Bachelor Thesis

Abir Bouraffa

Fast SAT-Count for Labelless Complementable BDDs in BDDStab

July 25, 2014

supervised by:
Prof. Dr. Sibylle Schupp
Sven Mattsen

Hamburg University of Technology (TUHH)
Technische Universität Hamburg-Harburg
Institute for Software Systems
21073 Hamburg
# Contents

**List of Figures**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Figures</td>
<td>iii</td>
</tr>
</tbody>
</table>

1. **Introduction**

2. **Binary Decision Diagrams**
   - 2.1. Notation and Examples
   - 2.2. Ordered Binary Decision Diagrams (OBDDs)
   - 2.3. Reduced Ordered Binary Decision Diagrams (ROBDDs)
   - 2.4. Sharing of ROBDDs
   - 2.5. Complemented Edges
   - 2.6. SAT-Count for ROBDDs

3. **Labelless Binary Decision Diagrams**
   - 3.1. Labelless Nodes
   - 3.2. BDDStab
   - 3.3. SAT-Count for Labelless BDDs
   - 3.4. SAT-Count for Labelless Complementable ROBDDs
   - 3.5. Implementation

4. **Evaluation**
   - 4.1. Correctness
   - 4.2. Speed and Size Evaluation

5. **Future Work**

6. **Conclusion**

**Bibliography**

A. **Appendix**
   - A.1. Correctness Proof
   - A.2. BDD (Original Version)
   - A.3. BDDs (Variant 1)
   - A.4. BDDint (Variant 2)
List of Figures

2.1. BDD representation of \( x_1 \) ......................................................... 4
2.2. BDD representation of \( x_1 \cdot x_2 \) ..................................................... 5
2.3. BDD representation of the singleton \( \{2\} \) ................................. 6
2.4. BDD representation of \( x_1 \cdot x_2 \) ..................................................... 6
2.5. OBDD representation of \( x_1 + x_2 \) with order \( x_2 < x_1 \) ............ 7
2.6. Elimination Rule ................................................................. 8
2.7. Merging Rule ................................................................. 8
2.8. Transformation from OBDD to ROBDD of the function \( x_1 \cdot x_2 + x_3 \).... 9
2.9. Construction of shared OBDD for \( f_1 \) and \( f_2 \) .......................... 10
2.10. OBDD and CROBDD of the function \( x_1 + \overline{x_1} \cdot x_2 \) .............. 11
2.11. Pairs of equivalent sub-graphs ................................................. 12
2.12. Satisfiability count of the function \( x_1 \cdot x_2 + \overline{x_1} \cdot x_2 \) ........... 13
2.13. Satisfiability count for an ROBDD with ordering gap .................. 13
2.14. ROBDD with ordering gap at the root node .................................. 14
2.15. Example of cardinality count for an 8-Bit integer set ..................... 14

3.1. Labelless representation for OBDD of function \( x_1 \cdot x_2 \) .............. 16
3.2. Sharing of a collection of labelless OBDDs .................................... 17
3.3. Example of binary bitwise OR on ROBDDs ................................. 18
3.4. Labelless ROBDD of the function \( x_1 + x_2 \) ................................ 19
3.5. SAT-Count for ROBDD of the function \( x_1 + x_2 \) ....................... 20
3.6. SAT-Count for labelless ROBDD ............................................... 21
3.7. Labelless complemented ROBDD of the function \( x_1 + x_2 \) .......... 23

4.1. Time consumption as a function of the number of BDD samples ....... 27

5.1. Widening example on Labelless BDDs ...................................... 29
5.2. Widening example on Labelless BDDs ...................................... 30
Eidesstattliche Erklärung

Ich, Abir Bouraffa, versichere an Eides statt, dass ich die vorliegende Bachelorarbeit mit dem Titel *Fast SAT-Count for Labelless Complementable BDDs in BDDStab* selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Diese Arbeit wurde in dieser oder ähnlichen Form bisher noch keiner anderen Prüfungskommission vorgelegt.


(Unterschrift)
1. Introduction

Over the last decade, significant progress has been made in developing program analysis tools for high-level programming languages. The effectiveness of these tools opened the perspective of an application on executable binaries. The value analysis tool developed by Sven Mattsen and Arne Wichmann\(^1\), which motivated this thesis, is one of such applications. In this tool, each variable is represented by a set of its possible values. The variable operations are translated to set operations in order to verify the validity of the variable value in each state. For the following C-code example, we assume that the values of \(x\) and \(y\) are contained in the respective sets \(\{10, 40\}\) and \(\{210, 212, 34\}\).

```
unsigned char foo(unsigned char x, unsigned char y) {
    unsigned char z = x + y;
    return z & x;
}
```

At line 2, the variable \(z\) is assigned the set \(\{44, 74, 220, 222, 250, 252\} = \{10, 40\} +_o \{210, 212, 34\}\), where \(+_o\) denotes the addition on sets. Similarly, at line 3, the set containing the values of the return variable is calculated by applying the bitwise AND on the sets representing the two operands. The return variable is represented by the set \(\{8, 10, 40\} = \{44, 74, 220, 222, 250, 252\} \&_o \{10, 40\}\). The set operations require an application on every tuple generated by the cross product of the operand sets. Because the sets can extend to large sizes, it is impractical to explicitly store all the elements of every set. Instead it is possible to represent the elements as different paths of a directed acyclic graph where every path can be used more than once. Such a graph, called Binary Decision Diagram (BDD), contains all the elements of a set in a compressed form. The BDD representation enables precise arithmetic and bitwise operations in comparison to other set representations such as intervals, which usually return a superset of the result. For multiplication and division, however, the computation cost is such that these two operations are only applicable if one or both operands are relatively small. To be able to perform both operations on large BDDs nonetheless, we resort to reducing their size with the consequence that the set they represent becomes larger. This corresponds to a tolerable loss in precision for the result. The precision gap can be attenuated by choosing a criterion on the basis of which to decide whether these reductions are indeed necessary. In this thesis, the decision is made based on an implementation of the cardinality count for integer sets. In fact by knowing the number of elements a set contains, we hope to avoid supersetting relatively small sets thereby preventing unnecessary imprecisions of the subsequent calculation. Furthermore, we carry out the cardinality count as a single operation with a constant cost \(O(1)\), thereby preventing a slowdown of the operations using it. In the next chapter, we present the standard BDD structure first introduced by Bryant\(^2\) and detail the methods commonly in use today to increase its efficiency. Chapter 2 also describes the basic algorithm used to determine the satisfiability count for

\(^1\) http://www.sts.tuhh.de/research/bddstab.html
standard BDDs. Chapter 3 introduces a customized BDD structure involving labelless nodes and addresses its implementation in BDDStab. The satisfiability count of labelless BDDs, constituting the actual contribution of this work, is outlined in this chapter alongside an elaborate description of its implementation. The implementation is subsequently tested for correctness and evaluated in association with the time and space performance of the BDD data structure in Chapter 4. We conclude this work by a short insight into a future application on the widening operation for BDDs in Chapter 5, followed by a concise summary of the milestones of this thesis in Chapter 6.
2. Binary Decision Diagrams

In this chapter, we attempt to familiarize the reader with the original data structure of binary decision diagrams. BDDs are a relatively new representation of boolean functions. They offer an efficient alternative to truth tables and Karnaugh diagrams. BDDs are regarded as an improvement to binary decision trees (BDTs), which are based on a graphical description of the Shannon expansion formula. The Shannon expansion formula states that a boolean function \( f(x_1, ..., x_n) \) can be represented as \( \overline{x_n}g + x_nh \) where \( g(x_1, ..., x_{n-1}) = f(x_1, ..., x_{n-1}, 0) \) and \( h(x_1, ..., x_{n-1}) = f(x_1, ..., x_{n-1}, 1) \). The binary decision tree (BDT) for this function is the result of a recursive application of the Shannon expansion where each function is represented by a node with edges leading to its subfunctions. As recognized by several studies, this basic form of boolean functions can contain redundancies and is therefore open to optimization. Among these studies, the work of Bryant has gained due distinction for introducing manipulation algorithms and restrictions resulting in a canonical form known as Reduced Ordered Binary Decision Diagrams (ROBDDs). This in turn simplified otherwise expensive operations such as testing for equality or satisfiability of boolean functions. Nowadays, the term BDD usually refers to ROBDDs, which we deal with in the third section of this chapter.

2.1. Notation and Examples

A binary decision diagram consists of nodes and directed edges arranged to form an acyclic connected graph with one root node. We differentiate between terminal nodes labeled 0 or 1, and decision nodes representing variables. Each decision node has exactly two outgoing edges: one true-edge and one false-edge.

![Figure 2.1.: BDD representation of \( x_1 \)](image)

Figure 2.1. illustrates a simple boolean function that depends only on the variable \( x_1 \). In this figure, non-terminal nodes are represented by circles labeled by a variable. The true-edge of a non-terminal node is depicted as a solid edge whereas the dashed edges illustrate false-edges. The terminal nodes are represented by squares containing their value. This convention is maintained for all the figures of this thesis. For the function in Figure 2.1, the assignment of the value true to \( x_1 \) is represented by the true-edge
and yields the sub-function 1 or true. Analogously, the assignment of the value false is represented by the false-edge and yields the boolean sub-function 0 or false.

More generally, given a BDD corresponding to a boolean function $f$ with input variables $x_1, ..., x_n$ and an assignment $(0, 1, ..., 0)$ of the variables, we can determine the output by traversing the BDD starting from the root node as follows:

1. For a decision node with label $x_i$, if the assignment of $x_i$ is true, then proceed to the node under the true-edge. Otherwise to the false-edge.

2. For a terminal with label 0, the output corresponding to the given variable assignment is considered false. If, however, the label of the terminal node is 1, then the output is true.

Figure 2.2 shows a possible BDD-representation of the boolean function $f = \overline{x_1} \cdot x_2$. From the two rules stated above, we ascertain that the paths leading to a sink node 1 are assignments that satisfy the function (models) and consequently all paths leading to the sink node 0 are assignments having false for output. The BDD of the function $f$ may also be regarded as a set of integers. In fact, $x_1$ and $x_2$, being boolean variables, can represent the single bits of an integer. In general, the elements of an integer set of type $T$ can be transformed into a set of Boolean tuples by means of a mapping $t: T \rightarrow \mathbb{B}^n$ where $n$ is the size of the integer type $T$. We are then able to define the indicator function $f_s: \mathbb{B}^n \rightarrow \mathbb{B}$ of the resulting set $S$. The function $f_s$ takes a tuple of Booleans for input and returns true if it is contained in the set $S$, false otherwise. The indicator function is a boolean function and can therefore be represented by a BDD. Consequently, the composition of the functions $t$ and $f_s$ allows us to obtain a BDD representation of an integer set of a certain type $T$. For BDDs representing integer sets, it holds that all variable assignments leading to the sink node 1 represent integers contained in the set and the contrary holds for variable assignments with paths leading to the sink node 0. By establishing the correspondence $x_1 \rightarrow \text{least significant bit}$ and $x_2 \rightarrow \text{first bit}$ for 2-bit integers, we are now able to reconstruct the set of integers $\{(1, 0)\} \equiv \{2\}$ from the previous example. Figure 2.3 recaptures this new interpretation in detail. In this figure, the variables are replaced with the corresponding integer bit. However, for the remainder of this thesis and for the sake of coherence, we continue to use the original notation of the nodes with variables as labels.
2. Binary Decision Diagrams

The BDD structure in its simplest form, which we outline in this section has several limitations. The most notable limitation is that a single boolean function can have multiple BDD representations. This adds to the complexity of basic operations such as testing for equivalence. The example from Figure 2.2 has another possible BDD representation proposed in Figure 2.4. A closer look at this BDD shows that the node labeled $x_2$ whose two outgoing edges point to the same terminal node can be omitted without any implications on the semantic of the BDD. In fact, if we choose an ordering in which to arrange the nodes in a top-down manner based on the variables they represent, we are able to reconstruct the value of the omitted nodes. In the previous example, the variables follow the order $x_1 < x_2$. Then, we can assume that both values, 0 and 1, of the no longer existing node $x_2$ lead to the same terminal 0 thereby coming to the same output as the Figure 2.2. The transition between the two BDDs is known as reduction.

Both, ordering and reduction, which we briefly mention here, are the two decisive aspects that helped achieve the canonical form proposed by Bryant, which we explain in more details in the subsequent sections.

2.2. Ordered Binary Decision Diagrams (OBDDs)

In this section, we introduce the representation data structure of Ordered Binary Decision Diagrams, called OBBDs for short. They result from the application of ordering restrictions proposed by Bryant on the already existent binary decision diagrams (then
2. Binary Decision Diagrams

called branching programs) developed by Lee and Ackers [2, 3].

Figure 2.5.: OBDD representation of \( x_1 + x_2 \) with order \( x_2 < x_1 \)

OBDDs and BDDs share the same notation and graphical representation with the exception of one additional property for OBDDs. Each OBDD is associated with a total order \( \pi \) on the set of variables \( \{x_1, ..., x_n\} \) featured in the corresponding boolean function. This order is maintained by the nodes along each path in the OBDD, i.e., for each edge leading from a node labeled \( x_i \) to a node labeled \( x_j \) it holds that \( x_i \prec_\pi x_j \) [4]. Figure 2.5 presents an OBDD of the function \( x_1 + x_2 \) with respect to the variable order \( x_2 < x_1 \). The horizontal dashed lines are drawn to delimit the different layers corresponding to variables in the ordering. The decision nodes included in a certain layer have the same variable for label [5].

OBDDs are therefore important because they constitute a first step toward a standard, more compact data structure that can be manipulated efficiently. The second, not less important step is the reduction of OBDDs resulting in what is known as Reduced Ordered Binary Decision Diagrams or ROBDDs for short.

2.3. Reduced Ordered Binary Decision Diagrams (ROBDDs)

In this section, we introduce the reduction algorithm published by Bryant in order to minimize the size of OBDDs. This algorithm consists of two basic rules aimed at avoiding redundancies in the OBDD data structure. An exhaustive application of these two rules results in a reduced ordered binary decision diagram (ROBDD) with the property that a boolean function has a unique ROBDD representation for a certain chosen ordering. This property, called canonicity, helps simplify complex operations such as the check for equality of two boolean functions. To specify the reduction rules used in ROBDDs, the following definition is necessary. For this definition, we denote the true-edge successor of an internal node \( u \) by \( \text{high}(u) \) and the false-edge successor by \( \text{low}(u) \). We also make use of \( \text{var}(u) \) to denote the variable label.

**Definition**  Let \( B_1 \) and \( B_2 \) be two OBDDs. \( B_1 \) and \( B_2 \) are called isomorphic iff there exists a bijective mapping \( \phi \) from the set of nodes of \( B_1 \) to the set of nodes of \( B_2 \) such that for each node \( u \) either:

1. the two nodes \( v \) and \( \phi(v) \) are sinks having the same label, or
2. Binary Decision Diagrams

2. \( \text{var}(v) = \text{var}(\phi(v)), \, \phi(\text{high}(v)) = \text{high}(\phi(v)), \, \phi(\text{low}(v)) = \text{low}(\phi(v)) \) \[4\]

Taking into account the previous definition of isomorphic OBDDs, we now formulate the reduction rules:

1. **Elimination Rule** If both, the true-edge and the false-edge of an internal node \( u \) point to the same sub-node \( v \), then eliminate \( u \) and redirect the incoming edges toward the node \( v \).

2. **Merging Rule** If an OBDD presents two isomorphic sub-OBDDs, starting respectively at the nodes \( u \) and \( v \), then eliminate one of the two sub-OBDDs, say the one with root node \( u \), and redirect the incoming edges of \( u \) to the node \( v \). \[4\]

---

The figures 2.6 and 2.7, respectively, illustrate a single example of the elimination rule and the merging rule. The first reduction rule eliminates all but one sink node labeled 1, since all 1-sinks are by definition isomorphic. This holds also for 0-sinks. The resulting graph contains internal nodes whose two outgoing edges point to the same sink. This is the case for the node labeled \( x_2 \) of Figure 2.6, here colored in red. The application of one of the reduction rules can render the other rule applicable. In Figure 2.7, the merging
of the two isomorphic sub-OBDDs (colored in red) makes both higher nodes labeled $x_2$ redundant as stated by the elimination rule. For this reason, the algorithm proceeds repetitively (recursively) until none of the reduction conditions hold. Figure 2.8 shows the result of an exhaustive application of the reduction rules on an OBDD representing the function $x_1 \cdot x_2 + x_3$.

![Figure 2.8: Transformation from OBDD to ROBDD of the function $x_1 \cdot x_2 + x_3$.](image)

In the previous example, the OBDD was drastically reduced from 15 nodes to only 5 nodes. This size minimization is one aspect of reduction. Another, more interesting aspect, especially relevant for the complexity of the operations we intend to apply on OBDDs, is canonicity. Indeed, for a certain known ordering of the variables, a boolean function has a unique ROBDD representation. This property allows us to test two boolean functions for equivalence by testing whether the corresponding ROBDDs match exactly instead of expensively checking for isomorphism.

### 2.4. Sharing of ROBDDs

So far, the focus has been on the structural properties of ROBDDs and reduction has been the only scheme used in order to limit their memory consumption. In general, this measure alone is not enough to meet the high demands on the performance of OBDD-operations. BDDStab, on which this study is based, is one example where these expectations are justifiable as it represents every code-variable with a set, i.e., an ROBDD.

A variety of ROBDD packages propose solutions to implement ROBDDs more efficiently. The most notable of which is the BDD package developed by Brace et. al. [6] where they introduce the concept of shared BDDs. In fact, identical sub-graphs can occur in several different ROBDDs. A shared ROBDD is a directed acyclic graph where only one instance of these sub-graphs is represented. To differentiate the ROBDDs (functions), an additional root node is implemented with a single edge pointing to the root node. In other terms, a shared ROBDD is a collection of ROBDDs having one or more identical sub-graphs. Figure 2.9 illustrates an example of a shared ROBDD resulting from the merging of the identical sub-graphs (in red) of the given ROBDDs.
Shared ROBDDs present an improvement in space consumption since every sub-function (sub-graph) is uniquely represented. A major advantage of this approach is the strong canonical form of the ROBDDs. This term conveys that two equivalent functions \( f \) and \( g \) not only have the same ROBDD representation, but also that in an implementation, they are represented by a pointer to exactly the same memory cell [4]. Checking two functions for equivalence is therefore a single operation, namely a pointer comparison. Concretely, canonicity is maintained through a redundancy check every time a new node is to be added to the shared ROBDD. For the new node, the label \( x_i \) as well as the right successor and the left successor nodes are specified. It is then tested whether a node with the same characteristics is available in the shared ROBDD. If so, no new node will be created and the already existing one is used. This test is performed by a unique table most commonly implemented as a hash table following the standard set by Brace et al. in their 1991 BDD package [?] in their 1991 BDD package [?]. In the unique table, a node \( v = (x_i, \text{high, low}) \) is mapped to a position \( h(x_i, \text{high, low}) \) using the hash function \( h \). At this position, the node in question is stored. This process does not rule out collisions of the hash values, which necessitates the choice of a well-suited hashing function to ensure a fair distribution. Examples of these functions include linear hashing suitable for dynamic hashing tables and Cuckoo hashing for fixed-size hashing tables.

ROBDDs are often used with another optimization consisting in an additional attribute to each edge known as complement bit. The resulting ROBDD is said to integrate complement edges and stores both, the function it represents and its complement function. This hybrid data structure is thoroughly explained in the next section.
2. Binary Decision Diagrams

2.5. Complemented Edges

ROBDDs hold an important property that simplifies the negation of the boolean function they represent. In fact, merely interchanging the values in the sink nodes of an ROBDD with a function $f$ results in an ROBDD representing its complement $\overline{f}$. This property was recognized by the early studies on BDDs [2] and was exploited to obtain another variant of ROBDDs known as Complementable Reduced Ordered Binary Decision Diagrams (CROBDD). The complement bit of a CROBDD edge indicates whether the underlying sub-graph is to be interpreted as the complement $\overline{f}$ or the ordinary sub-function $f$. Using this method, a function and its complement can be stored within the same graph. To achieve this, the complement function is represented by an additional complemented edge to the root of the original function. Understandably, this results in a decrease of the number of nodes, which represents a second advantage of CROBDDs.

Figure 2.10 shows a CROBDD of the function $x_1 + \overline{x_2}$. A small circle on an edge indicates that the edge is complemented. The 0-sink node no longer needs to be represented since it is equivalent to a complemented edge pointing to the 1-sink. A comparison of the node counts reveals that for the same amount of nodes, a CROBDD can store both the original boolean function and its complement function.

In general, both true-edges and false-edges may be complemented. This creates an issue concerning the canonicity of the representation. Figure 2.11 shows pairs of equivalent sub-graphs resulting from switching the complement bits between the different edges. To restore canonicity, a single representation is chosen from each pair. The left member from each pair in Figure 2.11 is the standard representation. It is maintained by disallowing the 1-edge of the ROBDD to be complemented. Upon the generation of a new ROBDD, the edge attributes are chosen so that the condition is always satisfied.

The CROBDD data structure described so far has been present in several early studies relating to BDDs. A formal proof of equivalence for CROBDDs and ROBDDs is given by Madre and Billion affecting circuit correctness represented by BDDs [1]. Introducing complement edges to the ROBDD data structure has helped improve the time and space.
2. Binary Decision Diagrams

2.6. SAT-Count for ROBDDs

Some of the operations on ROBDDs can only be applied if one or both operands have a restricted size due to the exponential size growth of the output ROBDD. To be able to apply these operations on a wider range of operands, a reduction is performed on the input ROBDDs. For ROBDDs representing integer sets, this reduction corresponds to a supersetting and thus a loss of precision for the subsequent operations. So far, this reduction has been performed arbitrarily using the same scheme for all ROBDDs, which has a negative effect on the precision of the results. A way to optimize reduction is to implement an alternative selective scheme, based on the number of elements the set represented by a sub-ROBDD encloses. The number of elements inside a set corresponds to the number of satisfying assignments the indicator function has. When these functions are represented in an ROBDD, the calculation is done by recursively traversing the ROBDD bottom-up. For the correctness of this approach, it is necessary to define the number of models at the sink 1 to be equal to one and the number of models at the sink 0 to be zero. Figure 2.12 shows an example of satisfiability count on an ROBDD with the ordering $x_1 < x_2$. The counting algorithm starts at the lowest level of the ROBDD where the values at the sinks are given by definition. In this example, the number of elements in every internal node is trivially given by the sum of elements in both its sub-nodes. Thus, at the $x_2$-layer, the number of models in the respective nodes is the sum of the number of models in the successors i.e. $1 = 1 + 0$. This follows also for the layer containing the root node where we counted $2 = 1 + 1$.

However, due to reduction, an ordering gap can exist between a node and its successors, this case is not covered by the previous example. When the ROBDD structure presents...
such gaps, an adjustment is necessary to take into account the value of the eliminated nodes, assuming the ordering is well-known. Figure 2.13 depicts an ROBDD following the same ordering $x_1 < x_2$ with an order gap between the node $x_1$ and the sink node 1. In this case the first reduction rule is responsible of eliminating a redundant node $x_2$ at the true-edge of the root node. The absence of an $x_2$-representative at this position creates the previously mentioned order gap. Nevertheless, we are able to correctly count the number of models at the root node. For this, we need to define the ordering index at the sink nodes 1 and 0 to be the highest, i.e., for an ordering of $n$ variables $x_1, x_2, ..., x_{n-1}, x_n$, the sink nodes would have the index $n + 1$ in the ordering. Consequently, the sink node 1 of Figure 2.11 has the ordering index 3. By multiplying the number of models at the sink node 1 by the number of models in the eliminated node, we obtain the correct model count at the root node. The number of models at the missing node is given by $2$ to the power of the difference of ordering between the 1-sink node and the root node.

Figure 2.13: Satisfiability count for an ROBDD with ordering gap

Furthermore, ordering gaps can occur at the root. This is the case in Figure 2.14 with the same ordering as in the previous examples. After the satisfiability count is performed on the ROBDD, if the index of the root node is not the smallest of the ordering, the result is adjusted by the factor of 2 to the power of the difference between the smallest index and the index of the root node with respect to the chosen ordering. As before, this factor represents the nodes eliminated by reduction.

More generally, the satisfiability count of an ROBDD representing a function $f$ with $n$ variables $x_1, ..., x_n$ having each an index inside the set $1, ..., n$ with respect to an ordering
\( \pi \) is inductively given by the following definitions.

For each node \( u \) in the ROBDD:

\[
\text{count}(u) = \begin{cases} 
0, & \text{if } u \text{ is a 0-sink} \\
1, & \text{if } u \text{ is a 1-sink} \\
2^{\text{index}(\text{low}(u)) - \text{index}(u) - 1} \times \text{count}(\text{low}(u)) + 2^{\text{index}(\text{high}(u)) - \text{index}(u) - 1} \times \text{count}(\text{high}(u)) & \text{if } u \text{ is an internal node}
\end{cases}
\]

Where \( \text{index}(u) \) is the index of the variable the node \( u \) represents with respect to the ordering \( \pi \) and \( \text{high}(u) \) and \( \text{low}(u) \) respectively represent the true-edge successor and the false-edge successor of \( u \).

Additionally, let \( v \) be the root node of the ROBDD, then the number of models of the function \( f \) is given by:

\[
\text{SATcount}(f) = 2^{\text{index}(v) - 1} \times \text{count}(v)
\]

A recursive SAT-Count algorithm for ROBDDs, on which these definitions are based, is formulated by R.H.Andersen \cite{Andersen1979}. As already mentioned in the beginning of this section, this algorithm is also used to determine the number of elements inside an integer set represented by an ROBDD. Figure 2.15 reconstructs the steps of the algorithm throughout the different nodes of an ROBDD representing a set of 8-Bit integers. We assume that the ROBDD follows the ordering \( x_1 < x_2 < \ldots < x_7 < x_8 \) and that a correspondence
between the variables and the bits is chosen in advance. It is important to note that this last detail has no influence on the result of the count. In fact, the algorithm only depends on the structure of the ROBDD and the ordering in which the variables occur.

The SAT-Count or Cardinality-Count algorithm is used to optimize other operations based on the additional information that it delivers. However, for large ROBDDs, it is often depreciated because it becomes time consuming. One way to overcome this side effect is to accept a tradeoff between time and space consumption by storing the result of the algorithm directly inside every new node. By doing so, the satisfiability count becomes a single operation. We make use of this method to implement the cardinality count in BDDStab. For this, we adapted the SAT-Count algorithm to the data structure in BDDStab, which differs from standard ROBDD by omitting the labeling of the nodes. We explain the differences between the two data structures in more detail in the next chapter.
3. Labelless Binary Decision Diagrams

In the last chapter, we have dealt with the standard, most commonly used form of BDDs. So far, we have used a combination of inter-graph sharing and reduction to limit the space complexity of BDDs. In this chapter, we continue using inter-graph sharing on a slightly different version of OBDDs with a modified approach for reduction. Converse to the reduction mechanism explained in Section 2.3, this new method compares sub-graphs for structural but not necessarily semantic equivalence. In the next section, we introduce the announced data structure and explain its specificity in more detail. We then elucidate the implementation of labelless BDDs in BDDStab in section 3.2 and tackle our main topic of interest, which is the SAT-Count algorithm for labelless BDDs, over the sections 3.3, 3.4 and 3.5 where we provide a proof of correctness for our chosen implementation.

3.1. Labelless Nodes

In this section, we present a variant of the OBDD data structure encountered in Section 2.2. This new variant foregoes the use of node labeling. To maintain a correct interpretation of the nodes, the chosen ordering is stored at a node with an outgoing edge to the root of the labelless BDD. For this representation, the elimination rule is only applicable to internal nodes whose two outgoing edges point to the same sink. On the other hand, the merging rule can be applied without restrictions. The most notable advantage of this modified data structure is the possibility to merge structurally equivalent sub-graphs which do not necessarily share the same semantic. Weakening the reduction rules gives labelless ROBDDs yet another property not encountered for standard ROBDDs. In fact, by avoiding the elimination rule between internal nodes, ordering gaps no longer exist between these nodes. This property is put to use in the next section to determine the satisfiability count of labelless BDDs. An example of the new representation is shown in Figure 3.1.

![Figure 3.1: Labelless representation for OBDD of function $x_1 \cdot x_2$](image-url)
Compared to ROBDDs, the labelless OBDD representation has, in most cases, a greater node count. However, it has an advantage over the standard data structure when used alongside BDD-sharing. In fact, sub-graphs which are structurally but not semantically equivalent occur more frequently in shared BDDs and their number is more likely to increase with the number of BDDs that are shared \[5\]. Figure 3.2 illustrates the construction of a shared BDD from the collection of labelless OBDDs. By merging structurally equivalent sub-graphs, here colored in green and red, we are able to construct a shared labelless BDD containing 7 nodes fewer. Using ROBDDs, the shared BDD would consist of three disjoint ROBDDs since they do not share any semantically identical sub-graphs, thus not providing an improvement in space complexity.

![Diagram](image)

Figure 3.2.: Sharing of a collection of labelless OBDDs

BDDStab makes use of the labelless BDD structure with a few changes that are mostly aimed at simplifying the implementation of certain operations.

### 3.2. BDDStab

As mentioned earlier in this thesis, BDDStab uses shared, labelless ROBDDs to represent integer sets. However, instead of allowing different orderings of the ROBDDs and storing the order in a higher node, BDDStab uses an implicit order from most significant to least significant bit for all ROBDDs. This convention is mainly applied to reduce the complexity of the operations by ensuring the operands have the same ordering. It also implements complement edges to enable complementation in constant time. Furthermore, inter-graph sharing is used by BDDStab to optimize the memory consumption. BDDStab provides a set of operations on integer sets which we can categorize into symmetric and non-symmetric operations \[5\]. The result of a symmetric operation is calculated by recursively calling the operation with same-type sub-graphs from both operands. For example, if \(r\) is the node resulting from the application of a ROBDD operation \(\circ\) on the nodes \(u\) and \(v\) then \(\text{high}(r) = \text{high}(u) \circ \text{high}(v)\) and analogously \(\text{low}(r) = \text{low}(u) \circ \text{low}(v)\).

Union and intersection on sets, provided by BDDStab, are examples of symmetric
3. Labelless Binary Decision Diagrams

operations. They are translated respectively into binary disjunction and conjunction of BDDs on the implementation level [5].

Non-symmetric operations, on the other hand, require an application on each pair generated by the cross-product of the true-edge and false-edge sub-graphs from both operands. In fact, in the case of symmetric operations two of these pairs are redundant since the boolean function corresponding to the operation is commutative. Binary bitwise operations are examples of non-symmetric operations implemented in BDDStab. Figure 3.3 shows a simplified step of the bitwise OR operation. The corresponding algorithm generates tuple lists of ROBDDs. The elements of these lists are then distributed either to the true-edge successor or to the false-edge successor of the result node based on the application of the OR operation. BDDStab also implements the addition operation. In fact the result sum, and the overflow are calculated in separate ROBDDs and eventually merged by the union operation. BDDStab's operations reach beyond the scope of this thesis and will therefore not be discussed in more details.

3.3. SAT-Count for Labelless BDDs

In Section 2.6 we presented the SAT-Count algorithm for standard ROBDDs. In this section, we adapt the algorithm to the labelless BDD representation of which we make use in BDDStab. SAT-Count, in its original form, depends heavily on the node label to determine the number of models the corresponding sub-function has. This constitutes the main difficulty since the new data structure does not use labels. The main advantage of labelless BDDs is that nodes can be used by distinct BDDs at different levels and represent more than one variable, provided inter-graph sharing is applied. To determine the satisfiability count of this BDD-representation, we therefore make use of a node characteristic that does not depend on its interpretation, namely the node depth. More precisely, we have come to observe that the depth of a node is the same throughout the different BDDs sharing this node, provided the depth is defined in a bottom-up manner, i.e., if the terminal nodes are assigned the least depth value. We also noted that the ordering property announced in Section 3.1 suggests a correlation between the satisfiability count and the lengths of the paths from the root node to a terminal, which in turn can be expressed in terms of the difference in depth.

To clarify matters further, we consider the example from Figure 3.4, illustrating a labelless ROBDD of the function \( f(x_1, x_2, x_3) = x_1 + x_2 \). As is the case in the original
SAT-Count algorithm, the satisfiability count at the true terminal node is defined to be 1 and that of the false terminal node is defined as 0. For internal nodes with both edges pointing to terminal nodes, the count is trivially given by the sum of the successors’ count. However, for internal nodes with at least one internal successor node, the trivial calculation is no longer correct. In fact, it is possible that the successors of a node do not have the same depth. The depth difference corresponds to nodes that have been omitted by the modified elimination rule (Section 3.1). Figure 3.5 shows variants of the labelless BDD example before and after reduction, emphasizing the elimination of the redundant node. The reduction of this node creates an ambiguity concerning the count at the root of the reduced BDD. To resolve this problem for labelless ROBDDs, we calculate the depth of the successor nodes and use the depth difference to adjust the calculation of the satisfiability count. The depth of a node is given by the following inductive definitions.

The variables \( h \) and \( l \) respectively represent the true-edge ROBDD successor and the false-edge ROBDD successor of the node \( \text{Node}(h, l) \). We also denote the true sink and the false sink by \( T \) and \( F \) respectively:

\[
\begin{align*}
    d(T) &= d(F) = 0 \\
    d(\text{Node}(h, l)) &= \max\{d(h), d(l)\} + 1 
\end{align*}
\]  

(3.1)

The node depth is defined in an increasing order starting from the terminal nodes with depth 0 up to the root with the highest depth in the ROBDD. To adjust the previous calculation, the count at the successor node with lesser depth is multiplied by a factor depending on the depth difference. The satisfiability count at a certain node is then
defined as follows:
\[
e(F) = 0
\]
\[
e(T) = 1
\]
\[
e(\text{Node}(h, l)) = \begin{cases} 
2^{d(h) - d(l)} \cdot e(l) + e(h), & \text{if } d(h) \geq d(l) \\
2^{d(l) - d(h)} \cdot e(h) + e(l), & \text{oth.}
\end{cases}
\]

(3.2)

The depth and count quantities of each node are formulated independent of the ordering chosen for the given BDD and constitute therefore two invariant node attributes. This property allows us to cache both values for each node. Consequently, the count operation for a new node is reduced to a single operation. The involved time and space complexity of caching is discussed in further details in Section 3.5.

Figure 3.5 revisits the previous example and specifies both the depth (blue) and the count (red) at each node. The count at the root node is given by \( 3 = 1 \cdot 2^{1-0} + 1 \)
3. Labelless Binary Decision Diagrams

following the previous definition. However this value does not correspond to the number of satisfying assignments of the function \( f \) with input variables \( x_1, x_2, x_3 \). Indeed, the number of models a boolean function has does not always coincide with the count at the root node of the corresponding ROBDD. In fact, it may occur that the depth at the root node is less than the number of variables the function requires, which is yet another side effect of reduction. In the previous example and according to the chosen variable ordering, the root node represents the variable \( x_1 \) and its false-edge successor the variable \( x_2 \). However all nodes representing \( x_3 \) have been eliminated by reduction since both variable assignments of \( x_3 \) lead to the same output. To take into account the assignments of such variables, an additional step is needed in the SAT-Count algorithm. The final result of the satisfiability count depends not only on the count \( e(r) \) at the root node \( r \) but also on the number of input variables \( n \) of the corresponding function and is given by the following term:

\[
\text{models}(n, r) = 2^{n-d(r)} \cdot e(r)
\]  

(3.3)

For the example of Figure 3.5 of the function \( x_1 + x_2 \) with three input variables the satisfiability count is then given by \( 2^{3-2} \cdot 3 = 6 \).

Figure 3.6.: SAT-Count for labelless ROBDD

The different steps of the SAT-Count algorithm can be formulated within a single operation. This operation takes a labelless ROBDD representing a boolean function \( f \) as well as the number of input variables \( n \) the function takes and recursively computes the number of satisfying assignments of \( f \). The operation is defined as follows:

\[
\begin{align*}
\text{models}_R(n, T) &= 2^n \\
\text{models}_R(n, F) &= 0 \\
\text{models}_R(n, \text{Node}(h, l)) &= \text{models}_R(n - 1, h) + \text{models}_R(n - 1, l)
\end{align*}
\]  

(3.4)

To verify that \( \text{models}_R \) indeed computes the satisfiability count for labelless BDDs, the different steps of the SAT-Count algorithm described so far must be fulfilled by the operation. This comes down to proving that the operation is equivalent to the one in 3.3 when the inductive definitions of depth and count at the single nodes are given respectively by the definitions in 3.1 and 3.2. The proof is performed inductively on the ROBDD structure where terminal nodes are considered basic and internal nodes are defined as the association of two sub-nodes. The induction holds for the following base cases:

\[
\forall n : \text{models}(n, T) = 2^n = \text{models}_R(n, T) \\
\forall n : \text{models}(n, F) = 0 = \text{models}_R(n, F)
\]

For the inductive case, we assume that the two subsequent statements hold for the two
Sub-ROBDDs \( r_1 \) and \( r_2 \):

\[
\forall n : \text{models}(n, r_1) = \text{modelsR}(n, r_1) \\
\forall n : \text{models}(n, r_2) = \text{modelsR}(n, r_2)
\]

Then:

\[
\forall n : \text{modelsR}(n, \text{Node}(r_1, r_2)) = \text{modelsR}(n - 1, r_1) + \text{modelsR}(n - 1, r_2) \\
= 2^{n - d(r_1) - 1} \cdot e(r_1) + 2^{n - d(r_2) - 1} \cdot e(r_2)
\]

For depths of the sub-ROBDDs \( r_1 \) and \( r_2 \), we distinguish between two cases. First case \( d(r_1) \geq d(r_2) \):

\[
\text{modelsR}(n, \text{Node}(r_1, r_2)) = 2^{n - d(r_1) - 1} \cdot (2^{d(r_1) - d(r_2)} \cdot e(r_2) + e(r_1)) \\
= \frac{2^{n - d(r_1) - 1}}{2^{n - d(\text{Node}(r_1, r_2))}} \cdot (e(r_1) + 2^{d(r_1) - d(r_2)} \cdot e(r_2))
\]

Second case \( d(r_1) < d(r_2) \):

\[
\text{modelsR}(n, \text{Node}(r_1, r_2)) = 2^{n - d(r_2) - 1} \cdot (2^{d(r_2) - d(r_1)} \cdot e(r_1) + e(r_2)) \\
= \frac{2^{n - d(r_2) - 1}}{2^{n - d(\text{Node}(r_1, r_2))}} \cdot (e(r_2) + 2^{d(r_2) - d(r_1)} \cdot e(r_1))
\]

\( \square \)

### 3.4. SAT-Count for Labelless Complementable ROBDDs

The SAT-Count operation defined in the last section can be adapted to labelless ROBDDs with the complement edge extension. As mentioned in Section 2.5 and to preserve canonicity, only the false-edges of an ROBDD may be complemented. If the complement bit of a false-edge is set, the corresponding sub-ROBDD is interpreted as the complement of the original function it represents. Consequently, the satisfiability count of this sub-ROBDD is the number of models the complement function has. If the incoming edge to a sub-ROBDD is not complemented, the satisfiability count can be calculated by the operation defined in the last section. Consider the example of Figure 3.7, illustrating the function \( f(x_1, x_2, x_3) = x_1 + x_2 \). The false-edge successor node of the root represents the sub-function \( f_1(x_2, x_3) = x_2 \) with 2 satisfying assignments. However, since the incoming false-edge is complemented, the node is interpreted as the complement function \( \overline{f_1}(x_2, x_3) = \overline{x_2} \). The count at the root node depends, therefore, on the number of models the complement function has, which is given by \( 2^4 - 2 = 2 \).

The satisfiability count at the nodes is modified to take into account the complement edge extension. We continue using the depth operation from the last chapter since the negation of sub-functions does not affect the depth of their node representation. For the count operation at the nodes, we revise the definition from the last chapter by...
3. Labelless Binary Decision Diagrams

![Diagram](Image)

Figure 3.7.: Labelless complemented ROBDD of the function $x_1 + x_2$

distinguishing between two cases based on the value of the false-edge’s complement bit. For the modified count operation, we change the notation of the nodes to include the value $c$ of the false-edge complement bit:

$$e(F, c) = \begin{cases} 0, & \text{if } c = 0 \\ 1, & \text{oth.} \end{cases}$$

$$e(T, c) = \begin{cases} 1, & \text{if } c = 0 \\ 0, & \text{oth.} \end{cases}$$

$$e(\text{Node}(h, l, c)) = \begin{cases} 2^{d(l)-d(h)} \cdot e(h) + (2^{d(l)} - e(l)), & \text{if } c = 1 \\ 2^{d(l)-d(h)} \cdot e(h) + e(l), & \text{oth.} \end{cases}$$

if $d(l) \geq d(h)$

$$e(\text{Node}(h, l, c)) = \begin{cases} 2^{d(h)-d(l)} \cdot (2^{d(l)} - e(l)) + e(h), & \text{if } c = 1 \\ 2^{d(h)-d(l)} \cdot e(l) + e(h), & \text{oth.} \end{cases}$$

if $d(h) > d(l)$

(3.5)

Using the previous definition, we are able to formulate the satisfiability count operation of boolean functions represented by labelless complemented ROBDDs as follows:

$$\text{modelsR}(n, (T, 0)) = 2^n$$

$$\text{modelsR}(n, (T, 1)) = 0$$

$$\text{modelsR}(n, \text{Node}(h, l, c)) = \begin{cases} \text{modelsR}(n - 1, h) + \text{modelsR}(n - 1, l), & \text{if } c = 0 \\ \text{modelsR}(n - 1, h) + 2^{n-1} - \text{modelsR}(n - 1, l), & \text{oth.} \end{cases}$$

(3.6)
The correctness of the previous definition is also verified by proving its equivalence with the operation in 3.3 holding for the definitions of the depth and the count at the nodes adapted to labelless complementable BDDs. The proof is detailed in section A.1 of the enclosed appendix.

3.5. Implementation

The satisfiability count operations for labelless ROBDDs dealt with so far are generally implemented as a recursive function that requires the full traversal of the data structure. However, the ROBDD representations generated by BDDStab can take large sizes. This is the case of ROBDDs representing 32 and 64-bit integers used in program analysis tools. Therefore, in our implementation of SAT-Count, we opted for a trade-off between space and time complexity by adding new node attributes representing the count as well as the depth. This enables us to reduce the satisfiability count to a single operation.

The challenge remains to find a suitable data structure to store the new node information. In fact, BDDStab offers ROBDD representations for integer sets of an arbitrary size. This suggests the use of dynamic data structures for the depth and count, which can build a considerable overhead. For our case study, we concentrate on the use of BDDStab for the representation of 32-bit integers. We therefore use a 32-bit integer to represent the count attribute while using the maximum number of elements at the given depth to deal with overflows. However, for the depth attribute ranging from $[0, ..., 32]$, the most suitable data structure provided is 8-bit large. We therefore attempt to store other node attributes within the same data structure, which can then be retrieved through masking. In the next chapter, we give an assessment of the chosen implementation in terms of speed and memory consumption.
4. Evaluation

4.1. Correctness

To test the correctness of our implementation, we use the scalaCheck library \[1\] more specifically the class Property to assert that the number of elements of a given 32-bit integer set corresponds to the satisfiability-count value of the corresponding BDD. The correctness test is given by the following code:

\[
\text{property} \left( \text{"set cardinality"} \right) = \text{forall} \{ \\
\text{\( a \in \text{Set[Int]} \)} \Rightarrow \\
\text{val b = IntSet(a)} \\
\text{a.size == b.size}
\}
\]

The class IntSet stands for a 32-bit integer set and uses BDDs implementing SAT-Count as a representation. The function IntSet.size is modified to return the SAT-Count of the BDD. The class property uses a customized random value generator to create a total of 100 samples, on which the forall-assertion is tested.

4.2. Speed and Size Evaluation

We expect the new implementation to have repercussions on the speed of the node-construction and consequently on the performance of the BDD-operations. We therefore concentrate in our evaluation on comparing the time needed for construction by the original and the new implementation. For the new data structure, we differentiate between two variants. The first variant uses a 16-bit Short to store the depth, the complement bit as well as the tag value to which the node corresponds in the unique table. This variant uses masking by means of bitwise operations to extract the different components, which represents an additional computational load. The second variant stores the node attributes separately and uses a Byte to represent the depth at each node. Both variants use a 32-bit integer to store the value of the count at the nodes.

To evaluate the construction speed, we generate a collection of random BDD-like data structures and construct the corresponding BDDs with the different implementations while measuring the performance. The BDD-like data structure is a simple description of BDDs defined using only terminals and nodes. It is created by means of a random seed where the probability of generating a node is ten times higher than that of a terminal. The construction stops when a depth upper bound specified beforehand is reached. It may however end before this final condition is met, thus enabling a variation in both depth and size of the resulting data structure.

\[\text{http://scalacheck.org/}\]
### Table 4.1: Performance results in milliseconds

<table>
<thead>
<tr>
<th>Max. Depth</th>
<th>No. of Samples</th>
<th>Original (ms)</th>
<th>Variant 2 (ms)</th>
<th>Variant 1 (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>100 000 (14.8 MB)</td>
<td>1775</td>
<td>1847</td>
<td>85662</td>
</tr>
<tr>
<td></td>
<td>(14.7 MB)</td>
<td>1649</td>
<td>1771</td>
<td>84392</td>
</tr>
<tr>
<td></td>
<td>(14.8 MB)</td>
<td>1521</td>
<td>2099</td>
<td>86394</td>
</tr>
<tr>
<td>10 000 (1.5 MB)</td>
<td>242</td>
<td>330</td>
<td>1062</td>
<td></td>
</tr>
<tr>
<td>(1.5 MB)</td>
<td>196</td>
<td>263</td>
<td>1066</td>
<td></td>
</tr>
<tr>
<td>(1.5 MB)</td>
<td>260</td>
<td>283</td>
<td>1036</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>5000 (45.1 MB)</td>
<td>5783</td>
<td>6809</td>
<td>-</td>
</tr>
<tr>
<td>(44.2 MB)</td>
<td>6208</td>
<td>8430</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>(43.7 MB)</td>
<td>5540</td>
<td>6104</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1000 (9.1 MB)</td>
<td>977</td>
<td>1175</td>
<td>39872</td>
<td></td>
</tr>
<tr>
<td>(8.9 MB)</td>
<td>918</td>
<td>1215</td>
<td>37846</td>
<td></td>
</tr>
<tr>
<td>(8.7 MB)</td>
<td>978</td>
<td>1235</td>
<td>36637</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>1000 (41.1 MB)</td>
<td>3947</td>
<td>6107</td>
<td>-</td>
</tr>
<tr>
<td>(40.7 MB)</td>
<td>5544</td>
<td>7490</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>(40.4 MB)</td>
<td>5864</td>
<td>6048</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>100 (4.0 MB)</td>
<td>527</td>
<td>578</td>
<td>7114</td>
<td></td>
</tr>
<tr>
<td>(3.8 MB)</td>
<td>490</td>
<td>668</td>
<td>6942</td>
<td></td>
</tr>
<tr>
<td>(3.6 MB)</td>
<td>459</td>
<td>647</td>
<td>6190</td>
<td></td>
</tr>
</tbody>
</table>

The first testing attempt was implemented as a single application assuming the task of generating the random data and successively running the different BDD-implementations while measuring the time consumption for each version. Each implementation has been restricted to the basic functions needed for the construction of BDDs and the maintenance of the unique table. This testing approach often showed contradictory results. In fact, we noted that the results depended on the order in which the different BDD-implementations are called, suggesting the intervention of several possible factors including compiler optimization and garbage collection. Due to these limitations, we opted for another strategy where the different tests are run separately.

To reduce the impact of compiler optimization and other factors affecting the time difference between the implementations, an application generates the collection of random BDD-descriptions and stores it in a serialized form on the hard-drive. The file collection is then fully deserialized and processed by an application running a variant of the BDD-implementation. Defining separate applications for the different implementations helps provide identical initial states needed for a reliable comparison of the performance results. Table 4.1 illustrates the duration the BDD-construction with the different implementations in milliseconds. Each test consists of a number of random BDD samples with depths ranging from 0 to a given maximal value.

The experiment shows a clear difference between the original version and the first variant in terms of time consumption, which is mostly due to the heavy cost of bitwise operations needed for masking. These standard operations, offered by the Java libraries
and integrated in the Scala implementation, are only available for 32-bit integer types and their multiples. This involves the application of two conversions from and to 32-bit integers to accompany every used bitwise operation, which contributes to the increase in runtime. The difference factor between the two implementations is not constant and increases proportionally to the sample size. The second variant, however, presents a much lower runtime disparity with the original BDD implementation. The data from Table 4.1 indicates an average runtime increase of 25%, which suggests linearity. This is confirmed by the graph of Figure 4.1 where all samples have a maximal depth of 8. The graph also shows an exponential trend for the runtime needed in the second implementation.

To investigate this last result, we inspected the code running the additional attributes and did not detect any inadvertent recursion that might be responsible for the growth behavior. We therefore came to the conclusion that it is the outcome of another factor, which can be determined through further analyses.

![Graph](image)

**Figure 4.1.:** Time consumption as a function of the number of BDD samples

The speed test of the implementations is insufficient to determine the overall cost of the SAT-Count operation and decide about the best suited implementation variant. The results presented in this section are to be considered alongside the memory consumption. The additional memory overhead is, however, well-known since it mainly consists in the memory needed for the new node attributes (and the corresponding operations). For the first variant, the supplementary memory overhead consists of 5 Byte for each node or terminal, which are used to represent the depth (1 Byte) and count (4 Byte). The second variant also uses 4 Byte to represent the count. However, it compresses the already existent attributes tag (Int) and complement bit (Boolean) respectively to 1 bit and 5 bits, which are stored in a 16-bit Short where the 9 remaining bits are occupied by the depth attribute. This variant has therefore a lower memory consumption. Nevertheless, because of its unfavorable runtime, a trade-off between time and space consumption
4. Evaluation

is not worthwhile. The tests conducted so far clearly favor the second variant for the implementation of SAT-Count, which in turn can be made more efficient with further optimization with regard to type conversions.
5. Future Work

Integration into the Widening-Operation

The purpose of this thesis has been to implement the satisfiability count for BDDs used to represent integer sets. As mentioned in Section 1, this implementation finds its main application in the widening operation of BDDs. Widening consists in cutting a BDD at a certain depth and redirecting the outgoing edges to terminal nodes in such a way that the resulting BDD has more truth assignments and consequently more elements in the corresponding set. This operation is mainly used to reduce the size of the BDD-operands, thus enabling the application of expensive operations such as multiplication and division of integer sets. However, this method is accompanied by a loss of precision due to the involved supersetting. An example of the widening operation is given in Figure 5.1. In this figure, the depth, at which the cut is performed, is represented by a dashed red line. The depth argument may be purely arbitrary and does not take into account the specificity of each sub-BDD structure. In the last example, the false-edge sub-BDD of the root node represents a set of 7 elements and is transformed into a sub-BDD of a superset containing 8 elements. For the true-edge successor representing a singleton set, the superset is 3 elements larger. Thus, the imprecision is not uniformly distributed throughout the different sections of the resulting BDD. The solution proposed in this thesis makes use of the satisfiability count at each node to predict the effect of a potential cut at a certain depth thus preventing high imprecision rates.

The new approach consists in traversing the BDD starting from the root node, and comparing at each node the actual count value to the maximum count value deduced
from the node depth. A cut is only conducted if the difference between these two values is not considerable. Figure 5.2 illustrates the application of this new method on the previous example. In this figure, the value of the count at each node is represented in red. The level of the cut for the false-edge successor of the root node remains since the count value differs from the maximum count value by 1. However, for the true-edge successor, the level of the cut is lowered, taking into account a considerable difference between the count value at the nodes and the maximal number of satisfying assignments at the corresponding depth.

Using this new approach for widening based on the satisfiability count, we hope to produce smaller supersets thereby reducing the error margin of expensive BDD-operations such as multiplication and division. The integration of such a procedure into the basic widening operation is expected to increase its computational load. Therefore, it is necessary to focus, in future analyses, on choosing a suitable implementation keeping the cost increase to a minimum.
6. Conclusion

Over the course of this work, we recapture the basic construction and optimization concepts of BDDs describing boolean functions and emphasize the aptitude of BDDs for the representation of integer sets. This property is exploited by program analysis tools integrating BDDStab, which we address in depth by outlining the customized labelless BDD data structure it implements. The challenge of this representation is the precise implementation of expensive integer operations, so far approximated by BDDStab. Our contribution has been the detailed implementation of an efficient satisfiability count for labelless BDDs, with the help of which we aim to achieve more precision in the approximated results. The algorithms introduced in this work are supported by a mathematical proof establishing correctness. We also maximize the performance of the new operation by caching the results as node attributes. The corresponding implementation is provided in two variants differing in terms of time and space consumption, as confirmed by the tests comparing the impact on the performance of the data structure. In the future, we hope to integrate the satisfiability count in the widening operation responsible for the approximation of expensive integer operations such as multiplication and division.
Bibliography


A. Appendix

A.1. Correctness Proof

SAT-Count for Labelless Complementable BDDs

\[ \forall n \in \mathbb{N}, \forall r \in \{(T, 0), (T, 1), \text{Node}(l, r, c)\} : models(n, r) = modelsR(n, r) \]

Base Cases:

\[ \forall n : modelsR(n, (T, 0)) = 2^n = 2^{n-0} \cdot e(T, 0) = models(n, (T, 0)) \]

\[ \forall n : modelsR(n, (T, 1)) = 0 = 2^{n-1} \cdot (2^0 - e(T, 1)) = models(n, (T, 1)) \]

Induction Hypothesis:

\[ \forall n : models(n, v_1) = modelsR(n, v_1) \]

\[ \forall n : models(n, v_2) = modelsR(n, v_2) \]

First case: \( c = 0, \; d(v_1) \geq d(v_2) \)

\[ \forall n : modelsR(n, \text{Node}(v_1, v_2, 0)) = modelsR(n, v_1) + modelsR(n, v_2) \]

\[ = 2^{n-d(v_1)-1} \cdot e(v_1) + 2^{n-d(v_2)-1} \cdot e(v_2) \]

\[ = 2^{n-d(v_1)-1} \cdot (2^{d(v_1)} - d(v_2)) \cdot e(v_2) + e(v_1) \]

\[ = models(n, \text{Node}(v_1, v_2)) \]

Second case: \( c = 0, \; d(v_1) < d(v_2) \)

\[ \forall n : modelsR(n, \text{Node}(v_1, v_2, 0)) = modelsR(n, v_1) + modelsR(n, v_2) \]

\[ = 2^{n-d(v_1)-1} \cdot e(v_1) + 2^{n-d(v_2)-1} \cdot e(v_2) \]

\[ = 2^{n-d(v_2)-1} \cdot (2^{d(v_2)} - d(n_1)) \cdot e(v_1) + e(v_2) \]

\[ = models(n, \text{Node}(v_1, v_2)) \]

Third case: \( c = 1, \; d(v_1) \geq d(v_2) \)

\[ models(n, \text{Node}(v_1, v_2, 1)) = 2^{n-d(v_1)-1} \cdot (2^{d(v_1)} - d(v_2)) \cdot (2^{d(v_2)} - e(v_2)) + e(v_1) \]

\[ = 2^{n-d(v_1)-1} \cdot e(v_1) + 2^{n-d(v_2)-1} \cdot (2^{d(v_2)} - e(v_2)) \]

\[ = 2^{n-d(v_1)-1} \cdot e(v_1) + 2^{n-1} - 2^{n-d(v_2)-1} \cdot e(v_2) \]

\[ = modelsR(n-1, v_1) + 2^{n-1} - modelsR(n-1, v_2) \]

\[ = modelsR(n, \text{Node}(v_1, v_2, 1)) \]
Fourth case: \( c = 1, \ d(v_1) < d(v_2) \)

\[
\text{models}(n, \text{Node}(v_1, v_2, 1)) = 2^{n-d(v_2)-1} \cdot (2^{d(v_2)} - d(v_1) \cdot e(v_1) + 2^{d(v_2)} - e(v_2)) \\
= 2^{n-d(v_1)-1} \cdot e(v_1) + 2^{n-d(v_2)-1} \cdot (2^{d(v_2)} - e(v_2)) \\
= 2^{n-d(v_1)-1} \cdot e(v_1) + 2^{n-1} - 2^{n-d(v_2)-1} \cdot e(v_2) \\
= \text{modelsR}(n-1, v_1) + 2^{n-1} - \text{modelsR}(n-1, v_2) \\
= \text{modelsR}(n, \text{Node}(v_1, v_2, 1))
\]
### A.2. BDD (Original Version)

sealed abstract trait BDD {
  def toString(c: Boolean): String
  def compl: Boolean
  def tag: Int
}

class CBDD(val bdd: BDD, val compl: Boolean) {
  ...
}

object Terminal extends BDD {
  override def compl = false
  override def tag = 0
  ...
}

object True extends CBDD(Terminal, false) {
  def unapply(cbdd: CBDD) =
    if (cbdd.bdd == Terminal && !cbdd.compl) Some(cbdd)
    else None
}

object False extends CBDD(Terminal, true) {
  def unapply(cbdd: CBDD) =
    if (cbdd.bdd == Terminal && cbdd.compl) Some(cbdd)
    else None
}

final class Node(val set: BDD, val uset: BDD,
  val compl: Boolean, val tag: Int) extends BDD {

  override def hashCode = (set.tag, uset.tag, compl).hashCode

  override def equals(that: Any) = that match {
    case t: Node => compl == t.compl && (set eq t.set) && (uset eq t.uset)
    case _ => false
  }
}
object Node {
  private[this] val cacheBDD = WeakHashMap.empty[BDD, BDD]
  private[this] var tagCounter: Int = 1
  def apply(set: CBDD, use: CBDD) = {
    val ibit = set.compl
    if (set.compl == use.compl &&
        set.bdd == Terminal && use.bdd == Terminal)
      new CBDD(Terminal, ibit)
    else {
      val compl: Boolean = set.compl != use.compl
      val tentative = new Node(set.bdd, use.bdd, compl, tagCounter)
      val hashconsed = cacheBDD.getOrElseUpdate(tentative, tentative)
      if (tentative eq hashconsed) tagCounter += 1
      new CBDD(hashconsed, ibit)
    }
  }
  ...
}
A.3. BDDs (Variant 1)

sealed abstract trait BDDs {
  val depComp: Short
  val count: Int
  def depth: Int
  def compl: Boolean
  def tag: Int
  def invalid: Boolean

  ...
}

class CBDDs(val bdd: BDDs, val compl: Boolean) {
  def depth = bdd.depth
  def count: Long = {
    val count: Long = if(bdd.count >= 0) bdd.count
    else pow(2,bdd.depth).toLong + bdd.count
    if(compl) pow(2,bdd.depth).toLong - count else count
  }
  ...
}

object Terminals extends BDDs {
  val depComp: Short = 0
  val count = 1
  override def hashCode = depComp
  override def compl = false
  override def depth = 0
  override def tag = 0
  override def invalid = false
  ...
}

object TrueS extends CBDDs(Terminals, false) {
  def unapply(cbdd: CBDDs) =
    if (cbdd.bdd == Terminals && !cbdd.compl) Some(cbdd)
    else None
}

object FalseS extends CBDDs(Terminals, true) {
  def unapply(cbdd: CBDDs) =
    if (cbdd.bdd == Terminals && cbdd.compl) Some(cbdd)
    else None
}
final class Nodes(val set: BDDs, val uset: BDDs,  
    val depComp: Short) extends BDDs {
  private def countR: Int = uset match {
    case Terminals if(compl) => 0  
    case Terminals if(!compl) => 1  
    case _ => if(compl) (pow(2, uset.depth).toLong - uset.count).toInt
    else uset.count
  }

  private def countL = set.count
  val count: Int = (uset.depth - set.depth) match {
    case x if (x > 0) => (pow(2, uset.depth - set.depth).toLong
        * countL + countR).toInt
    case x if (x < 0) => (pow(2, set.depth - uset.depth).toLong
        * countR + countL).toInt
    case _ => (countR + countL).toInt
  }

  override def hashCode = (set.tag, uset.tag, compl).hashCode
  override def invalid = (depComp % 2) == 1
  override def depth = (depComp & Mask.DEPTH) >> Mask.SHIFT_DEP
  override def compl = (depComp & Mask.COMPLEMENT) != 0
  override def tag = depComp & Mask.TAG
}

object Int2Short {
  implicit def int2Short(i: Int): Short = i.toShort
}

object Nodes {
  import Int2Short._
  private[this] val caches = WeakHashMap.empty[BDDs, BDDs]
  private[this] var tagCounter: Int = 0
  def apply(set: CBDDs, uset: CBDDs) = {
    val ibit = set.compl
    if (set.compl == uset.compl &&
        set.bdd == Terminals && uset.bdd == Terminals)
      new CBDDs(Terminals, ibit)
else {
    val compl: Boolean = set.compl != uset.compl
    var depcomp: Int = if(compl) 2 else 0
    depcomp |= (max(set.bdd.depth, uset.bdd.depth) + 1) << Mask.SHIFT_DEP
    depcomp |= tagCounter

    val tentative = new Nodes(set.bdd, uset.bdd, depcomp)
    val hashconsed = caches.getOrElseUpdate(tentative, tentative)
    if (tentative eq hashconsed) tagCounter += Mask.TAG_STEP
    new CBDDs(hashconsed, ibit)
}

...
A.4. BDDint (Variant 2)

sealed abstract trait BDDint {
  val count: Int
  val depth: Byte
  val tag: Int
  ...
}

class CBDDint(val bdd: BDDint, val compl: Boolean) {
  ...
  def depth = bdd.depth
  def count: Long = {
    val count: Long = if(bdd.count >= 0) bdd.count
      else pow(2, bdd.depth).toLong + bdd.count
      if(compl) pow(2, bdd.depth).toLong - count else count
  }
  ...
}

object TerminalInt extends BDDint {
  val count = 1
  override def hashCode = tag
  val compl = false
  val depth: Byte = 0
  val tag = 0
}

object TrueInt extends CBDDint(TerminalInt, false) {
  def unapply(cbdd: CBDDint) =
    if (cbdd.bdd == TerminalInt && !cbdd.compl) Some(cbdd)
      else None
}

object FalseInt extends CBDDint(TerminalInt, true) {
  def unapply(cbdd: CBDDint) =
    if (cbdd.bdd == TerminalInt && cbdd.compl) Some(cbdd)
      else None
}

object Int2Byte {
  implicit def int2Byte(i: Int): Byte = i.toByte
}


final class NodeInt(val set: BDDint, val uset: BDDint, 
    val compl: Boolean, val tag: Int) extends BDDint {
    import Int2Byte._ 
    val depth: Byte = max(set.depth, uset.depth) + 1 
    private def countR: Int = uset match {
        case TerminalInt if(compl) => 0
        case TerminalInt if(!compl) => 1
        case _ => if(compl)
            (pow(2, uset.depth).toLong - uset.count).toInt
        else uset.count
    }
    private def countL = set.count 
    val count: Int = (uset.depth - set.depth) match {
        case x if (x > 0) => (pow(2, uset.depth - set.depth).toLong
            * countL + countR).toInt
        case x if (x < 0) => (pow(2, set.depth - uset.depth).toLong
            * countR + countL).toInt
        case _ => (countR + countL).toInt
    }
    override def hashCode = (set.tag, uset.tag, compl).hashCode
    ...
}
object NodeInt {
    private[this] val cacheInt = WeakHashMap.empty[BDDint, BDDint]
    private[this] var tagCounter: Int = 1 
    def apply(set: CBDDint, uset: CBDDint) = {
        val ibit = set.compl 
        if (set.compl == uset.compl &&
            set.bdd == TerminalInt && uset.bdd == TerminalInt)
            new CBDDint(TerminalInt, ibit)
        else {
            val compl: Boolean = set.compl != uset.compl 
            val tentative = new NodeInt(set.bdd, uset.bdd, compl, tagCounter) 
            val hashconsed = cacheInt.getOrElseUpdate(tentative, tentative) 
            if (tentative eq hashconsed) tagCounter += 1
            new CBDDint(hashconsed, ibit)
        }
    }
    ...
}