Graph-Based Algorithms for Boolean Function Manipulation[1][2]

Randal E. Bryant[3]

Abstract

In this paper we present a new data structure for representing Boolean functions and an associated set of manipulation algorithms. Functions are represented by directed, acyclic graphs in a manner similar to the representations introduced by Lee [1] and Akers [2], but with further restrictions on the ordering of decision variables in the graph. Although a function requires, in the worst case, a graph of size exponential in the number of arguments, many of the functions encountered in typical applications have a more reasonable representation. Our algorithms have time complexity proportional to the sizes of the graphs being operated on, and hence are quite efficient as long as the graphs do not grow too large. We present experimental results from applying these algorithms to problems in logic design verification that demonstrate the practicality of our approach.

Index Terms: Boolean functions, symbolic manipulation, binary decision diagrams, logic design verification

1. Introduction

Boolean Algebra forms a cornerstone of computer science and digital system design. Many problems in digital logic design and testing, artificial intelligence, and combinatorics can be expressed as a sequence of operations on Boolean functions. Such applications would benefit from efficient algorithms for representing and manipulating Boolean functions symbolically. Unfortunately, many of the tasks one would like to perform with Boolean functions, such as testing whether there exists any assignment of input variables such that a given Boolean expression evaluates to 1 (satisfiability), or two Boolean expressions denote the same function (equivalence) require solutions to NP-Complete or coNP-Complete problems [3]. Consequently, all known approaches to performing these operations require, in the worst case, an amount of computer time that grows exponentially with the size of the problem. This makes it difficult to compare the relative efficiencies of different approaches to representing and manipulating Boolean functions. In the worst case, all known approaches perform as poorly as the naive approach of representing functions by their truth tables and defining all of the desired operations in terms of their effect on truth table entries. In practice, by utilizing more clever representations and manipulation algorithms, we can often avoid these exponential computations.

A variety of methods have been developed for representing and manipulating Boolean functions. Those based on classical representations such as truth tables, Karnaugh maps, or canonical sum-of-products form [4] are quite

---

[1]This research was funded at the California Institute of Technology by the Defense Advanced Research Projects Agency ARPA Order Number 3771 and at Carnegie-Mellon University by the Defense Advanced Research Projects Agency ARPA Order Number 3597. A preliminary version of this paper was presented under the title "Symbolic Manipulation of Boolean Functions Using a Graphical Representation" at the 22nd Design Automation Conference, Las Vegas, NV, June 1985.

[2]**Update:** This paper was originally published in ***IEEE Transactions on Computers, C-35-8***, pp. 677-691, August, 1986. To create this version, we started with the original electronic form of the submission. All of the figures had to be redrawn, since they were in a now defunct format. We have included footnotes (starting with "**Update:**") discussing some of the (minor) errors in the original version and giving updates on some of the open problems.

[3]Current address: Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213

impractical---*every* function of n arguments has a representation of size 2^n or more. More practical approaches utilize representations that at least for many functions, are not of exponential size. Example representations include as a reduced sum of products [4], (or equivalently as sets of prime cubes [5]) and factored into unate functions [6]. These representations suffer from several drawbacks. First, certain common functions still require representations of exponential size. For example, the even and odd parity functions serve as worst case examples in all of these representations. Second, while a certain function may have a reasonable representation, performing a simple operation such as complementation could yield a function with an exponential representation. Finally, none of these representations are *canonical forms*, i.e. a given function may have many different representations. Consequently, testing for equivalence or satisfiability can be quite difficult.

Due to these characteristics, most programs that process a sequence of operations on Boolean functions have rather erratic behavior. They proceed at a reasonable pace, but then suddenly "blow up", either running out of storage or failing to complete an operation in a reasonable amount of time.

In this paper we present a new class of algorithms for manipulating Boolean functions represented as directed acyclic graphs. Our representation resembles the binary decision diagram notation introduced by Lee [1] and further popularized by Akers [2]. However, we place further restrictions on the ordering of decision variables in the vertices. These restrictions enable the development of algorithms for manipulating the representations in a more efficient manner.

Our representation has several advantages over previous approaches to Boolean function manipulation. First, most commonly-encountered functions have a reasonable representation. For example, all symmetric functions (including even and odd parity) are represented by graphs where the number of vertices grows at most as the square of the number of arguments. Second, the performance of a program based on our algorithms when processing a sequence of operations degrades slowly, if at all. That is, the time complexity of any single operation is bounded by the product of the graph sizes for the functions being operated on. For example, complementing a function requires time proportional to the size of the function graph, while combining two functions with a binary operation (of which intersection, subtraction, and testing for implication are special cases) requires at most time proportional to the product of the two graph sizes. Finally, our representation in terms of *reduced* graphs is a canonical form, i.e. every function has a unique representation. Hence, testing for equivalence simply involves testing whether the two graphs match exactly, while testing for satisfiability simply involves comparing the graph to that of the constant function **0**.

Unfortunately, our approach does have its own set of undesirable characteristics. At the start of processing we must choose some ordering of the system inputs as arguments to all of the functions to be represented. For some functions, the size of the graph representing the function is highly sensitive to this ordering. The problem of computing an ordering that minimizes the size of the graph is itself a coNP-Complete problem. Our experience, however, has been that a human with some understanding of the problem domain can generally choose an appropriate ordering without great difficulty. It seems quite likely that using a small set of heuristics, the program itself could select an adequate ordering most of the time. More seriously, there are some functions that can be represented by Boolean expressions or logic circuits of reasonable size but for all input orderings the representation as a function graph is too large to be practical. For example, we prove in an appendix to this paper that the functions describing the outputs of an integer multiplier have graphs that grow exponentially in the word size regardless of the input ordering. With the exception of integer multiplication, our experience has been that such functions seldom arise in digital logic design applications. For other classes of problems, particularly in combinatorics, our methods seem practical only under restricted conditions.

A variety of graphical representations of discrete functions have be presented and studied extensively. A survey of the literature on the subject by Moret [7] cites over 100 references, but none of these describe a sufficient set of algorithms to implement a Boolean function manipulation program. Fortune, Hopcroft, and Schmidt [8] studied the properties of graphs obeying similar restrictions to ours, showing that two graphs could be tested for functional equivalence in polynomial time and that some functions require much larger graphs under these restrictions than under milder restrictions. Payne [9] describes techniques similar to ours for reducing the size of the graph representing a function. Our algorithms for combining two functions with a binary operation, and for composing two functions are new, however, and these capabilities are central to a symbolic manipulation program.

The next section of this paper contains a formal presentation of function graphs. We define the graphs, the functions they represent, and a class of "reduced" graphs. Then we prove a key property of reduced function graphs: that they form a canonical representation of Boolean functions. In the following section we depart from this formal presentation to give some examples and to discuss issues regarding to the efficiency of our representation. Following this, we develop a set of algorithms for manipulating Boolean functions using our representation. These algorithms utilize many of the classical techniques for graph algorithms, and we assume the reader has some familiarity with these techniques. We then present some experimental investigations into the practicality of our methods. We conclude by suggesting further refinements of our methods.

1.1. Notation

We assume the functions to be represented all have the same n arguments, written $x_1, \ldots, x_n$. In expressing a system such as a combinational logic network or a Boolean expression as a Boolean function, we must choose some ordering of the inputs or atomic variables, and this ordering must be the same for all functions to be represented.

The function resulting when some argument x_i of function f is replaced by a constant b is called a *restriction* of f, (sometimes termed a *cofactor* [10]) and is denoted $f|_{x_i=b}$. That is, for any arguments $x_1, \ldots, x_n$,

$$f|_{x_i=b}(x_1, \ldots, x_n) = f(x_1, \ldots, x_{i-1}, b, x_{i+1}, \ldots, x_n)$$

Using this notation, the Shannon expansion [11] of a function around variable x_i is given by

$$f \;=\; x_i \cdot f|_{x_i=1} + \overline{x}_i \cdot f|_{x_i=0} \tag{1}$$

Similarly, the function resulting when some argument x_i of function f is replaced by function g is called a *composition* of f and g, and is denoted $f|_{x_i=g}$. That is, for any arguments $x_1, \ldots, x_n$,

$$f|_{x_i=g}(x_1, \ldots, x_n) = f(x_1, \ldots, x_{i-1}, g(x_1, \ldots, x_n), x_{i+1}, \ldots, x_n)$$

Some functions may not depend on all arguments. The *dependency set* of a function f, denoted I_f, contains those arguments on which the function depends, i.e.

$$I_f \;=\; \{\, i \mid f|_{x_i=0} \neq f|_{x_i=1} \}$$

The function which for all values of the arguments yields 1 (respectively 0) is denoted **1** (respectively **0**). These two Boolean functions have dependency sets equal to the empty set.

A Boolean function can also be viewed as denoting some subset of Boolean n-space, namely those argument values for which the function evaluates to 1. The *satisfying set* of a function f, denoted S_f, is defined as:

$$S_f \;=\; \{\, (x_1, \ldots, x_n) \mid f(x_1, \ldots, x_n) = 1 \,\}.$$

2. Representation

In this section we define our graphical representation of a Boolean function and prove that it is a canonical form.

Definition 1: A function graph is a rooted, directed graph with vertex set V containing two types of vertices. A *nonterminal* vertex v has as attributes an argument index $index(v) \in \{1, \ldots, n\}$, and two children $low(v), high(v) \in V$. A *terminal* vertex v has as attribute a value $value(v) \in \{0,1\}$.

Furthermore, for any nonterminal vertex v, if $low(v)$ is also nonterminal, then we must have $index(v) < index(low(v))$. Similarly, if $high(v)$ is nonterminal, then we must have $index(v) < index(high(v))$.

Due to the ordering restriction in our definition, function graphs form a proper subset of conventional binary decision diagrams. Note that this restriction also implies that a function graph must be acyclic, because the nonterminal vertices along any path must have strictly increasing index values.

We define the correspondence between function graphs and Boolean functions as follows.

Definition 2: A function graph G having root vertex v denotes a function f_v defined recursively as:

1. If v is a terminal vertex:
   a. If $value(v)$=1, then $f_v = \mathbf{1}$
   b. If $value(v)$=0, then $f_v = \mathbf{0}$
2. If v is a nonterminal vertex with $index(v)=i$, then f_v is the function

$$f_v(x_1, \ldots, x_n) \;=\; \overline{x}_i \cdot f_{low(v)}(x_1, \ldots, x_n) + x_i \cdot f_{high(v)}(x_1, \ldots, x_n).$$

In other words, we can view a set of argument values $x_1, \ldots, x_n$ as describing a path in the graph starting from the root, where if some vertex v along the path has $index(v) = i$, then the path continues to the low child if $x_i = 0$ and to the high child if $x_i = 1$. The value of the function for these arguments equals the value of the terminal vertex at the end of the path. Note that the path defined by a set of argument values is unique. Furthermore, every vertex in the graph is contained in at least one path, i.e. no part of the graph is "unreachable."

Two function graphs are considered isomorphic if they match in both their structure and their attributes. More precisely:

Definition 3: Function graphs G and G' are *isomorphic* if there exists a one-to-one function σ from the vertices of G onto the vertices of G' such that for any vertex v if $\sigma(v)=v'$, then either both v and v' are terminal vertices with $value(v) = value(v')$, or both v and v' are nonterminal vertices with $index(v) = index(v')$, $\sigma(low(v)) = low(v')$, and $\sigma(high(v))=high(v')$.

Note that since a function graph contains only 1 root and the children of any nonterminal vertex are distinguished, the isomorphic mapping σ between graphs G and G' is quite constrained: the root in G must map to the root in G', the root's low child in G must map to the root's low child in G', and so on all the way down to the terminal vertices. Hence, testing 2 function graphs for isomorphism is quite simple.

Definition 4: For any vertex v in a function graph G, the *subgraph rooted by* v is defined as the graph consisting of v and all of its descendants.

Lemma 1: If G is isomorphic to G' by mapping σ, then for any vertex v in G, the subgraph rooted by v is isomorphic to the subgraph rooted by $\sigma(v)$.

The proof of this lemma is straightforward, since the restriction of σ to v and its descendants forms the isomorphic mapping.

A function graph can be reduced in size without changing the denoted function by eliminating redundant vertices and duplicate subgraphs. The resulting graph will be our primary data structure for representing a Boolean function.

> **Definition 5:** A function graph G is *reduced* if it contains no vertex v with $low(v)=high(v)$, nor does it contain distinct vertices v and v' such that the subgraphs rooted by v and v' are isomorphic.

The following lemma follows directly from the definition of reduced function graphs.

> **Lemma 2:** For every vertex v in a reduced function graph, the subgraph rooted by v is itself a reduced function graph.

The following theorem proves a key property of reduced function graphs, namely that they form a canonical representation for Boolean functions, i.e. every function is represented by a unique reduced function graph. In contrast to other canonical representations of Boolean functions, such as canonical sum-of-products form, however, many "interesting" Boolean functions are represented by function graphs of size polynomial in the number of arguments.

> **Theorem 1:** For any Boolean function f, there is a unique (up to isomorphism) reduced function graph denoting f and any other function graph denoting f contains more vertices.

Proof : The proof of this theorem is conceptually straightforward. However, we must take care not to presuppose anything about the possible representations of a function. The proof proceeds by induction on the size of I_f.

For $|I_f| = 0$, f must be one of the two constant functions **0** or **1**. Let G be a reduced function graph denoting the function **0**. This graph can contain no terminal vertices having value 1, or else there would be some set of argument values for which the function evaluates to 1, since all vertices in a function graph are reachable by some path corresponding to a set of argument values. Now suppose G contains at least one nonterminal vertex. Then since the graph is acyclic, there must be a nonterminal vertex v where both $low(v)$ and $high(v)$ are terminal vertices, and it follows that $value(low(v)) = value(high(v)) = 0$. Either these 2 vertices are distinct, in which case they constitute isomorphic subgraphs, or they are identical, in which case v has $low(v) = high(v)$. In either case, G would not be a reduced function graph. Hence, the only reduced function graph denoting the function **0** consists of a single terminal vertex with value 0. Similarly, the only reduced function graph denoting **1** consists of a single terminal vertex with value 1.

Next suppose that the statement of the theorem holds for any function g having $|I_g| < k$, and that $|I_f| = k$, where $k > 0$. Let i be the minimum value in I_f, i.e. the least argument on which the function f depends. Define the functions f_0 and f_1 as $f|_{x_i=0}$ and $f|_{x_i=1}$, respectively. Both f_0 and f_1 have dependency sets of size less than k and hence are represented by unique reduced function graphs. Let G and G' be reduced function graphs for f. We will show that these two graphs are isomorphic, consisting of a root vertex with index i and with low and high subgraphs denoting the functions f_0 and f_1. Let v and v' be nonterminal vertices in the two graphs such that $index(v) = index(v') = i$. The subgraphs rooted by v and v' both denote f, since f is independent of the arguments $x_1, \ldots, x_{i-1}$. The subgraphs rooted by vertices $low(v)$ and $low(v')$ both denote the function f_0 and hence by induction must be isomorphic according to some mapping σ_0. Similarly, the subgraphs rooted by vertices $high(v)$ and $high(v')$ both denote the function f_1 and hence must be isomorphic according to some mapping σ_1.

We claim that the subgraphs rooted by v and v' must be isomorphic according to the mapping σ defined as

$$\sigma(u) = \begin{cases} v', & u = v, \\ \sigma_0(u) & u \text{ in subgraph rooted by } low(v) \\ \sigma_1(u), & u \text{ in subgraph rooted by } high(v) \end{cases}$$

To prove this, we must show that the function σ is well-defined, and that it is an isomorphic mapping. Observe that if vertex u is contained in both the subgraph rooted by *low*(v) and the subgraph rooted by *high*(v), then the subgraphs rooted by $\sigma_0(u)$ and $\sigma_1(u)$ must be isomorphic to the one rooted by u and hence to each other. Since G' contains no isomorphic subgraphs, this can only hold if $\sigma_0(u) = \sigma_1(u)$, and hence there is no conflict in the above definition of σ. By similar reasoning, we can see that σ must be one-to-one---if there were distinct vertices u_1 and u_2 in G having $\sigma(u_1) = \sigma(u_2)$, then the subgraphs rooted by these two vertices would be isomorphic to the subgraph rooted by $\sigma(u_1)$ and hence to each other implying that G is not reduced. Finally, the properties that σ is onto and is an isomorphic mapping follows directly from its definition and from the fact that both σ_0 and σ_1 obey these properties.

By similar reasoning, we can see that graph G contains exactly one vertex with *index*(v) = i, because if some other such vertex existed, the subgraphs rooted by it and by v would be isomorphic. We claim in fact that v must be the root. Suppose instead that there is some vertex u with *index*(u) = $j < i$, but such that there is no other vertex w having $j <$ *index*(w) $< i$. The function f does not depend on the x_j and hence the subgraphs rooted by *low*(u) and *high*(u) both denote f, but this implies that *low*(u) = *high*(u) = v, i.e. G is not reduced. Similarly, vertex v' must be the root of G', and hence the two graphs are isomorphic.

Finally, we can prove that of all the graphs denoting a particular function, only the reduced graph has a minimum number of vertices. Suppose G is not a reduced graph. Then we can form a smaller graph denoting the same function as follows. If G contains a vertex v with *low*(v) = *high*(v), then eliminate this vertex, and for any vertex having v as child, make *low*(v) be the child instead. If G contains distinct vertices v and v' such that the subgraphs rooted by v and v' are isomorphic, then eliminate vertex v' and for any vertex having v' as a child, make v be the child instead.

3. Properties

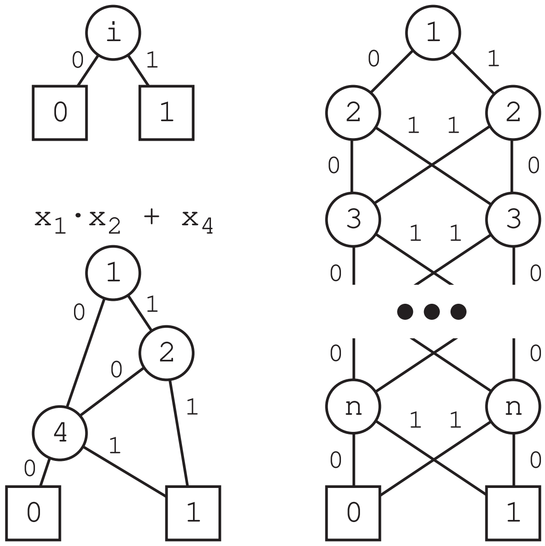


Figure 1.Example Function Graphs

In this section we explore the efficiency of our representation by means of several examples. Figure 1 shows several examples of reduced function graphs. In this figure, a nonterminal vertex is represented by a circle containing the index with the two children indicated by branches labeled 0 (low) and 1 (high). A terminal vertex is represented by a square containing the value.

3.1. Example Functions

The function which yields the value of the ith argument is denoted by a graph with a single nonterminal vertex having index i and having as low child a terminal vertex with value 0 and as high child a terminal vertex with value 1. We present this graph mainly to point out that an input variable can be viewed as a Boolean function, and hence can be operated on by the manipulation algorithms described in this paper.

The odd parity function of n variables is denoted by a graph containing $2n+1$ vertices. This compares favorably to its representation in reduced sum-of-products form (requiring 2^n terms.) This graph resembles the familiar parity ladder contact network first described by Shannon [11]. In fact, we can adapt his construction of a contact network to implement an arbitrary symmetric function to show that any symmetric function of n arguments is denoted by a reduced function graph having $O(n^2)$ vertices.

As a third example, the graph denoting the function $x_1 \cdot x_2 + x_4$ contains 5 vertices as shown. This example illustrates several key properties of reduced function graphs. First, observe that there is no vertex having index 3, because the function is independent of x_3. More generally, a reduced function graph for a function f contains only vertices having indices in I_f. There are no inefficiencies caused by considering all of the functions to have the same n arguments. This would not be the case if we represented functions by their truth tables. Second, observe that even for this simple function, several of the subgraphs are shared by different branches. This sharing yields efficiency not only in the size of the function representation, but also in the performance of our algorithms---once some operation has been performed on a subgraph, the result can be utilized by all places sharing this subgraph.

3.2. Ordering Dependency

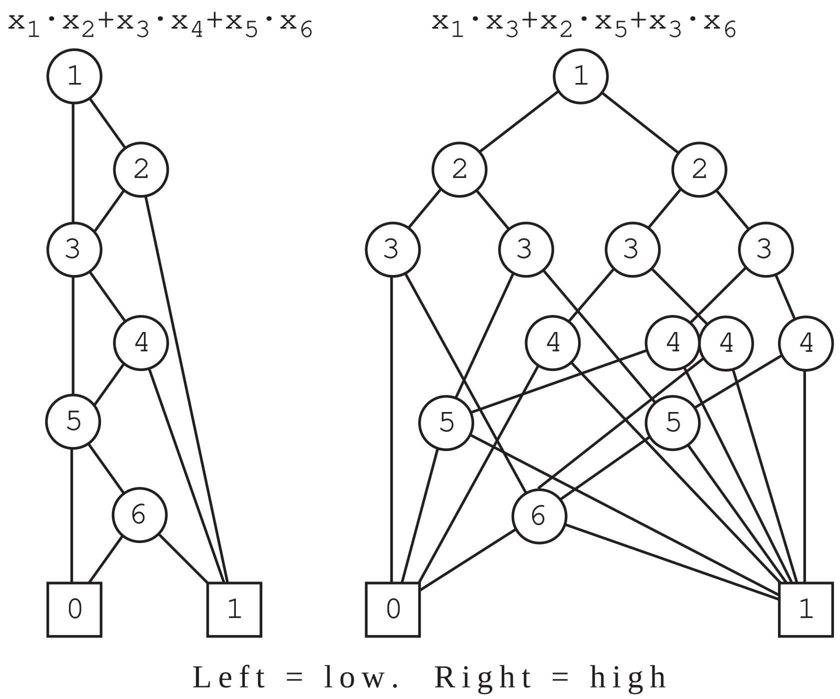


Figure 2.Example of Argument Ordering Dependency

Figure 2 shows an extreme case of how the ordering of the arguments can affect the size of the graph denoting a function. The functions $x_1 \cdot x_2 + x_3 \cdot x_4 + x_5 \cdot x_6$ and $x_1 \cdot x_4 + x_2 \cdot x_5 + x_3 \cdot x_6$ differ from each other only by a permutation of their arguments, yet one is denoted by a function graph with 8 vertices while the other requires 16 vertices.

Generalizing this to functions of $2n$ arguments, the function $x_1 \cdot x_2 + \cdots + x_{2n-1} \cdot x_{2n}$ is denoted by a graph of $2n+2$ vertices, while the function $x_1 \cdot x_{n+1} + \cdots + x_n \cdot x_{2n}$ requires 2^{n+1} vertices. Consequently, a poor initial choice of input ordering can have very undesirable effects.

Upon closer examination of these two graphs, we can gain a better intuition of how this problem arises. Imagine a bit-serial processor that computes a Boolean function by examining the arguments x_1, x_2, and so on in order, producing output 0 or 1 after the last bit has been read. Such a processor requires internal storage to store enough information about the arguments it has already seen to correctly deduce the value of the function from the values of the remaining arguments. Some functions require little intermediate information. For example, to compute the parity function a bit-serial processor need only store the parity of the arguments it has already seen. Similarly, to compute the function $x_1 \cdot x_2 + \cdots + x_{2n-1} \cdot x_{2n}$, the processor need only store whether any of the preceding pairs of arguments were both 1, and perhaps the value of the previous argument. On the other hand, to compute the function $x_1 \cdot x_{n+1} + \cdots + x_n \cdot x_{2n}$, we would need to store the first n arguments to correctly deduce the value of the function from the remaining arguments. A function graph can be thought of as such a processor, with the set of vertices having index i describing the processing of argument x_i. Rather than storing intermediate information as bits in a memory, however, this information is encoded in the set of possible branch destinations. That is, if the bit-serial processor requires b bits to encode information about the first i arguments, then in any graph for this function there must be at least 2^b vertices that are either terminal or are nonterminal with index greater than i having incoming branches from vertices with index less than or equal to i. For example, the function $x_1 \cdot x_4 + x_2 \cdot x_5 + x_3 \cdot x_6$ requires 2^3 branches between vertices with index less than or equal to 3 to vertices which are either terminal or have index greater than 3. In fact, the first 3 levels of this graph must form a complete binary tree to obtain this degree of branching. In the generalization of this function, the first n levels of the graph form a complete binary tree, and hence the number of vertices grows exponentially with the number of arguments.

To view this from a different perspective, consider the family of functions:

$$f_{b_1, \ldots, b_n}(x_{n+1}, \ldots, x_{2n}) \;=\; b_1 \cdot x_{n+1} + \cdots + b_n \cdot x_{2n}.$$

For all 2^n possible combinations of the values $b_1, \ldots, b_n$, each of these functions is distinct, and hence they must be represented by distinct subgraphs in the graph of the function $x_1 \cdot x_{n+1} + \cdots + x_n \cdot x_{2n}$.

To use our algorithms on anything other than small problems (e.g. functions of 16 variables or more), a user must have an intuition about why certain functions have large function graphs, and how the choice of input ordering may affect this size. In Section 5 we will present examples of how the structure of the problem to be solved can often be exploited to obtain a suitable input ordering.

3.3. Inherently Complex Functions

Some functions cannot be represented efficiently with our representation regardless of the input ordering. Unfortunately, the functions representing the output bits of an integer multiplier fall within this class. The appendix contains a proof that for any ordering of the inputs $a_1, \ldots, a_n$ and $b_1, \ldots, b_n$, at least one of the $2n$ functions representing the integer product $a \cdot b$ requires a graph containing at least $2^{n/8}$ vertices. While this lower bound is not very large for word sizes encountered in practice (e.g. it equals 256 for n=64), it indicates the exponential complexity of these functions. Furthermore, we suspect the true bound is far worse.

Empirically, we have found that for word sizes n less than or equal to 8, the output functions of a multiplier require no more than 5000 vertices for a variety of different input orderings. However, for $n > 10$, some outputs require graphs with more than 100,000 vertices and hence become impractical.

Given the wide variety of techniques used in implementing multipliers (e.g. [12]), a canonical form for Boolean functions (along with a set of manipulation algorithms) that could efficiently represent multiplication would be of great interest for circuit verification. Unfortunately, these functions seem especially intractable.

4. Operations

We view a symbolic manipulation program as executing a sequence of commands that build up representations of functions and determine various properties about them. For example, suppose we wish to construct the representation of the function computed by a combinational logic gate network. Starting from graphs representing the input variables, we proceed through the network, constructing the function computed at the output of each logic gate by applying the gate operator to the functions at the gate inputs. In this process, we can take advantage of any repeated structure by first constructing the functions representing the individual subcircuits (in terms of a set of auxiliary variables) and then composing the subcircuit functions to obtain the complete network functions. A similar procedure is followed to construct the representation of the function denoted by some Boolean expression. At this point we can test various properties of the function, such as whether it equals **0** (satisfiability) or **1** (tautology), or whether it equals the function denoted by some other expression (equivalence). We can also ask for information about the function's satisfying set, such as to list some member, to list all members, to test some element for membership, etc.

Procedure	**Result**	**Time Complexity**
Reduce	G reduced to canonical form	$O(\lvert G\rvert \cdot \log\lvert G\rvert)$
Apply	$f_1 <\mathrm{op}> f_2$	$O(\lvert G_1\rvert \cdot \lvert G_2\rvert)$
Restrict	$f\lvert_{x_i=b}$	$O(\lvert G\rvert \cdot \log\lvert G\rvert)$
Compose	$f_1\lvert_{x_i=f_2}$	$O(\lvert G_1\rvert^2 \cdot \lvert G_2\rvert)$
Satisfy-one	some element of S_f	$O(n)$
Satisfy-all	S_f	$O(n \cdot \lvert S_f\rvert)$
Satisfy-count	$\lvert S_f\rvert$	$O(\lvert G\rvert)$

Table 1.Summary of Basic Operations

In this section we will present algorithms to perform basic operations on Boolean functions represented as function graphs as summarized in Table 1.[4] These few basic operations can be combined to perform a wide variety of operations on Boolean functions. In the table, the function f is represented by a reduced function graph G containing $|G|$ vertices, and similarly for the functions f_1 and f_2. Our algorithms utilize techniques commonly used in graph algorithms such as ordered traversal, table look-up and vertex encoding. As the table shows, most of the algorithms have time complexity proportional to the size of the graphs being manipulated. Hence, as long as the functions of interest can be represented by reasonably small graphs, our algorithms are quite efficient.

4.1. Data Structures

We will express our algorithms in a pseudo-Pascal notation. Each vertex in a function graph is represented by a record declared as follows:

---

[4]**Update:** Some of the time complexity entries in this table are not strictly correct. An extra factor of $(\log|G_1|+\log|G_2|)$ should have been included in the time complexity of *Apply* and *Compose* to account for the complexity of reducing the resulting graph. In later research, Wegener and Sieling showed how to perform BDD reduction in linear time (*Information Processing Letters 48*, pp. 139-144, 1993.) Consequently, all of the log factors can be dropped from the table. In practice, most BDD implementations use hash tables rather than the sorting method described in this paper. Assuming retrieving an element from a hash table takes constant time (a reasonable assumption for a good hash function), these implementations also perform BDD reduction in linear time.

```
type vertex = record
        low, high: vertex;
        index: 1..n+1;
        val: (0,1,X);
        id: integer;
        mark: boolean;
end;
```

Both nonterminal and terminal vertices are represented by the same type of record, but the field values for a vertex *v* depend on the vertex type as given in the following table.

Field	**Terminal**	**Nonterminal**
low	null	*low*(*v*)
high	null	*high*(*v*)
index	*n*+1	*index*(*v*)
val	*value*(*v*)	X

The id and mark fields contain auxiliary information used by the algorithms. The id field contains a integer identifier which is unique to that vertex in the graph. It does not matter how the identifiers are ordered among the vertices, only that they range from 1 up to the number of vertices and that they all be different. The mark field is used to mark which vertices have been visited during a traversal of the graph. The procedure *Traverse* shown in Figure 3 illustrates a general method used by many of our algorithms for traversing a graph and performing some operation on the vertices. This procedure is called at the top level with the root vertex as argument and with the mark fields of the vertices being either all true or all false. It then systematically visits every vertex in the graph by recursively visiting the subgraphs rooted by the two children. As it visits a vertex, it complements the value of the mark field, so that it can later determine whether a child has already been visited by comparing the two marks. As a vertex is visited, we could perform some operation such as to increment a counter and then set the id field to the value of the counter (thereby assigning a unique identifier to each vertex.) Each vertex is visited exactly once, and assuming the operation at each vertex requires constant time, the complexity of the algorithm is $O(|G|)$, i.e. proportional to the number of vertices in the graph. Upon termination, the vertices again all have the same mark value.

```
procedure Traverse(v:vertex);
begin
        v.mark := not v.mark;
        ... do something to v ...
        if v.index ≤ n
        then begin {v nonterminal}
                if v.mark ≠ v.low.mark then Traverse(v.low);
                if v.mark ≠ v.high.mark then Traverse(v.high);
        end;
end;
```

Figure 3.Implementation of Ordered Traversal

4.2. Reduction

The reduction algorithm transforms an arbitrary function graph into a reduced graph denoting the same function. It closely follows an algorithm presented in Example 3.2 of Aho, Hopcroft, and Ullman [13] for testing whether two trees are isomorphic. Proceeding from the terminal vertices up to the root, a unique integer identifier is assigned to each unique subgraph root. That is, for each vertex *v* it assigns a label *id*(*v*) such that for any two vertices *u* and *v*, *id*(*u*) = *id*(*v*) if and only if $f_u = f_v$ (in the terminology of Definition 2.) Given this labeling, the algorithm constructs a graph with one vertex for each unique label.

By working from the terminal vertices up to the root, a procedure can label the vertices by the following inductive method. First, two terminal vertices should have the same label if and only if they have the same value attributes. Now assume all terminal vertices and all nonterminal vertices with index greater than i have been labeled. As we proceed with the labeling of vertices with index i, a vertex v should have $id(v)$ equal to that of some vertex that has already been labeled if and only if one of two conditions is satisfied. First, if $id(low(v)) = id(high(v))$, then vertex v is redundant, and we should set $id(v) = id(low(v))$. Second, if there is some labeled vertex u with $index(u) = i$ having $id(low(v)) = id(low(u))$, and $id(high(v)) = id(high(u))$, then the reduced subgraphs rooted by these two vertices will be isomorphic, and we should set $id(v) = id(u)$.

A sketch of the code is shown in Figure 4. First, the vertices are collected into lists according to their indices. This can be done by a procedure similar to *Traverse*, where as a vertex is visited, it is added to the appropriate list. Then we process these lists working from the one containing the terminal vertices up to the one containing the root. For each vertex on a list we create a key of the form (*value*) for a terminal vertex or of the form (*lowid*, *highid*) for a nonterminal vertex, where $lowid = id(low(v))$ and $highid = id(high(v))$. If a vertex has $lowid = highid$, then we can immediately set $id(v) = lowid$. The remaining vertices are sorted according to their keys. Aho, Hopcroft, and Ullman describe a linear-time lexicographic sorting method for this based on bucket sorting. We then work through this sorted list, assigning a given label to all vertices having the same key. We also select one vertex record for each unique label and store a pointer to this vertex in an array indexed by the label. These selected vertices will form the final reduced graph. Hence, we can obtain the reduced version of a subgraph with root v by accessing the array element with index $id(v)$ We use this method to modify a vertex record so that its two children are vertices in the reduced graph and to return the root of the final reduced graph when the procedure is exited. Note that the labels assigned to the vertices by this routine can serve as unique identifiers for later routines. Assuming a linear-time sorting routine, the processing at each level requires time proportional to the number of vertices at that level. Each level is processed once, and hence the overall complexity of the algorithm is linear in the number of vertices.

```
function Reduce(v: vertex): vertex;
        var subgraph: array[1..|G|] of vertex;
        var vlist: array[1..n+1] of list;
begin
        Put each vertex u on list  vlist[u.index]
        nextid := 0;
        for i := n+1 downto 1 do
        begin
                Q := empty set;
                for each u in vlist[i] do
                        if u.index = n+1
                                then add <key,u> to Q where key = (u.value) {terminal vertex}
                                else if u.low.id = u.high.id
                                        then u.id := u.low.id {redundant vertex}
                                        else add <key,u> to Q where key = (u.low.id, u.high.id);
                Sort elements of Q by keys;
                oldkey := (–1;–1);      {unmatchable key}
                for each <key,u> in Q removed in order do
                        if key = oldkey
                                then u.id:= nextid;   {matches existing vertex}
                                else begin  {unique vertex}
                                        nextid := nextid + 1; u.id := nextid; subgraph[nextid] := u;
                                        u.low := subgraph[u.low.id]; u.high := subgraph[u.high.id];
                                        oldkey := key;
                                end;
        end;
        return(subgraph[v.id]);
end;
```

Figure 4.Implementation of *Reduce*

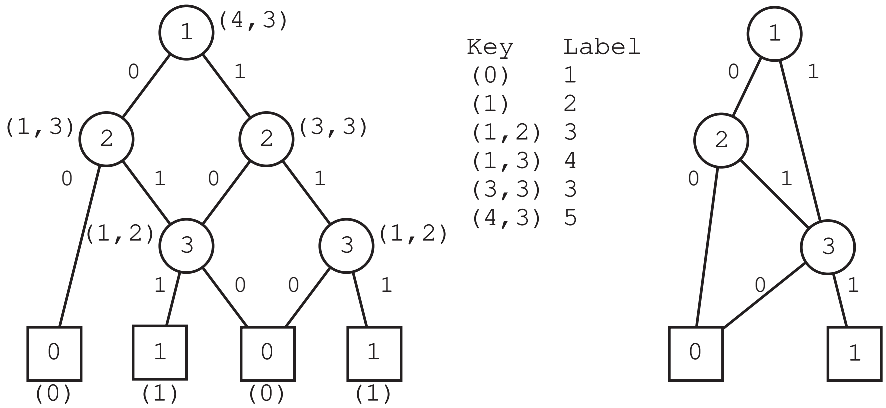


Figure 5.Reduction Algorithm Example

Figure 5 shows an example of how the reduction algorithm works. Next to each vertex we show the key and the label generated during the labeling process. Observe that both vertices with index 3 have the same key, and hence the right hand vertex with index 2 is redundant.

4.3. Apply

The procedure *Apply* provides the basic method for creating the representation of a function according to the operators in a Boolean expression or logic gate network. It takes graphs representing functions f_1 and f_2, a binary operator <op> (i.e. any Boolean function of 2 arguments) and produces a reduced graph representing the function $f_1 <op> f_2$ defined as

$$[f_1 <op> f_2](x_1, \ldots, x_n) \;=\; f_1(x_1, \ldots, x_n) \;<op>\; f_2(x_1, \ldots, x_n).$$

This procedure can also be used to complement a function (compute $f \oplus \mathbf{1}$)[5] to test for implication (compare $f_1 \cdot \neg f_2$ to $\mathbf{0}$), and a variety of other operations. With our representation, we can implement all of the operators with a single algorithm. In contrast, many Boolean function manipulation programs [6] require different algorithms for complementing, intersecting (<op> = ·), and unioning (<op> = +) functions, and then implement other operators by combining these operations.

The algorithm proceeds from the roots of the two argument graphs downward, creating vertices in the result graph at the branching points of the two arguments graphs. First, let us explain the basic idea of the algorithm. Then we will describe two refinements to improve the efficiency. The control structure of the algorithm is based on the following recursion, derived from the Shannon expansion (equation 1)

$$f_1 <op> f_2 \;=\; \overline{x}_i \cdot (f_1|_{x_i=0} <op> f_2|_{x_i=0}) \;+\; x_i \cdot (f_1|_{x_i=1} <op> f_2|_{x_i=1})$$

To apply the operator to functions represented by graphs with roots v_1 and v_2, we must consider several cases. First, suppose both v_1 and v_2 are terminal vertices. Then the result graph consists of a terminal vertex having value $value(v_1) <op> value(v_2)$. Otherwise, suppose at least one of the two is a nonterminal vertex. If $index(v_1) = index(v_2) = i$, we create a vertex u having index i, and apply the algorithm recursively on $low(v_1)$ and $low(v_2)$ to generate the subgraph whose root becomes $low(u)$, and on $high(v_1)$ and $high(v_2)$ to generate the subgraph whose root becomes $high(u)$. Suppose, on the other hand, that $index(v_1) = i$, but either v_2 is a terminal vertex or $index(v_2) > i$, Then the function represented by the graph with root v_2 is independent of x_i, i.e.

$$f_2|_{x_i=0} \;=\; f_2|_{x_i=1} \;=\; f_2.$$

[5] Alternatively, a function can be complemented by simply complementing the values of the terminal vertices.

Hence we create a vertex u having index i, but recursively apply the algorithm on $low(v_1)$ and v_2 to generate the subgraph whose root becomes $low(u)$, and on $high(v_1)$ and v_2 to generate the subgraph whose root becomes $high(u)$. A similar situation holds when the roles of the two vertices in the previous case are reversed. In general the graph produced by this process will not be reduced, and we apply the reduction algorithm to it before returning.

If we were to implement the technique described in the previous paragraph directly we would obtain an algorithm of exponential (in n) time complexity, because every call for which one of the arguments is a nonterminal vertex generates two recursive calls. This complexity can be reduced by two refinements.

First, the algorithm need not evaluate a given pair of subgraphs more than once. Instead, we can maintain a table containing entries of the form (v_1,v_2,u) indicating that the result of applying the algorithm to subgraphs with roots v_1 and v_2 was a subgraph with root u. Then before applying the algorithm to a pair of vertices, we first check whether the table contains an entry for these two vertices. If so, we can immediately return the result. Otherwise, we proceed as described in the previous paragraph and add a new entry to the table. This refinement alone drops the time complexity to $O(|G_1|\cdot|G_2|)$, as we will show later. This refinement shows how we can exploit the sharing of subgraphs in the data structures to gain efficiency in the algorithms. If the two argument graphs each contain many shared subgraphs, we obtain a high "hit rate" for our table. In practice we have found hit rates to range between 40% and 50%. Note that with a 50% hit rate, we obtain a speed improvement far better than the factor of 2 one might first expect. Finding an entry (v_1,v_2,u) in the table counts as only one "hit", but avoids the potentially numerous recursive calls required to construct the subgraph rooted by u.

Second, suppose the algorithm is applied to two vertices where one, say v_1, is a terminal vertex, and for this particular operator, $value(v_1)$ is a "controlling" value, i.e. either $value(v_1)$ <*op*> $a = 1$ for all a, or $value(v_1)$ <*op*> $a = 0$ for all a. For example, 1 is a controlling value for either argument of OR, while 0 is a controlling value for either argument of AND. In this case, there is no need to evaluate further. We simply create a terminal vertex having the appropriate value. While this refinement does not improve the worst case complexity of the algorithm, it certainly helps in many cases. In practice we have found this case occurs around 10% of the time.

A sketch of the code is shown in Figure 6. For simplicity and to optimize the worst case performance, the table is implemented as a two dimensional array indexed by the unique identifiers of the two vertices. In practice, this table will be very sparse, and hence it is more efficient to use a hash table. To detect whether one of the two vertices contains a controlling value for the operator, we evaluate the expression v1.value <op> v2.value using a three-valued algebra where X (the value at any nonterminal vertex) represents "don't care". That is, if b <op> 1 = b <op> 0 = a, then b <op> X = a, otherwise b <op> X = X. This evaluation technique is used in many logic simulators. [14] Before the final graph is returned, we apply the procedure *Reduce* to transform it into a reduced graph and to assign unique identifiers to the vertices.

To analyze the time complexity of this algorithm when called on graphs with $|G_1|$ and $|G_2|$ vertices, respectively, observe that the procedure *Apply-step* only generates recursive calls the first time it is invoked on a given pair of vertices, hence the total number of recursive calls to this procedure cannot exceed $2\cdot|G_1|\cdot|G_2|$. Within a single call, all operations (including looking for an entry in the table) require constant time. Furthermore the initialization of the table requires at time proportional to its size, i.e. $O(|G_1|\cdot|G_2|)$. Hence, the total complexity of the algorithm is $O(|G_1|\cdot|G_2|)$.[6] In the worst case, the algorithm may actually require this much time, because the reduced graph for the function f_1 <op> f_2 can contain $O(|G_1|\cdot|G_2|)$ vertices. For example, choose any positive integers m and n and define the functions f_1 and f_2 as:

---

[6]**Update:** See the footnote for Table 1 discussing the inaccuracy of this complexity measure.

$$f_1(x_1, \ldots, x_{2n+2m}) = x_1 \cdot x_{n+m+1} + \cdots + x_n \cdot x_{2n+m}$$
$$f_2(x_1, \ldots, x_{2n+2m}) = x_{n+1} \cdot x_{2n+m+1} + \cdots + x_{n+m} \cdot x_{2n+2m}$$

These functions are represented by graphs containing 2^{n+1} and 2^{m+1} vertices, respectively, as we saw in Section 3. If we compute $f = f_1 + f_2$, the resulting function is

$$f(x_1, \ldots, x_{2n+2m}) = x_1 \cdot x_{n+m+1} + \cdots + x_{n+m} \cdot x_{2n+2m}$$

which is represented by a graph with $2^{n+m+1} = 0.5 \cdot |G_1| \cdot |G_2|$ vertices. Hence the worst case efficiency of the algorithm is limited by the size of the result graph, and we cannot expect any algorithm to achieve a better performance than ours under these conditions. Empirically, we have never observed this worst case occurring except when the result graph is large. We conjecture the algorithm could be refined to have worst case complexity $O(|G_1|+|G_2|+|G_3|)$, where G_3 is the resulting reduced graph.[7]

```
function Apply(v1, v2: vertex; <op>: operator): vertex
    var T: array[1..|G1|, 1..|G2|] of vertex;

    {Recursive routine to implement Apply}
    function Apply-step(v1, v2: vertex): vertex;
    begin
        u := T[v1.id, v2.id];
        if u ≠ null then return(u); {have already evaluated}
        u := new vertex record; u.mark := false;
        T[v1.id, v2.id] := u; {add vertex to table}
        u.value := v1.value <op> v2.value;
        if u.value ≠ X
        then begin {create terminal vertex}
            u.index := n+1; u.low := null; u.high := null;
        end
        else begin {create nonterminal and evaluate further down}
            u.index := Min(v1.index, v2.index);
            if v1.index = u.index
                then begin vlow1 := v1.low; vhigh1 := v1.high end
                else begin vlow1 := v1; vhigh1 := v1 end;
            if v2.index = u.index
                then begin vlow2 := v2.low; vhigh2 := v2.high end
                else begin vlow2 := v2; vhigh2 := v2 end;
            u.low := Apply-step(vlow1, vlow2);
            u.high := Apply-step(vhigh1, vhigh2);
        end;
        return(u);
    end;

begin   {Main routine}
    Initialize all elements of T to null;
    u := Apply-step(v1, v2);
    return(Reduce(u));
end;
```

Figure 6. Implementation of *Apply*

Figure 7 shows an example of how this algorithm would proceed in applying the "or" operation to graphs representing the functions $\neg(x_1 \cdot x_3)$ and $x_2 \cdot x_3$. This figure shows the graph created by the algorithm before reduction. Next to each vertex in the resulting graph, we indicate the two vertices on which the procedure

[7]**Update:** No one has improved on the worst case complexity of the algorithm described here. Disproving a conjecture such as this one would be difficult, however.

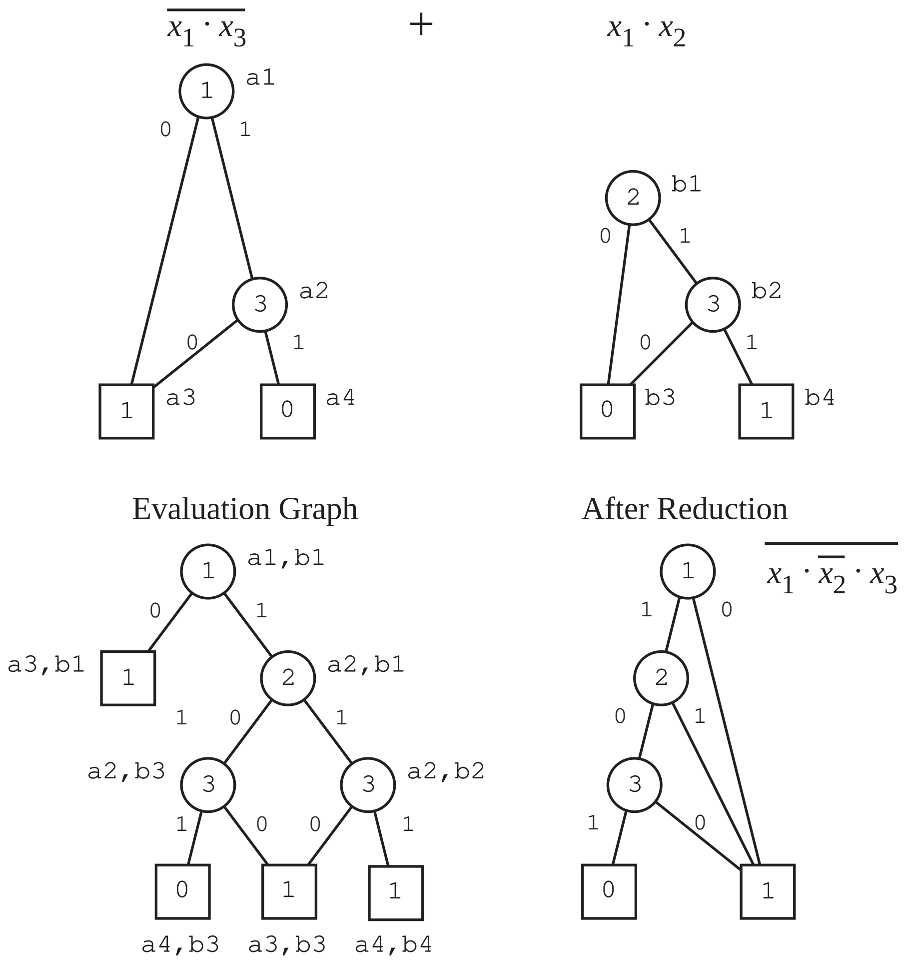


Figure 7.Example of *Apply*

Apply-step was invoked in creating this vertex. Each of our two refinements is applied once: when the procedure is invoked on vertices a3 and b1 (because 1 is a controlling value for this operator), and on the second invocation on vertices a3 and b3. For larger graphs, we would expect these refinements to be applied more often. After the reduction algorithm has been applied, we see that the resulting graph indeed represents the function $\neg(x_1 \cdot \overline{x}_2 \cdot x_3)$.

4.4. Restriction

The restriction algorithm transforms the graph representing a function f into one representing the function $f|_{x_i=b}$ for specified values of i and b. This algorithm proceeds by traversing the graph in the manner shown in the procedure *Traverse* looking for every pointer (either to the root of the graph or from some vertex to its child) to a vertex v such that $index(v) = i$. When such a pointer is encountered, it is changed to point either to $low(v)$ (for $b = 0$) or to $high(v)$ (for $b = 1$.) Finally, the procedure *Reduce* is called to reduce the graph and to assign unique identifiers to the vertices. The amount computation required for each vertex is constant, and hence the complexity of this algorithm is $O(|G|)$. Note that this algorithm could simultaneously restrict several of the function arguments without changing the complexity.

4.5. Composition

The composition algorithm constructs the graph for the function obtained by composing two functions. This algorithm allows us to more quickly derive the functions for a logic network or expression containing repeated structures, a common occurrence in structured designs. Composition can be expressed in terms of restriction and Boolean operations, according to the following expansion, derived directly from the Shannon expansion (equation 1):

$$f_1|_{x_i=f_2} \;=\; f_2 \cdot f_1|_{x_i=1} + (\neg f_2) \cdot f_1|_{x_i=0} \tag{2}$$

Thus, our algorithms for restriction and application are sufficient to implement composition. However, if the two functions are represented by graphs G_1 and G_2, respectively, we would obtain a worst case complexity of $O(|G_1|^2 \cdot |G_2|^2)$. We can improve this complexity to $O(|G_1|^2 \cdot |G_2|)$ by observing that equation 2 can be expressed in terms of a ternary Boolean operation *ITE* (short for if-then-else)

$$ITE(a, b, c) \;=\; a \cdot b + (\neg a) \cdot c$$

This operation can be applied to the three functions f_2, $f_1|_{x_i=1}$, and $f_1|_{x_i=0}$ by an extension of the *Apply* procedure to ternary operations. The procedure *Compose* shown in Figure 8 utilizes this technique to compose two functions. In this code the recursive routine *Compose-step* both applies the operation *ITE* and computes the restrictions of f_1 as it traverses the graphs.

It is unclear whether the efficiency of this algorithm truly has a quadratic dependence on the size of its first argument, or whether this indicates a weakness in our performance analysis. We have found no cases for which composition requires time greater than $O(|G_1| \cdot |G_2|)$.[8]

In many instances, two functions can be composed in a simpler and more efficient way by a more syntactic technique.[9] That is suppose functions f_1 and f_2 are represented by graphs G_1 and G_2, respectively. We can compose the functions by replacing each vertex v in graph G_1 having index i by a copy of G_2, replacing each branch to a terminal vertex in G_2 by a branch to $low(v)$ or $high(v)$ depending on the value of the terminal vertex. We can do this however, only if the resulting graph would not violate our index ordering restriction. That is, there can be no indices $j \in I_{f_1}$, $k \in I_{f_2}$, such that $i < j \leq k$ or $i > j \geq k$. Assuming both G_1 and G_2 are reduced, the graph resulting from these replacements is also reduced, and we can even avoid applying the reduction algorithm. While this technique applies only under restricted conditions, we have found it a worthwhile optimization.

4.6. Satisfy

There are many questions one could ask about the satisfying set S_f for a function, including the number of elements, a listing of the elements, or perhaps just a single element. As can be seen from Table 1, these operations are performed by algorithms of widely varying complexity. A single element can be found in time proportional to n, the number of function arguments, assuming the graph is reduced. Considering that the value of an element in S_f is specified by a bit sequence of length n, this algorithm is optimal. We can list all elements of S_f in time proportional to n times the number of elements, which again is optimal. However, this is generally not a wise thing to do---many functions that are represented by small graphs have very large satisfying sets. For example, the function **1** is represented by a graph with one vertex, yet all 2^n possible combinations of argument values are in its satisfying set.

---

[8]**Update:** Jain *et al.* (J. Jain, *et al.*, "Analysis of Composition Complexity and How to Obtain Smaller Canonical Graphs", *37th Design Automation Conference*, 2000, pp. 681-686.) have shown an example for which the resulting graph is of size $O(|G_1|^2 \cdot |G_2|)$. In the worst case, the algorithm *does* have a quadratic dependence on the size of its first argument.

[9]**Update:** Elena Dubrova (private correspondence, Feb., 2000) pointed out that this approach can yield a non-reduced graph.

```
function Compose(v1, v2: vertex; i:integer): vertex
    var T: array[1..|G1|, 1..|G1|, 1..|G2|] of vertex;

    {Recursive routine to implement Compose}
    function Compose-step(vlow1, vhigh1, v2: vertex): vertex;
    begin
        {Perform restrictions}
        if vlow1.index = i then vlow1 := vlow1.low;
        if vhigh1.index = i then vhigh1 := vhigh1.high;
        {Apply operation ITE }
        u := T[vlow1.id, vhigh1.id, v2.id];
        if u ≠ null then return(u); {have already evaluated}
        u := new vertex record; u.mark := false;
        T[vlow1.id, vhigh1.id, v2.id] := u; {add vertex to table}
        u.value := (¬ v2.value · vlow1.value) + (v2.value · vhigh1.value);
        if u.value ≠ X
        then begin {create terminal vertex}
            u.index := n+1; u.low := null; u.high := null;
        end
        else begin {create nonterminal and evaluate further down}
            u.index := Min(vlow1.index, vhigh1.index, v2.index);
            if vlow1.index = u.index
                then begin vll1 := vlow1.low; vlh1 := vlow1.high end
                else begin vll1 := vlow1; vlh1 := vlow1 end;
            if vhigh1.index = u.index
                then begin vhl1 := vhigh1.low; vhh1 := vhigh1.high end
                else begin vhl1 := vhigh1; vhh1 := vhigh1 end;
            if v2.index = u.index
                then begin vlow2 := v2.low; vhigh2 := v2.high end
                else begin vlow2 := v2; vhigh2 := v2 end;
            u.low := Compose-step(vll1, vhl1, vlow2);
            u.high := Compose-step(vlh1, vhh1, vhigh2);
        end;
        return(u);
    end;

begin   {Main routine}
    Initialize all elements of T to null;
    u := Compose-step(v1, v1, v2);
    return(Reduce(u));
end;
```

Figure 8.Implementation of *Compose*

Hence, care must be exercised in invoking this algorithm. If we wish to find an element of the satisfying set obeying some property, it can be very inefficient to enumerate all elements of the satisfying set and then pick out an element with the desired characteristics. Instead, we should specify this property in terms of a Boolean function, compute the Boolean product of this function and the original function, and then use the procedure *Satisfy-one* to select an element. Finally, we can compute the size of the satisfying set by an algorithm of time proportional to the size of the graph (assuming integer operations of sufficient precision can be performed in constant time.) In general, it is much faster to apply this algorithm than to enumerate all elements of the satisfying set and count them.

The procedure *Satisfy-one* shown in Figure 9 is called with the root of the graph and an array x initialized to some arbitrary pattern of 0's and 1's. It returns the value false if the function is unsatisfiable ($S_f = \emptyset$), and the value true if it is. In the latter case, the entries in the array are set to a set of values denoting some element in the satisfying set. This procedure utilizes a classic depth-first search with backtracking scheme to find a terminal vertex in the graph having value 1. This procedure will work on any function graph. However, when called on an unreduced function graph, this could require time exponential in n (consider a complete binary tree where only the final terminal vertex in the search has value 1). For a reduced graph, however, we can assume the following property:

Lemma 3: Every nonterminal vertex in a reduced function graph has a terminal vertex with value 1 as a descendant.

The procedure will only backtrack at some vertex when the first child it tries is a terminal vertex with value 0, and in this case it is guaranteed to succeed for the second child. Thus, the complexity of the algorithm is $O(n)$.

```
function Satisfy-one(v: vertex; var x: array[1..n] of integer): boolean
begin
    if v.value = 0 then return(false); {failure}
    if v.value = 1 then return(true); {success}
    x[i] := 0;
    if Satisfy-one(v.low, x) then return(true);
    x[i] := 1;
    return(Satisfy-one(v.high, x))
end;
```

Figure 9.Implementation of *Satisfy-one*

To enumerate all elements of the satisfying set, we can perform an exhaustive search of the graph, printing out the element corresponding to the current path every time we reach a terminal vertex with value 1. The procedure *Satisfy-all* shown in Figure 10 implements this method. This procedure has three arguments: the index of the current function argument in the enumeration, the root vertex of the subgraph being searched, and an array describing the state of the search. It is called at the top level with index 1, the root vertex of the graph, and an array with arbitrary initialization. The effect of the procedure when invoked with index i, vertex v, and with the array having its first $i-1$ elements equal to $b_1, \ldots, b_{i-1}$ is to enumerate all elements in the set

$$\{(b_1, \ldots, b_{i-1}, x_i, \ldots, x_n) \mid f_v(b_1, \ldots, b_{i-1}, x_i, \ldots, x_n) = 1\}.$$

As with the previous algorithm, this procedure will work for any function graph, but it could require time exponential in n for an unreduced graph regardless of the size of the satisfying set (consider a complete binary tree with all terminal vertices having value 0.) For a reduced graph, however, we are guaranteed that the search will only fail when the procedure is called on a terminal vertex with value 0, and in this case the recursive call to the other child will succeed. Hence at least half of the recursive calls to *Satisfy-all* generate at least one new argument value to some element in the satisfying set, and the overall complexity is $O(n \cdot |S_f|)$.

Finally, to compute the size of the satisfying set, we assign a value α_v to each vertex v in the graph according to the following recursive formula:

1. If v is a terminal vertex: $\alpha_v = value(v)$
2. If v is a nonterminal vertex[10]:

$$\alpha_v = \alpha_{low(v)} \cdot 2^{index(low(v)) - index(v)} + \alpha_{high(v)} \cdot 2^{index(high(v)) - index(v)}$$

3. where a terminal vertex has index $n+1$.

This computation can be performed by a procedure that traverses the graph in the manner of the procedure *Traverse*. The formula is applied only once for each vertex in the graph, and hence the total time complexity is

[10]**Update:** Adnan Darwiche of UCLA (private correspondence, 2001) observed that the formula given in the paper is incorrect. Here is the correct one:

$$\alpha_v = \alpha_{low(v)} \cdot 2^{index(low(v)) - index(v) - 1} + \alpha_{high(v)} \cdot 2^{index(high(v)) - index(v) - 1}$$

```
procedure Satisfy-all(i: integer; v: vertex; x: array[1..n] of integer):
begin
    if v.value = 0 then return; {failure}
    if i = n+1 and v.value = 1
    then begin {success}
        Print element x[1],...,x[n];
        return;
    end;
    if v.index > i
    then begin {function independent of x_i}
        x[i] := 0; Satisfy-all(i+1, v, x);
        x[i] := 1; Satisfy-all(i+1, v, x);
    end
    else begin {function depends on x_i}
        x[i] := 0; Satisfy-all(i+1, v.low, x);
        x[i] := 1; Satisfy-all(i+1, v.high, x);
    end;
end;
```

Figure 10.Implementation of *Satisfy-all*

$O(|G|)$ Once we have computed these values for a graph with root v, we compute the size of the satisfying set as

$$|S_f| \;=\; \alpha_v \cdot 2^{index(v)-1}.$$

5. Experimental Results

As with all other known algorithms for solving NP-hard problems, our algorithms have a worst-case performance that is unacceptable for all but the smallest problems. We hope that our approach will be practical for a reasonable class of applications, but this can only be demonstrated experimentally. We have already shown that the size of the graph representing a function can depend heavily on the ordering of the input variables, and that our algorithms are quite efficient as long as the functions are represented by graphs of reasonable size. Hence, the major questions to be answered by our experimental investigation are: how can an appropriate input ordering be chosen, and given a good ordering how large are the graphs encountered in typical applications.

We have implemented the algorithms described in this paper and have applied them to problems in logic design verification, test pattern generation, and combinatorics. On the whole, our experience has been quite favorable. By analyzing the problem domain, we can generally develop strategies for choosing a good ordering of the inputs. Furthermore, it is not necessary to find *the* optimal ordering. Many orderings will produce acceptable results. Functions rarely require graphs of size exponential in the number of inputs, as long as a reasonable ordering of the inputs has been chosen. In addition, the algorithms are quite fast, remaining practical for graphs with as many as 20,000 vertices.

For this paper, we consider the problem of verifying that the implementation of a logic function (in terms of a combinational logic gate network) satisfies its specification (in terms of Boolean expressions.) As examples we use a family of Arithmetic Logic Unit (ALU) designs constructed from 74181 and 74182 TTL integrated circuits [15]. The ’181 implements a 4 bit ALU slice, while the ’182 implements a lookahead carry generator. These chips can be combined to create an ALU with any word size that is a multiple of 4 bits. An ALU with an n bit word size has $6+2n$ inputs: 5 control inputs labeled m, s_0, s_1, s_2, s_3 to select the ALU function, a carry input labeled *cin*, and 2 data words of n bits each, labeled $a_0, \ldots, a_{n-1}$ and $b_0, \ldots, b_{n-1}$. It produces $n+2$ outputs: n function outputs labeled $f_0, \ldots, f_{n-1}$, a carry output labeled *cout*, and a comparison output labeled $A=B$ (the logical AND of the function outputs.)

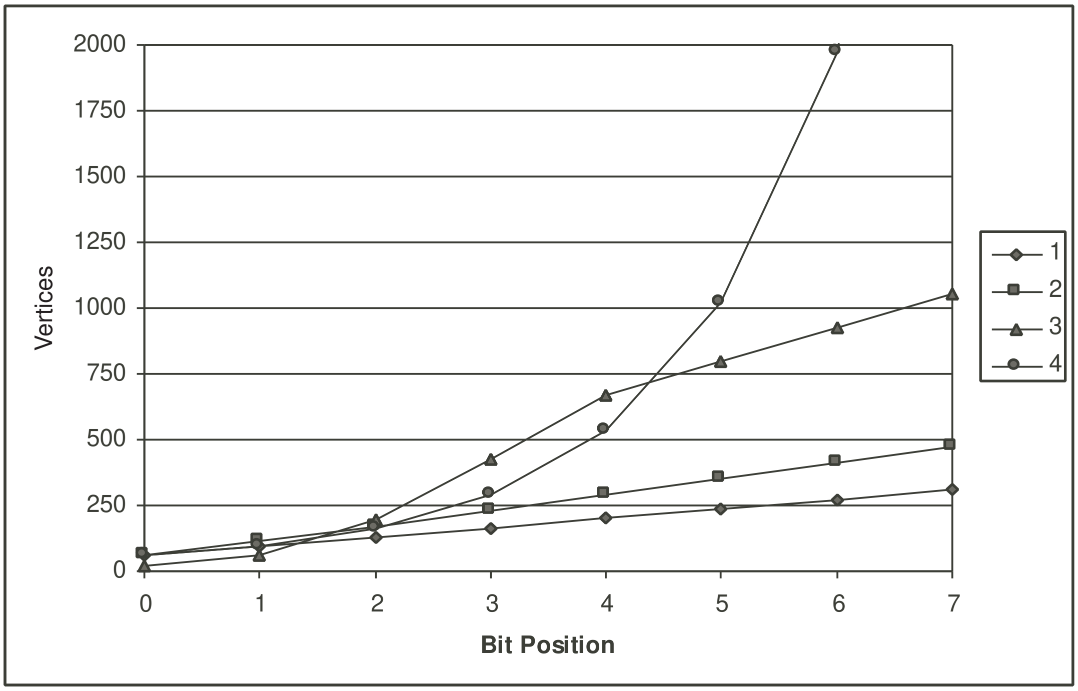
Word Size	**Gates**	**Patterns**	**CPU Minutes**	***A*=*B* Graph**
4	52	1.6×10^{4}	1.1	197
8	123	4.2×10^{6}	2.3	377
16	227	2.7×10^{11}	6.3	737
32	473	1.2×10^{21}	22.8	1457
64	927	2.2×10^{40}	95.8	2897

Table 2.ALU Verification Examples

For our experiments, we derived the functions for the two chips from their gate-level descriptions and then composed these functions to form the different ALU's according to the chip-level interconnections in the circuit manual. We then compared these circuit functions to functions derived from Boolean expressions obtained by encoding the behavioral specification in the circuit manual. We succeeded in verifying ALU's with word sizes of 4, 8, 16, 32, and 64 bits. The performance of our program for this task is summarized Table 2. These data were measured with the best ordering we were able to find, which happened to be the first one we tried: first the 5 control inputs, then the carry input, and then an interleaving of the two data words from the least significant to the most. In this table, the number of gates is defined as the number of logic gates in the schematic diagrams for the two chips times the number of each chip used. The number of patterns equals the number of different input combinations. CPU time is expressed in minutes as measured on a Digital Equipment Corporation VAX 11/780 (a 1 MIP machine.) The times given are for complete verification, i.e. to construct the functions from both the circuit and the behavioral descriptions and to establish their equivalence. The final column shows the size of the reduced graph for the $A=B$ output. In all cases, this was the largest graph generated.

As can be seen, the time required to verify these circuits is quite reasonable, in part because the basic procedures are fast. Amortizing the time used for memory management, for the user interface, and for reducing the graphs, each call to the evaluation routines *Apply-step* and *Compose-step* requires around 3 milliseconds. For example, in verifying the 64 bit ALU, these two procedures were called over 1.6×10^{6} times. The total verification time grows as the square of the word size. This is as good as can be expected: both the number of gates and the sizes of the graphs being operated on grow linearly with the word size, and the total execution time grows as the product of these two factors. This quadratic growth is far superior to the exponential growth that would be required for exhaustive analysis. For example, suppose that at the time the universe first formed (about 20 billion years ago [16]) we started analyzing the 32 bit ALU exhaustively at a rate of one pattern every microsecond. By now we would be about half way through! For the 64 bit ALU, the advantage over exhaustive analysis is even greater.

These ALU circuits provide an interesting test case for evaluating different input orderings, because the successive bits of the function output word are functions of increasingly more variables. Figure 11 shows how the sizes of these graphs depend on the ordering of circuit inputs. The best case was obtained for the ordering: $m, s_0, s_1, s_2, s_3, cin, a_0, b_0, \ldots, a_{n-1} b_{n-1}$. This ordering is also what one would choose for a bit-serial implementation of the ALU: first read in the bits describing the function to be computed, and then read in the successive bits of the two data words starting with the least significant bits. Hence, our bit-serial computer analogy presented in Section 3 guides us to the best solution. The next best case tested occurred with the ordering: $m, s_0, s_1, s_2, s_3, cin, a_{n-1}, b_{n-1}, \ldots, a_0 b_0$, i.e. the same as before but with the data ordered with the most significant bit first. This ordering represents an alternative but often successful strategy: order the bits in decreasing order of importance. The i th bit of the output word depends more strongly on the i th bits of the input words than any lower order bits. As can be seen, this strategy also works quite well. The next case shown is for an ordering with the control inputs last: $cin, a_0, b_0, \ldots, a_{n-1} b_{n-1}, m, s_0, s_1, s_2, s_3$. This ordering could be expected to produce a rather poor result, since the outputs depend strongly on the control inputs. However, the complexity of the graphs still grows linearly, due to the fact that the number of control inputs is a constant. To explain this linear growth in terms of the



Input Orderings:

1: $m,s_0,s_1,s_2,s_3,cin,a_0,b_0, \ldots ,a_{n-1}b_{n-1}$

2: $m,s_0,s_1,s_2,s_3,cin,a_{n-1},b_{n-1}, \ldots ,a_0b_0$

3: $cin,a_0,b_0, \ldots ,a_{n-1}b_{n-1},m,s_0,s_1,s_2,s_3$

4: $m,s_0,s_1,s_2,s_3,cin,a_0, \ldots ,a_{n-1}b_0, \ldots ,b_{n-1}$

Figure 11.ALU Output Graph Sizes for Different Input Orderings

bit-serial processor analogy, we could implement the ALU with the control inputs read last by computing all 32 possible ALU functions and then selecting the appropriate result once the desired function is known. The final case shows what happens if a poor ordering is chosen, in this case the ordering $m,s_0,s_1,s_2,s_3,cin,a_0, \ldots ,a_{n-1}b_0, \ldots ,b_{n-1}$. This ordering requires the program to represent functions similar to the function $x_1{\cdot}x_{n+1} + \cdots + x_n{\cdot}x_{2n}$ considered in Section 3, with the same exponential growth characteristics.

These experimental results indicate that our representation works quite well for functions representing addition and logical operations on words of data, as long as we choose an ordering in which the successive bits of the input words are interleaved. Our representation seems especially efficient when compared to other representations of Boolean functions. For example, a truth table representation would be totally impractical for ALU's with word sizes greater than 8 bits. A reduced sum-of-products representation of the most significant bit in the sum of two n bit numbers requires about 2^{n+2} product terms, and hence a reduced sum-of-products representation of this circuit would be equally impractical.

6. Conclusion

We have shown that by taking a well-known graphical representation of Boolean functions and imposing a restriction on the vertex labels, the minimum size graph representing a function becomes a canonical form. Furthermore, given any graph representing a function, we can reduce it to a canonical form graph in linear time. Thus our reduction algorithm not only minimizes the amount of storage required to represent a function and the time required to perform symbolic operations on the function, it also makes such tasks as testing for equivalence, satisfiability, or tautology very simple. We have found this property valuable in many applications.

We have presented a set of algorithms for performing a variety of operations on Boolean functions represented by our data structure. Each of these algorithms obeys an important closure property---if the argument graphs satisfy our ordering restrictions, so does the result graph. By combining concepts from Boolean algebra with techniques from graph algorithms, we achieve a high degree of efficiency. That is, the performance is limited more by the sizes of the data structures rather than by the algorithms that operate on them.

Akers [17] has devised a variety of coding techniques to reduce the size of the binary decision diagrams representing the output functions of a system. For example, he can represent the functions for all 8 outputs of the 74181 ALU slice by a total of 35 vertices, whereas our representation requires 918. Several of these techniques could be applied to our representation without violating the properties required by our algorithms. We will discuss two such refinements briefly.

Most digital systems contain multiple outputs. In our current implementation we represent each output function by a separate graph, even though these function may be closely related and therefore have graphs containing isomorphic subgraphs. Alternatively, we could represent a set of functions by a single graph with multiple roots (one for each function.)[11] Our reduction algorithm could be applied to such graphs to eliminate any duplicate subgraphs and to guarantee that the subgraph consisting of a root and all of its descendants is a canonical representation of the corresponding function. For example, we could represent the n+1 functions for the addition of two n-bit numbers by a single graph containing $9n-1$ vertices (assuming the inputs are ordered most significant bits first), whereas representing them by separate graphs requires a total of $3n^2+6n+2$ vertices. Taking this idea to an extreme, we could manage our entire set of graphs as a single shared data structure, using an extension of the reduction algorithm to merge a newly created graph into this structure. With such a structure, we could enhance the performance of the *Apply* procedure by maintaining a table containing entries of the form $(v_1, v_2, \langle op \rangle, u)$ indicating that the result of applying operation $\langle op \rangle$ to graphs with roots v_1 and v_2 was a graph with root u. In this way we would exploit the information generated by previous invocations of *Apply* as well as by the current one. These savings in overall storage requirements and algorithm efficiencies would be offset somewhat by a more difficult memory management problem, however.

Akers also saves storage by representing functions in decomposed form. That is, we can represent the function $f|_{x_i=g}$ in terms of f and g (which may themselves be represented in decomposed form). Unfortunately, a given function can be decomposed in many different ways, and hence this technique would not lead to a canonical form. However, as noted on page 16 there are certain instances in which functions f and g can be composed in a straightforward way by simply replacing each vertex representing the composition variable x_i with the graph for g. For such decompositions, functions could be stored in decomposed form and expanded into canonical form dynamically as operations are performed on them. In some instances, the storage savings could be considerable. For example the graph for the function

$$x_1 \cdot x_{2n} + x_2 \cdot x_{2n-1} + \ . \ . \ . \ + x_n \cdot x_{n+1}$$

requires a total of 2^{n+1} vertices. This function can be decomposed as a series of functions where $f_n = x_n \cdot x_{n+1}$,

$$f_i \ = \ (x_i \cdot x_{2n-i+1} + x_{i+1})|_{x_{i+1}=f_{i+1}}$$

for $n > i \geq 1$, and f_1 equals the desired function. Each of these functions can be represented by graphs with 6 vertices. It is unclear, however, how often such decompositions occur, how easy they are to find, and how they would affect the efficiency of the algorithms.

[11] **Update:** This idea was subsequently termed a "Shared BDD" (S. Minato, N. Ishiura, and S. Yajima, "Shared Binary Decision Diagram with Attributed Edges," *27th Design Automation Conference*, 1990, pp. 52-57.) It has now become the most common implementation method.

Appendix: The Complexity of Integer Multiplication

In this appendix, we prove that the functions representing the outputs of an integer multiplier provide a difficult case for our representation, i.e. the graph sizes grow exponentially in the word size regardless of the ordering of the input variables. Given that there are $(2n)!$ possible orderings of the input variables, we could not hope to derive this result experimentally, and hence we must provide a detailed proof.

Our proof is based on principles similar to those used in proving area-time lower bounds on multiplier circuits [18, 19]. However, we must show not just that a large amount of information must be transferred from the set of inputs to the set of outputs in performing multiplication, but that certain individual outputs require high information transfer.

Consider a multiplier with inputs $a_1, \ldots, a_n$ and $b_1, \ldots, b_n$ corresponding to the binary encoding of integers a and b with a_1 and b_1 being the least significant bits. This circuit has $2n$ outputs corresponding to the binary encoding of the product $a \cdot b$, described by functions $mul_i(a_1, \ldots, a_n, b_1, \ldots, b_n)$ for $1 \le i \le 2n$. For a permutation π of $\{1, \ldots, 2n\}$, let $G(i,\pi)$ be a graph which for inputs $x_1, \ldots, x_{2n}$ denotes the function $mul_i(x_{\pi(1)}, \ldots, x_{\pi(2n)})$

Theorem 2: For any π there exists an i, $1 \le i \le 2n$ such that $G(i,\pi)$ contains at least $2^{n/8}$ vertices.

Proof : Informally, our proof proceeds as follows. If one input (the "control") to a multiplier is a power of 2 then the circuit acts as a shifter, transferring the bits of the other input (the "data") to the output with some offset. For example, if $b = 2^j$, then

$$[a \cdot b]_i = \begin{cases} a_{i-j} & j < i \le j+n \\ 0 & \text{else} \end{cases}$$

Graph $G(i,\pi)$ must contain enough vertices to encode all values of a_{i-j} for which a_{i-j} occurs in the first half of the input sequence while b_j occurs in the second. Furthermore, we can show that for any ordering of input variables, we can choose which input (a or b) is control and which is data such that for some output i, this undesirable splitting occurs for at least $n/8$ values of j.

More formally, for permutation π let

$$t = |\{ j \mid 1 \le j \le n, \pi(j) \le n \}|,$$

i.e. the number of bits of argument a occurring in the first half of the input sequence. If $t \ge n/2$ define sets F and L as

$$F = \{ \pi(j) \mid 1 \le j \le n, \pi(j) \le n \}$$

$$L = \{ \pi(j) \mid n+1 \le j \le 2n, \pi(j) > n \}$$

That is, F represents those the indices of argument a occurring in the first half of the input sequence, while L represents those indices of b (with n added to them) occurring in the second half. If $t < n/2$ then define F and L as

$$F = \{ \pi(j) \mid 1 \le j \le n, \pi(j) > n \}$$

$$L = \{ \pi(j) \mid n+1 \le j \le 2n, \pi(j) \le n \}$$

That is, F represents those indices of b (with n added to them) occurring in the first half while L represents those indices of a occurring in the second. In either case the sets F and L will each contain at least $n/2$ elements. We will consider the elements of F to be data inputs and those of L to be control. Since multiplication is commutative, we are free to choose which argument is considered the control input and which is considered the data in our proof.

For $1 \le i \le 2n-1$ define the set F_i as

$$F_i = \{\, j \mid j \in F,\ \exists k \in L\,(j+k=i+n+1)\,\}$$

and let $q_i = |F_i|$. That is, for output i, F_i represents those indices of the data input occurring in the first half of the input sequence such that the corresponding bits of the control input occur in the second half.

Now consider the set of sequences

$$S_i = \{x_1, \ldots x_n \mid x_j = 0 \text{ if } \pi(j) \notin F_i\}$$

This set contains 2^{q_i} possible values for the first n inputs. We claim that $G(i,\pi)$ must contain a unique vertex for each element of S_i. If this were not the case, then we could choose two sequences $x_1, \ldots, x_n$ and $x'_1, \ldots, x'_n$ leading to the same vertex in $G(i,\pi)$ such that for some value j, $\pi(j) \in F_i$ and $x_j \neq x'_j$. Now consider the sequences $x_{n+1}, \ldots, x_{2n}$ and $x'_{n+1}, \ldots, x'_{2n}$ defined as

$$x_k = x'_k = \begin{cases} 1, & \pi(j)+\pi(k) = i+n+1 \\ 0, & \text{else} \end{cases}$$

Note that x_k and x'_k equal 1 for exactly one value of k. The sequences $x_1, \ldots, x_{2n}$ and $x'_1, \ldots, x'_{2n}$ lead to the same terminal vertex in $G(i,\pi)$, but

$$mul_i\,(x_{\pi(1)}, \ldots, x_{\pi(2n)}) = x_j$$

while

$$mul_i\,(x'_{\pi(1)}, \ldots, x'_{\pi(2n)}) = x'_j \neq x_j.$$

This contradiction forces us to conclude that graph $G(i,\pi)$ must contain at least 2^{q_i} vertices.

To get the final result, we need only show that for some value of i, $q_i \geq n/8$. This involves a counting argument expressed by the following lemma.

Lemma 4: Suppose $A, B \subseteq \{1, \ldots, n\}$ each contain at least $n/2$ elements. For $1 \leq i \leq 2n-1$ let

$$q_i = |\{\, \langle a,b \rangle \mid a \in A,\ b \in B,\ a+b = i+1 \,\}|$$

Then there is some i such that $q_i \geq n/8$.

Proof : Observe that since the sets A and B each contain at least $n/2$ elements, there are at least $n^2/4$ ordered pairs $\langle a,b \rangle$ with $a \in A$, $b \in B$. Hence

$$\sum_{j=1}^{2n-1} q_j \geq \frac{n^2}{4}$$

For some value of i, q_i must be at least as large as the average value of the q_j's:

$$q_i \geq \frac{1}{2n-1} \cdot \frac{n^2}{4} \geq \frac{n}{8}.$$

This theorem shows that for any ordering, some multiplier output will have a graph of exponential size. This leaves open the possibility that for each output, there could be some ordering giving a polynomial size graph for this output. We conjecture, however, that this is not the case, namely that for certain outputs (e.g. output n), the graph for this function is of exponential size regardless of the ordering.[12] A proof of this conjecture would require something stronger than our simple shifter argument. Such a proof would also lead to an interesting area-time lower bound on circuits computing single bits in the product of two binary numbers.

---

[12]**Update:** This conjecture was subsequently proved by the author (R. E. Bryant, "On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Application to Integer Multiplication," *IEEE Transactions on Computers* 40-2, pp. 205-213, Feb., 1991.)

References

1. C.Y. Lee, ‘‘Representation of Switching Circuits by Binary-Decision Programs’’, *Bell System Technical Journal*, Vol. 38, July 1959, pp. 985-999.

2. S.B. Akers, ‘‘Binary Decision Diagrams’’, *IEEE Transactions on Computers*, Vol. C-27, No. 6, June 1978, pp. 509-516.

3. M.R. Garey and D.S. Johnson, *Computers and Intractibility: A Guide to the Theory of NP-Completeness,* Freeman, New York, 1979.

4. F.J. Hill and G.R. Peterson, *Introduction to Switching Theory and Logical Design,* Wiley, New York, 1974.

5. J.P. Roth, *Computer Logic, Testing, and Verification,* Computer Science Press, Potomac, MD., 1980.

6. R. Brayton, *et al*, ‘‘Fast Recursive Boolean Function Manipulation’’, *International Symposium on Circuits and Systems*, IEEE, Rome, Italy, May 1982, pp. 58-62.

7. B.M.E. Moret, ‘‘Decision Trees and Diagrams’’, *ACM Computing Surveys*, Vol. 14, No. 4, December 1982, pp. 593-623.

8. S. Fortune, J. Hopcroft, and E.M. Schmidt, ‘‘The Complexity of Equivalence and Containment for Free Single Variable Program Schemes’’, in *Automata, Languages, and Programming,* Goos, Hartmannis, Ausiello, and Boehm, eds., Springer-Verlag, Lecture Notes in Computer Science, Vol. 62, 1978, pp. 227-240.

9. R.W. Payne, ‘‘Reticulation and Other Methods of Reducing the Size of Printed Diagnostic Keys’’, *Journal of General Microbiology*, Vol. 98, 1977, pp. 595-597.

10. R. Brayton, *et al*, *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, 1984.

11. C.E. Shannon, ‘‘A Symbolic Analysis of Relay and Switching Circuits’’, *Transactions of the AIEE*, Vol. 57, 1938, pp. 713-723.

12. C.S. Wallace, ‘‘A Suggestion for a Fast Multiplier’’, *IEEE Transactions on Electronic Computing*, Vol. EC-13, No. 1, January 1964, pp. 14-17.

13. A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms,* Addison-Wesley, Reading, MA., 1974.

14. J.S. Jephson, R.P. McQuarrie, and R.E. Vogelsberg, ‘‘A Three-Level Design Verification System’’, *IBM Systems Journal*, Vol. 8, No. 3, 1969, pp. 178-188.

15. Texas Instruments, *TTL Data Book,* 1976.

16. M. Rowan-Robinson, *Cosmology,* Oxford Univ. Press, 1977.

17. S.B. Akers, ‘‘Functional Testing with Binary Decision Diagrams’’, *Eighth Annual Conference on Fault-Tolerant Computing*, IEEE, 1978, pp. 75-82.

18. R.P. Brent, and H.T. Kung, ‘‘The Area-Time Complexity of Binary Multiplication’’, *Journal of the ACM*, Vol. 28, No. 3, July 1981, pp. 521-534.

19. H. Abelson, and P. Andreae, ‘‘Information Transfer and Area-Time Trade-Offs for VLSI Multiplication’’, *Communications of the ACM*, Vol. 23, No. 1, January 1980, pp. 20-23.

Table of Contents

List of Figures

List of Tables