OBDD–based Set Implementations
Abstract
Storing huge amount of data in a memory has always been one of the main challenges of software development process. Using an approach for saving data that is not efficient enough may result in consumption of huge amount of memory and time. One of the examples of such applications where the amount of used memory and time is one of the crucial properties of the system is Points-to analysis.

Points-to analysis computes reference information which is an essential input to different client applications, e.g., static garbage collection or metric analysis. This type of information is the basis for many different analysis and transformations. One of the challenges of implementing high-precise analysis is increasing of memory consumption. The possible solution is to store context-sensitive points-to analysis information in a compact, non-redundant way that will make it efficient enough to perform on benchmarks of significant sizes.

Ordered-Binary Decision Diagrams (OBDDs) have been shown very useful for representing large sets of objects and solving memory consuming problems. OBDD is a data structure for showing Boolean functions by directed acyclic graph with a set of manipulation algorithms. The use of OBDD-based sets as a mean to store points-to analysis results can lead to memory reduction in the analysis phase.

This thesis is focused on implementation of OBDDs for capturing and manipulating different amount of sets of objects that could represent points-to analysis information. Additionally, we compare OBDDs with a yet another implementation known as BitSet where each set of objects is represented as a vector of bits.

The goal of this thesis was to show that OBDDs can also be quite effective for storing and manipulating of points-to analysis information of large programs. However, the results are not that impressive as it was expected after theoretical investigation.

Keywords:
Ordered Binary Decision Diagram (OBBD), Graph, Points-to analysis, reduce, satisfy-all, satisfy-one, apply, vertex, BitSet, Lattice, Boolean function, memory, create, DAG(Directed Acyclic Graph), Reduced Binary Decision Diagram (ROBDD), Redundant, isomorphic sub-graph, meet, join
Acknowledgement

First, we would like to offer sincerest gratitude to our supervisors, PhD student Antonina Danylenko and Dr. Jonas Lundberg for providing this interesting topic and helping us throughout this thesis. We would never have been able to finish our thesis without their guidance. We also appreciate help of Mathias Hedenborg during this thesis.

We would like to thank our parents for supporting us towards this journey.
## Contents

1 Introduction .................................................................................................................. 1  
1.1 Research goals ..................................................................................................... 1  
1.2 Outline of Intended Approach ............................................................................. 1  
1.3 Motivation ............................................................................................................. 1  
1.4 Thesis Outline ..................................................................................................... 2  
2 Related Work ............................................................................................................. 3  
2.1 Points-to analysis in BDDs ..................................................................................... 3  
2.2 Model Checking .................................................................................................... 3  
3 Set Representation Approach .................................................................................... 5  
3.1 OBDD Representation ........................................................................................... 5  
3.2 OBDD: Construction and Manipulation ............................................................... 8  
3.2.1 The Traverse Operation ................................................................................... 8  
3.2.2 The Reduce Operation .................................................................................... 9  
3.2.3 The Apply Operation ....................................................................................... 12  
3.2.4 The Satisfy-one Operation ............................................................................. 16  
3.2.5 The Satisfy-all Operation .............................................................................. 17  
3.3 BitSet Representation .......................................................................................... 18  
3.4 Summary .............................................................................................................. 19  
4 Implementation .......................................................................................................... 21  
4.1 Total Overview ..................................................................................................... 21  
4.2 OBDD Creation ..................................................................................................... 22  
4.2.1 Apply Implementation .................................................................................... 24  
4.3 Lattice .................................................................................................................. 25  
4.4 OBBD Lattice ....................................................................................................... 26  
5 Experimental Results .............................................................................................. 29  
5.1 Metrics .................................................................................................................. 29  
5.2 Results .................................................................................................................. 29  
6 Conclusion and Future Work .................................................................................... 37  
6.1 Conclusion ............................................................................................................ 37  
6.2 Future Work ......................................................................................................... 37
List of Figures

Figure 3.1: Representation of the function $f(x_1, x_2, x_3)$ ........................................... 6
Figure 3.2: Representation of Variable Ordering ......................................................... 7
Figure 3.3: Implementation of Traverse Operation ...................................................... 9
Figure 3.4: Traverse of Graph .............................................................................. 10
Figure 3.5: Implementation of Reduce Operation ...................................................... 11
Figure 3.6: Reduction Algorithm Example ............................................................... 12
Figure 3.7: Implementation of Apply .................................................................... 15
Figure 3.8: Apply Example .................................................................................. 16
Figure 3.9: Implementation of Satisfy-one ............................................................... 17
Figure 3.10: Satisfy-one Traversal .......................................................................... 17
Figure 3.11: Implementation of Satisfy-all ............................................................... 18
Figure 3.12: Traverse of Satisfy-all Operation ....................................................... 19
Figure 4.1: Packages dependencies .................................................................. 21
Figure 4.2: Representation of OBDD for Sets \{1,2\} and \{3,4\} ............................. 24
Figure 4.3: Reduce Graph Figure 4.2 ................................................................. 25
Figure 4.4: Class Diagram and Dependencies ....................................................... 28
List of Tables

Table 3.1: Truth Table of $f(x_1, x_2, x_3)$ ................................................................. 5
Table 3.2: All Cases of Apply Algorithm ................................................................. 13
Table 4.1: The Binary Form of Set ............................................................................ 23
Table 5.1: First Result of OBBD for Memory Consumption ................................. 30
Table 5.2: First Result of OBBD for Execution Time ............................................ 31
Table 5.3: Set Size-Memory (OBDD) ...................................................................... 32
Table 5.4: Set Size-Time (OBDD) ........................................................................... 33
Table 5.5: Set Size-Memory (BitSet) ...................................................................... 34
Table 5.6: Set Size-Time (BitSet) ........................................................................... 35
List of Charts

Chart 5.1: Chart of table 5.1 ................................................................. 30
Chart 5.2: Chart of table 5.2 ................................................................. 31
Chart 5.3: Chart of table 5.3 ................................................................. 33
Chart 5.4: Chart of table 5.4 ................................................................. 34
Chart 5.5: Chart of table 5.5 ................................................................. 35
Chart 5.6: Chart of table 5.6 ................................................................. 36
1 Introduction
This chapter represents a brief introduction of thesis goals, lists out the goals and discusses the motivation behind thesis topic. At the end the thesis outline will be explained.

1.1 Research Goals
Memory consumption is an important issue when we want to have and store enormous sets of data. Using an optimized approach to store huge amount of information can be helpful in order to consume less memory. Different approaches such as BitSet implementation for decreasing the memory consumption have been already implemented. Thus, the goal of this thesis is to theoretically analyze and try to implement efficient approach to store significant sizes of data.

This goal can further be divided into two sub-goals:

1. Theory: Analyzing existing approaches for storing large amounts of data on particular examples of sets of objects. Theoretically adjusting the representation of Ordered Binary Decision Diagrams (OBDDs) to efficiently capture and manipulate the sets of different objects according to the existing approaches that use OBDDs for decreasing memory consumption.

2. Implementations: Implementing OBDDs along with a set of manipulation algorithms e.g., reduce and apply according to our theoretical implications and comparing it to BitSet (yet another data structure which is implemented efficiently to store huge amounts of sets) in terms of memory and time for different size of data.

1.2 Outline of Intended Approach
This thesis’s approach to fulfill the two research goals above could be outlined as following. In order to reach the first goal the Brayant model [1] for OBDDs have been analyzed theoretically and our approach have been implemented for large sets of data based on Lhot’ak [5]. The BitSet approach that is an efficient data structure has been examined also.

To fulfill the second goal the OBDDs and the associated set of manipulation algorithms have been implemented according to Brayant [1] to store different sets of objects. There are plenty of methods and algorithms such as reduce, apply and satisfy-all to reach this aim. The second restriction of Bryant model checking [2] has been used and consequently reduce algorithm [3] has been applied in order to decrease the memory. Then power lattice which is a container for representing OBDDs have been implemented. At the end we measured the time and memory for significant amount of data and then we compared our results to BitSet to observe the differences.

1.3 Motivation
During the procedure of storing massive amount of data, memory consumption is always an issue. Using an inefficient approach for saving huge amount of information might cause
a situation that the system runs out of memory. So finding an optimum solution for storing a high number of sets is the main idea of this thesis. Our effort to reduce the memory consumption became the starting point for representing a data structure that is called OBDD.

OBDD stands for Ordered Binary Decision Diagram. OBDD is a directed acyclic graph (DAG) with a set of manipulation algorithms. The set of manipulation algorithms are used for optimizing the OBDD representation. OBDD is used frequently because of its easy way of representing and performing Boolean functions. It also represents the large sets of data concisely.

BitSet is one of the java data structure and has been used for representing and implementing the huge amount of data. An efficient implementation for BitSet has been developed in Linnaeus University by Software Technology group. Later on, we will compare our implementation (OBDD Set) results to BitSet implementation results.

OBDD Set and BitSet implementations are using same interface in some parts, thus it is feasible to compare the two together. Our implementation allows studying results and benefits from this comparison in a way that would help to improve OBDD implementation for presenting an efficient approach in the future.

1.4 Thesis Outline
The rest of the report has the following outline. Chapter 2 discuses related work that has been done previously. It gives a short overview of points-to analysis and Model Checking. In chapter 3 we go deeper in Set Representation. This chapter (OBDD Representation) represents OBBD’s definition and how they have been created. We describe OBDD’s set of manipulation algorithms and it is followed by a representation of BitSet. Chapter 4, called Implementation, shows how we implemented the OBDD data structure and its operations in details. Lattice set and power lattice have been also described. This includes some mathematical definition. In addition it provides implementation issues of power lattice. In chapter 5 we represent our OBBD algorithms results for different sizes of sets, especially big sizes of data. We also use some metrics to evaluate our approach as well as BitSet implementation. At the end of this chapter we illustrate the memory and time results in different charts. There are some charts to compare OBDD to BitSet. And finally in the last chapter, Conclusion and Future Work, we summarize what we have done during this thesis and the results have been categorized. A short description of how this implementation could be improved in the future has been provided as well.
2 Related Work
In this chapter we describe what points-to analysis is and how it helps to reduce the memory. We also discuss about some approaches that are efficient in case of memory consumption.

Bryant introduced the Ordered Binary Decision Diagram (OBDD) in 1986 [3]. Some rules and limitations on arranging of BDDs have been applied by Lee[7] and has been developed later by Aker [8]. OBDD has an order or arrangement on all vertices of BDD from root to leaf vertices [1].

There exist different domains where OBDD can improve memory consumption. In what follows we will discuss about one of this domains.

2.1 Points-to analysis in BDDs
Each program consists of many objects which may point to different other objects. Each object has a reference that points to an address in memory, in other words it consumes memory. Points-to analysis is a program which is used for calculating the references of objects in the program. One of the useful applications that use Points-to analysis is garbage collector. Garbage collector collects the de-allocated objects which results in memory reduction [4].

There are some applications where the huge amount of memory consumption is a problem. During analyzing those applications, the main problem is the number of points-to sets and size of each set would become very large. As it was mentioned before, Points-to analysis is an example of application that the amount of used memory is an important issue.

In order to find a compact way of using points-to sets different approaches have been tried and investigated. There are some papers [13], [14], [15] that show BDDs can be very effective for developing a points-to analysis. One example of BDD is bddbddb that is also an efficient approach for implementing large sets of data. Another efficient approach is BitSet BDD-based that it will be explained more in Model Checking section.

2.2 Model Checking
Based on the researches that have been done before, BDD-based approaches are efficient and compact ways of representing large sets. BDD stands for Binary Decision Diagram, and we will go through its details in next chapter.

Here in this thesis we work on Ordered Binary Decision Diagrams (OBDDs). OBDD is a BDD-based implementation. As it was mentioned before BitSet is a BDD-based data structure also for storing huge sets of binary vectors. We will compare the memory and time results of these two implementations at the end.

BitSet representation stores context-sensitive points-to analysis information in a compact, non-redundant way that will make it efficient enough to perform on benchmarks of significant size. One of the important factors that make BitSet efficient for storing large
sets of objects is to apply reduce algorithm. The reduced BitSet shows the same set as the original unreduced BitSet but it is smaller.

In section 3.2.2 we will completely explain how this algorithm works and results in memory reduction by using points-to analysis and garbage collection.

We exactly use the same reduce algorithm for decreasing the memory, but our approach for creating the graph is different. Therefore, in theory we expected to get better memory result comparing to BitSet representation.

In next chapters we will discuss our approach and implementation more. We also give more details about BitSet representation.
3 Set Representation Approach

A set is a group of different objects which they have been completely defined. There are some rules for every set of objects that makes it easy to observe whether an object is a member of the set or not. It is possible to apply different operations for creating new sets from existing sets.

- Union (U): Two sets can be merged together. The union of set A and set B is a set that consists of all elements of A and B.
  \[ A \cup B = \{ x \mid x \in A \lor x \in B \} \]

- Intersection (∩): The intersection of set A and set B is a set of elements which belong to both set A and B.
  \[ A \cap B = \{ x \mid x \in A \land x \in B \} \]

- Difference: The difference of set A and set B which is denoted by A-B is a set of elements of A that they don’t belong to set B.
  \[ A - B = \{ x \mid x \in A \land x \notin B \} \]

OBDDs and BitSet representations are different implementations of sets.

3.1 OBDD Representation and Construction

In 1978 Akers [8] has introduced a more efficient way of implementing BDDs based on Boolean functions which consumes less memory [6]. A binary decision diagram (BDD) “is a rooted, directed acyclic graph (DAG).” [3] Each BDD is formed of two kinds of vertices, terminal vertices and non-terminal vertices.

Every non-terminal vertex v is a vertex with two outgoing edges which have been assigned to 0 and 1 [2]. The edge with label 0 goes to low(v) and the edge with label 1 goes to high(v) or in other words:

- low(v) is a condition where vertex v is set to 0 and high(v) is a condition where vertex v is set to 1.

Every terminal vertex v has a value which is 1 or 0.

Figure 3.1 shows a representation of the function \( f(x_1,x_2,x_3) \) defined by truth table 3.1.

<table>
<thead>
<tr>
<th>( X_1 )</th>
<th>( X_2 )</th>
<th>( X_3 )</th>
<th>( F )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3.1: Truth Table of \( f(x_1,x_2,x_3) \)
As it is shown in figure 3.1 if the vertex is assigned to 0, it will go to low(v), the child on the left. But if it is assigned to 1 it will go to high(v), the child on the right. The value of the terminal vertex shows the value of function for the special path. If the value is 1 it means the special path from root to that terminal vertex satisfies the Boolean function. If the value is 0 it means the path does not satisfy the Boolean function.

As an example the path 011 leads to a terminal with value 1 and based on table 3.1 it is true. At the same time path 000 goes to a terminal with value 0 and it is false. As figure 3.1 shows there are a lot of redundancies in such graphs. Binary decision diagrams do not provide a very compact representation for Boolean functions. In figure 3.1 all 3 sub-graphs with root $x_3$ are exactly the same (isomorphic sub-graphs).

Bryant applied two rules on binary decision diagrams in order to reach a more compact representation for binary decision diagram [2].

- “All vertices in the graph from root to leaf vertices must obey an equal order”.
- “The graph does not include any isomorphic sub-graph or redundant vertices”.

In the following the works that should be done to achieve these two rules will be explained. It also will be explained later what isomorphic sub-graphs and redundant vertices are.

For applying the first rule some ordering should be done on the BDD.

There is a characteristic for all non-terminal vertices of the graph which is called index $1 \leq \text{index} \leq n$. The index of root vertex is always 1 and all the other non-terminals based on
their level will get different indices. All terminals have always index n+1 (n is considered as the last level(index) of non-terminals). In this thesis ordering is done among the vertices based on their indices.

All non-terminal children of vertex v have a greater index that v, for instance if u is a non-terminal child of v, then index(u) > index(v). By this variable ordering: index(root) < index (x_2) < index (x_3) < Terminal on figure 3.1 the result will be figure 3.2. The numbers in the figure 3.2 are the level of each vertex (index). The vertices in a same level have the same index (order). All the terminals have the same, biggest index in the graph.

![Figure 3.2: Representation of variable ordering](image)

A Reduced Ordered Binary Decision Diagram (ROBDD) is an OBDD which is a reduced and does not consist of any isomorphic sub-graphs or redundant vertices.

The second rule of Bryant representation that was mentioned above is achieved by these three rules [1]:

**“Remove duplicate terminals”:** All terminal vertices except for one distinct terminal vertex are removed. In Reduced Ordered Binary Decision Diagrams there is just one terminal vertex with value 0 and there is one terminal vertex with value 1, and then forward all received edges of the removed vertices to the this lasting one.

**“Remove duplicate non-terminals”:** If there are two sub-graphs with roots u and v and the following conditions: index(u) == index(v), low(u) == low(v) and high(u) == high(v) is valid for them, then they are isomorphic. One of them is removed and all received edges of removed vertex are forwarded to the lasting one.
“Remove redundant tests”: If there is this condition: \( \text{low}(v) = \text{high}(v) \) for a non-terminal vertex \( v \), then \( v \) is removed and all received edges of \( v \) are forwarded to \( \text{low}(v) \).

### 3.2 OBDD: Construction and Manipulation

In this chapter a set of manipulation algorithms is explained. The basic operations traverse, reduce, apply, satisfy-one and satisfy-all can be combined to perform a wide variety of operations on Boolean functions. These algorithms will be explained in details later on.

Vertex is a basic part of a graph that has been used frequently to create the graph. Vertex has a group of characteristics. Some important ones are listed below:

- Vertex low, high: Each non-terminal vertex has both low and high children. These children are also vertices. Terminal vertices have no children.

- Index: is an integer value between 1 and \( n+1 \). As it was discussed previously, depending on the variable ordering each level of graph has a special integer value. The indices start from root (it has 1 as index) and it continues until the terminal vertices that they have the biggest integer value as index. All terminal vertices have the same index \( n+1 \), where \( n \) is the last level of non-terminal vertices. In figure 3.1 \( n \) equals to 3, you can see all indices for all vertices in figure 3.2.

- Value: All terminal vertices have value 0 or 1. The value of a terminal depends on whether the special path of graph that reaches to this terminal has a false value in the Boolean function or the result is true. In this implementation the value of non-terminals is -1.

- Id: Each vertex has an id that is unique for every vertex.

- Mark: It is a Boolean attribute that is initialized with false. When a vertex is visited the value of mark is changed to true.

#### 3.2.1 The Traverse Operation

Considering figure 3.3 [3], the traverse algorithm has an input argument. This procedure is started with the mentioned argument. This argument is a vertex which is the root of graph or sub-graph that will be visited. After visiting the root vertex, the procedure will visit the left sub-graph and right sub-graph recursively. This procedure is a pre-order traversal algorithm. When the procedure visits a vertex, it changes the value of “mark” to true (cf. line 1), which is by default has been set to false. Sometimes during different procedures it is needed to know if a vertex has been visited or not. For performing this task the value of “mark” can be evaluated, if it is true, it means, it has been visited (cf. line 2.1 and line 2.2). This operation meets every vertex one time. If the number of vertices is \( n \) the complexity of algorithm would be \( O(n) \).

Algorithm Traverse\((v : \text{Vertex})\){
   1. \( v.\text{mark} = \text{true} \)
2. If $v.index \leq n$
   2.1 If $v.mark \neq v.low.mark$
       2.1.1 Traverse($v.low$)
   2.2 If $v.mark \neq v.high.mark$
       2.2.1 Traverse($v.high$)

Figure 3.3: Implementation of Traverse Operation

Figure 3.4 represents the traversal of figure 3.1. It starts from the root, and then goes to the left part and finally visits the right part of the graph.

3.2.2 The Reduce Operation
The reduce algorithm is designed to reduce a binary graph [3]. Although it changes the graph’s form, but it still represents the similar functions.

The purpose of reduce algorithm is to eliminate redundant vertices and isomorphic subgraphs which they will be explained later. Figure 3.5 represents the reduce algorithm. For implementing this algorithms the below steps have been followed.

- It is needed to have a list of all vertices based on their indices. It could be done by using a procedure same as traverse. In this thesis all the vertices have been added to this list during the creation of the OBDD.

- The next step is to process the above list. It starts from the terminal vertices and ends in root vertex. During this step, all vertices will be evaluated based on their indices. According to the value of the index it will be decided whether some vertices are redundant (cf. line 5.2.2.1). If any redundant vertex will be discovered, its id will be set to the id of its low child (cf. line 5.2.2.1). There are two rules, which based on those rules it will be decided whether some vertices are redundant or there are isomorphic sub-graphs:
  If two terminal vertices have the same value, they should have the same id (Based on rule “Remove duplicate terminals” that has been explained already in section 3.1).
During this algorithm the two vertices with the same index will get the same id if and only if these two restrictions happen: “Remove duplicate non-terminals” and “Remove redundant test”. The both rules are explained in section 3.1.
In order to detect redundant vertices and isomorphic sub-graphs a special container is needed. This container saves both lowid and highid for non-terminal vertices and the value for terminal vertices. Another list is needed (cf. line 5.1) which will be used for saving the keys. A new key will be created for non-terminal (cf. line 5.2.3.1) and terminal vertices (cf. line 5.2.1.1).
The next step, is sorting the list of keys (keylist, cf. line 5.3). A new id will be set for the vertices that have the similar key (sc. Line 5.5.1.1). An array of Vertex has been created before (cf. line 1), to keep all the unique vertices (cf. line 5.5.2.2). This unique and its children will be saved in this array (cf. lines 5.5.2.3 and 5.5.2.4). The subgraph[v.id] returns the root of reduces graph, it includes all the vertices of reduces graph also.

The sorting procedure has a linear time complexity and it depends to the number of vertices at that level. If the number of vertices is n, the time complexity is $O(n)$.

\[\text{Algorithm Reduce}(v: \text{Vertex}) : \text{Vertex}\{\]
\begin{align*}
1. & \quad \text{subgraph: array}[1\ldots|G|] \text{ of Vertex} \\
2. & \quad \text{vlist: array}[1\ldots n+1] \text{ of list} \\
3. & \quad \text{Put each vertex u on list vlist}[u.index] \\
4. & \quad \text{nextid} := 0 \\
5. & \quad \text{for (int i = n+1; i>= 1; i--)}{
5.1 & \quad \text{Create an empty set named keyList} \\
5.2 & \quad \text{For (Vertex u : vList)}{
5.2.1 & \quad \text{if u.index = n+1} \\
5.2.1.1 & \quad \text{key = u.value \{Terminal Vertex\}} \\
5.2.1.2 & \quad \text{keyList.add(<key,u>)}
\end{align*}
5.2.2 else if(u.low.id = u.high.id)
5.2.2.1 u.id := u.low.id {redundant vertex}
5.2.3 else{
5.2.3.1 Key = (u.low.id, u.high.id)
5.2.3.2 keyList.add(<key,u>)
}
5.3 Sort all elements of Q by keys;
5.4 oldkey = (-1,-1) {Unmatchable key}
5.5 for (Key k : keyList){
5.5.1 if key = oldkey
5.5.1.1 u.id := nextid; {matches existing vertex}
5.5.2 else {
5.5.2.1 nexid = nextid+1; u.id = nextid
5.5.2.2 subgraph[nexid] = u
5.5.2.3 u.low = subgraph[u.low.id]
5.5.2.4 u.high = subgraph[u.high.id]
5.5.2.5 oldkey = key;
}
}
return (subgraph[v.id]);
}

Figure 3.5: Implementation of Reduce Operation

Figure 3.6 illustrates reduced graph of figure 3.1. All sub-graphs with root x_3 in figure 3.1 are isomorphic based on the second rule. So the algorithm just keeps on of them. The vertex x_2 on the left is redundant, considering the first rule, so this one is removed as well.

3.2.3 The Apply Operation

In algorithm apply we will create a new graph according to the variety of operators in a Boolean expression. Any binary operator of two arguments can be applied over two function graphs also. With this representation it is possible to implement different operators (i.e. union which has been implemented with operator OR (∨), intersection has been implemented with operator AND (∧) and difference which has been implemented with operator AndNot(Combination of operator & and !). As the code is shown in figure 3.7 [3], two vertices and one operator are the inputs of the algorithm. The two vertices are the roots of the graphs or sub-graphs that will be applied together based on the mentioned input operator. There are several steps to implement this algorithm.

- This algorithm starts from the roots of graphs F_1 and F_2. The procedure is followed by calling recursively procedure apply-step on other sub-graphs.
During the procedure of applying this algorithm on the different sub-graphs of graph $F_1$ and $F_2$ there is no need to evaluate a pair of sub-graph that has been computed before. To prevent this, a two dimensional array has been created (cf. line 1). At first all elements of array are initialized to null (cf. line 2.2). The array includes the indices of the two vertices that have been calculated before. So when the algorithm is called, first it checks in this array whether these two vertices have been calculated before or not. If the bucket in this array with their indices is null it means that they have not been calculated yet, so it goes through the algorithm and save the vertex result in this array with the indices of vertices. But if there is a result for them, there is no need to go over the algorithm again, it just returns the result of the array.

During this algorithm there are some situations that based on these conditions the implementation and result will be different. Table 3.2 represents all these different situations. The first condition is when the both vertices are non-terminals then the result vertex $u$ is a non-terminal with a unique index. If the both indices of vertex $v_1$ and $v_2$ are the same the index of result vertex $u$ will be index $v_1$, otherwise the minimum index of $v_1$ and $v_2$ will be assigned to the new vertex $u$ (cf. line 8.1). The next step is to finding the $\text{low}(u)$ and $\text{high}(u)$ (cf. line 9 - 10). $\text{Low}(u)$ is a root for the left sub-graph and $\text{high}(u)$ is a root for the right sub-graph of result graph. Another condition is where one of the vertices
is terminal and the other one is non-terminal (Situation 2, 3 in table 3.2). In this case the result vertex could be a terminal or non-terminal vertex based on the operator and the value of the terminal vertex. For example if the operator is OR and the value of terminal vertex is 1, the result vertex is always will be terminal vertex with value 1. Because 1 is a controlling value in OR operator as well as 0 for AND operator. But if the terminal does not have a controlling value, the result vertex is a non-terminal with minimum index which is the index of non-terminal vertex. The procedure of setting low(u) and high(u) is same as previous condition. The last case is where both are terminal vertices. The result vertex is a terminal vertex with index n+1 (the same index for all terminal vertices cf. line 7.1).

<table>
<thead>
<tr>
<th>Situation</th>
<th>Vertex v1</th>
<th>Vertex v2</th>
<th>Vertex u</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Non-Terminal</td>
<td>Non-Terminal</td>
<td>Non-Terminal</td>
</tr>
<tr>
<td>2</td>
<td>Terminal</td>
<td>None-Terminal</td>
<td>Based on the operator and value of Terminal Vertex could be terminal or non-terminal</td>
</tr>
<tr>
<td>3</td>
<td>Non-Terminal</td>
<td>Terminal</td>
<td>Based on the operator and value of Terminal Vertex could be terminal or non-terminal</td>
</tr>
<tr>
<td>4</td>
<td>Terminal</td>
<td>Terminal</td>
<td>Terminal</td>
</tr>
</tbody>
</table>

Table 3.2: All Cases of Apply Algorithm

Imagine there are graphs $F_1$ and $F_2$ with $|n_1|$ and $|n_2|$ vertices, in algorithm apply creating and initializing the array is $O(|n_1|.|n_2|)$. Considering that algorithm apply calls apply-step only once, and procedure apply-step consists of several recursive calls so the total number of recursive calls can be $O(|n_1|.|n_2|)$. So the time complexity of the algorithm is $O(|n_1|.|n_2|)$.

In order to have the reduced form of the result graph after applying two graphs (sub-graphs) together the reduce method is called (cf. line 4). Figure 3.8 represents the original graphs $F_1$ and $F_2$, applied sub-graph and finally the reduced form of applied sub-graph. The root of result graph is the outcome of applying operator OR on roots of graphs $F_1$ and $F_2$. As it is shown in the last picture of figure 3.8, the result is a small graph with just 3 vertices.

Algorithm Apply($v_1,v_2$: vertex; <op> : operator): Vertex{
    1. T: array[1...|G1|, 1...|G2|] of vertex
    2. for (int row =0; row< number of elements in T; row++)
        2.1 for (int col =0; col < number of elements in T; col++)
            2.2 T[row][col] = null;
    3. Vertex u = apply-step ($v_1,v_2$);
    4. return (Reduce(u));
}
Algorithm apply-step (v1,v2: Vertex): Vertex;
{
  1. Vertex u = T[v1.id, v2.id];
  2. if u != null
     2.1 return (u); { It has been calculated before} 
  3. u = new vertex record;
  4. u.mark := false;
  5. T[v1.id , v2.id] = u;
  6. u.value = v1.value <op> v2.value;
  7. if u.value != x
     7.1 u.index = n+1;
     7.2 u.low = null;
     7.3 u.high = null;
  } 
  8. else {
     8.1 u.index = Min(v1.index , v2.index);
     8.2 if v1.index = u.index {
     8.2.1 vlow1 = v1.low;
     8.2.2 vhigh1 = v1.high 
    }
     8.3 else{
     8.3.1 vlow1 = v1;
     8.3.2 vhigh1 = v1 
   }
   8.4 if v2.index = u.index{
     8.4.1 vlow2 = v2.low;
     8.4.2 vhigh2 = v2.high 
  }
  8.5 else {
     8.5.1 vlow2 = v2;
     8.5.2 vhigh2 = v2 end;
  }
  9. u.low = apply-step(vlow1,vlow2);
  10. u.high = apply-step(vhigh1,vhigh2);
}
  11. return(u);
}

Figure 3.7: Implementation of Apply
Graph $g_1$

Graph $g_2$

Evaluation Graph

Reduced Graph
3.2.4 The Satisfy-one Operation

Algorithm satisfy-one represented in figure 3.9 [3], like all of the procedures that have been implemented in this thesis starts from the root of graph and consists of several recursive calls in order to find the terminal vertex with value one. This procedure returns false if none of terminal vertices have value 1, but when it finds the first terminal with value 1 returns true (cf. line 2.1). There is an array that is initialized with 0 and 1 during the process. If we follow finding the value 1 with recursive calls on the left sub-graph (sub-graph with root v.low), 0 will be saved in array otherwise 1 is stored in this array (finding value 1 in sub-graph with root v.high). At the end array shows the path to reach the 1. If the number of vertices in graph g is n, whether the graph is reduced or not, the time complexity of algorithm is O(n).

Algorithm Satisfy-one(v: Vertex; x: array[1…n] of integer) : Boolean{
1. if (v.value == 0)
2. then return (false);
3. if (v.value == 1)
4. then return (true);{found the first terminal vertex with value 1 and return}
5. x[i] = 0;
6. if (Satisfy-one (v.low,x))
7. then return true;
8. x[i] = 1;
9. return (Satisfy-one (v.high,x));
}

Figure 3.9: Implementation of Satisfy-one

Figure 3.10 represent the traversal of the Satisfy-one algorithm. The purpose of this algorithm as it was mentioned before is finding the first terminal with value 1.
3.2.5 The Satisfy-all Operation
For implementing this method, the procedure shown in figure 3.11 [3] is used. This
algorithm starts from the root with index 1 and represents all paths to reach value one by
the array that will be initialized during the Satisfy-all process. This algorithm uses several
recursive calls to find paths which reach to one. Figure 3.12 illustrates the Satisfy-all
algorithm.

Algorithm Satisfy-all(i: integer; v: vertex; x: array[1..n] of integer) {
  1. if (v.value == 0 )
  2. return; {failure}
  3. if (i == n+1 && v.value == 1){
      3.1 Show element x[1],…,x[n] on console;
      3.2 return;
    }
  4. if (v.index > i){
      4.1 x[i] = 0; Satisfy-all(i+1,v,x);
      4.2 x[i] = 1; Satisfy-all(i+1,v,x);
    }
  5. else{
5.1 \( x[i] = 0; \text{Satisfy-all}(i+1, v.\text{low}, x); \)
5.2 \( x[i] = 1; \text{Satisfy-all}(i+1, v.\text{high}, x); \)
}

Figure 3.11: Implementation of Satisfy-all

Figure 3.12 illustrates the traversal of Satisfy-all algorithm. This algorithm prints all the paths from root to terminals with value 1 on console.

3.3 BitSet Representation

“BitSet class (java.util.BitSet) implements a vector of bits”. [10] There is a Boolean attribute for all elements of BitSet that is used as value. A new BitSet can be created by applying a logical operation such as AND, OR and XOR over two other BitSets. Number of bits that currently are in use by the bit set is flexible, it depends on the implementation. When a BitSet is created all the Boolean values are set to false, later on by applying other logical operations on BitSet the Boolean values may change.

As it was mentioned before BitSet implementation has been shown very efficient for storing large sets of data in terms of memory and time. There are different implementations of BitSet. One of the efficient implementations has been developed before [11, 16].

All the operations that are applicable on OBBDs are applicable here also, like AND, OR, and ANDNOT. But the implementation is different. These Boolean operators called Bitwise operators.
3.4 Summary
In this section it is explained why based on theoretical analysis OBDDs are expected to be efficient during the process of saving enormous amount of data.

Imagine we want to store for example 1000 different sets of objects. In this approach it is not created a specific OBDD for each set of objects, but instead one OBDD has been created and then all the given sets as different sub-graphs have been added to this OBDD.

During the process of creating sub-graphs for different sets, the chance of having repetitive sets are high. As it was mentioned before reduce algorithm is used for decreasing the memory. Reduce algorithm removes isomorphic sub-graphs and redundant vertices. For all equal sub-graphs the reduce algorithm gives the reference to the first sub-graph that has been created. So no matter how many equal sets and sub-graphs are in OBDD, reduce algorithm will just keep one of them and for the rest the algorithm gives reference to the remained sub-graph. By use of points-to analysis, garbage collector collects the memory of these de-allocated objects and results in memory reduction.

In comparison BitSet implementation creates all the given sets as separate vectors. For each set it creates one single vector and reduce algorithm is applied on every single set separately.

Therefore, on theory it is expected that OBDD implementation should be an efficient way for storing large sets of information.
4 Implementation
In this chapter the implementations of OBDD graph has been explained. The class hierarchy which is used in this implementation is represented also.

4.1 Total Overview
This implementation consists of three packages, obdd package, operator package and test package.

Obdd package is the main package. It includes the creation part, how the OBBDs has been created, how they are stored and how algorithms have been implemented. It is also represents how implemented algorithms are used. It also includes lattice and power lattice interfaces and their implementations.

Another one is operators package that includes implementations of different operators that are used in Boolean functions. The last one is named test. It is used for testing the graphs and creating the large sets making the huge OBDDs and measures the time and memory of these significant OBDDs. Test package includes both main and JUnit classes for testing the OBBD. Figure 4.1 shows package dependencies in this implementation.

Figure 4.1: Packages dependencies
In this thesis the vertex class is a vital part of the implementation. To represent the graph and show all the characteristics the vertex is used. Each vertex includes different attributes that explain the graph and its characteristics.

There is another class named Leaf. This class has been used for showing the terminal vertices. They are special type of vertices. Leaf class extends Vertex class and inherits all its features. Vertex class has a constructor that Leaf class will use it as a super constructor.

As it was discussed before all terminal vertices have value 1 or 0, so the leaf class needs a constructor to set the value of terminals.

4.1.2 OBDD Creation
Class OBDDGraph has some different kinds of methods that they are used for creating different types of OBDDs like Collections, array and a single object.

First of all it has a constructor that has 3 arguments as input. The inputs are an array of objects (consisting of all elements that obdd will be formed of), the total number of graphs and the maximum depth of graph. To make it more clear, they will be explained one by one.

- Array of Objects: It is an array filled with all the elements that the OBDD will be formed of. It is called top-set.

- Total Number of Graphs: In this implementation a different approach comparing to BitSet implementation has been used. Before when they want to create for example 3 different OBDDs they create 1 separate graph for each given set, but in this thesis an OBDD will be created and all the given sets will be added to this OBDD. Every OBDD or sub-graph has a special id to make them different from each other.

- Maximum depth of graph: Before creating the graphs it is necessary to know the biggest element of the graph and based on this biggest element decide what the maximum depth of graph is. Suppose the top set (all input elements) is \{1, 2, 3, 4, 5, 6,\}. Here the biggest element is 6 and the binary form of it is: 110 so the length is 3. Now length of all the other elements of top set will be 3, if the length of any of them in binary form is less than 3, this implementation automatically will add enough 0 at the beginning of all needed elements to make all the elements length the same.

Considering the 6 as the biggest element and its length, the binary presentation for the rest of the top set would look like table 3.1.

It is needed to do that because in some of the algorithms, for instance reduce algorithm is working based on the index that each vertex has. The index (depth) for each vertex is set during the process of creation elements in the set. For example if the binary representation of 1 does not change to 001 (the real binary representation of 1 is 1, we add two 0s at the beginning of it to make all the lengths equal), then index of terminal vertex would be 2. However the binary representation for 6 is 110 and consequently the terminal’s index will
be 4. So the terminal index for these two elements would be different, in spite of the fact the all terminal vertices have an equal index.

<table>
<thead>
<tr>
<th>Element</th>
<th>Actual Binary Form</th>
<th>Required Binary Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>001</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>010</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>011</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>101</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>110</td>
</tr>
</tbody>
</table>

Table 4.1: The binary form of set \{1, 2, 3, 4, 5, 6\}

Now imagine we want to create an OBDD with two sets: \{1,2\} and \{3,4\}. The top set is: \{1, 2, 3, 4\}. Each set shows the elements of one sub-graph. Figure 4.2 shows the OBDD representation. As figure 4.2 represents every sub-graph has a unique id. The ids have been shown with orange color. The id for set \{1, 2\} is 0 and for \{3, 4\} is 1. Root of each sub-graph has a blue color. The roots are recursively v2, v6. That is the reason that is needed to know the total number of OBDD graphs (for giving a specific id to each set). For example for the first sub-graph the id is 0, for the second one the id will be 1 and for the nth sub-graph the id is n-1. The id of each graph is converted to binary form also. If it is asked to create 10 OBDDs so the id of last one is 10 – 1 = 9, and 9 in binary system is: 1001. The rule of adding 0 at the beginning of id works here also. So the id for the first sub-graph is 0, the binary form is: 0, after adding 0s it will be: 0000.

Figure 4.3 illustrates the reduced form of this graph. In figure 4.3 there are more than one terminal vertex with value 0, that in reality it is not like that. In real form there is just one vertex with value 0. Here to prevent having a complex figure it is preferred to have more than one terminal vertex with value 0(just in the picture, not reality). Also the right child of v7 goes to v4 that is in left sub-graph of OBDD. But again in this picture is just put the vertex v4 as right child of v7 to prevent complexity.

As it was described before there are several create methods in OBBDGraph class, that they have been used for different types of inputs.

There are two kinds of inputs for creating OBDDs:

- int: All the elements of graph in this part are instances of int (positive or negative digits). It could be array of integers or Collections, like array list or hash map, that include elements of int.
- Object: The inputs here are all instances of Objects, they could be Integer, String or even Object of any class. This group of inputs also could be array or Collection of elements.

![Diagram of OBDD for sets {1, 2} and {3, 4}]

Figure 4.2: Representation of OBDD for sets {1, 2} and {3, 4}

So that is the reason of using different methods for creation.

OBDD is a binary graph, so every non-terminal vertex has two children. During the creation of graph some non-terminal nodes might just have one child, for these kinds of vertices after creation the whole OBDD, another terminal child with value 0 must be added.

For adding these terminal vertices to the graph there is a special method that traverse the whole OBDD and checks all the vertices if they have one child or not. For all terminals with one child the method adds a terminal node with value 0 to all those vertices.

All the algorithms, such as reduce, traverse, that have been described before in chapter 3 are implemented in this class. There are some difference in apply algorithm implementation regarding this thesis approach, that it will be explained in details in next section.

4.1.3 Apply Implementation

Apply algorithm have been discussed in section 3.2.3 of chapter 3, but its implementation is slightly different that it is explained in this chapter.

As it has been written before, in this thesis another approach is used for creating the OBDDs. In this approach all different OBBDs are added a big graph. This changes affect apply algorithms also.
Based on apply algorithm in figure 3.6, the result of applying two different OBBDs will be a new OBDD graph. But in this apply implementation the changes will be added to the first OBDD. For example if we want to apply graph \(g_1\) and graph \(g_2\) together, the result graph based on the apply algorithm would be a new graph \(g_3\) depending on the operator. But in this approach the changes will be saved in the first graph, here graph \(g_1\).

![Reduced graph figure 4.2](image)

Figure 4.3: Reduced graph figure 4.2

The nodes that are the same will be kept, the ones that are not needed any more will be removed and some vertices will be created for the new ones that are necessary to be in the graph.

Suppose that two vertices from graphs \(g_1\) and \(g_2\) are applied together. If the vertex from \(g_1\) is a terminal with value 1 and the other vertex from graph \(g_2\) is a terminal with value 0, and the operator is AND, the result vertex is a terminal vertex with value 0. So the value of vertex in graph \(g_1\) will be changed to 0. But sometimes it is necessary to create a new terminal or non-terminal vertex, depending on the operator, no matter what the type of vertex was before. Always after finishing apply algorithm the reduction method is called.

During this process some attributes of vertices are changed, so it is needed to reset them. Doing a traverse over the OBDD and reset the characteristics that are necessary would be an appropriate solution.

### 4.2 Lattice

An arranged set of objects that every two elements in this set have a specific “lower bound” or meet and “upper bound” or join is a lattice[12].

24
A lattice set is known for two operation join (\(\lor\)) and meet (\(\land\)).

- Binary join: For joining two members of Lattice Set \(\{x, y\}\) the join will be: \(x \lor y = \text{sup} \{x, y\}\). The result is set with all members of \(x\) and \(y\). Another name for join is "upper bound".

- Binary meet: For meeting two members of Lattice Set \(\{x, y\}\), the meet will be: \(x \land y = \text{inf} \{x, y\}\). The result is a set with common members of \(x\) and \(y\). Another name for meet is "lower bound".

If a lattice has an “upper bound” it will be shown by \(\top\) (called top in this implementation) and if has a “lower bound” will be presented by \(\bot\) (called bottom in this implementation).

Every set can be considered as a lattice if and only if all the included sets have both join and meet operators. This fact is applied to empty sets also.

Based on mathematical rules join of every set with top is top and meet of bottom with every set is bottom. These rules are implemented and used in this thesis also.

A set that includes all sets and sub-sets is a power set. If \(|\text{Lattice}| = n\), then the number of all sub-sets is: \(2^n\).

Lattice is a data structure for storing different kinds of sets, even an empty set(bottom). Lattice is used in both OBDD and BitSet to store different sets. Lattice can be used to keep track of all stored sets (sub-sets). After this mathematical introduction about Lattice and Power Lattice, the implementation will be explained in next sections.

### 4.2.2 OBDD Lattice

There is an overview of class diagram and dependencies in figure 4.4.

OBDD Lattice class implements below interface, named Power Lattice Interface.

**Power Lattice Interface**

- `LatticeSet getTop()`
- `LatticeSet getBot()`
- `int countElements()`
- `Iterator iterator()`
- `LatticeSet create(Object single_element)`
- `LatticeSet create(Collection elements)`
- `Boolean equals(LatticeSet s1, LatticeSet s2)`
- `LatticeSet join(LatticeSet s1, LatticeSet s2)`
- `LatticeSet meet(LatticeSet s1, LatticeSet s2)`
- `LatticeSet diff(LatticeSet s1, LatticeSet s2)`

OBBD Lattice class is a class that consists of our OBDD and top-set (set of all elements for creating the whole OBDD). We use the create methods for creating subgraphs. OBBDLattice class use OBBDSet class as well as OBBDGraph class. OBBDLattice contains the set operations such as join, meet and difference for applying
Boolean functions over lattice. All the methods that have been written above as Power Lattice Interface are implemented in this class and all of them have been used.

When we want to create a new graph, we check if this given set has already been created or not.

There is another interface, named LatticeSet that is the return type of some methods in Power Lattice Interface. This interface is implemented by OBBDSet class. It is a container for each set; we can measure the size of the set, iterate over it or apply Boolean functions on a set. OBBDSet class implements another interface, named LatticeValue. You can read them in the next page for more details.

**LatticeSet Interface**
- `LatticeSet intersection(Latticeset s)`
- `LatticeSet union(Latticeset s)`
- `LatticeSet diff(Latticeset s)`
- `int size()`
- `Boolean isEmpty()`
- `Boolean contains(Object o)`
- `Iterator iterator()`
- `Object getSingelton()`
- `Boolean addElement()`

**LatticeValue Interface**
- `Boolean equalValues(LatticeValue v)`
- `String toString()`
Figure 4.4: Class diagram and dependencies
5 Experimental Results
In this chapter we will practically present the results of the benchmarks for different size of sets and compare it with BitSet Implementation [16]. This implementation is part of a big package named Grail.

5.1 Metric
One of the goals of this thesis is comparing our results to BitSet implementations. The main metrics here are memory and time. These mentioned metrics are explained in details:

- Memory: A good approach for calculating the memory consumption is using benchmarks and JUnit for different size of sets and mostly for significant sets of data. For calculating the memory first a runtime command has been used and then free memory has been subtracted from total memory to get the actual consumed memory.

- Time: For measuring the execution time specific commands have been put before and after each method that we want to have its execution time. We measured the system’s time before and after each method, and then we subtracted these two numbers to have the execution time.

5.2 Results
Two different charts have been illustrated in this chapter, memory and time. These two charts belong to OBDD implementation and BitSet implementation.

During this thesis we tried different approaches for creating the OBBD in order to achieve better results. The first group of memory and time charts represents the consumed memory and execution time of our first approach. But they are not the final results.

It should be mentioned also that each OBDD consists of different numbers of sets (subgraphs). The number of sets is different from 100 up to 2000. Each set has a random number of elements. For generating the elements of each set, a method that randomly generates objects has been used. The size of set is depends on the random number of elements that the method generates. It is a random number between 50 and maximum number of sets in the OBDD.

Table 5.1 shows our first approach results for execution time and charts 5.1 represents the chart of table 5.1. As it is shown in chart 5.1, there is a time leaking in set with size 1400. When we changed the approach we could solve this problem also.
<table>
<thead>
<tr>
<th>Number of Sets</th>
<th>Memory (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.4009</td>
</tr>
<tr>
<td>200</td>
<td>0.7147</td>
</tr>
<tr>
<td>300</td>
<td>1.0624</td>
</tr>
<tr>
<td>400</td>
<td>1.5045</td>
</tr>
<tr>
<td>500</td>
<td>2.0904</td>
</tr>
<tr>
<td>600</td>
<td>2.5804</td>
</tr>
<tr>
<td>700</td>
<td>3.3275</td>
</tr>
<tr>
<td>800</td>
<td>4.2055</td>
</tr>
<tr>
<td>900</td>
<td>4.6914</td>
</tr>
<tr>
<td>1000</td>
<td>5.9449</td>
</tr>
<tr>
<td>1100</td>
<td>6.7916</td>
</tr>
<tr>
<td>1200</td>
<td>8.2054</td>
</tr>
<tr>
<td>1300</td>
<td>9.3887</td>
</tr>
<tr>
<td>1400</td>
<td>10.0147</td>
</tr>
<tr>
<td>1500</td>
<td>12.1989</td>
</tr>
</tbody>
</table>

Table 5.1: First Result of OBBD for Memory Consumption

 ![Chart 5.1: Chart of table 5.2](image-url)
<table>
<thead>
<tr>
<th>Number of Sets</th>
<th>Time(S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.148</td>
</tr>
<tr>
<td>200</td>
<td>1.41</td>
</tr>
<tr>
<td>300</td>
<td>6.854</td>
</tr>
<tr>
<td>400</td>
<td>16.207</td>
</tr>
<tr>
<td>500</td>
<td>47.056</td>
</tr>
<tr>
<td>600</td>
<td>99.812</td>
</tr>
<tr>
<td>700</td>
<td>176.431</td>
</tr>
<tr>
<td>800</td>
<td>310.729</td>
</tr>
<tr>
<td>900</td>
<td>457.337</td>
</tr>
<tr>
<td>1000</td>
<td>712.962</td>
</tr>
<tr>
<td>1100</td>
<td>1040.037</td>
</tr>
<tr>
<td>1200</td>
<td>2779.597</td>
</tr>
<tr>
<td>1300</td>
<td>4265.341</td>
</tr>
<tr>
<td>1400</td>
<td>4536.109</td>
</tr>
<tr>
<td>1500</td>
<td>4726.152</td>
</tr>
</tbody>
</table>

Table 5.2: First Result of OBBD for Execution Time

As it is represented in chart 5.2 there a sudden, unexpected increase in the time after 1000. One of the reasons could be that adding enormous amount of data at once would take a lot of time to execute and build the graph. By changing the approach it will becomes possible to solve this issue in future. In current approach the reduce algorithm has been
applied to the whole graph after adding all the elements(sets) to the graph. But in next approach after adding or creating each set(sub-graph) the reduce algorithm has been applied to the graph.

Table 5.1 represents the memory consumption of this group. Chart 5.1 illustrates the chart of table 5.1.

Number of sets for the first group was 1500 which is different from the other test cases. The number of sets in next groups is 2000. As it was explained before and is shown in chart 5.2 after set number 1500 the execution time was that big that it was impossible and meaningless to proceed.

The second group of charts is the final results of OBBD after trying to improve our approach. Table 5.3 represent the memory consumption, consequently chart 5.3 illustrated the memory consumption of table 5.3. Table 5.4 represent the execution time, consequently chart 5.4 illustrated the time consumption of table 5.4.

<table>
<thead>
<tr>
<th>Number of Sets</th>
<th>Memory(MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.370</td>
</tr>
<tr>
<td>200</td>
<td>0.535</td>
</tr>
<tr>
<td>300</td>
<td>0.746</td>
</tr>
<tr>
<td>400</td>
<td>0.960</td>
</tr>
<tr>
<td>500</td>
<td>1.258</td>
</tr>
<tr>
<td>600</td>
<td>1.629</td>
</tr>
<tr>
<td>700</td>
<td>1.994</td>
</tr>
<tr>
<td>800</td>
<td>2.343</td>
</tr>
<tr>
<td>900</td>
<td>2.838</td>
</tr>
<tr>
<td>1000</td>
<td>3.200</td>
</tr>
<tr>
<td>1100</td>
<td>3.865</td>
</tr>
<tr>
<td>1200</td>
<td>4.587</td>
</tr>
<tr>
<td>1300</td>
<td>5.032</td>
</tr>
<tr>
<td>1400</td>
<td>5.562</td>
</tr>
<tr>
<td>1500</td>
<td>6.230</td>
</tr>
<tr>
<td>1600</td>
<td>6.877</td>
</tr>
<tr>
<td>1700</td>
<td>7.714</td>
</tr>
<tr>
<td>1800</td>
<td>8.628</td>
</tr>
<tr>
<td>1900</td>
<td>9.128</td>
</tr>
<tr>
<td>2000</td>
<td>10.054</td>
</tr>
</tbody>
</table>

Table 5.3: Set Size-Memory (OBDD)
It was expected to get a more straight line here for memory consumption and also to have a same amount of memory consumption for different number of sets after one special number (set number). However it was not possible to reach this point, but there is a difference between chart 5.2 and 5.3 which they both show the memory consumption for different approaches.

<table>
<thead>
<tr>
<th>Number of Sets</th>
<th>Time(S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.042</td>
</tr>
<tr>
<td>200</td>
<td>0.279</td>
</tr>
<tr>
<td>300</td>
<td>0.863</td>
</tr>
<tr>
<td>400</td>
<td>1.815</td>
</tr>
<tr>
<td>500</td>
<td>3.598</td>
</tr>
<tr>
<td>600</td>
<td>6.738</td>
</tr>
<tr>
<td>700</td>
<td>9.951</td>
</tr>
<tr>
<td>800</td>
<td>14.011</td>
</tr>
<tr>
<td>900</td>
<td>22.076</td>
</tr>
<tr>
<td>1000</td>
<td>26.028</td>
</tr>
<tr>
<td>1100</td>
<td>38.289</td>
</tr>
<tr>
<td>1200</td>
<td>54.933</td>
</tr>
<tr>
<td>1300</td>
<td>67.52</td>
</tr>
<tr>
<td>1400</td>
<td>80.262</td>
</tr>
<tr>
<td>1500</td>
<td>99.62</td>
</tr>
<tr>
<td>1600</td>
<td>118.701</td>
</tr>
<tr>
<td>1700</td>
<td>151.306</td>
</tr>
<tr>
<td>1800</td>
<td>175.637</td>
</tr>
<tr>
<td>1900</td>
<td>201.479</td>
</tr>
<tr>
<td>2000</td>
<td>243.293</td>
</tr>
</tbody>
</table>

Table 5.4: Set Size-Time (OBDD)
Comparing charts 5.2 and 5.4, shows that we could manage to decrease the execution
time more than 45 times. By comparing two charts 5.1 and 5.3 we can see we reduce the
memory consumption 2 times. Table 5.5 shows the memory consumption of BitSet
implementation. The chart of this table is represented in chart 5.5.

<table>
<thead>
<tr>
<th>Number of Sets</th>
<th>Memory (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.12</td>
</tr>
<tr>
<td>200</td>
<td>0.27</td>
</tr>
<tr>
<td>300</td>
<td>0.32</td>
</tr>
<tr>
<td>400</td>
<td>0.37</td>
</tr>
<tr>
<td>500</td>
<td>0.421</td>
</tr>
<tr>
<td>600</td>
<td>0.485</td>
</tr>
<tr>
<td>700</td>
<td>0.543</td>
</tr>
<tr>
<td>800</td>
<td>0.629</td>
</tr>
<tr>
<td>900</td>
<td>0.705</td>
</tr>
<tr>
<td>1000</td>
<td>0.77</td>
</tr>
<tr>
<td>1100</td>
<td>0.88</td>
</tr>
<tr>
<td>1200</td>
<td>0.965</td>
</tr>
<tr>
<td>1300</td>
<td>1.04</td>
</tr>
<tr>
<td>1400</td>
<td>1.14</td>
</tr>
<tr>
<td>1500</td>
<td>1.25</td>
</tr>
<tr>
<td>1600</td>
<td>1.36</td>
</tr>
<tr>
<td>1700</td>
<td>1.46</td>
</tr>
<tr>
<td>1800</td>
<td>1.52</td>
</tr>
<tr>
<td>1900</td>
<td>1.63</td>
</tr>
<tr>
<td>2000</td>
<td>1.75</td>
</tr>
</tbody>
</table>

Table 5.5: Set Size-Memory (BitSet)
Chart 5.2 shows the memory-size result of BitSet Implementation.

![Chart of Table 5.2](image-url)

<table>
<thead>
<tr>
<th>Number of Sets</th>
<th>Time(S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.009</td>
</tr>
<tr>
<td>200</td>
<td>0.009</td>
</tr>
<tr>
<td>300</td>
<td>0.012</td>
</tr>
<tr>
<td>400</td>
<td>0.016</td>
</tr>
<tr>
<td>500</td>
<td>0.022</td>
</tr>
<tr>
<td>600</td>
<td>0.029</td>
</tr>
<tr>
<td>700</td>
<td>0.543</td>
</tr>
<tr>
<td>800</td>
<td>0.629</td>
</tr>
<tr>
<td>900</td>
<td>0.705</td>
</tr>
<tr>
<td>1000</td>
<td>0.77</td>
</tr>
<tr>
<td>1100</td>
<td>0.88</td>
</tr>
<tr>
<td>1200</td>
<td>0.965</td>
</tr>
<tr>
<td>1300</td>
<td>1.04</td>
</tr>
<tr>
<td>1400</td>
<td>1.14</td>
</tr>
<tr>
<td>1500</td>
<td>1.25</td>
</tr>
<tr>
<td>1600</td>
<td>1.33</td>
</tr>
<tr>
<td>1700</td>
<td>1.48</td>
</tr>
<tr>
<td>1800</td>
<td>1.64</td>
</tr>
<tr>
<td>1900</td>
<td>1.76</td>
</tr>
<tr>
<td>2000</td>
<td>1.95</td>
</tr>
</tbody>
</table>

Table 5.6: Set Size-Time (BitSet)
Considering the charts above, the memory usage and execution time in BitSet Implementation are lower than used memory and consumed execution time in OBDD implementation.

Based on theoretical analysis we expected to reach a point that memory consumption after that follow a same value for all different sets.

As it was mentioned before we tried different approaches such as reduce the whole OBDD after adding each new set to OBDD or reduce each set individually and we could manage to reduce the memory and specially time, but still the results are not comparable to BitSet.

The reasons of results differences between OBDD implementation and BitSet implementation could be:

- Lack of theoretical analysis for finding efficient approach for implementing OBDDs
- Not efficient implementation of prototype
- The only goal of thesis was not to make the memory consumption and execution time lower than BitSet. First goal was to create the OBBD and its set of manipulation of algorithms.
6 Conclusion and Future Work

In this final chapter, the thesis will be concluded and the works that should be done in the future will be explained.

6.1 Conclusion

Memory consumption has always been an important issue during storing enormous sets of data. There are plenty of efficient approaches for saving large sets of information.

Based on theoretical analysis Ordered Binary Decision Diagrams (OBDDs) have been shown very effective in terms of memory usage and time execution. This thesis is focused on implementation of OBDDs for capturing and manipulating different sizes of sets. Then we compare OBDDs memory and time results to BitSet implementation [16].

As the first part of this thesis theoretical analysis was done to create an OBDD and a set of manipulation algorithms. Then OBDD and its algorithms have been implemented like reduce, Apply, satisfy-all, etc. Reduce algorithm can be considered as the most important one in terms of decreasing memory and time.

As the next step we did plenty of benchmarks for capturing memory consumption of OBBD for different sets of data. During this process we tried different approaches to get the more efficient results. In chapter 5 there are different charts of memory and time for 2 different approaches. Charts 5.1 (memory) and 5.2 (time) are belong to the first approach. It is discussed before that time consumption was so high that it was needed to take a new approach. Charts 5.3 and 5.4 show the new results for memory and time respectively.

The final part was to comparing OBDD’s time and memory results to BitSet’s memory consumption and execution time. First we measured the memory consumption and execution time of BitSet which they have been shown in charts 5.5 and 5.6 respectively.

The results were not the same as we expected based on theoretical analysis. We changed our approach and we could succeed to reduce memory consumption and execution time (Charts 5.3 and 5.4), but still the results were not as improved as we expected.

However these theoretical analysis and OBBD implementation gave us a good idea to do a better theoretical analysis and find an efficient approach for the future.

6.2 Future Work

These results can help us to get new insights about how to find a more efficient approach for implementing OBDDs. In future we can theoretically analyze the OBDDs in details and discover a new and effective implementation for OBBD and its set of manipulation algorithms. After theoretical implications we can have a better idea to create an OBDD and implement its set of manipulation algorithms in an efficient way. These results will be used and compared to the future implementation of OBDD’s results.
References


[16] An efficient implementation for BitSet has been developed in Linnaeus University by Software Technology group.
