDs with the following remarks:

- UPBDDs stay canonical, i.e. two UPBDDs $\langle f, f^I \rangle$ and $\langle g, g^I \rangle$ are semantically equal if and only if the pointers f and g are equal and f^I and g^I are equal.

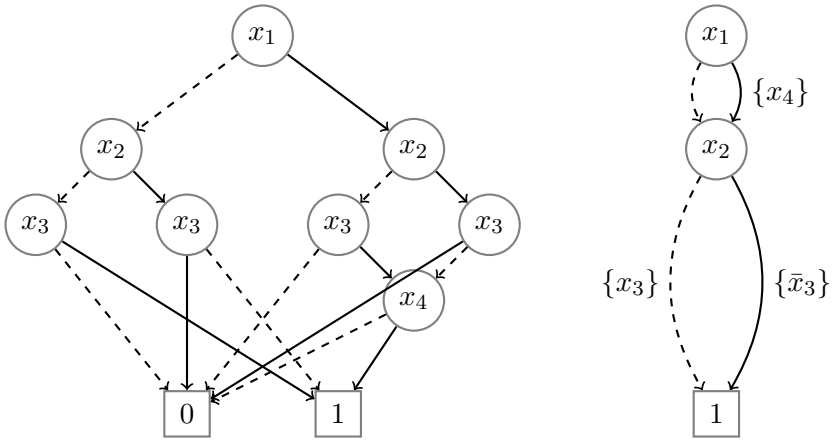

Figure 4.9: BDD size 8, UPBDD size 5

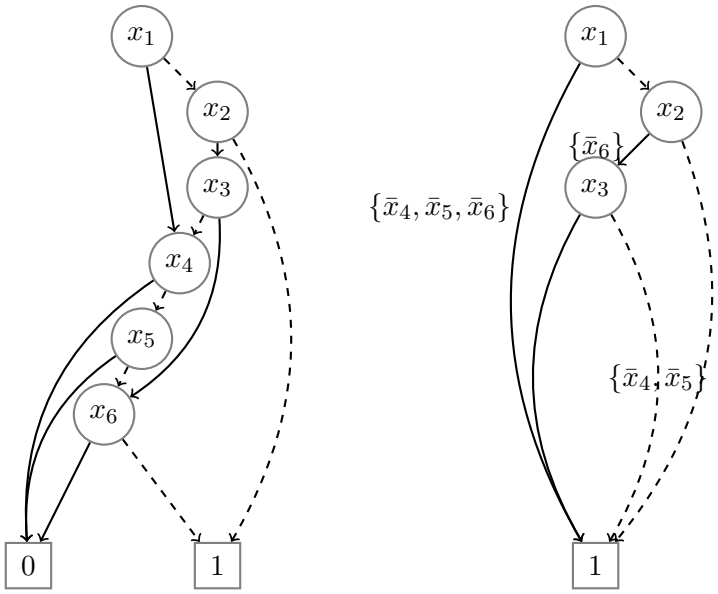

Figure 4.10: BDD size 6, UPBDD size 8

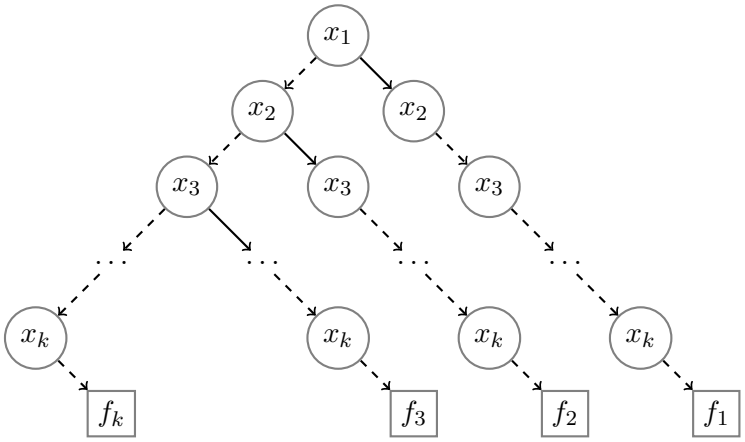

Figure 4.11: The BDD to Example 4.15. Links to the 0 sink have been omitted for clarity.

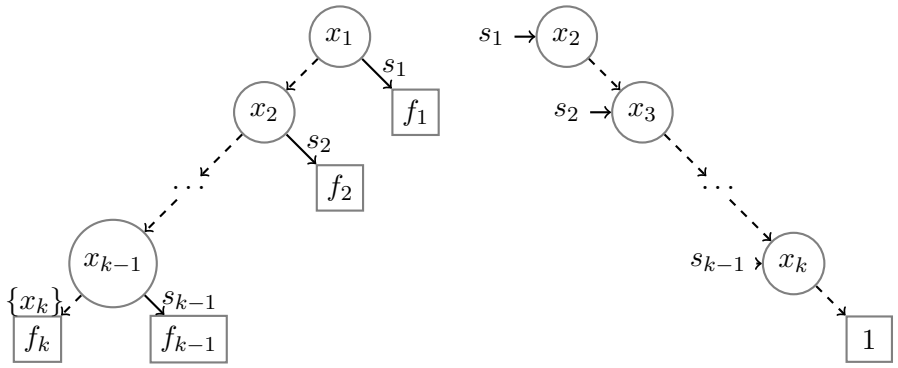

Figure 4.12: The UPBDD to Example 4.15. $s_1, s_2 \dots s_k$ are pointers to the set sharing BDD

- The BDD algorithms can be transformed to UPBDDs. The UPBDD specific algorithms include only minor overhead to handle the implication sets. In many cases the implication sets can be used to identify base cases earlier in the recursion.
- The size of UPBDDs tends to decrease wrt. to its corresponding BDD.
- Different variable orderings for UPBDD can have exponential different sizes in the UPBDD (as for BDDs).

5 Evaluation

In this chapter we compare the size of BDD and UPBDD representations and apply a prototypical implementation of a monolithic and search based BDD solver on three different instances of the Pseudo Boolean Evaluation: Pigeon Hole, Weighted NQueens and Equality Knapsack. These problems are classical benchmark problems of both the SAT and CSP community.

First each PBC is translated to a BDD. Then the monolithic approach consist of conjoining these BDD representations. The computed single BDD contains a compact representation of all solutions to the initial problem. The search based approach clusters the BDD representations to some certain size and then searches on theses larger constraints for a solution. We instantiate the variable in the same order as the BDDs. We have not yet included the propagation mechanism discussed in the previous chapter.

This chapter is organized as follows: First we present and discuss our experiments to compare sizes between BDDs and UPBDDs. Then we present the obtained results from our monolithic/search based implementation and describe for each class of benchmarks problems the PBC encoding. For the discussion to the pigeon hole problem, we additionally introduce an alternative approach, the cutting plane proof system.

Choices of the Implementation The monolithic and search based approaches are implemented in Java with the BDD framework JavaBDD[1]. This framework interfaces to popular C/C++ BDD frameworks like CUDD[2] or BuDDy [3]. We conducted our experiments on a computer running Windows XP with 1GB RAM and 1.6 GHz CPU.

Moreover, we have a prototypical BDD/UPBDD implementation in order to compare sizes, which was written in SWI-Prolog[4]. This implementation does not aim at performance, but suffices as a prototype and is kept simple and clear.

5.1 Comparing Size: UPBDDs vs. BDDs

We compare the different representation sizes of BDDs and UPBDDs wrt. to a selection of benchmark problems from the Pseudo Boolean evaluation, see Table 5.1: army, blast-floppy, chnl, fpga, pigeon holes, and weighted nqueens. The number in brackets show the number of benchmark problems over which the mean is taken. We have two main columns labeled "no clustering" and "clustering(500)". In "no clustering" we consider the BDDs that are output of the transformation algorithm. In "clustering(500)" we conjoin

[1] www.javabdd.sourceforge.net
[2] www.vlsi.colorado.edu/~fabio/CUDD
[3] www.sourceforge.net/projects/buddy
[4] www.swi-prolog.org

	no clustering			**clustering(500)**		
	decrease	nodes/up	set/up	**decrease**	nodes/up	set/up
army (9)	0,54%	77,05%	22,95%	59,53%	90,57%	9,43%
blast-floppy (3)	10,72%	78,30%	21,70%	59,85%	78,66%	21,34%
chnl (9)	0,00%	64,78%	35,22%	32,58%	81,92%	18,08%
fpga (7)	0,00%	78,97%	21,03%	16,28%	95,91%	4,09%
pigeon (8)	0,00%	65,37%	34,63%	27,07%	79,66%	20,34%
nqueens (1)	-9,48%	71,19%	28,81%	14,79%	75,97%	24,03%

Table 5.1: Comparison of different sizes of BDDs vs UPBDDs

the BDDs until they exceed a size 500. The variable ordering and the conjunction order is the same as in the problem specification.

Each BDD representation of a problem is translated to the equivalent UPBDD representation and then these structures are compared in size. In the column "decrease" we state the decrease in percent of nodes that were saved by the UPBDD representation. The columns "nodes/up" and "set/up" show the share of nodes in the UPBDD, that are real UPBDD nodes and those that belong to the set sharing BDD (these two entries add up to 100%).

We state the following observations:

- In the initial translation, there is not a large decrease in nodes (in the Nqueens even an increase). On average 25% of BDD nodes are implications.
- the advantage is visible after some clustering (conjunction of constraints). Here the number of nodes decrease by using the UPBDD representation, with up to 60%.

5.2 The Pigeon Hole Problem

The pigeon hole problem is a simple, but still hard problem for search based solvers. The problem itself seems trivial: Can we cannot put m pigeons into n holes, when $m > n$ and maximally one pigeon fits into each hole? This is clearly unsatisfiable.

5.2.1 Encoding

The PBC encoding of the pigeon hole problem (m, n) is straight forward. We have for each combination of pigeon $1 \leq i \leq m$ and hole $1 \leq j \leq n$ a Boolean variable $x_{i,j}$, see Figure 5.2.1 for a visualization. For every row i (the pigeon) at least one $x_{i,j}$ must be true and for every column j (the holes) at most one $x_{i,j}$ is true:

$$\sum_{j=1}^{n} x_{i,j} \geq 1 \quad \text{for} \quad i = 1 \ldots m$$

$$\sum_{i=1}^{m} x_{i,j} \geq 1 \quad \text{for} \quad j = 1 \ldots n$$

$x_{1,1}$	$x_{1,2}$	$x_{1,3}$	≥ 1
$x_{2,1}$	$x_{2,2}$	$x_{2,3}$	≥ 1
$x_{3,1}$	$x_{3,2}$	$x_{3,3}$	≥ 1
$x_{4,1}$	$x_{4,2}$	$x_{4,3}$	≥ 1
≤ 1	≤ 1	≤ 1	

Figure 5.1: Pigeon Hole $(4, 3)$ example. Each row requires to have et least one cell and each column at most one cell to be 1.

5.2.2 Solving Pigeon Hole with the BDD based approach

According to [12] solving pigeon hole with a resolution based approach is not very promising. In general SAT solvers perform poor on such instances and can not solve for more than $m = 15$. The same observation was also done in our approach. We tried to solve the pigeon hole with the monolithic and search based approach. However the search based method performs considerably worse than the monolithic, so we concentrate on the latter here.

In Table 5.2 we show the number of nodes produced with the monolithic method wrt. two different orderings of variables and constraints. However, in all tried cases our approach has exponential behavior and solves only up to 18 pigeons. In the original benchmark, the largest instance has 101 pigeons. So for our case, we generated smaller instances up to 20.

5.2.3 The Cutting Plane Proof System

For comparison to a completely different inference system, we briefly present the *cutting plane* proof system, which was introduced by Gomory [14]. It is a complete proof systems for problems given as PBCs.

A proof with resolution can be simulated and there are examples where a resolutions proof requires exponential more steps (like the pigeon hole). Thus it is strictly *stronger* than the resolution proof system. There are variants to the initial cutting plane proof systems and several solvers use this powerful inference system, e.g. see [8, 12].

There are two inference steps of the cutting plane proof system. First there is the possibility to derive a new constraint by taking a linear combination of two others ($q_1, q_2 > 0$)

$$\frac{\begin{array}{c}\sum a_i x_i \geq t_1 \\ \sum b_i x_i \geq t_2\end{array}}{\sum q_1 a_i x_i + \sum q_2 b_i x_i \geq q_1 t_1 + q_2 t_2}$$

If $q_1 = q_2 = 1$, we basically add the two left sides and the two right sides and derive a

pigeons	var	con	ordering1	ordering2
6-5	30	11	258	510
7-6	42	13	642	1560
8-7	56	15	1538	4458
9-8	72	17	3586	12108
10-9	90	19	62343	31630
11-10	110	21	18434	80144
12-11	132	23	40962	198162
13-12	156	25	90114	480276
14-13	182	27	196610	1144854
15-14	210	29	425986	2691096
16-15	240	31	917506	timeout
17-16	272	33	1966082	timeout
18-17	306	35	4194306	timeout

Table 5.2: Results from solving the pigeon hole problem with the monolithic approach. The largest occurring BDD in the computation is shown with respect to two different orderings.

new (weaker) inequality.

The second inference in the proof system gives fractional rounding ($d > 0$ and $\lceil \cdot \rceil$ is the round up operation):

$$\frac{\sum a_i x_i \geq t}{\sum \lceil \frac{a_i}{d} \rceil x_i \geq \lceil \frac{t}{d} \rceil}$$

A derived constraint from this rule is called a *cut*. These two rules are a sufficient proof system. Moreover, they can also be used two generate new constraints in solving pseudo Boolean problems.

5.2.4 Proof of the Pigeon Hole by Pseudo Boolean Inference

The difference between the two reasoning mechanism (BDD conjunction and cutting planes) becomes visible when realizing the simplicity behind the cutting plane proof of the pigeon hole problem.

Here we will show a short proof of the pigeon hole problem, that simply uses the first rule of the cutting plane proof system on all constraints. This results in a contradiction $0 \geq 1$ (or in the general case, $n \geq m$).

First we *sum* up the constraints, that state that in each row there has to be at least one pigeon. Here the first two rows:

$$\frac{\sum_{k=1}^{n} x_{1,k} \geq 1 \quad \sum_{k=1}^{n} x_{2,k} \geq 1}{\sum_{\substack{1 \leq i \leq 2 \\ 1 \leq k \leq n}} x_{i,k} \geq 2}$$

We do this consecutively to all constraints and get

$$\sum_{\substack{1\leq i\leq m\\1\leq k\leq n}} x_{i,k} \geq m$$

i.e. the sum of all variables is greater than m. This means that at least m pigeons have to be put in n holes, this does not yet contain that not several pigeons can go into the same hole. Now we add all the column constraints:

$$\frac{\begin{array}{c}\sum_{i=1}^{m} -x_{i,1} \geq -1\\ \sum_{i=1}^{m} -x_{i,2} \geq -1\\ \vdots\\ \sum_{i=1}^{m} -x_{i,n} \geq -1\end{array}}{\sum_{\substack{1\leq i\leq m\\1\leq k\leq n}} -x_{i,k} \geq -n}$$

I.e. in total in all holes there can be n pigeons. The final step is to add these two inequalities.

$$\frac{\frac{\begin{array}{c}\sum_{\substack{1\leq i\leq m\\1\leq k\leq n}} x_{i,k} \geq m\\ \sum_{\substack{1\leq i\leq m\\1\leq k\leq n}} -x_{i,k} \geq -n\end{array}}{\sum_{\substack{1\leq i\leq n\\1\leq k\leq m}} (x_{i,k} - x_{i,k}) \geq m - n}}{n \geq m}$$

Since it is given $m > n$ in the pigeon hole, we have derived an unsatisfiable inequality. The derivation is linear and uses only one of the two inference rules.

The pigeon hole brings down the huge difference between mathematical reasoning (arithmetic) and logical (resolution). This makes the problem interesting also from a theoretical perspective.

5.3 The Weighted NQueens

The NQueens problem is a well known and popular example for constraint programming. The problem requires to place n queens on a $n \times n$ chessboard s.t. they do not attack each other. The weighted NQueens extends this problem by assigning a weight to each cell on the board and asks for maximizing/minimizing the sum of weights wrt. the cells that contain a queen.

5.3.1 Encoding

Encoding of the NQueens problem is similar to the pigeon hole. Each cell on the board is modeled by one Boolean variable $x_{i,j}$, e.g. see a 8×8 chessboard in Figure 5.2. In each row and column there is exactly one queen, encoded by the following equality constraints:

$x_{1,1}$	$x_{1,2}$	$x_{1,3}$	$x_{1,4}$	$x_{1,5}$	$x_{1,6}$	$x_{1,7}$	$x_{1,8}$
$x_{2,1}$	$x_{2,2}$	$x_{2,3}$	$x_{2,4}$	$x_{2,5}$	$x_{2,6}$	$x_{2,7}$	$x_{2,8}$
$x_{3,1}$	$x_{3,2}$	$x_{3,3}$	$x_{3,4}$	$x_{3,5}$	$x_{3,6}$	$x_{3,7}$	$x_{3,8}$
$x_{4,1}$	$x_{4,2}$	$x_{4,3}$	$x_{4,4}$	$x_{4,5}$	$x_{4,6}$	$x_{4,7}$	$x_{4,8}$
$x_{5,1}$	$x_{5,2}$	$x_{5,3}$	$x_{5,4}$	$x_{5,5}$	$x_{5,6}$	$x_{5,7}$	$x_{5,8}$
$x_{6,1}$	$x_{6,2}$	$x_{6,3}$	$x_{6,4}$	$x_{6,5}$	$x_{6,6}$	$x_{6,7}$	$x_{6,8}$
$x_{7,1}$	$x_{7,2}$	$x_{7,3}$	$x_{7,4}$	$x_{7,5}$	$x_{7,6}$	$x_{7,7}$	$x_{7,8}$
$x_{8,1}$	$x_{8,2}$	$x_{8,3}$	$x_{8,4}$	$x_{8,5}$	$x_{8,6}$	$x_{8,7}$	$x_{8,8}$

Figure 5.2: How to put 8 queens on a chess board, such that they do not attack each other?

$$\sum_{j=1}^{n} x_{i,j} = 1 \quad \text{for} \quad i = 1 \dots n$$
$$\sum_{i=1}^{n} x_{i,j} = 1 \quad \text{for} \quad j = 1 \dots n$$

Note that $=$ can be encoded by two inequalities. Furthermore, queens are not allowed to attack each other on the diagonals:

$$\sum_{1 \le i,j \le n, i=j+k} x_{i,j} \le 1 \quad \text{for} \quad k \in [-(n-2), \dots, (n-2)]$$
$$\sum_{1 \le i,j \le n, i+j=k} x_{i,j} \le 1 \quad \text{for} \quad k \in [3, \dots 2n-1]$$

For the weighted NQueens an additional constraint has to be full filled: To every cell (i, j) is attached a weight $w_{i,j}$ and the sum over the weights that contain a queen, has to be greater than a certain given threshold t. Note that here we state the decision version of the weighted NQueens:

	Monolithic	**1000**	**10000**	**50000**
constraints	1	11	7	5
clustering (time)	3779	19	149	590
search (time)	0	4873	471	220
clustering nodes	279236	20523	88650	328177

Table 5.3: Average numbers of the cluster and search method from solving all 100 problems given in the Benchmark. Time in milisec.

	Monolithic				**Search based**		
					1000	**10000**	**50000**
instance	**solutions**	**time**	**max nodes**	**unsat**	**time**	**time**	**time**
1110966483	0	704	91191	80	4266	469	156
1110966688	0	1141	106398	77	5047	328	140
1110967142	0	13109	774410	87	9938	2281	766
1110967327	1	6953	521744		3156	625	328
1110967523	0	3343	296691	81	4282	594	187
1110967729	0	1000	106871	72	3594	312	63
1110968094	0	1938	173850	82	4953	594	250
1110968431	1	1172	110611		2953	343	141
1110968561	0	3890	379283	85	4656	718	266
1110968898	0	969	89373	81	6016	141	46
1110969170	0	1203	145720	68	4984	250	156
1110969860	0	5109	472007	80	9782	1328	625
1111217374	31	34453	1632340		766	140	141
1111217380	3	5344	415127		3281	797	281
1111217392	0	2187	201105	85	3953	359	188

Table 5.4: A selection of weighted 13queens problem. Comparison of the monolithic with a search based approach. Times in milliseconds

$$\sum_{1 \leq i,j \leq n} w_{i,j} x_{i,j} \geq t \quad \text{with} \quad w_{i,j}, t \in \mathbb{Z}$$

5.3.2 Monolithic vs. Simple Search

The weighted NQueens problem was one of the benchmark class where our solver performed well. We were able to solve all 100 problems with both approaches.

For the variable ordering is unchanged from the specification of the benchmark and the order of conjunction was wrt. the size of the constraint, decreasing. In the search based approach we tried three different threshold for the cluster size, 1000, 10000 and 50000 nodes.

In Table 5.3 we show the average solution time and size of BDDs. In Table 5.4 we present a more detailed view on a selection of all instances. For the monolithic approach, there is also the number of models (column "solutions") and after how many constraints contradiction was detected out of 101 constraints (column "unsat").

We state the following observations:

- The monolithic approach can compete with the other solvers in the benchmark. The case of unsatisfiability is normally detected before all constraints were conjoined.
- In the search based approach, after clustering we find the same number of global constraint in all instances. Comparing the three columns in Table 5.3 we observe a tradeoff between threshold of cluster sizes and search time. This tradeoff seems to best between 10000 and 50000 nodes for each cluster. With this approach we were able to solve problems on average under 1 sec. (which is considerable better than most of the other solvers in the benchmark).
- Note that the weight constraints seem to be bounded polynomial and our transformation method was particulary useful in this benchmark, since the SAT solver with a similar transformation performs much worse, see [13]. The BDD representation of the global weight constraint was about 5000 nodes.

5.4 Knapsack Problems

The knapsack problem is a typical combinatorial optimization problem. The name follows from the problem of choosing objects $i = 1, 2 \dots n$ of size w_i and value p_i to put in a knapsack of size t s.t. the sum of the values is maximized. Formally:

$$\begin{aligned} \text{maximize } & \sum_{i=1}^{n} p_i x_i \\ \text{subject to } & \sum_{i=1}^{n} w_i x_i \leq t, \qquad x_i, \quad i = 1, \dots, n \end{aligned}$$

The problem instances given in the Pseudo Boolean Evaluation have an origin in [1], and are hard integer equality constrained knapsack problems. This special class (also called subset sum problems) are decision problems, where weights w_i and values p_i are equal (here denoted by a_i), such that the PBC encoding contains two constraints

$$\begin{aligned} \sum_{i=1}^{n} a_i x_i \geq t \\ \sum_{i=1}^{n} a_i x_i \leq t \end{aligned}$$

Name	**Var**	**Solutions**	**Time** in sec	**BDD size** in 100000	**Competitors** succ./all	**BestTime** in sec
prob1	72	8.59E8	20.5	13	4/7	0.06
prob2	68	2.04E6	21.6	13	4/7	0.6
prob3	52	0	0.875	1.0	0/7	-
prob4	82	6.31E7	10.4	79	3/7	254.8
prob5	80	2.17E10	94.0	19	5/7	13.9
prob6	93	2.18E5	9.8	7.6	4/7	0.1
prob7	96	4.19E12	34.0	2.0	1/7	498.8
prob8	86	6.74E6	21.1	1.4	1/7	7.5
cuww1	60	1	0.13	0.01	6/7	0.01
cuww2	70	1	0.14	0.01	1/7	24.1
cuww3	68	2	0.13	0.01	1/7	16.5
cuww4	82	1	0.16	0.03	1/7	15.2
cuww5	86	1	0.17	0.02	1/7	36.3

Table 5.5: Results obtained from instances of the BigIntegerSatUnsat Category. The times are compared to the solution times of the best solver in the benchmark.

According to [1], the weights are chosen such that these problems become especially hard for usual branch and bound algorithms. In Table 5.5 we present our obtained results. The variable ordering was chosen decreasing wrt. the coefficients.

We state the following observations.

- The BDD approach performs well on these kind of problems. Most of the spend time is on the translation. None of the solvers attending the benchmark, seem to use our transformation, because for most of the solvers these problems are not solvable.
- With the monolithic approach, we can count the number of solutions. This is possible since model counting of BDDs is linear in the size. These model counts are of interest for e.g. heuristics and we might have found a different way to compute these number, as e.g. [26]. Our solution times for these problems are even comparable to special tools for model counting LATTE [24].
- We suspect, that our transformation algorithm has relation to the special dynamic programming algorithm for the subset sum problem, which might explain this good performance on these instances.

6 Conclusion

6.1 Summary

We have presented a transformation algorithm from PBCs to BDDs, which seems to have not yet been used in the PBC community. Furthermore, we have given a novel background understanding to this algorithm and have shown its relation to theoretical results of PBCs.

Moreover, we have implemented the first steps in our general plan of a BDD based solver and experimented with a monolithic conjunction and a search based approach. Some of the benchmark problems are solvable and the performance is competitive. However, the majority of problems in the pseudo Boolean evaluation stay unsolvable with this simple approach.

Furthermore, we have made some theoretical analysis on how to find implied variables in BDDs. This was thought to enhance the backtrack search algorithm with some form of propagation. Finding the implied variables requires a top down or bottom up traversal of a BDD and optimizations are possible in terms of caching and other techniques. However, we were not able get competitive results with the initial implementation of the propagation mechanism.

To improve propagation and implication detection, we investigated an idea on how to integrate propagation more tightly with BDDs and proposed a change in the data structure. We named this data structure UPBDDs, analyzed various ways of implementing it and showed first attempts for a conjunction on two UPBDDs. Furthermore, we were interested in the difference of size between BDDs and UPBDDs representation.

6.2 Further Work

The knowledge behind the transformation algorithm can be used for a reverse engineering approach: We can now formulate the problem to test if a BDD represents a PBC directly by using the unique bounds of the value interval.

Moreover, There is considerable work behind completing the initial plan of a BDD based solver. First, there must be a proper propagation mechanism by either of the proposed algorithms or by using UPBDDs. Then the applicability of other techniques from SAT should be investigated in context of BDDs, like conflict analysis, nogood learning and random restarts. E.g. for nogoods on BDDs see [27].

In general, our approach lacks heuristics for the various nondeterminism (variable ordering of the BDDs, order of conjunction, variable and value selection etc.). Already for a single PBC it is NP hard to determine the best order. The estimation of a good global variable ordering is necessary, algorithm like FORCE or MINCE,[2], can be transformed to work with PBCs instead of CNF. Model counting is linear in the size of a BDD and

could be a good indicator for various heuristics, and be useful in variable selection or order of conjunction. For a simple heuristic in which order to cluster, see for example [22].

Another important step is to extend the approach to solve optimization problems. This would require treating a linear function to be minimized wrt. a conjunction of PBCs. The optimization problem could be transform to a sequence of satisfiability problems through binary search wrt. objective value, e.g. [13]. We could also use the objective function to prune directly BDDs by branch and bound.

6.2.1 Hybridization

There are several attempts to combine BDDs with SAT solvers. Regarding our initial idea of having a solver completely based on BDDs, we tend now to a hybridization as well. The transformation algorithm proposed here could enhance the translation from PBCs to a CNF representation. To go a step further, we could treat the more complex PBCs as global constraints and perform self defined propagation methods on them and translate the shorter PBCs into SAT.

The initial clustering can also be used as a preprocessing step before translation to CNF. GAC on the BDD representing a PBC, can also be maintained by unit propagation on the CNF encoding from [13] and [4]. However, it does not work for general BDDs. For maintaining GAC on general BDDs, we need to add additional clauses to the CNF encoding, see [21].

6.2.2 UPBDDs

From this initial proposal for UPBDDs we can indicate several directions to work on. It is still missing a mature implementation, in order to be comparable to the BDD frameworks. For this we have to analyze in depth the essential difference between BDDs and UPBDDs and find out how to use knowledge behind the optimized BDD implementation for UPBDDs, e.g. good hashing functions. Connection between dynamic variable reordering techniques and UPBDDs should be explored, since there is a correspondence. Further, we are especially interested in transforming the BDD algorithms to work for UPBDDs without loosing their performance. In several cases UPBDDs could lead to improvements but this is still subjected to be shown.

UPBDDs could also be useful in the compilation from BDDs to CNF as a temporary step, since normally the size of UPBDDs is smaller than of BDDs. We could expect "faster" propagation due to smaller encoding, if first the BDD is translated to UPBDDs.

6.2.3 Integration of Cutting Planes

From the pigeon hole problem we know the limits of a search/resolution based approach for solving PBCs and considerable improvements could be achieve when combining the logical and mathematical paradigms. For example, we could imagine a hybrid approach, where cutting planes are integrated into BDDs in a similar manner as our implications sets, i.e. cutting planes as an implied knowledge in each BDD node. We have through our construction from a PBCs to BDDs, in each node a equivalent cutting plane (the *rest* PBC). This cutting plane can be used further. E.g. for each combination of nodes that

is visited in the the conjunction algorithm, we could construct new cuts and this might detect contradiction earlier.

Bibliography

[1] K. Aardal and A. K. Lenstra. Hard equality constrained integer knapsacks. In *Lecture Notes in Computer Science*, pages 350–366. Springer-Verlag, 2002.

[2] F. A. Aloul, I. Markov, and K. Sakallah. Force: A fast and easy-to-implement variable-ordering heuristic. In *Great Lakes Symposium on VLSI*, pages 116–119, April 2003.

[3] R. Bagnara. A reactive implementation of pos using robdds. In *PLILP '96: Proceedings of the 8th International Symposium on Programming Languages: Implementations, Logics, and Programs*, pages 107–121, London, UK, 1996. Springer-Verlag.

[4] O. Bailleux, Y. Boufkhad, L. Universit, and O. Roussel. Research note a translation of pseudo-boolean constraints to sat, 2005.

[5] M. Behle. On threshold bdds and the optimal variable ordering problem. In *COCOA*, pages 124–135, 2007.

[6] L. Bordeaux, Y. Hamadi, and L. Zhang. Propositional satisfiability and constraint programming: A comparative survey. *ACM Comput. Surv.*, 38(4), 2006.

[7] R. Bryant. Graph-based algorithms for boolean function manipulation. pages 677–691. IEEE Transactions on Computers, 1986.

[8] D. Chai and A. Kuehlmann. A fast pseudo-boolean constraint solver, 2003.

[9] K. C. K. Cheng and R. H. C. Yap. Maintaining generalized arc consistency on ad-hoc n-ary boolean constraints. In *ECAI*, pages 78–82, 2006.

[10] R. Damiano and J. Kukula. Checking satisfiability of a conjunction of bdds. In *DAC '03: Proceedings of the 40th conference on Design automation*, pages 818–823, New York, NY, USA, 2003. ACM.

[11] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[12] H. E. Dixon and M. L. Ginsberg. Inference methods for a pseudo-boolean satisfiability solver. In *AAAI/IAAI*, pages 635–640, 2002.

[13] N. Een and N. Soerensson. Translating pseudo-boolean constraints into sat. In *JSAT*, volume 2, pages 1–26, 2006.

[14] R. Gomory. An algorithm for integer solutions to linear programs. In *Recent Advances in Mathematical Programming*, pages 635–640, 1963.

[15] S. Gopalakrishnan, V. Durairaj, and P. Kalla. Integrating cnf and bdd based sat solvers. In *HLDVT '03*, page 51, Washington, DC, USA, 2003. IEEE Computer Society.

[16] P. Hawkins, V. Lagoon, and P. J. Stuckey. Solving set constraint satisfaction problems using ROBDDs. *Journal of Artificial Intelligence Research (JAIR)*, 24:109–156, 2005.

[17] P. Hawkins and P. J. Stuckey. A hybrid bdd and sat finite domain constraint solver. In *PADL*, pages 103–117, 2006.

[18] S. ichi Minato. Zero-suppressed bdds for set manipulation in combinatorial problems. In *DAC '93: Proceedings of the 30th international conference on Design automation*, pages 272–277, New York, NY, USA, 1993. ACM.

[19] S. ichi Minato, N. Ishiura, and S. Yajima. Shared binary decision diagram with attributed edges for efficient boolean function manipulation. In *DAC '90: Proceedings of the 27th ACM/IEEE conference on Design automation*, pages 52–57, New York, NY, USA, 1990. ACM.

[20] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.

[21] J. C. Jung, P. Barahona, G. Katsirelos, and T. Walsh. Two encodings of dnnf theories. In *ECAI Workshop on Inference methods based on Graphical Structures of Knowledge*, 2008.

[22] G. Katsirelos and F. Bacchus. Gac on conjunctions of constraints. In *CP '01: Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming*, pages 610–614, London, UK, 2001. Springer-Verlag.

[23] H. Kazuhisa, T. Yasuhiko, and Y. Shuzo. On the size of ordered binary decision diagrams representing threshold functions. *RIMS Kokyuroku*, 871:87–93, 1994.

[24] J. A. D. Loera, R. Hemmecke, J. Tauzer, and R. Yoshida. Effective lattice point counting in rational convex polytopes. *J. Symb. Comput.*, 38(4):1273–1302, 2004.

[25] V. M. Manquinho and O. Roussel. The first evaluation of pseudo-boolean solvers (pb05). *Journal on Satisfiability, Boolean Modeling and Computation*, 2:103–143, 2006.

[26] A. Morgado, P. J. Matos, V. M. Manquinho, and J. P. M. Silva. Counting models in integer domains. In *SAT*, pages 410–423, 2006.

[27] S. Subbarayan. Efficient reasoning for nogoods in constraint solvers with bdds. In *PADL*, pages 53–67, 2008.

[28] S. Tani, K. Hamaguchi, and S. Yajima. The complexity of the optimal variable ordering problems of shared binary decision diagrams. In *ISAAC: 4th International Symposium on Algorithms and Computation*, 1993.

[29] I. Wegener. *Branching programs and binary decision diagrams: theory and applications*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

List of Figures

List of Tables

List of Algorithms

Statement of Academic Honesty

Hereby I declare to have referenced all my sources and identified their origin in the bibliography. Moreover, I have acknowledged the people that contributed to this thesis. Apart from that, I have not made use of any ideas of others, and this thesis is to that extent a product of my own creation

Valentin Christian Johannes Kaspar Mayer-Eichberger

September 1, 2008 Lisbon