5-1993

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

Edmund M. Clarke
Carnegie Mellon University

M. Fujita
Fujitsu Labs

P C. McGeer
University of California - Berkeley

K. McMillan
Carnegie Mellon University

J C-Y Yang
Stanford University

See next page for additional authors

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

E. M. Clarke † M. Fujita‡ P. C. McGeer§ K. McMillan ¶ J. C.-Y. Yang‖
X. Zhao **

February 17, 1993

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 block matrices, band matrices, and 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 and 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.

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 knowlege of BDD's on the part of virtually any reader.

BDD's were introduced by S. B. Akers in 1959[2]. In the early 1980's, searching for a data structure to undergird his switch-level simulator, R. E. 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.[7] demonstrated that

*This Research sponsored by Fujitsu Research
†Computer Science Department, Carnegie-Mellon University
‡Fujitsu Laboratories, Kawasaki, Japan
§EECS Department, University of California, Berkeley
¶Computer Science Department, Carnegie-Mellon University
‖EECS Department, Stanford University
**Computer Science Department, Carnegie-Mellon University

BDD's could be used to verify large multi-level combinational logic functions. In 1990. Berthet. Coudert. and Madre[5] 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.[10]. In 1992. Coudert and Madre[6] demonstrated a method by which the primes of very large Boolean functions could be implicitly represented using BDD's. This year, these techniques were extended to a full implicit logic minimizer.

This year also saw the exploration of the relationship between BDD's and matrices. In [8], it is argued that any BDD can be thought of as a vector of length 2^n, or as a binary matrix of of size $2^{n-1} \times 2^{n-1}$, or, for that matter, as an object of almost any dimensionality. In [8], 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], 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 [8], 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] 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 $f : \mathbf{B}^n \rightarrow B$, the *cofactors* of f are the functions obtained by partial evaluation of f. For example, the cofactor $f_{x_1\bar{x}_2x_4}$ 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 $f(x_1, ..., x_n)$, 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 $\bar{x_i}$ 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:

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

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 $\{0, ..., m-1\}$, or the case where it is the finite set $\{0, ..., m-1\} \times \{0, ..., n-1\}$. 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 $\{x_1, ..., x_s\}$, 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 $\{x_1, ..., x_s\}$ and the column index bits with the variables $\{y_1, ..., y_t\}$, 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 $f(x_1, y_1, x_2, y_2, ...)$ or $f(y_1, x_1, ...)$. 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:

$$\begin{pmatrix} f_{\bar{x_1}\bar{y_1}} & f_{\bar{x_1}y_1} \\ f_{x_1\bar{y_1}} & f_{x_1y_1} \end{pmatrix}$$

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 identity 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 an 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 $\mathbf{B}^N \mapsto \tilde{R}$. 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. ∎

We begin by comparing MTBDD's to the dense-matrix representation of a matrix of total dimension n: clearly a lower bound on the size of such a matrix is $O(n)$. We have the following:

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

Proof: Since there are n elements, there are at most n leaves of the MTBDD. The worst-case size of an MTBDD occurs when there is no sharing of internal nodes, i.e., when the MTBDD becomes a binary tree. It is well-known that a binary tree with n leaves has n internal nodes, giving the result. ∎

This comparison is somewhat of a straw man; dense matrices do not occur in practice very often. A more interesting comparison is to sparse matrices: matrices where there are m nonzero elements and total dimension n, for $m << n$.

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 at 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 representation 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 nontrivial entries, $m << n$. Then there exists at least one matrix M of total dimension n with m nontrivial entries, $m << n$, 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 $\begin{pmatrix} n \\ m \end{pmatrix}$ such matrices. Since $m << n$. $\begin{pmatrix} n \\ m \end{pmatrix} \approx n^m$ 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 $h_{\bar{x}_i} == f_{\bar{x}_i} \diamond g_{\bar{x}_i}$.*

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 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.

4.4.1 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:

$$[f_{\bar{x}_1} f_{x_1}] \begin{bmatrix} g_{\bar{x}_1} \\ g_{x_1} \end{bmatrix} \tag{3}$$

and hence:

$$f \circ g = f_{\bar{x}_1} \circ g_{\bar{x}_1} + f_{x_1} \circ g_{x_1} \tag{4}$$

(4) forms the basis of the recursion procedure.

```
Vector_Multiply(f, g, i, n) {
    if((Result = Lookup(f, g, i)) ≠ NULL)
        return Result;
    if(i > n) Result = fg;
    else      Result = Vector_Multiply(f_{x̄_i}, g_{x̄_i}, i + 1, n) + Vector_Multiply(f_{x_i}, g_{x_i}, i + 1, n)
    Store(f, g, i, Result);
    return Result;
}
```

The initial call on two length$-n$ vectors f and g is `Vector_Multiply`$(f, g, 0, n)$.

The termination conditions are fairly simple. When $i > n$, the recursion has bottomed: f and g are constant nodes, and the result is simply fg. The second condition is more interesting. Note that after each computation, the result is stored; one termination condition is simply the case where a stored result is retrieved: in this case, the computation proceeds no further – the stored result is simply returned. 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 $O(|f||g||n|)$.

4.4.2 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:

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

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

$$\begin{bmatrix} f_{\bar{x}_1 \bar{y}_1} & f_{\bar{x}_1 y_1} \\ f_{x_1 \bar{y}_1} & f_{x_1 y_1} \end{bmatrix}$$

and the following picture of matrix multiplication:

$$\begin{bmatrix} h_{\bar{x}_1} \\ h_{x_1} \end{bmatrix} = \begin{bmatrix} f_{\bar{x}_1 \bar{y}_1} & f_{\bar{x}_1 y_1} \\ f_{x_1 \bar{y}_1} & f_{x_1 y_1} \end{bmatrix} \begin{bmatrix} g_{\bar{y}_1} \\ g_{y_1} \end{bmatrix}$$

Hence:

$$\begin{aligned} h_{\bar{y}_1} &= f_{\bar{x}_1\bar{y}_1} \circ g_{\bar{x}_1} + f_{x_1\bar{y}_1} \circ g_{x_1} \\ h_{y_1} &= f_{\bar{x}_1 y_1} \circ g_{\bar{x}_1} + f_{x_1 y_1} \circ g_{x_1} \end{aligned}$$

This forms the basis of the recursion procedure.

```
Multiply_Matrix_By_Vector(f, g, i, n) {
    if((Result = Lookup(f, g, i)) ≠ NULL)
        return Result;
    if(i > n) Result = fg;
    else    R1 = Multiply_Matrix_By_Vector(f_{x̄_i ȳ_i} g_{ȳ_i}, i + 1, n) + Multiply_Matrix_By_Vector(f_{x̄_i y_i} g_{y_i}, i + 1,
        R2 = Multiply_Matrix_By_Vector(f_{x_i ȳ_i} g_{ȳ_i}, i + 1, n) + Multiply_Matrix_By_Vector(f_{x_i y_i} g_{y_i}, i + 1, n)
        Result = x̄_i R1 + x_i R2;
    Store(f, g, i, Result);
    return Result;
```

4.4.3 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:

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

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, i, n) {
    if(Result = lookup(f, g, i, MULT)) return Result;
    if(f and g are both constants) return fg2^n;
    Q_1 = Apply( Matrix_Multiply(f_{x̄_i ȳ_i}, g_{ȳ_i z̄_i}, i + 1, n − 1),
          Matrix_Multiply(f_{x̄_i y_i}, g_{y_i z̄_i}, i + 1, n − 1), ADD);
    Q_2 = Apply( Matrix_Multiply(f_{x̄_i ȳ_i}, g_{ȳ_i z_i}, i + 1, n − 1),
          Matrix_Multiply(f_{x̄_i y_i}, g_{y_i z_i}, i + 1, n − 1), ADD);
    Q_3 = Apply( Matrix_Multiply(f_{x_i ȳ_i}, g_{ȳ_i z̄_i}, i + 1, n − 1),
          Matrix_Multiply(f_{x_i y_i}, g_{y_i z̄_i}, i + 1, n − 1), ADD);
    Q_4 = Apply( Matrix_Multiply(f_{x_i ȳ_i}, g_{ȳ_i z_i}, i + 1, n − 1),
          Matrix_Multiply(f_{x_i y_i}, g_{y_i z_i}, 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, 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[9]products at each level of the recursion, rather than the simple, naive block method outlined here.

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 $P_{ij} = 0$. 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 $O(\left(\begin{array}{c} n \\ k \end{array} \right)(k! - 2^{k-1}))$ 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.

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* $O(k \log^2 n)$. This is a loose upper bound. In fact, it seems likely that the tight bound is $O(k \log n)$ (i.e., MTBDD's are optimal), but this has yet to be shown.

A permutation matrix is constructed precisely in the manner given in the proof above; specifically, the "deviation" 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 Inverting An Upper or Lower Triangular Matrix

Lower (upper) triangular matrices are always invertible, of course, and the result is always lower (upper) triangular. To see this in the case of the lower triangular matrix, we decompose into quadrants, as always, to obtain:

$$\left(\begin{array}{cc} L_{00} & 0 \\ L_{10} & L_{11} \end{array} \right) \left(\begin{array}{cc} M_{00} & M_{01} \\ M_{10} & M_{11} \end{array} \right) = \left(\begin{array}{cc} I & 0 \\ 0 & I \end{array} \right)$$

Simple multiplication yields:

$$\begin{aligned} L_{00}M_{01} &= L_{10}M_{00} + L_{11}M_{10} = 0 \\ L_{00}M_{00} &= L_{10}M_{01} + L_{11}M_{11} = I \end{aligned}$$

Solving yields $M_{01} = 0$, thus demonstrating that M is in fact lower-triangular, and that $M_{00} = L_{00}^{-1}$ and $M_{11} = L_{11}^{-1}$. By induction, M_{00} and M_{11} exist and are lower-triangular, and M_{10} is easily found as $-(M_{11}L_{10}M_{00})$.

The naive algorithm suggested by these equations has complexity equivalent to that of matrix multiplication, as can be seen by the following argument. Let $T(n)$ be the number of operations required to invert an $n \times n$ lower (upper) triangular matrix. Casual inspection of the equations yields a recurrence equation:

$$T(n) = 2T(n/2) + 2M(n/2) \tag{5}$$

where $M(n)$ is the complexity of multiplying two $n \times n$ matrices. A simple induction shows $T(n) \leq cM(n)$ for constant c, and so inverting an $n \times n$ upper- or lower-triangular matrix is about the asymptotic cost of a constant number of matrix multiplications.

7 L/U Decomposition

The problem of L/U decomposition is that of factoring a (non-singular) $m \times m$ matrix A into three matrices P, L, U such that $A = PLU$ and L is unit lower-triangular (L has zeroes everywhere above the diagonal and ones on the diagonal), U is upper-triangular (zeroes everywhere above the diagonal), and P is a permutation matrix. The classic iterative algorithm for L/U Decomposition is well-known, and is based on Gaussian Elimination.

```
LU_Decomp(A){
    n —|A|; U—A; L—I_n; P—I_n
    for(i—0; i < n; ++i)
        U_ji is the maximal element of U_{i..n,i};
        P_ji is the associated permutation matrix;
        U—U P_ji; P—P P_ji;
        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/Uii;
```

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_{ji}, for $i < j < n$.

The important thing to notice about the iterative LUP decomposition algorithm is the observation that the upper-triangular matrix U is found by Gaussian Elimination on the original matrix A; the multiplicands form the column entries of L.

Now consider the recursive-descent procedure; as usual, we break A into

$$\begin{pmatrix} A_{\bar{x}\bar{y}} & A_{\bar{x}y} \\ A_{x\bar{y}} & A_{xy} \end{pmatrix}$$

We begin by factoring $A_{\bar{x}\bar{y}}$ into $L_{\bar{x}\bar{y}}$ and $U_{\bar{x}\bar{y}}$; this may involve some pivoting, so we also pass the submatrix $A_{\bar{x}y}$ as the source for the required pivots and compute an initial permutation matrix P_1.

At this point we have:

$$AP_1 = \begin{pmatrix} L_{\bar{x}\bar{y}} & 0 \\ 0 & I \end{pmatrix} \begin{pmatrix} U_{\bar{x}\bar{y}} & U_{\bar{x}y} \\ (AP_1)_{x\bar{y}} & (AP_1)_{xy} \end{pmatrix}$$

We are left to compute $L_{x\bar{y}}$: following the logic of the iterative procedure, and noting that as an upper-triangular, nonzero matrix, $U_{\bar{x}\bar{y}}$ is invertible, we write:

$$(AP_1)_{x\bar{y}} = L_{x\bar{y}} U_{\bar{x}\bar{y}}$$

And so we derive:

$$L_{x\bar{y}} = (AP_1)_{x\bar{y}} U_{\bar{x}\bar{y}}^{-1}$$

We now find:

$$B = (AP_1)_x - (AP_1)_x U_{\bar{x}\bar{y}}^{-1} U_{\bar{x}}$$

Note that:

$$\begin{aligned} B_{\bar{y}} &= ((AP_1)_x - (AP_1)_x U_{\bar{x}\bar{y}}^{-1} U_{\bar{x}})_{\bar{y}} \\ &= (AP_1)_{x\bar{y}} - (AP_1)_{x\bar{y}} U_{\bar{x}\bar{y}}^{-1} U_{\bar{x}\bar{y}} \\ &= 0 \end{aligned}$$

We recursively factor B_y to produce $L_{xy}P_3^{-1}$, $U_{xy}P_3^{-1}$ and P_2, a a permutation matrix of the form:

$$\begin{pmatrix} I & 0 \\ 0 & P_3 \end{pmatrix}$$

The identity matrix in the top-left corner of P_2 indicates that the first half of the columns are left untouched by the second factorization.

The final product is returned as:

$$\begin{aligned} L &= \begin{pmatrix} L_{\bar{x}\bar{y}} & 0 \\ L_{x\bar{y}} & L_{xy} \end{pmatrix} \\ U &= \begin{pmatrix} U_{\bar{x}\bar{y}} & U_{\bar{x}y}P_3 \\ 0 & U_{xy} \end{pmatrix} \\ P &= P_1 P_2 \end{aligned}$$

The algorithm is simple. Loosely, given cubes c, d, and integer k, where c is a vertex over the space $\{x_1, ..., x_{k-1}\}$ and d is the corresponding vertex over the space $\{y_1, ..., y_{k-1}\}$, `factor` LU factorizes the submatrix A_{cd}, pivoting as necessary.

```
factor(d, c, A, k) {
1.  if(k = m)
2.      s is the minterm with the maximum element of A;
3.      P— permute(d, s);
4.      L—1;
5.      U—A ∘ P;
6.  else
7.      (K, V, Q)— factor(d_ȳk, c_x̄k, A_x̄k, k + 1);
8.      B—A_xk ∘ Q^-1;
9.      J—B_dȳk ∘ (V_dȳk)^-1;
```

10. C—$B - J \circ V$;
11. (M, W, R)— `factor`$(dy_k, cx_k, C, k+1)$;
12. L—$\overline{x_k y_k} K + x_k \overline{y_k} J + x_k y_k M$;
13. U—$\overline{x_k}(V \circ R^{-1}) + x_k W$;
14. P—$Q \circ R$;
15. `return`(L, U, P)

We have the following lemma and proof, adapted to MTBDD's from the original in [1].

Lemma 7.1 *Let A be a matrix with 2^{m-k} rows, rank 2^{m-k}, such that every column left of the first column indexed by cube d is 0. Then* `factor`(c, d, A, k) *returns L, U, P such that:*

1. *$A = L \circ U \circ P$; and*
2. *P is a $2^m \times 2^m$ permutation matrix and $P_{ii} = 1$ for every i preceding the first integer in the cube c; and*
3. *if $k < m$ then $U_{y_k d\overline{x_k}} = 0$; and*
4. *U_d is an upper-triangular non-singular matrix of dimension $2^{m-k} \times 2^{m-k}$; and*
5. *L is a unit lower-triangular non-singular matrix of dimension $2^{m-k} \times 2^{m-k}$.*

Proof: Induction on $m - k$. If $m - k = 0$, then $k = m$, A has one row, and is of rank one. Since A is of rank one, it has at least one nonzero element, and since all the columns to the left of d are zero the nonzero element s must lie to the right of d. By construction P is a $2^m \times 2^m$ permutation matrix that permutes two elements, d and $s > d$; hence all the elements left of d are not permuted, establishing 2. U_d and L are nonzero scalars, establishing 4 and 5. Since $k = m$, 3 is irrelevant. For 1. by construction $U = A \circ P$; since $L = 1$, $L \circ U = A \circ P$, i.e., $L \circ U \circ P^{-}1 = A$. Since P permutes two elements. $P = P^{-1}$, establishing the result.

Now suppose for all $m - k < N$ and consider $m - k = N$. Since A is a matrix of 2^N rows of rank 2^N, $A_{\overline{x_k}}$ is a matrix of 2^{N-1} rows of rank 2^{N-1}. By induction, $K \circ V \circ Q = A_{\overline{x_k}}$, K is unit lower-triangular, $V_{d\overline{y_k}}$ is upper-triangular. We claim:

$$A = \begin{pmatrix} K & 0 \\ B_{d\overline{y_k}} \circ V_{d\overline{y_k}}^{-1} & I \end{pmatrix} \circ \begin{pmatrix} V \\ C \end{pmatrix} \circ Q \tag{6}$$

To establish the claim, we note that, since $V_{d\overline{y_k}}$ is upper-triangular and nonsingular, it is invertible and hence $V_{d\overline{y_k}}^{-1}$ exists; similarly, since Q is a permutation matrix, Q^{-1} exists. Hence B and J are well-defined, and hence C is well-defined. Multiplying (6), we see:

$$A_{\overline{x_k}} = K \circ V \circ Q$$

and

$$A_{x_k} = (B_{d\overline{y_k}} \circ V_{d\overline{y_k}}^{-1} \circ V + C) \circ Q$$

and (6) is established if both these hold. The first is immediate from induction. Substituting for B, C and J, we have:

$$A_{x_k} = ((A_{x_k} \circ Q^{-1})_{d\overline{y_k}} \circ V_{d\overline{y_k}}^{-1} \circ V + A_{x_k} \circ Q^{-1} - (A_{x_k} \circ Q^{-1})_{d\overline{y_k}} \circ (V_{d\overline{y_k}})^{-1} \circ V) \circ Q$$

or, simplifying:

$$A_{x_k} = A_{x_k} \circ Q^{-1} \circ Q = A_{x_k}$$

and done, establishing (6). Now, since A is of rank 2^{m-k}, the right hand side of (6) must be of rank 2^{m-k}, and therefore $\begin{pmatrix} V \\ C \end{pmatrix}$ must be of rank 2^{m-k}, and so C must be of rank 2^{m-k-1}. Further, $C_{d\overline{y_k}} = (B - J \circ V)_{d\overline{y_k}} = B_{d\overline{y_k}} - (B_{d\overline{y_k}} \circ (V_{d\overline{y_k}})^{-1} \circ V)_{d\overline{y_k}})$. Since the y variables index the columns, it is easy to establish that this is equivalent to:

$$B_{d\overline{y_k}} - B_{d\overline{y_k}} \circ (V_{d\overline{y_k}})^{-1} \circ V_{d\overline{y_k}})$$

which is of course 0. The recursive call to **factor** on line 11 therefore meets the premises of the lemma, and so by induction 1-5 hold for the recursive call. We must establish 1-5. For 1, note from (6):

$$A = \begin{pmatrix} K & 0 \\ B_{d\overline{y_k}} \circ V_{d\overline{y_k}}^{-1} & I \end{pmatrix} \circ \begin{pmatrix} V \\ C \end{pmatrix} \circ Q$$

and by induction:

$$C = M \circ W \circ R$$

Noting that $V = I \circ V \circ R^{-1} \circ R$, we may write:

$$\begin{pmatrix} V \\ C \end{pmatrix} = \begin{pmatrix} I & 0 \\ 0 & M \end{pmatrix} \circ \begin{pmatrix} V \circ R^{-1} \\ W \end{pmatrix} \circ R$$

and substituting:

$$A = \begin{pmatrix} K & 0 \\ B_{d\overline{y_k}} \circ V_{d\overline{y_k}}^{-1} & I \end{pmatrix} \circ \begin{pmatrix} I & 0 \\ 0 & M \end{pmatrix} \circ \begin{pmatrix} V \circ R^{-1} \\ W \end{pmatrix} \circ R \circ Q$$

Multiplying the first two matrices together, and the last two, and noting $P = R \circ Q$, we get:

$$A = \begin{pmatrix} K & 0 \\ J & M \end{pmatrix} \circ \begin{pmatrix} V \circ R^{-1} \\ W \end{pmatrix} \circ P$$

Noting that

$$L = \begin{pmatrix} K & 0 \\ J & M \end{pmatrix}$$

and

$$U = \begin{pmatrix} V \circ R^{-1} \\ W \end{pmatrix}$$

We have $A = LUP$, establishing 1. Further, since K and M are unit lower-triangular, so is L, establishing 5. Since neither Q nor R perturb any elements preceding d, neither does $R \circ Q$, establishing 2. Since $V_{d\overline{y_k}}$ is upper-triangular non-singular, and since R does not perturb any elements preceding dy_k, it follows that $(V \circ R)_{d\overline{y_k}} = V_{d\overline{y_k}}$ is upper-triangular and nonsingular. Further, W_{dy_k} is upper-triangular and nonsingular. Now:

$$V_d = \begin{pmatrix} (V \circ R)_{d\overline{y_k}} & (V \circ R)_{dy_k} \\ W_{d\overline{y_k}} & W_{dy_k} \end{pmatrix}$$

Since $C_{d\overline{y_k}} = 0$, and $C = M \circ W \circ R$, and since R did not perturb the columns of $d\overline{y_k}$, it follows that $(M \circ W)_{d\overline{y_k}} = 0$. It is easy to see that $(M \circ W)_{d\overline{y_k}} = M \circ W_{d\overline{y_k}}$, and since M is nonsingular it follows $W_{d\overline{y_k}} = 0$, i.e.,

$$V_d = \begin{pmatrix} (V \circ R)_{d\overline{y_k}} & (V \circ R)_{dy_k} \\ 0 & W_{dy_k} \end{pmatrix}$$

and this is plainly nonsingular upper-triangular, establishing 4. Further, $U_{x_k d\overline{y_k}} = W_{d\overline{y_k}} = 0$, establishing 3, and this gives the result. ∎

8 Conclusions and Further Work

We have implemented most of the major algorithms in this package and are currently engaged in an intensive experimental program. The theory developed herein gives us strong reason to believe that MTBDD's are a superior data structure for the representation of general, sparse, and block matrices. However, we still must discover whether the advantages are extensive enough to warrant re-engineering our standard matrix packages.

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] 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.

[6] O. Coudert and J.-C. Madre. A new implicit graph-based prime and essential prime computation technique. In T. Sasao, editor, *New Trends in Logic Synthesis and Optimization.* Kluwer Academic Publishers, 1992.

[7] 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.

[8] Y. Matsunaga, P. C. McGeer, and R. K. Brayton. On computing the transitive closure of a state transtion relation. In *Design Automation Conference*, 1993.

[9] V. Strassen. Gaussian elimination is not optimal. *Numer. math*, 13, 1969.

[10] 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.