Formal Methods in System Design 10, 149–169 (1997)

Multi-Terminal Binary Decision Diagrams: An Efficient Data Structure for Matrix Representation

M. FUJITA
Fujitsu Laboratories of America, 3350 Scott Blvd., Bldg #34, Santa Clara, CA 95054

P.C. MCGEER
Cadence Berkeley Laboratories, 1919 Addison St. #303, Berkeley, CA 94704

J.C.-Y. YANG
Center for Integrated Systems, Stanford University
On Leave Currently at YAHOO! Inc.

Abstract. In this paper, we discuss the use of binary decision diagrams to represent general matrices. We demonstrate that binary decision diagrams are an efficient representation for every special-case matrix in common use, notably sparse matrices. In particular, we demonstrate that for any matrix, the BDD representation can be no larger than the corresponding sparse-matrix representation. Further, the BDD representation is often smaller than any other conventional special-case representation: for the $n \times n$ Walsh matrix, for example, the BDD representation is of size $O(\log n)$. No other special-case representation in common use represents this matrix in space less than $O(n^2)$. We describe termwise, row, column, block, and diagonal selection over these matrices, standard an Strassen matrix multiplication, and LU factorization. We demonstrate that the complexity of each of these operations over the BDD representation is no greater than that over any standard representation. Further, we demonstrate that complete pivoting is no more difficult over these matrices than partial pivoting. Finally, we consider an example, the Walsh Spectrum of a Boolean function.

Keywords: Binary Decision Diagrams, Matrix Algorithms, Multi-Terminal BDD's, Walsh Transform, Spectral Methods

1. Introduction

Binary Decision Diagrams (BDD's) are a data structure that has been used for years to provide a cogent representation of Boolean functions. Indeed, they are now such a common part of computer-aided design research that one can safely assume a working knowledge of BDD's on the part of virtually any reader.

BDD's were introduced by Akers in 1959 [2]. In the early 1980's, searching for a data structure to undergird his switch-level simulator, Bryant [3] demonstrated how a BDD could be modified to become a canonical representation of a Boolean function. Bryant also demonstrated, given BDD's representing two functions f and g, how to compute the functions $f \diamond g$, where $\diamond$ is any of the common binary Boolean operators. Finally, Bryant demonstrated, in a famous theorem, that $|f \diamond g| \leq |f||g|$.

Bryant's paper, together with the observation that almost every common function could be built with a reasonably-sized BDD, opened the floodgates. In 1988, Malik et al. [9]

demonstrated that BDD's could be used to verify large multi-level combinational logic functions. In 1990, Coudert, and Madre [6] demonstrated that one could represent sets of finite-state machine states cogently using BDD's, and then described an interation in which one could find the reachable states of a finite-state machine, again represented as a BDD. The key idea was confirmed in separate experiments by Touati et al. [14]. In 1992, Coudert and Madre [7] demonstrated a method by which the primes of very large Boolean functions could be implicitly represented using BDD's. In 1993, these techniques were extended to a full implicit logic minimizer [13, 8].

This year also saw the exploration of the relationship between BDD's and matrices. In [10], it is argued that any n-variable BDD can be thought of as a vector of length 2^n, or as a binary matrix of size $2^{n-1} \times 2^{n-1}$, or, for that matter, as an object of almost any dimensionality. In [10], a BDD for a finite-state machine transition function is thought of as a two-dimensional matrix, and a classic matrix technique is used to find its transitive closure. In [4, 5], it is observed that though a BDD is generally thought to take only terminals 0 and 1, there is no reason for this restriction; a BDD can have arbitrary integer terminals, and there is no reason why a BDD should be restricted to only two terminals; using independently the observations of [10], they argue that any integer matrix or vector can be represented as a BDD, and give BDD representations for the Walsh matrix and for the Walsh spectrum of a Boolean function.

We take the ideas introduced in [4, 5] and extend and expand upon them. Specifically, we observe that there is no reason to restrict the terminals of a BDD to the integers; any finite set will do. This is a minor contribution. Our central contribution, however, is to begin to answer the following question. Given that we *can* represent vectors and matrices as BDD's, should we? Is the BDD representation cogent? Do our matrix algorithms translate well onto this new representation?

Our initial answer to these question is remarkably affirmative. Over the next several pages, we will demonstrate that BDD's are a cogent representation of matrices and vectors. Further, we will demonstrate that many popular matrix algorithms translate easily and naturally onto the BDD representation, occasionally enjoying a clean advantage over other representations.

We do not purport to "prove" in any real sense that BDD's are a superior representation of general matrices. This paper is not the end, but rather the beginning of our inquiry. Rather, our hope is to demonstrate some advantages and spur further research into this area. It is our hope that the BDD representation of vectors and matrices will prove as fruitful and fulfilling an area for CAD researchers in synthesis, physical design, and simulation as research into sparse matrices has been.

2. Multi-terminal binary decision diagrams

The root of BDD's lies in the Shannon Cofactor Expansion of a Boolean function. Given a Boolean function $f : \mathbf{B}^n \mapsto \mathbf{B}$, the *cofactors* of f are the functions obtained by partial evaluation of f. For example, the cofactor $f_{x_1\overline{x_2}x_4}$ is the function obtained by setting $x_1 = 1, x_2 = 0, x_4 = 1$ and evaluating f.

Cofactors are given their importance by the famous Shannon Expansion of a function. Given any boolean function $f(x_1, \ldots, x_n)$, and any variable x_i, we may write:

$$f = x_i f_{x_i} + \overline{x_i} f_{\overline{x_i}} \tag{1}$$

(1), carried out recursively, gives a full binary tree with leaves 0 and 1. Each internal node represents a function, its left child its cofactor with respect to $\overline{x_i}$ for some variable x_i, and its right child its cofactor with respect to x_i. This tree is called a *Shannon Tree* of f.

Bryant's seminal contribution was to order the variables of the function, so at the kth level from the root the cofactors with respect to x_k were taken. Then, applying the well-known theorem:

$$g \equiv h \Longleftrightarrow g_{x_k} \equiv h_{x_k} \quad \text{and} \quad g_{\overline{x_k}} \equiv h_{\overline{x_k}} \tag{2}$$

and working up from the leaves, he found the set of distinct functions in the tree. He then obtained the BDD for f by folding together nodes representing identical functions into a single node.

The central idea behind BDD's can be easily adapted to more general functions. It is straightforward to define functions from the boolean space $\mathbf{B}^n$ onto $\tilde{R}$, where $\tilde{R}$ is any finite set; in general, we consider $\tilde{R}$ to be an arbitrary finite subset of the reals. It is fairly easy to see that (1)–(2) hold for any function $f : \mathbf{B}^n \mapsto \tilde{R}$, and so in precisely the same manner as in the two-terminal case, the Shannon Tree for such a function may be formed and converted into its BDD representation; the only distinction between a BDD representing a function onto the Booleans and one representing a function onto a finite subset of the reals is that the latter has multiple leaves, not two as in the former case. As a result, we christen BDD's with leaves other than 0 and 1 as *Multi-Terminal* BDD's, or MTBDD's.

A moment's thought persuades the reader that MTBDD's not only represent functions from a Boolean space $\mathbf{B}^n$ onto a finite set $\tilde{R}$, but, more generally, functions from any finite space $\tilde{D} \mapsto \tilde{R}$; one simply encodes the members of $\tilde{D}$ using $\lceil \lg |\tilde{D}| \rceil$ variables. Our interest is in the case where $\tilde{D}$ is the finite set of the integers $\{0, \ldots, m-1\}$, or the case where it is the finite set $\{0, \ldots, m-1\} \times \{0, \ldots, n-1\}$. In the former case, $f : \tilde{D} \mapsto \tilde{R}$ is a vector; in the latter, a matrix.

To make this picture concrete, consider a vector v of length m; the vector is indexed by an integer of $\lceil \lg m \rceil$ bits: One can think of each bit of the index as representing a separate Boolean variable; the vector thus becomes a function from the Boolean space $\mathbf{B}^{\lceil \lg m \rceil}$ onto the range of the vector, and this can be represented as an MTBDD.

In representing a vector as a BDD, of course, some information is lost: specifically, the dimensionality of the vector or matrix and the size of the vector/matrix in each dimension. However, this information can be kept implicitly in a number of ways, and for the remainder of this paper will be understood.

Implicitly, when given a vector with index bits $\{x_1, \ldots, x_s\}$, we will assume that x_1 is the most significant bit, and x_s the least. For two-dimensional matrices, we by convention denote the row index bits with the variables $\{x_1, \ldots, x_s\}$ and the column index bits with the variables $\{y_1, \ldots, y_t\}$, ordered as before most significant to least significant. A felicitous ordering interleaves the row and column variables; i.e., a matrix is conceptually a function

$f(x_1, y_1, x_2, y_2, \ldots)$ or $f(y_1, x_1, \ldots)$. This order on the boolean variables associated with the index bits is not required, and may not be ideal for some matrices. This order, however, leads to the following identification of the cofactors on the matrix M, when represented by the function f as an MTBDD:

$$\begin{pmatrix} f_{\overline{x_1}\,\overline{y_1}} & f_{\overline{x_1}\,\overline{y_1}} \\ f_{x_1\overline{y_1}} & f_{x_1 y_1} \end{pmatrix}$$

where each submatrix is similarly decomposed.

Note that this ordering identifies each node in the MTBDD at level k from the root with a rectangular submatrix of size 2^{s+t-k}. This identity is critical, for the following reasons:

1. Significant savings in the MTBDD representation occur when nodes in the BDD have more than one parent; this corresponds to different cofactors in the Shannon tree mapping on to identical functions. In matrix terms, this occurs when identical submatrices occur in various parts of the original matrix. Since many of the special-case matrices that we actually use (block matrices, sparse matrices, band matrices) have this property, this gives some intuition to the idea that MTBDD's are a cogent form of these special-case matrices; and
2. Many matrix algorithms (LU decomposition, Strassen and standard matrix multiplication) are naturally phrased in terms of recursive-descent procedures on submatrices. Since the submatrices map naturally onto cofactors of the BDD, the translation onto BDD procedures is direct.

In the sequel, we will demonstrate the effect of (1) quantitatively, and (2) by giving BDD versions of efficient matrix procedures.

Notation: In the sequel, we will use superscripts to denote vertices on the Boolean space $\mathbf{B}^n$ and subscripts to denote individual variables. Hence x^k stands for an assignment of values to the variables $x_1, \ldots, x_s$, while x_k refers to one such variable.

Remark. For matrices A of size other than $2^n \times 2^m$, we use the standard trick [1] of attaching an identify submatrix:

$$\begin{bmatrix} A & 0 \\ 0 & I \end{bmatrix}$$

For vectors v of length other than 2^n, we attach a special element ϕ to represent missing entries:

$$[v\phi]$$

3. Size of MTBDD's as a matrix representation

We can derive an upper bound on the size of an MTBDD by counting the number of paths in the data structure. Since each path involves at most $\log n$ nodes, where n is the dimension

of the matrix (number of rows for a vector, number of rows or number of columns for a matrix), and since each node must lie along some path, it follows that in the MTBDD with p paths, there are at most $p \log n$ nodes.

This is a worst-case upper bound, and corresponds to the case where the MTBDD is simply the Shannon Tree; i.e., where the folding process has not resulted in any reduction of the tree.

Lemma 3.1. *Let $f : \mathbf{B}^n \mapsto \tilde{R}$ be any boolean vector function. Choose any element r of $\tilde{R}$. If there are k elements $x^1, \ldots, x^k$ of $\mathbf{B}^n$ such that $f(x^i) = r$, then there are at most k paths from the root of the MTBDD for f to r.*

Proof: Induction on n. For $n = 0$, trivial, since the only Boolean vector functions are the constant functions over $\tilde{R}$, and the MTBDD's for these functions have a single node and no paths. Now suppose for all $n < N$, and consider some function from $\mathbf{B}^N \mapsto R$. Let the left child of f be denoted f^L and the right child f^R. If there are k elements $x^1, \ldots, x^k$ of $\mathbf{B}^N$ such that $f(x^i) = r$, there are $0 \leq l < k$ elements of $\mathbf{B}^{N-1}$ such that $f^L(x^i) = r$, and $k - l$ elements of $\mathbf{B}^{N-1}$ such that $f^R(x^i) = r$. By the inductive hypothesis, there are at most l paths from f^L to the terminal r, and at most $k - l$ paths from f^R to the terminal r. Since each path from the root to r is an extension of either a path from the right child or a path from the left child, it follows that there are at most $l + k - l = k$ paths from the root to r. □

Lemma 3.2. *In any MTBDD, each non-terminal node must be on at least one path to each of two distinct terminal elements.*

Proof: This is obvious from a casual inspection of any BDD. A particularly fastidious reader can construct an induction, if desired. □

Theorem 3.1. *The MTBDD representation of a matrix with total dimension n is of space complexity $O(n \log n)$.*

Proof: Since there are n elements, there are at most n paths through the MTBDD by Lemma 3.1. Further, each path is of length $O(\log n)$. □

Theorem 3.2. *The MTBDD representation of a matrix of dimension n and m nonzero elements is of space complexity $O(m \log n)$.*

Proof: By Lemma 3.1, there are at most m paths terminating in a nonzero terminal in such an MTBDD; since each path is of length $O(\log n)$, there are at most $O(m \log n)$ internal nodes on these paths. Further, by Lemma 3.2, each internal node must be on a path to at least two terminals, i.e., on at least one path to a nonzero terminal. There are most $O(m \log n)$ nodes on these paths, and hence at most $O(m \log n)$ nodes in the MTBDD. □

Note that the standard sparse-matrix representation has at least one pointer per row, and one pointer per column, as well as space complexity proportional to the number of

nonzero elements; this gives total space complexity of $O(m+n)$; note that if $m < \frac{n}{\log n}$, the worst-case complexity of the MTBDD respresentation is superior to that of the standard sparse-matrix representation. In fact, the following lemma is of interest.

Theorem 3.3. *Let R be any representation of matrices of total dimension n with m non-trivial entries, $m \ll n$. Then there exists at least one matrix M of total dimension n with m nontrivial entries, $m \ll n$, such that $|M|_R = O(m \log n)$.*

Proof: Given that there are n possible positions in the matrix, and m total non-trivial entries, it follows that even if each non-trivial entry is identical there are $\binom{n}{m}$ such matrices. Since $m \ll n$, $\binom{n}{m} \approx n^m$ over the domain of interest. Any representation valid for k objects must be of size $O(\log k)$, hence the size of at least one matrix under R must be at least $O(\log n^m) = O(m \log n)$. □

Note that this implies that MTBDD's are the optimal representation for very sparse matrices.

A similar, but much more complex and longer argument, demonstrates that for a matrix made up of m constant blocks, total dimension n, the size is at most $O(m \log n)$.

4. Operations

Of course, no data structure is complete without a definition of the operations over it. In this section we describe a set of operations over MTBDD's, with complexity results.

4.1. On hashing

Computations over BDD's derive great efficiency from the idempotency of operations; the results of a BDD operation depend only upon the operands, not upon the context. As a result, modern BDD packages ensure that every BDD node represents a distinct function, and memorizes the results of every operation, typically using a hash table. If the operation is ever repeated, no computation is done: the result is directly returned. As a result, no operation is ever performed twice. This leads to complexity bounds in the case of termwise operations that are linear in the sizes of the operands.

We use hashing extensively in our matrix package; virtually every operation has its results remembered for later re-use.

4.2. Accessing and setting submatrices

In a matrix represented as an MTBDD, one accesses individual elements, column, row, and diagonal vectors, and submatrices using a single mechanism: partial evaluation. Full evaluation—setting each variable—obtains an individual element. Setting each x variable obtains a single row, and setting each y variable obtains a single column. Setting some x variables and some y variables obtains a submatrix.

Elements, rows, columns, and blocks are set using a similar scheme. Any submatrix is simply an MTBDD free of some variables: these variables are the indices of the submatrix within the larger matrix. The set routine takes in three arguments: the matrix itself (here denoted f), the submatrix to be inserted (g) and a list of variables and values which indicate where the submatrix g is to be inserted in f. As with all MTBDD routines, underlying this is the Shannon Expansion. `Set` is a recursive-descent procedure, which walks through the MTBDD representing the matrix f and creates the MTBDD with the appropriate submatrix set to g.

At each level of the recursion, `Set` examines the top (most-significant) variable of each of its three arguments. There are four cases.

1. `top_var(list)` $>$ `top_var(`f`)`. Return a BDD in which one cofactor is the `Set` of f with respect to g and the remainder of `list`, and the other is simply f.
2. `top_var(`g`)` $>$ `top_var(`f`)`. Return a BDD whose cofactors are the `Set` of f with respect to the cofactors of g.
3. x_i `= top_var(`f`) = top_var(`g`)`. In this case, the children of f are `Set` recursively appropriately with the children of `g`.
4. x_i `= top_var(`f`) = top_var(list)`. In this case, the appropriate cofactor of f is recursively `Set` to g.

In the case where more than one of these cases apply, the identity of the top variable determines the action. For example, if the top variable belonged to `list` alone, case (1) would apply even if case (3) applied as well.

The terminal case occurs when `list` is empty: in this case g is returned as the result.

It is easy to see that the cost of `Set` is bounded above by $O(|f||g|)$; this is a gross upper bound, and the expected cost is $O(\log n)$, where n is the total dimension of f.

4.3. Termwise operations

Many operations are performed termwise over the elements of a matrix: examples are matrix addition, matrix inner product, scalar multiplication, and (in the case of the binary matrices) the Boolean operations. Briefly, a termwise operation $\diamond$ over a matrix is any operation such that, for any pair of $n \times m$ matrices M and M', $(M \diamond M')_{ij} = M_{ij} \diamond M'_{ij}$.

For termwise operations, we simply use Bryant's `Apply` operator. This derives from the following classic theorem:

Theorem 4.1. *Let f, g be any vector Boolean functions, $f, g : \mathbf{B}^n \mapsto \tilde{R}$, $\diamond$ any termwise operator $h = f \diamond g$ iff $h_{x_i} == f_{x_i} \diamond g_{x_i}$ and $h_{\overline{x_i}} == f_{\overline{x_i}} \diamond g_{\overline{x_1}}$.*

This theorem permits the use of Bryant's `Apply` procedure, or Rudell's subsequent improvements. In this code, `newMTBDD` is a procedure which takes as input a variable and two MTBDD's, and returns an MTBDD indexed with the variable and whose left and right children are the two arguments.

In our BDD package, `newMTBDD` does not simply create a new BDD, but, rather, keeps a lookup table of existing BDD's, and if one is found that matches the requested BDD, the old

BDD is returned. In this manner there is exactly one BDD per function, which simplifies greatly the lookup computations that permeate the matrix package.

```
Apply_Operator(f, g, Op) {
    Results = hash_table();
    Return Apply(f, g, Op);
}
Apply(f, g, Op) {
    if((Result = Lookup(f, g, Op, Results)) ≠ NULL) return Result;
    if(f is a terminal)
        if(g is a terminal) Result = f op g;
        else
            Result = newMTBDD(g.var, Apply(f, g.left, Op),
                              Apply(f, g.right, OP));
    elsif(g is a terminal)
         Result = newMTBDD(f.var, Apply(f.left, g, Op),
                           Apply(f.right, g, Op));
    elsif(top variables of f and g are equal)
         Result = newMTBDD(f.var, Apply(f.left, g.left, Op),
                           Apply(f.right, g.right, Op));
    elsif(top variable of f precedes top variable of g)
         Result = newMTBDD(f.var, Apply(f.left, g, Op),
                           Apply(f.right, g, Op));
    else
         Result = newMTBDD(g.var, Apply(f, g.left, Op),
                           Apply(f, g.right, Op));
    Store(Results, f, g, Op, Result);
}
```

Note immediately that there is at most one node in the resulting MTBDD for each call to this routine. The storage of results and their lookup ensure that there is at most one call per pair of nodes from f and g. Hence if $h = f \diamond g$ then $|h| \leq |f||g|$. This of course recaps Bryant's seminal theorem for two-terminal BDD's.

4.4. "Shadow" nodes

As mentioned above, MTBDD's themselves contain no hint of the dimensionality of the represented matrix or vector; at best, from a raw MTBDD, one can deduce the size of the *smallest* represented matrix or vector (2^n, where n is the number of variables of the MTBDD). However, each MTBDD represents an *infinite* number of matrices and vectors. To see the problem, consider the MTBDD 1. This surely represents a constant matrix or vector. But that matrix might be the scalar 1, the 2-vector [1 1], the two-by-two matrix $\left(\begin{smallmatrix}1 & 1\\ 1 & 1\end{smallmatrix}\right)$, or *any* constant matrix of size 2^m elements (any $m \geq 0$), of *any* dimensionality.

This difficulty arises even within well-defined MTBDD's; consider, for example, the matrix: $\left(\begin{smallmatrix}1 & 1\\ 0 & 1\end{smallmatrix}\right)$. This is represented by the MTBDD pictured in figure 1.

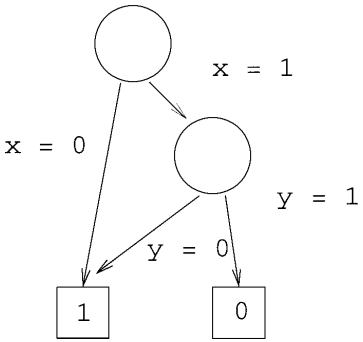

Figure 1. MTBDD illustrating paths of unequal length.

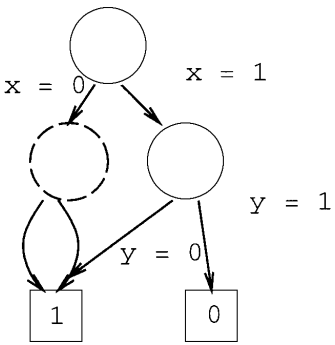

Figure 2. MTBDD with a shadow node.

There are many possible methods of resolving this difficulty. One method uses the variable associated with a node (the variable which labels its outgoing edges) to denote the "level" of the node; operations over the BDD keep track of the expected level, and note a discrepancy as the repetition of the BDD. The difficulty with this idea is that the resulting code is fairly complicated and filled with bookkeeping.

A cleaner and more elegant solution ensures that every node is at a well-defined distance from the root of the MTBDD; this is done by introducing, along edges that skip levels in the MTBDD, one node per level skipped. Both outgoing edges from these "shadow" nodes are directed to the appropriate successor node along the original edge. An example on our original MTBDD is given in figure 2. MTBDDs with these properties are called *Quasi-Reduced* MTBDDs; the property of quasi-reduction was first defined by Sasao [11], who applied it to BDD's and a variant, Ternary Decision Diagrams (TDDs).

```
Add_Shadow_Nodes(B, i) {
    if(B is a terminal) return;
    left ← B→ left; right ← B→right;
```

```
    for(node ← left, j ← topvar(left) - 1; j > i; --j)
        node ← newMTBDD(j, node, node);
    B→left ← node;
    Add_Shadow_Nodes(B→left, topvar(B→left));
    for(node ← right, j ← topvar(right) - 1; j > i; --j)
        node ← newMTBDD(j, node, node);
    B→right ← node;
    Add_Shadow_Nodes(B→right, topvar(B→right));
}
```

The procedure to add shadow nodes is quite straightforward, and is given above.

Though shadow nodes do add to the size of an MTBDD, they do not affect the complexity results derived above. Note that at most $O(\log n)$ shadow nodes are added to any edge in any MTBDD; hence the size of any MTBDD can grow by (at most) a multiplicative factor of $O(\log n)$. Even this grossly overstates the size of the size increase to an MTBDD. In order for an MTBDD to grow by a factor of $O(\log n)$, each path through the original MTBDD (before the addition of shadow nodes) must be of length $O(1)$; this can only hold for constant matrices and vectors. Further, it is critical to note that the major results on paths and sizes assume $O(\log n)$ nodes on every path through an MTBDD. Hence, even with the addition of shadow nodes, the maximum sizes of the MTBDD for sparse, dense, and permutation matrices are as given above.

4.5. Vector multiplication

Up until now, we have spoken only of the *total dimension* of a matrix, without considering its exact shape, or number of dimensions. For vector operations, of course, both the size and the shape of the matrix is relevant in determining the result. Of course, the shape of a matrix represented by an MTBDD is arbitrary, and must be specified separately.

4.5.1. Multiplication of a vector by a vector. The result of vector multiplication is quite straightforward. Given vectors, f and g, both of length m, we have:

$$f \circ g = \sum_{i=0}^{m-1} f_i g_i$$

When f, g are represented as Boolean functions, $f, g : \mathbf{B}^{\lceil \log m \rceil} \mapsto \tilde{R}$, this is rewritten:

$$[\, f_{\overline{x_1}}\, f_{x_1} \,] \begin{bmatrix} g_{\overline{x_1}} \\ g_{x_1} \end{bmatrix} \tag{3}$$

and hence:

$$f \circ g = f_{\overline{x_1}} \circ g_{\overline{x_1}} + f_{x_1} \circ g_{x_1} \tag{4}$$

Equation (4) forms the basis of the recursion procedure:

```
Vector_Multiply(f, g) {
    if(terminal cases) return(terminal_case(f, g));
    x_i = top_var(f, g);
    Result = Vector_Multiply(f_x̄i, g_x̄i) + Vector_Multiply(f_xi, g_xi)
    Store(f, g, Result);
    return Result;
}
```

The termination conditions need some scrutiny. If both f and g are terminals, the result is *not* simply fg—consider the case where $f_i = g_i = 1$ for every i. The vectors are represented by the constant function 1, but the result is m for m-length vectors. The reason for this disparity is that a constant terminal represents not a single entry, but rather a block of constant entries of the some size. Thus, when f and g are both terminals, the result is fgk where k is the size of the block.

The size of the constant block represented by the terminal can be deduced during the computation. The size is the size of the partial vector. At the top level of the recursion tree, this size is 2^m, and is reduced by half with each recursive call; we can keep track of the current size simply by passing the size into `Vector_Multiply` as an integer argument. The result follows.

```
Vector_Multiply(f, g, n) {
    if((Result = Lookup(f, g, n)) ≠ NULL)
          return Result;
    if(f is a terminal)
          Result = Multiply_Vector_by_Scalar(f, g, n);
    elsif(g is a terminal)
          Result = Multiply_Vector_by_Scalar(g, f, n);
    else
          x_i = top_var(f, g);
          Result = Vector_Multiply(f_x̄i, g_x̄i, n − 1)
                   + Vector_Multiply(f_xi, g_xi, n − 1)
    Store(f, g , n, Result);
    return Result;
}
Multiply_Vector_by_Scalar(f, g, n) {
    if((Result = Lookup(f, g, n)) ≠ NULL)
          return Result;
    if(g is a terminal)
          Result = f. value* g. value* 2^n;
    else
          x_i = top_var(g);
          Result = Multiply_Vector_by_Scalar(f, g_xi=0, n-1)
                   + Multiply_Vector_by_Scalar (f, g_xi=1, n-1);
```

```
    Store(f, g, n, Result);
    return Result;
}
```

We've broken the case where one of the two vectors is a scalar out for clarity, but they can be treated together for the purposes of complexity analysis. The storage and retrieval of results implies that there is at most one multiplication for each triple (μ, ν, k), where μ is a node of f, ν is a node of g, and k is an integer between 0 and n; hence there are at most $O(|f||g|n)$ multiplications. Further, there is at most one lookup on a triple (μ, ν, n) per unique pair (*Parent* (μ), *Parent* (ν)); hence there are at most $O(|f||g||n|)$ separate hash table lookups. Hence the complexity of vector multiplication is $O(|f||g|n|)$.

4.5.2. Multiplication of a matrix by a vector. The case of multiplying a matrix by a vector is almost identical to that of multiplying a vector by a vector. In general, we have:

$$h(x_1, \ldots, x_m) = f(x_1, y_1, \ldots, x_m, y_n) \circ g(y_1, \ldots, y_n)$$

The use of the variables $(y_1, \ldots, y_m)$ to index the rows of f gives us the following picture of f:

$$\begin{bmatrix} f_{\overline{x_1}\,\overline{y_1}} & f_{\overline{x_1}y_1} \\ f_{x_1\overline{y_1}} & f_{x_1y_1} \end{bmatrix}$$

and the following picture of matrix multiplication:

$$\begin{bmatrix} h_{\overline{x_1}} \\ h_{x_1} \end{bmatrix} = \begin{bmatrix} f_{\overline{x_1}\,\overline{y_1}} & f_{\overline{x_1}y_1} \\ f_{x_1\overline{y_1}} & f_{x_1y_1} \end{bmatrix} \begin{bmatrix} g_{\overline{y_1}} \\ g_{y_1} \end{bmatrix}$$

Hence:

$$\begin{aligned} h_{\overline{y_1}} &= f_{\overline{x_1}\,\overline{y_1}} \circ g_{\overline{x_1}} + f_{x_1\overline{y_1}} \circ g_{y_1} \\ h_{y_1} &= f_{\overline{x_1}y_1} \circ g_{\overline{x_1}} + f_{x_1y_1} \circ g_{y_1} \end{aligned}$$

This forms the basis of the recursion procedure. Again, the terminal cases must be carefully reviewed. We have the following:

1. f is a function of x variables only. In this case, each row of f is a constant function, and hence each position of h can be found by multiplying the relevant constant function of f by the sum of the entries of g. As a result, to obtain h we multiply each terminal of f by the entries of g.
2. g is a scalar function, f is a function of some y variables and possibly some x variables. In this case, g is a constant function, but the rows of f are not constants; we reduce the rows of f through the usual Shannon division.

In order to implement case (1) above, we need a routine which sums up vectors represented as MTBDD's.

```
Sum_Vector(g, n) {
    if((Result = Lookup(g, n)) ≠ NULL)
        return Result;
    if(g is a terminal)
        Result = g.value * 2^n;
    else
        y_i = top_var(g);
        Result = Sum_Vector(g_{y_i}, n-1)+ Sum_Vector(g_{y_i}, n-1);
    Store(g, n, Result);
    return Result;
}
```

With this in hand, we can easily write the routine to multiply a matrix by a vector:

```
Multiply_Matrix_By_Vector(f, g, n) {
   if((Result = Lookup(f, g, n)) ≠ NULL)
       return Result;
   if(f is independent of y variables)
       Sum = Sum_Vector(g, n);
       Result = Duplicate(f);
       foreach terminal R of Result
          R.value = R.value * Sum;
   elsif(g is a terminal)
       Result = Multiply_Matrix_by_Scalar(f, g, n);
   elsif top_var(f) > top_var(g)
       y_i = top_var(f);
       Result = newMTBDD(x_i, Multiply_Matrix_By_Vector(f_{x_i}, g, n),
          Multiply_Matrix_By_Vector(f_{y_i}, g, n))
   else
       y_i = top_var(g);
       Left = Multiply_Matrix_By_Vector(f_{y_i}, g_{y_i}, n-1);
       Right = Multiply_Matrix_By_Vector(f_{y_i}, g_{y_i}, n-1);
       Result = Apply_Operator(Left, Right, PLUS);
   Store(f, g, n, Result);
   return Result;
}
Multiply_Matrix_by_Scalar(f, g, n) {
   if((Result = Lookup(f, g, n)) ≠ NULL)
       return Result;
   if(f is independent of y variables)
       Sum = g.value * 2^n;
       Result = Duplicate(f);
       foreach terminal R of Result
          R.value = R.value * Sum;
```

```
    else
        y_i = top_var(f);
        Result = newMTBDD(x_i, Multiply_Matrix_by_Scalar(f_ȳi, g,
                 n-1),
            Multiply_Vector_by_Scalar(f_yi, g, n-1));
    Store(f, g, n, Result);
    return Result;
}
```

4.5.3. Multiplying a matrix by a matrix. Consider the problem of finding a matrix product: $h = fg$, where f and g are matrices. By now the recursion is familiar to the reader:

$$\begin{bmatrix} h_{\overline{x}\overline{z}} & h_{\overline{x}z} \\ h_{x\overline{z}} & h_{xz} \end{bmatrix} = \begin{bmatrix} f_{\overline{x}\overline{y}} & f_{\overline{x}y} \\ f_{x\overline{y}} & f_{xy} \end{bmatrix} \begin{bmatrix} g_{\overline{y}\overline{z}} & g_{\overline{y}z} \\ g_{y\overline{z}} & f_{yz} \end{bmatrix}$$

Once again the recursion suggested is the straightforward one given by the equations, and once again it is easy to see that an integer n is required to keep track of the sizes of the constant blocks. In the case of the matrix multiplication, the recursion is somewhat simpler due to the symmetry of the operands:

```
Matrix_Multiply(f, g, n, i) {
    if(Result = lookup(f, g, n, i, MULT)) return Result;
    if(f and g are both constants) return fg2^n;
    Q_1 = Apply( Matrix_Multiply(f_{x̄i ȳi}, g_{ȳi z̄i}, i+1, n-1),
          Matrix_Multiply(f_{x̄i yi}, g_{yi z̄i}, i+1, n-1), ADD);
    Q_2 = Apply( Matrix_Multiply(f_{x̄i ȳi}, g_{ȳi zi}, i+1, n-1),
          Matrix_Multiply(f_{x̄i yi}, g_{yi zi}, i+1, n-1), ADD);
    Q_3 = Apply( Matrix_Multiply(f_{xi ȳi}, g_{ȳi z̄i}, i+1, n-1),
          Matrix_Multiply(f_{xi yi}, g_{yi z̄i}, i+1, n-1), ADD);
    Q_4 = Apply( Matrix_Multiply(f_{xi ȳi}, g_{ȳi zi}, i+1, n-1),
          Matrix_Multiply(f_{xi yi}, g_{yi zi}, i+1, n-1), ADD);
    R1 = newMTBDD(z_i, Q_1, Q_2);
    R2 = newMTBDD(z_i, Q_3, Q_4);
    Result = newMTBDD (x_i, R1, R2);
    Store(f, g, n, i, MULT, Result);
    return Result;
}
```

The variable i in this case is used to track which set of variables is being used at this level of recursion. For clarity, here we have used the natural order of the variables, and have interleaved the x, y, and z variables.

This routine can be improved by using the Strassen [12] products at each level of the recursion, rather than the simple, naive block method outlined here. The applicability of the Strassen procedure is evident, since Strassen's method is a simple variant of the divide-and-conquer approach outlined above.

5. Permutation matrices

An interesting class of matrix well worth study is the *permutation matrix*. A permutation matrix is simply a square binary matrix with precisely one one in each row and one one in each column; its effect, when applied to a vector, is to permute the elements of the vector.

Permutation matrices arise most often in LU decomposition and Gaussian elimination. The most common representation is as a vector V of length n, where $V_{i=j}$ iff $P_{ij} = 1$. Such a representation is easily seen to be of size $O(n \log n)$; since there are $n!$ permutation matrices, this is optimal.

A simple consequence of the sparse matrix theorem is that the space complexity of the MTBDD representation of a permutation matrix is also $O(n \log n)$. This is not the most interesting measure, however. A more precise estimate is found for an $n \times n$ permutation matrix which permutes only k elements.

It is easy to see that there must be $O(\binom{n}{k}(k! - 2^{k-1}))$ such matrices, and hence any representation must be of size $O(k \log n)$. The most obvious efficient representation—a linked list of k elements, where the (i, j) coordinate of each permuted element is stored—is plainly $O(k \log n)$, i.e., optimal. Our purpose here is to investigate the MTBDD representation.

First, note that the k elements off the main diagonal form a Boolean function with k minterms: hence there are at most k paths through its BDD to the one terminal, and hence its size is at most $O(k \log n)$, optimal.

The measure given above requires some computational complexity when the permutation matrix is used, however, both in the MTBDD and in the linked-list case. Thus, it is often desired to represent the unmoved elements. It is interesting to note that when very few elements are moved, the MTBDD representation under the natural order is in fact smaller than the $O(n \log n)$ representation generally thought necessary.

A permutation matrix that moves exactly k elements has precisely k one elements off the main diagonal. Such a matrix can be written:

$$P = I_n \oplus M$$

where I_n is the $n \times n$ identity matrix, $\oplus$ is the termwise `exclusive-or` operation (addition modulo 2) and M is a matrix defined as follows:

$$M_{i,j} = \begin{cases} P_{i,j} & i \neq j \\ \bar{P}_{i,j} & i = j \end{cases}$$

Note that M has $2k$ one elements. By the sparse matrix theorem, therefore, M is of size $O(k \log n)$. Since $P = I_n \oplus M$, by Bryant's Theorem $|P| \leq |I_n||M|$. By construction, I_n is of size $O(\log n)$, and hence we conclude that P is of size at *most* $O(k \log^2 n)$. Note that this is smaller than $O(n \log n)$ when $k < n / \log n$.

5.1. *Construction of permutation matrices*

A permutation matrix is constructed precisely in the manner given in the proof above; specifically, the "deviation" M of the permutation matrix from the identity is constructed; the result is then exclusive-or'd to the identity matrix.

The deviation matrix is easy to form; it is easy to see that if k elements are permuted, it is equivalent to a logic function of $2k$ minterms.

6. L/U decomposition

L/U Decomposition on MTBDD's is a fascinating topic. It is trivial to implement the standard Gaussian-elimination-with-pivoting algorithm. It is worth noting, however, that it is easy to maintain the maximum element of an MTBDD with the root node, and compute it dynamically in $O(1)$ time. Since this is the case, it is easy to find the maximal element in an MTBDD in $O(1)$ time. This implies immediately that complete pivoting—where the largest element of a matrix is chosen as the pivot, not merely the largest element on the diagonal—is as easy as partial pivoting on an MTBDD. This is important, for complete pivoting is numerically more desirable than partial pivoting, but is avoided with standard packages due to the expense of searching the entire matrix for the pivots.

While the standard Gaussian-elimination algorithm is easily implemented, we can in fact do somewhat better. The optimal L/U factorization algorithm is found in [1], and is based on a recursive-descent paradigm. This suggests that the optimal algorithm may map nicely onto the MTBDD structure. In fact, this the case, as we show in the remainder of this section.

6.1. Recursive-descent LUP factorization

The classic recursive-descent LUP factorization algorithm is taken from [1] and adapted to MTBDD's. We give the algorithm here, and discuss its adaptation below.

```
/* ProcedureFactor(A, m, p) returns L, U, P such that
L is lower-triangular m × m, U is upper triangular m × p,
and P is a p × p permutation matrix such that A = LUP,
where A is an m × p matrix of rank m */
Factor(A, m, p)
1.  if m == 1 {
2.      L = [1];
3.      c is any column containing a nonzero element of A
4.      P = permute(1, c);
5.      U = AP;
6.      return L, U, P;
7.  } else {
8.      B is the top m/2 rows of A;
9.      C is the bottom m/2 rows of A;
10.     (L1, U1, P1) = Factor(B, m/2, p);
11.     D = C P1^-1;/* Hence D P1 = C */
12.     E is the leftmost m/2 columns of U1 /* E is
          upper-triangular, m/2 × m/2 */
13.     F is the leftmost m/2 columns of D;
14.     G = D − F E^-1 U1
```

```
        /* Note since E is square and upper triangular, E is invertible
        Also note that since U_1 = [E | K], where K is an m/2 × (p − m/2)
        matrix, E^{-1}U_1 = [I_{m/2} | E^{-1}K], and so FE^{-1}U_1 = [F | FE^{-1}K]
        Since D = [F | K_1], G = [0 | K_1 − FE^{-1}K] */
15.     (L_2, U_2, P_2) = FACTOR(K_1 − FE^{-1}K, m/2, p − m/2)
16.     P_3 = [ I_{m/2}   0 ]
              [   0      P_2 ]
17.     H = U_1 P_3^{-1}
18.     L = [ L_1        0 ]  /* L is m × m */
            [ FE^{-1}   L_2 ]
19.     U = [ H        ]  /* U is m × p */
            [ 0    U_2 ]
20.     P = P_3 P_1
21.     return (L, U, P)
    }
```

The initial call is `Factor`(A, n, n); note, as above, that we may assume that $n = 2^r$, for some integer $r > 0$.

It is easy to see that since the initial m is a power of 2, so is each subsequent m; further, cutting the matrix in half horizontally simply corresponds to evaluating any unevaluated row variable; for convenience, the top row variable will be chosen. Hence the initial split of matrix A into matrix B and C simply corresponds to taking two cofactors of A.

The only central difficulty with translation to an MTBDD form is the problem of cutting off the leftmost $m/2$ columns of G in the recursive call to `FACTOR` on line 15; since p is not cut on the way down in the same manner that m is, even if p is a power of 2 at some level it is not assured that $p - m/2$ is: consider $16 - 4 = 12$. Hence splitting the matrix vertically is not a matter of a simple evaluation. However, note we do not actually need to split the matrix; examination of the Aho-Hopcroft-Ullman algorithm reveals that every matrix actually factored into (L_1, U_1) is a square matrix of size $m/2$. Further, if these matrices are actually plotted on the original matrix A, it is seen that these square matrices are of size a power of 2, straddling the main diagonal of matrix A.

This picture gives a useful alternative notion of the Aho-Hopcroft-Ullman algorithm, and relates it to the classic iterative procedure for LU decomposition.

Consider the classic iterative algorithm:

```
LU_Decomp(A){
    n ← |A|;  U ← A;  L ← I_n;
    for(i ← 0;  i < n;  ++ i)
        pivot the maximal element of U_{i..n,i..n} into U_ii;
        for(j ← i + 1; j < n; ++j)
            L_ji ← U_ji / U_ii;
            U_ji ← 0;
            for(k ← i + 1;  k < n;  ++k)
                U_jk ← U_jk − U_ji / U_ii;
}
```

Conceptually, on the ith iteration, the ith column of U is turned to 0 from $i+1$ to n, and the multiplicands necessary to do that are stored in the elements L_{ij}, for $i < j < n$.

Now, the principle of the iterative algorithm is easily generalized to recursive descent:

$$\begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} = \begin{pmatrix} L_{00} & 0 \\ L_{10} & L_{11} \end{pmatrix} \begin{pmatrix} U_{00} & U_{01} \\ 0 & U_{11} \end{pmatrix}$$

where U_{00}, U_{11} are upper triangular, L_{00}, L_{11} are lower triangular, and L_{10}, U_{01} are general matrices.

2×2 Gaussian Elimination solves for L, U:

$$\begin{aligned} A_{00} &= L_{00}U_{00} \\ A_{01} &= L_{00}U_{01} \\ A_{10} &= L_{10}U_{00} \\ A_{11} &= L_{10}U_{01} + L_{11}U_{11} \end{aligned}$$

These equations are easily solved: one first recursively factors A_{00} into L_{00} and U_{00}, then solves:

$$\begin{aligned} U_{01} &= L_{00}^{-1}A_{01} \\ L_{10} &= A_{10}U_{00}^{-1} \end{aligned}$$

and then recursively factors $A_{11} - L_{10}U_{01}$ to obtain L_{11} and U_{11}.

Analysis of this naive algorithm gives:

$$S(n) = 2S(n/2) + 2T(n/2) + 2M(n/2)$$

where $T(n)$ is the time to factor an $n \times n$ matrix, as before, and $M(n)$ is time to multiply two $n \times n$ matrices. Noting that $T(n) \leq cM(n)$, we obtain:

$$S(n) \leq 2S(n/2) + c_1M(n/2)$$

and it is easy to see that $S(n) \leq c_2M(n)$ for some constant c_2.

Returning to the iterative procedure, the middle loop conceptually consists of three steps:

1. Computation of the multiplicand;
2. "Zeroing" the appropriate element of U; and
3. Computation of the row of U

The great insight of the AHU recursive-descent procedure was the recognition that the multiplicand need not simply be U_{ii}/U_{ji}, but could in fact be FE^{-1}, where E was an $m \times m$ submatrix centred about the diagonal and F was a $m \times m$ matrix directly below E. The computation of the new row of U—the innermost loop of the iterative algorithm—becomes the computation of the m rows of U including the $m \times m$ submatrix F (note that F is zeroed in a manner precisely analogous to U_{ji} in the standard algorithm).

The point, then to note is that the AHU algorithm is in fact quite similar to the iterative algorithm; a pivot is selected, and then moved to the main diagonal; the element immediately below the multiplicand is zeroed. This implies that the $m \times p$ matrix A is in fact the rightmost p columns of an $m \times n$ matrix $[0 \;\; A]$; conceptually, we may as easily deal with this supermatrix.

This picture of the AHU algorithm permits us to note the following fact: the matrix E, for every call of the algorithm, is a square matrix of size $2^k \times 2^k, 0 \leq k \leq n$. Further, E consists of the submatrix $U_{ii} \cdots U_{i+2^k-1,i+2^k-1}$ of the final upper triangular matrix U, for some $i = c2^{k+1}$. Since E is such a matrix, it represents a cofactor of the final upper-triangular matrix U with the variables x_{n-k}, y_{n-k} set to 0, the variables x_j, y_j set arbitrarily for $0 \leq j < k$ and the variables x_{n-i}, y_{n-i} not set for $1 \leq i < k$. It must therefore follow that F is found as a cofactor of U_1, by setting all the x and y values up to $n-k$ as for E, setting $x_{n-k} = 1, y_{n-k} = 0$ and not setting x_{n-i}, y_{n-i} for $i \leq k$.

This observation permits us to consider a minor variant of the AHU algorithm: `Factor` $(A, 2^m, 2^n, p, k)$ factors a $2^m \times 2^n$ matrix A whose leftmost $2^n - p$ columns are identically 0 into factors L, U, P such that:

1. L is $2^m \times 2^m$ lower-triangular
2. U is a $2^m \times 2^n$ matrix, $[0 \;\; V]$, where 0 is $2^m \times 2^n - p$ and V is $2^m \times p$, upper-triangular
3. P is a $2^n \times 2^n$ permutation matrix of the form $[\begin{smallmatrix} I_{2^n-p} & 0 \\ 0 & P' \end{smallmatrix}]$; i.e., P does not permute the first $2^n - p$ columns of A
4. $A = LUP$

In other words, we simply modify the AHU algorithm to retain *all* the columns of A, throughout the recursion; the fact that this procedure returns the result of the AHU procedure is to note that it *is* the AHU procedure in the case where $p = 2^n$.

The only significant modification to the AHU procedure is in lines 12–13 (which become 13–14), where E and F are computed. E and F are now not the leftmost 2^{m-1} columns of U_1 and D, but are a selection of columns with indices $2^n - p \cdots 2^n + 2^m - (p+1)$; as mentioned in the foregoing, these are selected by a an appropriate setting of the variables $x_0, \ldots, x_{n-(m+1)}, y_0, \ldots, y_{n-(m+1)}$. Note that the variables x_j, y_j are set before entry for $0 \leq j < n - (m+1)$.

This observation lets us adapt the AHU algorithm with only a little simple bookeeping, namely passing the values already set into FACTOR as a separate argument, which we denote here as S.

```
/* ProcedureFactor(A, m, n, S, k) returns L, U, P such that
L is lower-triangular 2^m × 2^m, U is 2^m × 2^n
of the form of (2) above,
and P is a 2^n × 2^n permutation matrix such that A = LUP,
where A is an 2^m × 2^n matrix of rank 2^m */
Factor(A, m, n, S, k)
1.  if m == 0 {
2.      L = [1];
3.      c is any column containing a nonzero element of A
```

```
4.      d is the column selected by S
5.      P = permute(d, c);
6.      U = AP;
7.      return L, U, P;
8.  } else {
9.      B ← A|_{x_k=0};
10.     C ← A|_{x_k=1};
11.     (L_1, U_1, P_1) = Factor(B, m − 1, n, S ⋃ {y_k = 0, x_k = 0}, k + 1);
12.     D = C P_1^{-1}; /* Hence D P_1 = C */
13.     E ← U_1|_{S ⋃ {y_k=0}} /* E is upper-triangular, 2^{m−1} × 2^{m−1} */
14.     F ← D|_{S ⋃ {y_k=0}};
15.     G = D − F E^{-1} U_1
        /* Note since E is square and upper triangular, E is
        invertible
        Also note that since U_1 = [0  E  K]
        E^{-1} U_1 = [0  I_{2^{m−1}}  E^{-1} K], and so F E^{-1} U_1 = [0  F  F E^{-1} K]
        Since D = [0  F  K_1], G = [0  0  K_1 − F E^{-1} K] */
16.     (L_2, U_2, P_2) = FACTOR(G, m − 1, n, S ⋃ {y_k = 1, x_k = 1}, k + 1);
17.     H = U_1 P_2^{-1}
18.     L = ((y_k = 0) ∧ ((x_k = 0) ∧ L_1) ∨ ((x_k = 1) ∧ F E^{-1})) ∨ ((y_k = 1) ∧ (x_k = 1) ∧ L_2)
19.     U = ((x_k = 0) ∧ H) ∨ ((x_k = 1) ∧ (y_k = 1) ∧ U_2)
20.     P = P_2 P_1
21.     return (L, U, P)
    }
```

The initial call is `Factor`$(A, n, n, \emptyset, 0)$.

Note that the assembly operations in the lines 18–19 are Boolean operations; these conform exactly to the matrix operations given in the original version of the AHU procedure, when the matrix is considered as a function from the Boolean space of index variables onto the range of the matrix. Note also that the permutation matrix P_3 has disappeared from this variant of the algorithm; this is because, it is now unnecessary to construct large permutation matrices from smaller ones: all permutation matrices generated at every stage of this algorithm are $2^n \times 2^n$, rendering the reconstruction operations superfluous.

As with all the other MTBDD operations, LUP factorization is idempotent, and hence the results of any call can be stored and retrieved for a subsequent call. Note that not only the matrix A, represented as an MTBDD, but also the parameters m and n must be used as a hash key, to resolve the dimensionality issues given above.

7. Conclusion

In the foregoing, we have demonstrated that MTBDD's are efficient representations of matrices. We have demonstrated that MTBDD's are the space-optimal representation of both dense and sparse matrices, and of permutation matrices. Further, we have demonstrated that the optimal time-complexity algorithms for the basic suite of matrix operations translate

directly and elegantly onto the MTBDD structure, without any reduction in the asymptotic efficiency of the algorithms.

Acknowledgments

This Research sponsored by Fujitsu Research.

References

1. A. Aho, J. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms*, Addision-Wesley, 1974.
2. S.B. Akers, "On a theory of boolean functions," *J. SIAM*, 1959.
3. R.E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, 1986.
4. E.M. Clarke, M. Fujita, K. McMillan, J. Yang, and X. Zhao, "Spectral transforms for large boolean functions with applications to technology mapping," in *Design Automation Conference*, 1993.
5. E.M. Clarke, M. Fujita, K. McMillan, J. Yang, and X. Zhao, "Spectral transforms for large boolean functions with applications to technology mapping," This Issue, 1996.
6. O. Coudert and J.C. Madre, "A unified framework for the formal verification of sequential circuits," in *IEEE International Conference on Computer-Aided Design*, 1990.
7. O. Coudert and J.-C. Madre, "A new implicit graph-based prime and essential prime computation technique," in *New Trends in Logic Synthesis and Optimization*, T. Sasao (Ed.), Kluwer Academic Publishers, 1992.
8. O. Coudert, J.C. Madre, and H. Fraisse, "A new viewpoint on two-level minimization," in *Design Automation Conference*, 1993.
9. S. Malik, A. Wang, R.K. Brayton, and A.L. Sangiovanni-Vincentelli, "Logic verification using binary decision diagrams in a logic synthesis environment," in *IEEE International Conference on Computer-Aided Design*, 1988.
10. Y. Matsunaga, P.C. McGeer, and R.K. Brayton, "On computing the transitive closure of a state transition relation," in *Design Automation Conference*, 1993.
11. Tsutomu Sasao, "Ternary decision diagrams and their applications," in *International Workshop on Logic Synthesis*, 1993.
12. V. Strassen, "Gaussian elimination is not optimal," *Numer. Math*, 13, 1969.
13. G.M. Swamy, R.K. Brayton, and P.C. McGeer, "A fully implicit quine-mccluskey procedure using bdds," Technical Report UCB/ERL M92/127, Electronics Research Lab, University of California at Berkeley, 1992.
14. H. Touati, H. Savoj, B. Lin, R.K. Brayton, and A. Sangiovanni-Vincentelli, "Implicit state enumeration of finite state machines using bdd's," in *IEEE International Conference on Computer-Aided Design*, 1990.