Visualization of Binary Decision Diagram

1. Contents

1. Introduction

2. Background and Related Work

2.1 Formal Definition of Binary Decision Diagram

A Binary Decision Diagram (BDD) [BRY86] is a rooted, directed acyclic graph (DAG), *G(V,E).* The vertex set is made up two different types of vertices, terminal and non-terminal. Each BDD has one or two terminal vertices with out-degree zero, and multiple non-terminal vertices. A terminal vertex v, has attribute $value(v) \in \{0,1\}$. A non-terminal vertex v has attribute $index(v) \in \{1,2,..,n\}$ and two children $low(v), high(v) \in V$. The non-terminal vertex attribute *index(v)* implies a linear ordering of the variable ordering of the BDD, and it satisfies the property such that for any non-terminal vertex v, $index(v) < \{ index(low(v)), index(high(v)) \}$.

A Reduced Binary Decision Diagram (ROBDD) is a BDD with two additional constraints. Before we introduce these constraints, we will review the definition of *isomorphic sub-graphs*. Two sub-graphs rooted at *v* and *w*, *G'(v)* and *G''(w)* respectively are *isomorphic* if there exists a one-to-one mapping σ, from the vertices in *G'(v)* and *G''(w)* such that for every vertex v' in G', there is a vertex $w' = \sigma(v')$ which satisfies the following condition; either both v' and w' are terminal vertices with $value(v') = value(w')$ or v' and w' are both non-terminal vertices with $index(v') = index(w')$, $\sigma(low(v')) = low(w')$ and $\sigma(high(v')) = high(w')$. The above-mentioned constraints are presented as such; for *G(V,E)* to qualify as an ROBDD, there is no vertex $v \in V$ such that low(v) = high(v) nor does *G(V,E)* contain two distinct vertices *v* and *w* such that the sub-graphs rooted at v and w are isomorphic. For the rest of this report, BDD and ROBDD are used interchangeably to denote ROBDD unless stated otherwise.

A BDD rooted at vertex v, which denotes a function f_v is defined recursively as such;

- If v is a terminal vertex such that $value(v) = 0$, $f_v = 0$, else If v is a terminal vertex such that $value(v) = 1$, $f_v = 1$
- If v is a non-terminal vertex such that $index(v) = i$, then $f_v = x_i f_{high(v)} + x_i' f_{low(v)}$.

The definition given above describes a bottom-up construction of a BDD. It is also descriptive to provide a top-down definition of BDD as representation of Boolean function based on Shannon's Decomposition Theorem, which states that

$$f = x_i \cdot f_{xi} + x_i' \cdot f_{xi'}$$

where f_{xi} and $f_{xi'}$ are *f* evaluated at $x_i = 1$ and $x_i = 0$ respectively. This theorem allows us to associate a unique Boolean function with each vertex in the BDD *G(V,E)* such that the root of G represents the *main* function for which the BDD was constructed, and every other vertex is defined such that $high(f) = f_{xi}$ and $low(f) = f_{xi'}$. Furthermore, if v is the node that is associated with the function *f* stated above, $index(v) = i$.

Shannon's decomposition theorem also provides the foundation for the recursive *if-then-else* (*ite*) [BRB90] formulation of BDD. The *ite* operator is defined as follows

$$ite(f,g,h) = f.g + f'.h$$

where *f, g, h* are arbitrary Boolean functions. It is a well-known fact that *ite* operator can be used to implement all Boolean functions which operate on 2 variables, as shown in Table 1.

Table	Name	Expression	Equivalent Form
0000	0	0	0
0001	AND(*f,g)*	*f.g*	*ite(f,g,0)*
0010	$f > g$	$f.g'$	*ite(f,g',0)*
0011	*f*	*f*	*f*
0100	$f < g$	$f'.g$	*ite(f,0,g)*
0101	*g*	*g*	*g*
0110	XOR(*f,g*)	$f \oplus g$	*ite(f,g',g)*
0111	OR(*f,g*)	$f + g$	*ite(f,1,g)*
1000	NOR(*f,g*)	$(f + g)'$	*ite(f,0,g')*
1001	XNOR(*f,g*)	$(f \oplus g)'$	*ite(f,g,g')*
1010	NOT(*f,g*)	g'	*ite(g,0,1)*
1011	$f \geq g$	$f + g'$	*ite(f,1,g')*
1100	NOT(*f*)	f'	*ite(f,0,1)*
1101	$f \leq g$	$f' + g$	*ite(f,g,1)*
1110	NAND(*f,g*)	$(f.g)'$	*ite(f,g',1)*
1111	1	1	*1*

Table 1. Two Argument Operators Expressed in terms or *ite*

Let Z = *ite*(*f*,*g*,*h*) and let *v* be the top variable of functions *f, g, h.* Then

$$
\begin{aligned}
Z &= ite(f,g,h) \\
&= v.Z_v + v'.Z_{v'} \qquad \text{(Shannon Decomposition Theorem)} \\
&= v.(f.g + f'.h)_v + v'.(f.g + f'.h)_{v'} \\
&= v.(f_v.g_v + f'_v.h_v) + v'.(f_{v'}.g_{v'} + f'_{v'}.h_{v'}) \\
&= ite(v,\ ite(f_v,g_v,h_v),\ ite(f_{v'},g_{v'},h_{v'}))
\end{aligned}
$$

The terminal cases of this recursion are:

$$ite(1,f,g) = ite(0,g,f) = ite(f,1,0) = ite(g,f,f) = f$$

The pseudo-code for *ite-based* algorithm is presented in Appendix XXX .

2.2 Analysis of BDD packages

We will now investigate some important issues that have to be considered when implementing an efficient BDD package [BRB90], in terms of both memory and CPU requirements. The most efficient and hence popular implementation of a BDD package is based on the recursive *ite* formulation of BDDs described in Section 2.1 above. Table 2 summarizes some important fundamental concepts and data structures used in these implementations.

Concepts and Data Structures	Brief Description and Analysis
Shared BDD	• A multiple-output Boolean function is represented as a single multi-rooted BDD with a root for each function that we are explicitly interested in. • This concept is justified by the assumption that different Boolean functions have many common sub-expressions, and representing these sub-expressions using one sub-graph rather than multiple identical sub-graphs reduces memory requirements.
Unique Table	• A dictionary of all functions represented in the current DAG. • Implemented as a *Hashtable* with collision chains. • (*key, object*) of the *Hashtable* is ((*index(v), low(v), high(v)) , *v*) such that **v* is the pointer to a BDD vertex. • Collision chain is implemented as an additional entry in the BDD vertex data structure. • Guarantee that at *any time* there is no isomorphic (sub)graphs in the DAG. Maintains *strong canonicity* such that any two equivalent function always share the same sub-graph.
Computed Table	• Implemented as a Hash-based Cache. • Stores results of recent computations , i.e. informs the package if a certain *ite(f,g,h)* has recently been computed - and if so return the resultant vertex which represents the function *result = ite(f,g,h)*. • Reduces the complexity of the *ite* recursions from exponential in the size of input variables to polynomial in the size of operand sub-graphs.
BDD vertex	• Can be efficiently implemented using as few as 4 memory *word**. reference-count ‖ *index(v)*, ** high(v)*, **low(v)* and **next* • **next* implements the collision chain of the Unique Table • reference-count variable will be discussed in conjunction with garbage collection • Each vertex v, represents a distinct Boolean function $f(x_i,x_{i+1},..,x_n)$ where *index(v) = i.*
Garbage Collection	• Automatic garbage collection is based on the reference-count variable introduced above. When an upper threshold of BDD vertices have reference-count = 0, this mechanism is invoked which released the memory occupied by these dead vertices, and appropriately update (or rehash if necessary) the computed table and the unique table. • The reference-count of a vertex *v* maintains the sum of all other BDD vertices or user-defined functions which reference *v*. *v* is a

	dead vertex if its reference-count is zero. • The CPU cost garbage collection is amortized over all vertices which are freed.

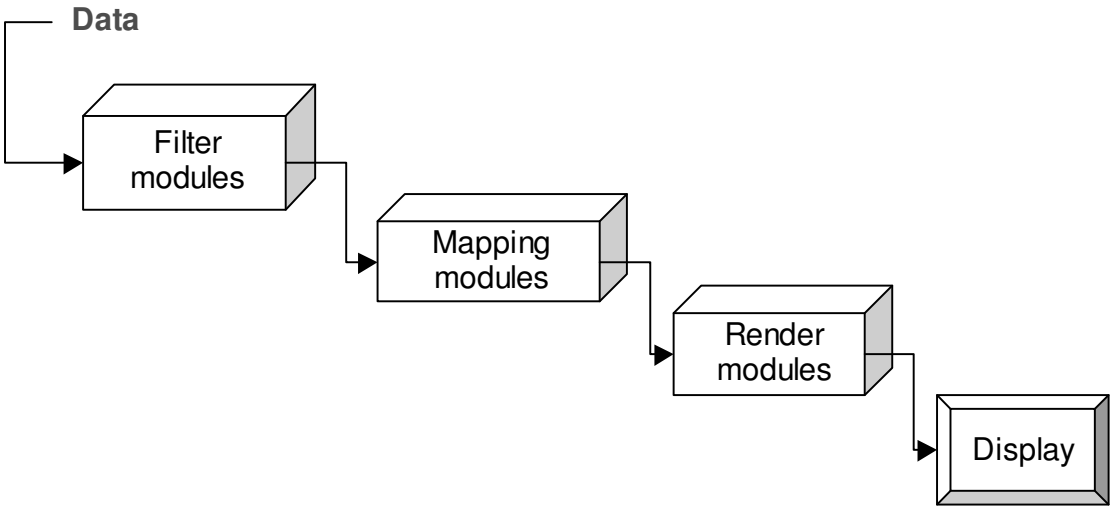
2.3 Variable Ordering of BDDs

A BDD is essentially an effort to arrive at an optimal trade-off between efficient memory usage for representing Boolean functions and effective algorithmic (in terms of time-complexity) to manipulate these functions. The effectiveness of BDD manipulation algorithms is directly related to the size of the BDD DAGs. Based on the definition of BDD, it is easy to see that the size of its DAG and hence the complexity of its algorithms are exponential in the worst case - this problem is commonly referred to as the "memory explosion" problem. However, in many practical cases, we can find *good* variable orderings for which the resultant BDDs have sizes that are polynomial in the number of its input variables. Figure XXX illustrates the importance of a good variable ordering for BDDs. (draw the figure after I have coded in the BDD-dump with software control stuff - look at pg 296 Symbolic Boolean ... Bryant paper)

Figure XXX shows the BDDs of the function $f = a_1b_1 + a_2b_2 + a_3b_3$ under two different BDD variable orderings. The first ordering $a_1 < b_1 < a_2 < b_2 < a_3 < b_3$, produces a BDD which is linear in the number of its input variables whereas the second $a_1 < a_2 < a_3 < b_1 < b_2 < b_3$ produces a BDD which is exponential in the number of its input variables.

Unfortunately the problem of finding an optimal variable ordering is co-NP complete [FS87].

In order for BDD to be efficient, we need to avoid the variable orderings that results in exponential BDDs. [BYR86] have shown that for ALU func63 0 0 1 384.84 461.24 Tm (3)Tj R12 11.6798 T



the visualization process is described is described by a series of computation modules (Figure XX).

Filtering: A process whereby raw data is transformed into data of "interest". Some examples of computation are interpolation, smoothing operations, data extractions etc.

Mapping: A set of modules that build geometric representation of the data.

Rendering: The geometric representation of data is converted to an image that can be displayed

Application developers *pick* from a library of modules, each categorized to one of the three types above, and "wires" them together into a network. In this system, a modules reads in data in a format which it understands and presents its output to a subsequent module. Data flows through the network, from the filter modules, through the mapping modules and finally to the render modules – hence the term "data flow" architecture.

Scalar Mapping Technique

The scalar mapping technique [HDB92] introduced by Hibbard *et al* performs visualization of a data-set based on user-defined mapping functions σ_i (s). Before we define these mapping functions, we will discuss some fundamental concepts of this visualization system. The system defines a finite set of primitive data types. All primitive data types are scalar types. However, the system provides a mechanism for user to specify data types as a hierarchical composition of elements from the set of system primitives. Graphical display as well as user interaction with the display is modeled as a special data set which is in turn composed from a finite set of primitive display scalar types. Currently supported primitive scalar types primitive display scalar types are included in appendix. We will now define σ_i (s) as mappings from scalar types to display scalar types. These scalar mapping functions σ_i (s) , provide a simple user interface for controlling how all data types are displayed, since the graphical depiction of any object can be derived from the them.

2.6 Graph Visualization Algorithms

Get the paper from the ATT website.
"A technique for drawing directed graphs"
"Applications to Graph Visualizations"

Graph visualization algorithms are formulated to draw graphs base
principles [DOTALG]:

- Expose hierarchical structure in graphs, expose general flow of
- Avoid edge crossings, sharp bends and other visual analomies
  information
- Keep edges as short as possible
- Expose symmetry, parallelism, re. 162 0 Td (l)Tj 3.3619 0 Td (

This are physical limits to the number of vertices that can be visualized, explain the difficulty with BDD, large data-set etc..

Write about the dot algorithm

2.7 Existing BDD Visualization Techniques

Currently, there is not much available in the way of BDD visualization. To our knowledge, besides traditional text printouts for debugging purposes, the only available utilities for the visualization of BDDs display the entire DAG using graph display algorithms. . All systems surveyed *dumps* the BDD DAG as it is, vertex for vertex and edge for edge. There is no work done in the area of data-set preprocessing to provide more meaningful abstraction of data.

Layout of BDD dumps is easier than general graph layout. We know the level (or rank) of all vertices to begin with. [Look at the dot algorithm and mention of the savings] systems then layout the DAG according to the visual principles specified in the previous section.