Formal Methods in System Design 10, 171–206 (1997)

Algebraic Decision Diagrams and Their Applications

R.I. BAHAR, E.A. FROHM, C.M. GAONA, G.D. HACHTEL, E. MACII*, A. PARDO AND F. SOMENZI
University of Colorado, Department of Electrical and Computer Engineering, Boulder, CO 80309

Abstract. In this paper we present theory and experimental results on Algebraic Decision Diagrams. These diagrams extend BDDs by allowing values from an arbitrary finite domain to be associated with the terminal nodes of the diagram. We present a treatment founded in Boolean algebras and discuss algorithms and results in several areas of application: Matrix multiplication, shortest path algorithms, and direct methods for numerical linear algebra. Although we report an essentially negative result for Gaussian elimination per se, we propose a modified form of ADDs which appears to circumvent the difficulties in some cases. We discuss the relevance of our findings and point to directions for future work.

Keywords: decision diagrams, graph algorithms, linear algebra

1. Introduction

Binary Decision Diagrams (BDDs) [3, 5] have significantly changed the landscape of synthesis, formal verification, and testing of digital circuits. When combined with symbolic graph algorithms, they have made the solution of many difficult problems possible. In symbolic graph algorithms, node and edge sets are stored symbolically, that is, in terms of their characteristic functions. BDDs provide an efficient and canonical form of representation of those functions. The major impact of BDDs derives from this canonicity property; conventional, optimal time complexity algorithms, such as breadth-first search, do not do well on large graphs, simply because they process vertices and edges on an individual basis. Consequently, despite their "optimal" complexity, such algorithms essentially run forever on graphs with, say, 2^{100} vertices. With symbolic BDD algorithms (which may not have optimal worst case complexity), however, there is a chance of handling the vertices in sets of smaller sizes, and in many cases this breaks the computational bottleneck.

Several examples of successful applications of symbolic graph algorithms to reachability analysis of finite state machines have been reported [6–9, 13, 23]. All these applications are *topological*; they address problems where connectivity is the principal issue. However, recent work [10, 11] introduced a class of algorithms for *arithmetic* symbolic computation, based on a new kind of BDD called a Multi-Terminal Binary Decision Diagram (MTBDD), whose main feature is the adoption of multiple terminal nodes. Computational results were reported in [10], but were limited to a very special class of matrices which arise in Walsh transform techniques. (All elements of a Walsh matrix are 1 or -1.)

*Enrico Macii is also with Politecnico di Torino, Dipartimento di Automatica e Informatica, Torino, Italy 10129.

In this paper we discuss theory and implementation issues for a broad class of symbolic algorithms applicable not only to arithmetic, but to many algebraic structures. Because of their applicability to different algebras, and because of their foundation in (large) Boolean algebras, we call the BDDs with multiple terminals *Algebraic Decision Diagrams* (ADDs).

We have used ADDs to solve successfully general mathematics and computer science problems. In this work, we describe in detail applications of ADDs to matrix multiplication, shortest-path computation, and solution of large systems of linear equations. In order to show the effectiveness of the ADD data structure, and the efficiency of the algorithms for its manipulation, we report experimental data concerning all the applications mentioned above.

The plan of the paper is as follows. In Section 2 we propose Algebraic Decision Diagrams (ADDs), an extension of BDDs, and we present the fundamental algorithms for their manipulation. In Section 3, we present a basic theory of ADD-based matrix multiplication in semi-rings and quasi-rings. In Section 4, this theory is applied to develop algorithms for single-source and all-pairs shortest path problems, and in Section 5 we apply ADDs to numerical linear algebra. We conclude in Section 6 with a discussion of our qualitative research findings and their significance for future research.

2. Algebraic Decision Diagrams (ADDs)

2.1. Overview

An ADD can be seen as a BDD whose leaves may take on values belonging to a set of constants different from 0 and 1. Let S be the set of constants. For example,

$$S = \{0, 1, 2\}, \qquad S = \{blue, red, green\}, \qquad S = \{a, b, c, d\}.$$

An ADD can also be seen as a Boolean function,

$$f : \{0, 1\}^n \mapsto Q,$$

where $S \subseteq Q$, and $|Q| = 2^m$, for some m. As a consequence, all theorems of Boolean algebra can be applied to ADDs. Specifically, Boole's expansion theorem [2]:

$$f(x, y, z, \ldots) = x \cdot f(1, y, z, \ldots) + x' \cdot f(0, y, z, \ldots),$$

which is the basis for all recursive algorithms that manipulate ADDs.

2.2. Formal definition

Let us consider an algebraic structure (S, O, D), composed of a finite carrier, S, a set of operations, O, and a set of distinguished elements, D. We will use S to identify the algebraic system as well as its carrier, when that does not generate confusion. ADDs are representations of functions from $\{0, 1\}^n$ to S defined by the following rules.

An ADD is a directed acyclic graph $(V \cup \Phi \cup T, E)$, representing a set of functions $f_i : \{0, 1\}^n \to S$, where S is the finite carrier of the algebraic structure over which the ADD

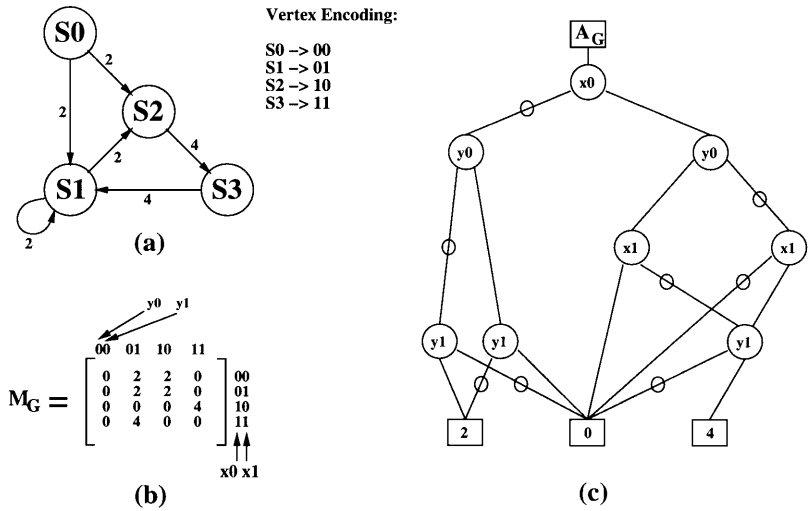

Figure 1. ADD representation of graphs and matrices.

is defined. V is the set of the internal nodes. The out-degree of $v \in V$ is 2. The two out-going arcs for a node $v \in V$ are labeled *then* and *else*, respectively. Every node $v \in V$ has a label $l(v) \in \{0, \ldots, n-1\}$. The label identifies a variable on which the f_i's depend. Φ is the set of the function nodes: The out-degree of $\phi \in \Phi$ is 1 and its in-degree is 0. The function nodes are in one-to-one correspondence with the f_i's. T is the set of terminal nodes; each terminal node, $t \in T$, is labeled with an element of the carrier S, $s(t)$. The out-degree of a terminal node is 0. E is the set of edges connecting the nodes of the graph; (v_i, v_j) is the edge connecting node v_i to v_j. The variables of the ADD are ordered; if v_j is a descendant of v_i (i.e., $(v_i, v_j) \in E$), then $l(v_i) < l(v_j)$.

An ADD represents a set of Boolean functions, one for each function node, defined as follows:

1. The function of a terminal node, t, is the constant function $s(t)$. The constant $s(t)$ is interpreted as an element of a Boolean algebra larger than or equal in size to S.
2. The function of a node $v \in V$ is given by $l(v) \cdot f_{then} + l(v)' \cdot f_{else}$, where '$\cdot$' and '$+$' denote Boolean conjunction and disjunction, and f_{then} and f_{else} are the functions of the *then* and *else* children.
3. The function of $\phi \in \Phi$ is the function of its only child.

ADDs are a natural symbolic representation of weighted directed graphs, which are in one to one correspondence with square sparse matrices. For example, the ADD $A_G(x, y)$ for the simple weighted directed graph $G = (V, E, W)$ of figure 1(a), whose adjacency matrix, M_G, is shown in figure 1(b), is illustrated in figure 1(c).

In this ADD, and throughout the sequel, the open circle on an edge emanating from a node identifies the *else* child of the node. In this particular case, we have $S \supseteq \{0, 2, 4\}$; each path corresponding to an edge not in G reaches the terminal value 0.

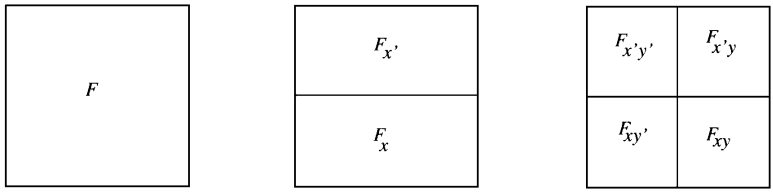

Figure 2. Matrix partitioning by cofactoring.

If there are N vertices in the given graph G, then there will be $2n$ encoding variables, where $n = |x| = |y| = \lceil \log(N) \rceil$. If every vertex has at least one fanout with a unique weight, then the ADD $A_G(x, y)$ will have at least $2^n = O(N)$ nodes. Of course, this is linear in the number of vertices in G, but it is nevertheless exponential in the number of ADD variables, i.e., variables used to encode the vertices of the graph.

Notice that in the example of figure 1, the specified matrix was a square matrix of size $n = 4$, that is, n was a power of 2. If this is not the case, then the matrix must be "padded" with appropriately valued dummy rows and columns so that the thus augmented matrix satisfies the power of 2 rule, that is, the number of rows and columns is exactly a power of 2. Else, ADD computations cannot be employed. However, for efficient ADD manipulation, the values assigned to dummy rows and columns depend on the type of operations which are to be applied. Therefore, we post-pone the question of how to "pad" the given matrix until ADD operations have been discussed (see Section 2.5).

As said before, ADDs can be used efficiently to store weighted directed graphs, which are in one-to-one correspondence with square sparse matrices. Similarly, ADDs naturally represent bi-partite graphs, which are in one-to-one correspondence with rectangular matrices. In this regard, cofactors of ADDs play an important role, as pointed out by McGeer et al. in [11]. This is illustrated in figure 2.

In the figure, x stands for the top variable of the ADD structure, and is identified as a row variable. Similarly, y stands for the next variable of the ADD structure, and is identified as a column variable. Note that cofactoring with respect to x partitions the matrix into two rectangular sub-matrices, the first (upper) represented by the else-child $F_{else} = F_{x'}$, and the second by $F_{then} = F_x$. Cofactoring with respect to both x and y leads similarly to a four way partition, in which the 4 square sub-matrices are represented by the 4 grand-children of the ADD function node F. Note that these need not be distinct—in fact recombination of identical children can lead to great efficiencies. Various classical algorithms (multiplication, LU factorization) are based on such recursive partitioning, called by some authors "recursive descent" [1].

If such recursive descent is continued until all the row and column variables have been cofactored, one arrives at a 1×1 sub-matrix, which of course is just one of the constant terminal elements of the given ADD. If the given ADD represents a $N \times N$ matrix, there are just $n = \log(N)$ row variables and $n = \log(N)$ column variables, so access to any of the non-zero elements can be attained in $O(n)$ operations.

As a final point, note that 4-way partitioning still occurs even if x and/or y are not the top variables in the row and column sets, respectively. For example, if x and y were the two bottom variables, an even-odd, or "checker-board" partitioning pattern would emerge.

However, we give special emphasis to top variable cofactoring, since this can be done in $O(1)$ time.

Because of the correspondence to matrices, ADD-based symbolic graph algorithms have further applications to logic synthesis, formal verification, and circuit simulation. Thus, we take the time to introduce some convenient notation. An $N \times M$ sparse (full) matrix is a matrix with less than (exactly) $N \times M$ non-zero elements. Matrices are in one to one correspondence with weighted bi-partite graphs. If $N = M$, matrices correspond to weighted di-graphs. In the sparse matrix context, we think of the zero value as a *background*, denoting no entry, or no connection in the corresponding graph. In other applications, for example shortest-path computations, it is convenient to use a large edge length as the background value. These different requirements are naturally taken into account by our approach.

2.3. Basis in Boolean algebra

In the sequel, we restrict our attention to the case of $S = \{0, \ldots, r-1\}$, where r is a power of 2. The most general case can be easily derived from this case. S can be thought of as the carrier of a Boolean algebra whose zero and one are the r-bit binary codes $0 \cdots 0$ and $1 \cdots 1$ respectively. Similarly, the r atoms of this algebra are the "1-hot" codes $0 \cdots 01$, $0 \cdots 10$, ..., $10 \cdots 0$.

In this context, the function $f(x)$ associated with a function node of an ADD can be viewed as a Boolean function of n variables, that is,

$$f(x) : \{0, 1\}^n \mapsto S.$$

Because the f we consider is a Boolean function, it satisfies Boole's expansion theorem:

Theorem 2.1. *If $f : \{0, 1\}^n \to \{0, 1\}$ is a Boolean function, then*

$$f(x_1, x_2, \ldots, x_n) = x_1' \cdot f(0, x_2, \ldots, x_n) + x_1 \cdot f(1, x_2, \ldots, x_n), \tag{1}$$

for all $(x_1, \ldots, x_n) \in \{0, 1\}^n$.

If we recursively apply Boole's expansion to a function, we eventually get the well known *minterm canonical form* [4]:

$$f(x_1, \ldots, x_{n-1}, x_n) = f(0, \ldots, 0, 0)\ x_1' \cdots x_{n-1}' x_n' + \cdots + f(1, \ldots, 1, 1)\ x_1 \cdots x_{n-1} x_n.$$

The values $f(0, \ldots, 0, 0), \ldots, f(1, \ldots, 1, 1)$ are elements of S, and they are called the *discriminants* of the function f; the elementary products:

$$x_1' \cdots x_{n-1}' x_n',\ x_1' \cdots x_{n-1}' x_n, \ldots,\ x_1 \cdots x_{n-1} x_n$$

are called the *minterms*. An ADD can then be viewed as an implicit enumeration of a minterm expansion. When recombination occurs, the minterms with the same discriminant are grouped together. In this sense, a matrix function or, equivalently, an ADD can be

viewed as a Boolean function, and the set of all such functions for a given carrier S, forms a Boolean function algebra. Thus, the manipulation of ADDs is equivalent to the manipulation of functions in such algebra. At the same time, the values of the terminal nodes belong to the algebraic system S. Therefore, two sets of operations are applicable to ADDs: Boolean operations, and operations on matrices built on S; the latter can be decomposed into arithmetic and abstraction operations. In the following, we first discuss general properties of ADDs, then we describe in detail ADD operations.

2.4. General properties

The fundamental characteristics of ADDs can be summarized as follows:

1. ADDs are canonical, which is important when dealing with finite and integer domains. When dealing with finite precision arithmetic, rounding errors may decrease the usefulness of this property.
2. Edge attributes, such as complementation flags, may be of limited utility, because complementation in the Boolean algebra may not have a meaningful counterpart in S.
3. Recombination efficiency, that is, number of shared nodes, is relatively small in comparison to BDDs. However, the advantages due to recombination may still be important, when comparing ADDs to competing data structures.
4. ADDs provide a uniform $\log(N)$ access time to the elements of the data structure, being N the total of real numbers stored in the ADD. (For example, the non-zero elements of a sparse matrix.)
5. ADDs cannot beat sparse matrix data structures in terms of worst-case space complexity. However, recombination of isomorphic sub-graphs may give a considerable practical advantage to ADDs over other data structures.

2.5. Operations

There are three sets of operations that can be used to manipulate ADDs:

1. Boolean operations;
2. Arithmetic operations;
3. Abstraction operations.

The use of such operations has made possible the implementation of symbolic algorithms for the solution of problems which are known to be computationally difficult.

2.5.1. Boolean operations. Since we have established that an ADD can be used to represent functions from $\{0, 1\}^n$ to S, all the usual Boolean operations are well defined.

A fundamental Boolean operation is the *if-then-else* operation (`ITE`). Similarly to the BDD version [5], operator `ITE` takes three arguments. The first is an ADD restricted to have only 0 or 1 as terminal values (that is, a conventional BDD). The second and third arguments are generic ADDs. `ITE` is defined by:

$$\texttt{ITE}(f, g, h) = f \cdot g + f' \cdot h,$$

and its implementation is shown in figure 3.

```
ITE (f,g,h) {
  if (f == 1)   return g;
  if (f == 0)   return h;
  if (g == h)   return g;
  if ((g == 1) and (h == 0))   return f;
  if (Table_Lookup (computed_table,(f,g,h),r))   return r;
  v = Top_Var (f,g,h);
  T = ITE (f_v,g_v,h_v);
  E = ITE (f_v',g_v',h_v');
  if (T == E)   return T;
  r = Find_Or_Add (v,T,E);
  Table_Insert (computed_table,(f,g,h),r);
  return r;
}
```

Figure 3. The ITE algorithm.

In the unique table, usually implemented by a hash table, the pointer to the node with label v and children g and h is stored in the entry corresponding to the key (v, g, h). In the recursive algorithm, we first find g and h and then ask whether we should introduce a new node labeled v and pointing to g and h. The unique table is looked-up to check if such a node already exists. If not, a new entry is created and is inserted in the table (procedure Find_Or_Add). The algorithm maintains the ADD reduced by checking if T equals E before calling procedure Find_Or_Add.

Concerning the computed table, the key for insertion and retrieval is the triple $\{f, g, h\}$. At every level of the recursion, we check the computed table for the result (procedure Table_Lookup). If it is already there, we just return. If not we recur and, before returning, we insert the newly computed result in the table (procedure Table_Insert).

ITE can be used to set elements of a matrix to desired values or to extract a sub-matrix. For example, let us apply the ITE operator to the ADDs representing the following matrices:

$$f = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}; \qquad g = \begin{bmatrix} 3 & 3 & 3 & 3 \\ 3 & 3 & 3 & 3 \\ 3 & 3 & 3 & 3 \\ 3 & 3 & 3 & 3 \end{bmatrix}; \qquad h = \begin{bmatrix} 4 & 4 & 4 & 4 \\ 4 & 4 & 4 & 4 \\ 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 \end{bmatrix}.$$

We obtain:

$$\mathtt{ITE}(f, g, h) = \begin{bmatrix} 3 & 3 & 4 & 4 \\ 3 & 3 & 4 & 4 \\ 2 & 2 & 3 & 3 \\ 2 & 2 & 3 & 3 \end{bmatrix}.$$

2.5.2. Arithmetic operations. In this section we discuss a generic operation, called Apply, that my be used to accomplish a large number of matrix operations. Its definition is

the following:

$$\texttt{Apply}(f, g, op) = f \; op \; g,$$

where f and g are generic ADDs, and *op* is an operator such as $+$, $-$, $*$, $/$, min, max, etc.

The meaning of operation $f \; op \; g$ is the following. Let

$$D^f = \left\{d_1^f, \ldots, d_q^f\right\} = \{f(0, \ldots, 0, 0),\ f(0, \ldots, 0, 1), \ldots,\ f(1, \ldots, 1, 1)\}$$

be the set of discriminants of function f, and let

$$D^g = \left\{d_1^g, \ldots, d_q^g\right\} = \{g(0, \ldots, 0, 0), g(0, \ldots, 0, 1), \ldots, g(1, \ldots, 1, 1)\}$$

be the set of discriminants of function g. Then, the result of $f \; op \; g$ is defined as the function whose set of discriminants, D, is obtained by applying *op* to corresponding pairs of discriminants of f and g. Formally,

$$D = \left\{d_1^f \; op \; d_1^g, \ldots, d_q^f \; op \; d_q^g\right\}.$$

Notice that, since operator *op* is applied term-wise, questions concerning the underlying algebraic system such as distributivity and associativity do not arise. Thus, `Apply` may be called with a wide variety of operators. Of course underflow and overflow (closure on practical computing systems) is still an issue.

Even though, by virtue of its term by term application of *op*, `Apply` has near-universal applicability, for some operations the result depends on the order of application of *op*; therefore, we adopt the convention that the result of `Apply`(f, g, op) is $f \; op \; g$.

Like `ITE`, `Apply` is a recursive procedure based on Boole's expansion theorem. In figure 4 we give the pseudo-code for the `Apply` algorithm.

The procedure begins with a test for the terminal cases, which is passed if both arguments are constant functions, or if one of the arguments is the identity or the annihilator for *op*. Then, the variable v of lowest index in the support of f and g is chosen as the splitting variable of the usual binary recursive expansion. After the two recursive calls, the functions

```
Apply (f,g,op) {
  if (Terminal_Case (f,g,op))   return (op(f,g));
  if (Table_Lookup (computed_table,(f,g,op), R))   return R;
  v = Top_Var (f,g);
  T = Apply (f_v,g_v,op);
  E = Apply (f_v',g_v',op);
  else   R = ITE (v,T,E);
  Table_Insert (computed_table,(f,g,op),R);
  return R;
}
```

Figure 4. The `Apply` algorithm.

T and E corresponding to the *then* and *else* children have been determined. If $T = E$, the procedure returns T, else it returns the result of `ITE` applied to v, T, and E.

The algorithm maintains the returned ADD in canonical form by calling `ITE`. Also, the usual computed table look-up/insertion prevents repetition of identical recursions.

As an application example of procedure `Apply`, let us compute `Apply`$(f, g, +)$ for the following matrices f and g:

$$f = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}; \qquad g = \begin{bmatrix} 4 & 4 & 4 & 4 \\ 4 & 4 & 4 & 4 \\ 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 \end{bmatrix}.$$

We obtain:

$$\texttt{Apply}(f, g, +) = \begin{bmatrix} 5 & 5 & 4 & 4 \\ 5 & 5 & 4 & 4 \\ 2 & 2 & 3 & 3 \\ 2 & 2 & 3 & 3 \end{bmatrix}.$$

2.5.3. Abstraction operations. In this section, we present a generic operation that allows us to reduce the dimensionality of its argument function by means of variable abstractions. For example, if the argument function represents a $N \times M$ matrix, and the abstraction variable set is the entire set of row encoding variables, then the result of abstraction is a row vector, that is, a $1 \times M$ matrix. Notice that, even if the abstraction variable set is not an entire row or column encoding variable set, there is still a loss of dimensionality in the sense that the cofactored variables are removed from the support of the operand.

We introduce the following definition and notation for abstraction. Let u be the support of a function $f(u)$, and let x and y be two sub-sets of u such that $x \cup y = u$. The generic abstraction of variables x from function $f(u)$, written $f_x^{op}(y) = \backslash_x^{op} f(u)$, is defined as follows. For each minterm y of $f_x(y)$, we define the discriminant set D^y as the set of discriminants of all the minterms (x, y) of $f(x, y)$. Then, the discriminant of minterm y of $f_x(y)$ is defined as the result of applying the operation *op* uniformly, in right to left order, over the members of set D^y. In other words, if $D^y = \{d_1, \ldots, d_q\}$,

$$f_x^{op}(y) = \backslash_x^{op} f(u) = (d_1 \; op \; (d_2 \; op \cdots (d_{q-1} \; op \; d_q)) \cdots).$$

The operation specified by this definition is easily implemented for familiar operations such as $+$ and $*$. However, it must be kept in mind that for some operations, the above definition depends on the order operator *op* is applied. Fortunately, there is a simple characterization of algebraic systems for which the result of abstraction is independent of the order operator *op* is applied. Let (S, op, I) be a monoid; then:

1. S is *closed* under *op*, that is, $(a \; op \; b) \in S, \quad \forall a, b \in S$.
2. S is *associative*, that is, $(a \; op \; (b \; op \; c)) = ((a \; op \; b) \; op \; c), \quad \forall a, b \in S$.
3. I is an *identity* for *op*, that is, $a \; op \; I = I \; op \; a = a, \quad \forall a \in S$.

Aside from the limitations of finite representations, the operators $+$, $*$, min, max, Boolean *and* (denoted as $\wedge$), and Boolean *or* (denoted as $\vee$), all form monoids.

The key point here is *associativity*. If operator *op* does not have this property, then the result of generic abstraction depends on the order of applying *op* to the discriminant set of the definition, which has specified right to left precedence. In the ADD context, we want the implementation of abstraction to return the correct result, without regard to the following factors:

- The ordering of the n variables used in constructing the ADDs;
- The order in which variables are selected for cofactoring at each step in the recursive Boole's expansion.

It is conceivable that, with appropriate padding of dummy variables and contrivance of variable orderings, a specified application order for the given operation might be achieved. However, it is generally true that efficiency considerations are paramount in ADD computations, so one needs to be free to select the ADD variable ordering to minimize storage requirements and execution times. Similarly, efficient top variable splitting needs to be employed. Consequently, we will restrict our ADD abstraction computations to monoids.

Notice that, as stated in Section 2.2, in cases where the specified matrix fails to satisfy the power of 2 rule, dummy rows or columns with values appropriate to the operation being applied must be added before an ADD representation can be obtained. For generic abstraction operations, the monoid identities are the appropriate values for the padding. When function $f = A(x, y)$ is an ADD representing a matrix, this leads us to the following partial list of the possible generic abstraction operations which can be performed using ADDs.

- $\backslash_x^+ A(x, y)$: take the sum over the rows of A.
- $\backslash_x^* A(x, y)$: take the product over the rows of A.
- $\backslash_x^{\min} A(x, y)$: take the minimum over the rows of A.
- $\backslash_x^{\max} A(x, y)$: take the maximum over the rows of A.
- $\backslash_x^{\wedge} A(x, y)$: take the meet over the rows of A.
- $\backslash_x^{\vee} A(x, y)$: take the join over the rows of A.

As an example of abstraction, let us compute $\backslash_x^+ A(x, y)$ and $\backslash_y^{\min} A(x, y)$ of the following matrix $A(x, y)$:

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 0 & 0 & 1 & 2 \\ 2 & 2 & 2 & 2 \end{bmatrix}.$$

We obtain:

$$\backslash_x^+ A(x, y) = [8 \quad 10 \quad 13 \quad 16]; \qquad \backslash_y^{\min} A(x, y) = [1 \quad 5 \quad 0 \quad 2]^T.$$

Now that the abstraction operation has been introduced, we can discuss its ADD realization. In figure 5 we present the pseudo-code for the `Abstract` algorithm. Notice that,

```
Abstract (f,x,op) {
  if (Is_Constant (x))
    return (f);
  top_x = Top_Var (x);
  top_f = Top_Var (f);
  if (top_x < top_f) {
    T = Abstract (f,x_{top_x},op);
    R = Apply (T,T,op);
    return R;
  }
  if (Table_Lookup (computed_table,(f,x,op),R))
    return (R);
  T = f_{top_f};
  E = f_{top_f'};
  if (top_x == top_f) {
    r1 = Abstract (T,x_{top_x},op);
    r2 = Abstract (E,x_{top_x},op);
    R = Apply (r1,r2,op);
    Table_Insert (computed_table, (f,x,op),R);
    return R;
  }
  else {     /* (top_x > top_f) */
    r1 = Abstract (T,x,op);
    r2 = Abstract (E,x,op);
    R = ITE (top_f,r1,r2);
    Table_Insert (computed_table, (f,x,op),R);
    return R;
  }
}
```

Figure 5. The `Abstract` algorithm.

for the sake of efficiency in the implementation, we regard the set x of variables being abstracted from function f as a set of positive literals, which may be interpreted as a cube, that is, a Boolean function.

The procedure has one terminal case: It checks to see if there is any abstraction variable left to be treated. If not, f is returned. Then, the top variable *top_f* of f is computed and compared to *top_x*, the abstraction variable of lowest index. There are three cases. If *top_x* < *top_f*, the function f is independent of *top_x*, so $f_{top\_x} = f$. However, the remaining abstraction variable set x_{top_x} still needs to be abstracted from f, and this is done by recursively calling procedure `Abstract`. On the other hand, if *top_x* ≥ *top_f*, f depends on *top_x*. The computed table is first checked to see if the abstraction of f has already been done in a previous node of the recursion and can be returned immediately. If this does not happen, we still have two cases to consider. If *top_x* = *top_f*, the merging of the two children, $r_1 =$ `Abstract(`f_{top_f}, x_{top_x}, *op*) and $r_2 =$ `Abstract(`$f_{top\_f'}$, x_{top_x},

op), is done with `Apply`, since *top_x* has to be abstracted. On the other hand, if *top_x* > *top_f*, the abstraction problem is referred downward to the two children, with the results being merged by the `ITE` operation. The procedure concludes by inserting the result into the computed table and by returning it.

3. Matrix multiplication

In this section we discuss matrix multiplication algorithms, when matrices are represented as ADDs. In Section 2.5 we have presented two generic ADD operations, `Apply` and `Abstract`, which could be composed to perform matrix multiplication. However, this leaves open the question of whether or not a more efficient procedure could be derived by intermixing the micro-operations of these two procedure calls. Thus, our main purpose in this section is to explore the algebraic restrictions which might constrain such intermixing. In particular, our goal is to define a top variable recursive descent algorithm based, as usual, on Boole's expansion theorem, with ADD variable ordering and splitting variable selection dedicated solely to efficiency considerations. To this end we resort to quasi-rings and semi-rings, which play a role in matrix multiplication similar to that played by monoids in the generic abstraction procedure. These algebraic systems are built on the combination of two monoids, for example, the $+$ and $*$ monoids in the case of quasi-rings, which underlie conventional matrix multiplication. This general approach stresses the common theoretical and implementation substrate of many problems of interest, for example shortest-path computation in weighted directed graphs (see Section 4).

3.1. Quasi-rings and semi-rings

In the following, we recall the definitions of quasi-rings and semi-rings. For a more detailed treatment of quasi-rings and semi-rings, the reader may refer to [1, 12].

Definition 3.1. A *quasi-ring* is an algebraic structure $(S, \sharp, \flat, 0, 1)$ such that:

1. It is closed under both $\sharp$ and $\flat$.
2. Both $\sharp$ and $\flat$ are associative.
3. 0 is the identity for $\sharp$ and 1 is the identity for $\flat$.
4. 0 is the annihilator for $\flat$.
5. $\sharp$ is commutative.
6. $\flat$ distributes over $\sharp$.

Definition 3.2. A (*closed*) *semi-ring* is a quasi-ring such that:

1. $\sharp$ is *idempotent*.
2. The application of $\sharp$ to a countable sequence of elements of S is defined and unique.
3. Associativity, commutativity, idempotence, and distributivity of $\flat$ apply to infinite sums.

Examples of quasi-rings are the field of the rational numbers, $(Q, +, *, 0, 1)$, and the field of the real numbers, $(\Re, +, *, 0, 1)$. Examples of semi-rings are all Boolean algebras and $(\Re^{\geq 0} \cup \{+\infty\}, \min, +, +\infty, 0)$, where $\Re^{\geq 0}$ is the set of the non-negative real numbers. (Most interesting examples have a richer algebraic structure than just those of a quasi-ring or a semi-ring. However, the additional properties are not important to our discussion. Also, we do not consider rings, because we do not discuss matrix multiplication algorithms like Strassen's.)

With the standard definitions of matrix addition and multiplication, we have the following important result:

Proposition 3.1. *Matrices over a quasi-ring form a quasi-ring. Matrices over a semi-ring form a semi-ring.*

For the purpose of our discussion, the most notable difference between a quasi-ring and a semi-ring is the idempotence of $\sharp$. We will present two matrix multiplication algorithms, one for quasi-rings and one for semi-rings. They differ precisely in how they handle the $\sharp$ operation: The assumption of idempotence makes the algorithm for semi-rings simpler; hence, we describe it first.

3.2. Matrix multiplication in semi-rings

In the following, we assume that the two matrices to be multiplied are $A(x, z)$ and $B(z, y)$. Without loss of generality, we assume that the numbers of rows and columns are powers of two. If this is not the case, the matrices can be padded with zeroes. The algorithm for semi-ring matrix multiplication is based on the recursive application of Boole's expansion theorem. Let v be the variable of least index among those appearing in either A or B. We need to distinguish two cases:

1. v is z_i, for some i. We have: $A(x, z) \flat B(z, y) = (A_v(x, z) \flat B_v(z, y)) \sharp (A_{v'}(x, z) \flat B_{v'}(z, y))$.
2. v is either x_i or y_i, for some i. We have: $A(x, z) \flat B(z, y) = v \cdot (A_v(x, z) \flat B_v(z, y)) + v' \cdot (A_{v'}(x, z) \flat B_{v'}(z, y))$.

If both A and B are constant, or if either A or B is either 0 or 1, $A \flat B$ can be computed directly. Since $\sharp$ is a term-wise operation, it can be computed by `Apply`.

The implementation of the recursive algorithm makes use of a cache, as most ADD functions. No special problem arises in using the cache, because the ADDs for the operands A and B uniquely determine the result. This is not the case, however, for quasi-ring matrix multiplication, as it will be shown in the following section.

3.3. Matrix multiplication in quasi-rings

A given ADD represents different matrices depending on what variables it is assumed to depend on. For instance,

$$f(x_0) = x_0 \cdot 5 + x_0' \cdot (-2)$$

represents a matrix with two rows and one column. However,

$$f(x_0, x_1, y_0, y_1) = x_1 \cdot 5 + x_1' \cdot (-2)$$

represents a 4×4 matrix with four identical columns and two pairs of identical rows. The difference is immaterial when $\sharp$ is idempotent, that is, in semi-rings, but it is important otherwise, for example in the case of quasi-rings.

The quasi-ring multiplication algorithm we propose, called in the following *Boulder* method, still splits on the top variable. (The one of least index among those appearing in either ADD.) Unlike the semi-ring multiplication algorithm of Section 3.2, however, it keeps track of missing z variables in the two operands. Specifically, the recursive function is passed the expansion variable of its caller. Before returning the result, it checks for the occurrence of z variables in the order between the caller's expansion variable and its expansion variable. If the number of such variables is p, then the function "doubles" the result p times before returning it. (In the case of real or rational matrices, this is efficiently implemented by multiplying the result by the scalar 2^p.) We refer to this operation as *scaling* of the result.

The results are stored in the cache before any scaling is applied. This implies that the cache stores the result of multiplying the minimum representation of the operands (no missing leading variables). When a result is retrieved from the cache, the appropriate scaling is performed before returning it. Obviously, no scaling is required if the result is 0.

The pseudo-code of procedure `Matrix_Multiply` is shown in figure 6. The input parameters are the two matrices to be multiplied, $A(x, z)$ and $B(z, y)$, the index of the

```
Matrix_Multiply (A(x,z),B(z,y),v_top,n) {
    if ((A == 0) or (B == 0))  return 0;
    if ((Is_Constant (A) and (Is_Constant (B))) {
        SF = Compute_Scaling_Factor (v_top,n);
        R = Value(A) × Value(B) × SF;
        return R;
    }
    v_top = Top_Var (A,B);
    if (Table_Lookup (cache,(Matrix_Multiply,A,B),R))
        SF = Compute_Scaling_Factor (v_top,n);
        if (SF > 1)   R = Apply (R,SF,*);
        return R;
    }
    T = Matrix_Multiply (A_v_top,B_v_top,v_top,n);
    E = Matrix_Multiply (A_v'_top,B_v'_top,v_top,n);
    if(v_top ∈ z)   R = Apply (T,E,+);
    else
        if (T == E)   R = T;
        else   R = ITE (v_top,T,E);
    Table_Insert (cache,(Matrix_Multiply,A,B),R);
    SF = Compute_Scaling_Factor (v_top,n);
    if (SF > 1)   R = Apply (R,SF,*);
    return R;
}
```

Figure 6. The `Matrix_Multiply` algorithm.

current top variable between A and B, v_{top}, and the number, n, of z variables, that is, the "abstraction" variables. The first time `Matrix_Multiply` is called, v_{top} is initialized to the value -1. The procedure returns, as a result of the computation, matrix $R(x, y)$, which has as many rows as A and as many columns as B.

3.4. Comparison to prior work

Matrix multiplication in Boolean semi-rings based on BDDs has been in use for some time. It has been applied to the computation of the image and pre-image of a multiple-output function, given its input-output relation. Such operations are at the core of verification algorithms for finite state systems [13] and considerable effort has gone into making them efficient. Our algorithm for semi-ring multiplication, when applied to Boolean semi-rings, is equivalent to the algorithm that combines conjunction and existential abstraction [6, 23].

Two algorithms based on ADDs have been presented for multiplication of real matrices [10, 11]. In this context, the set S of the algebraic system (i.e., the semi-ring), is the set $\Re$ of the real numbers, and operations O are just real addition and multiplication.

The algorithm of Clarke and co-workers [10], in the following called the *CMU* method, is based on the observation that each entry $R_{i,j}$ of matrix $R = A \times B$ can be computed as follows:

$$R_{i,j} = \sum_k (A_{i,k} * B_{k,j}),$$

where $*$ indicates term-wise product and Σ indicates summation in $\Re$. The formula above tells us that entry $R_{i,j}$ of R can be computed by taking the *inner product* of column i of A with each column j of B. Therefore, by using ADD operations `Apply` and `Abstract`, it is possible to symbolically calculate matrix R as:

$$R(x, y) = \backslash_z^+ (A(x, z) * B(z, y)).$$

In this method, one binary recursion is performed by procedure `Apply` to carry out all the multiplications, and another by procedure `Abstract` to carry out all the additions. Notice that the analog of this algorithm in the case of Boolean semi-rings is the one that does not combine conjunction and abstraction. Experimental results have indicated that keeping the two operations separated has advantages and disadvantages. On the one hand, the algorithm is remarkably simpler and makes good use of the cache. On the other hand, it is possible for the intermediate results to grow too large and hamper the computation.

McGeer's algorithm [11], which will be referred to as the *Berkeley* method, is based on four-way partitioning of the problem. Of the three ADD-based matrix multiplication methods compared in this section, it is the one with the most direct link to conventional matrix manipulation algorithms. At each successive recursive call, the operands are expanded with respect to a pair of variables. The pair is formed by corresponding row and column variables. The expansion is performed even if the operands do not depend on the variables. This eliminates the need for scaling, but has some disadvantages: The procedure may expand with respect to internal variables (not the ones on the top) if the x and y variables

are not interleaved; moreover, different algorithms have to be used when multiplying two matrices and when multiplying a matrix by a vector.

3.5. *Conventional sparse matrix algorithms*

For the sake of comparison, we have implemented an efficient conventional algorithm for sparse matrix multiplication due to Gustavson [19]. We will call this algorithm *Row_Mult*, due to its similarity to the well known row-wise, or Doolittle, form of Gaussian elimination [16].

Row_Mult takes, as input, two sparse matrices, A and B, represented as row-wise linear lists, and outputs $R = A \times B$ in the same format. Suppose A is a $N \times P$ matrix with $n_{zA} > 0$ non-zero elements, and B is a $P \times M$ matrix with $n_{zB} > 0$ non-zeros. Then, the complexity of computing R is $O(N, M, n_{zA}, n_{mults})$, where $n_{zA} < n_{mults} <$ NPM is the number of multiplications performed. This routine has optimal worst case complexity for arbitrary matrices (O(NPM)) or sparse matrices of specified density ($O(n_{mults})$).

The data structure consists of row pointers (one per row), column indices (one per non-zero), and values (one per non-zero). An example of the data structure for the simple matrix of figure 7 follows the matrix.

Gustavson's algorithm can be visualized as follows. For each row i of A, scan the non-zero elements $a_{ij} = vals(k),\ k = rptr(i), \ldots, rptr(i+1) - 1$. Multiply each of these non-zeros by the corresponding row of B, processing only the non-zeros b_{jm} of B. Each multiplication either starts the accumulation of a new element of R or adds to an already started accumulation. These two cases are distinguished by a marker array $m_{\text{acc}}(l),\ l = 1, \ldots, M$, which is initialized once to zero, and updated to i when each accumulation starts. The non-zeros of R are thus computed row-wise, and an array $j_{\text{acc}}(l),\ l = 1, \ldots, M$ is kept pointing to where the accumulation of r_{jl} is being put in the values array of R (and where to put the corresponding column index).

We note that this algorithm will not return the column indices of R in ascending order, even if A and B come in that way.

Finally, we observe that the same algorithm can be applied symbolically to determine exactly where the elements of R go in advance. In this case, the accumulations in a subsequent numerical computation can be done in a tight inner loop comprised of just one index

$$\begin{bmatrix} 0 & 0 & 0 & 2 & -1 \\ 0 & 0 & 0 & -1 & 2 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 2 & -1 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

rptr	(Row Pointers)	(1, 3, 5, 8, 11)									
colind	(Column indices)	3	4	3	4	1	2	3	1	2	4
vals	(Values)	2	−1	−1	2	−1	2	−1	2	−1	−1

Figure 7. A simple sparse matrix.

step, one multiplication, and one addition. Such an inner loop is ideal for exploiting the pipeline of many architectures. This idea is also exploited in the row-wise LU factorization performed by the Harwell Sparse Matrix Package *ma28* [15] (in fact, these two routines were co-developed), which is discussed further in Section 5.

3.6. Experimental results

We have compared the performance of our own implementation of all the quasi-ring multiplication algorithms discussed in this section on a large set of benchmark matrices, whose names and characteristics are summarized in the first four columns of Table 1. Many of these examples come from the Harwell-Boeing Benchmark Set [17]. This is a large collection of medium to large size problems from many fields of application, including oil exploration, economics, surveying, device physics, electrical power distribution, and operations research. Other matrices are randomly generated matrices. Finally, we present some results for Walsh transform matrices.

For all experiments, run on a DEC-Station 5000/200 with 80 MB of memory, we have computed $A \times A^T$, $A^T \times A$, and $A \times A = A^2$; the results we have obtained in the three cases are very similar, so in the last four columns of Table 1 we show only the ones concerning $A^T \times A$, which is a common case of matrix multiplication. For instance, it occurs in the solution of least-square problems. In the case of a Walsh matrix W, $W^T \times W = W^2$ because the matrix is symmetric. Though the occurrence of a Walsh matrix in a least-squares problem is unlikely, the experiment of computing W^2 is an interesting one. One can easily verify that $W^2 = 2^n \cdot I$, where n is the number of row variables. The extreme regularity of the problem makes it a very good test for the effectiveness of the various algorithms in utilizing the cache. Notice that the conventional algorithm could not handle Walsh matrices for any interesting value of n.

The first observation that can be made by looking at the numbers presented in Table 1 is that our implementation of the ADD based methods (columns *Boulder*, *Berkeley*, and *CMU*) are markedly slower than the traditional sparse multiplication algorithm (column *Row_Mult*), by a factor ranging from 20 to 100. Of course, in terms of the size of the problem that can be handled, the symbolic algorithms are superior. We believe the unfavorable time performance on small problems derives mainly from the canonicity of the ADD form of the product. As new or intermediate results are computed, the ADD algorithm must continually check that any new function does not already exist, and that the ADD is completely reduced.

The *Berkeley* algorithm we have implemented incorporates several optimizations that we developed for our multiplication scheme. As an expected result, the performance of the *Boulder* and *Berkeley* routines are very similar on most of the examples we have tried; *Berkeley* seems to have a little advantage in processing large random matrices, while *Boulder* is more flexible and general.

The *CMU* method, on the other hand, is appreciably slower than the other algorithms on regular sparse matrices and, in general, it appears to be much more memory consuming on random generated matrices; however, it is definitely the best on Walsh matrices.

As in all BDD-based methods, cache management is crucial for the matrix multiplication algorithms. In particular, the choice of the size of the cache has shown to be fundamental

Table 1. Matrix multiplication results.

Matrix	*Rows*	*Columns*	*Non-Zeroes*	*Row_Mult*	*Boulder*	*Berkeley*	*CMU*
chemwest1	156	156	371	—	0.27	0.33	0.43
chemwest2	167	167	507	—	0.39	0.44	0.62
chemwest3	381	381	2157	0.590	2.28	2.40	3.67
chemwest4	132	132	414	0.004	0.36	0.39	0.61
chemwest5	67	67	294	0.004	0.24	0.25	0.39
chemwest6	655	655	2854	0.035	4.99	5.29	5.78
chemwest7	479	479	1910	0.019	2.46	2.64	4.19
chemwest8	497	497	1727	0.023	2.18	2.23	4.63
chemwest9	1505	1505	5445	0.059	6.12	6.59	10.99
chemwest10	2021	2021	7353	0.082	9.63	10.37	16.10
chemwest11	989	989	3537	0.043	5.24	5.67	6.30
cirphys	991	991	6027	—	11.92	13.96	15.01
grenoble1	115	115	421	0.004	0.33	0.39	0.54
grenoble2	185	185	1005	0.008	0.66	0.74	1.02
grenoble3	216	216	876	0.008	0.23	0.27	0.39
grenoble4	216	216	876	0.008	0.27	0.30	0.44
grenoble5	343	343	1435	0.016	0.38	0.43	0.61
grenoble6	512	512	2192	0.024	0.43	0.49	0.73
grenoble7	1107	1107	5664	0.066	5.29	5.85	6.12
sherman1	1000	1000	3750	0.04	2.12	2.41	3.09
sherman2	1080	1080	23094	1.12	68.53	58.36	98.99
sherman3	5005	5005	20033	0.26	42.39	43.05	97.81
sherman4	1104	1104	3786	0.04	4.46	4.81	19.10
sherman5	3312	3312	20793	0.51	52.86	48.00	114.54
random1	100	100	965	—	3.42	3.83	4.95
random2	300	300	9066	—	129.79	139.98	247.46
random3	500	500	25070	—	684.22	709.75	Mem-Out
random4	800	800	64061	—	3714.42	3500.70	Mem-Out
random5	1000	1000	99944	—	8176.44	7991.06	Mem-Out
walsh1	2^{32}	2^{32}	2^{64}	—	0.23	0.29	0.18
walsh2	2^{64}	2^{64}	2^{128}	—	0.96	1.20	0.72
walsh3	2^{100}	2^{100}	2^{200}	—	2.36	3.02	1.80
walsh4	2^{200}	2^{200}	2^{400}	—	9.83	12.46	7.40
walsh5	2^{300}	2^{300}	2^{600}	—	22.89	29.92	17.51

for good performance. We have experimented on the benchmark matrices by varying the size of the cache. The major indication that we have got from our data is that a cache of about 10^5 entries allows a fair management of sufficiently large matrices.

4. Shortest path algorithms

Calculating shortest paths is a graph problem for which sophisticated solution algorithms are available in the literature (see, for example, [12, 18]). Dijkstra's and Bellman-Ford procedures efficiently treat the single-source shortest path problem (i.e., the problem of finding the weight of the shortest path from a given source vertex to all the other vertices in the graph), while Repeated-Squaring, Floyd-Warshall, and Johnson's algorithms focus on all-pairs shortest path (i.e., finding the weight of the shortest path between any pair of vertices in the graph).

All the methods above are based on traditional matrix manipulation techniques; therefore, their major limitation concerns the sizes of the graphs they are able to handle. In this section, we present ADD-based realizations of Bellman-Ford, Repeated Squaring, and Floyd-Warshall procedures that can be effectively used to process graphs with more than 10^{27} vertices and more than 10^{36} edges.

4.1. Weighted directed graphs

A *weighted directed graph* is a triple

$$G = (V, E, w),$$

where V is a finite set, E is a binary relation on V, and w is a function from E to R. The set V is called the *vertex set* (or *node set*) of G, and its elements are called *vertices* (or *nodes*). The set E is called the *edge set* of G; its elements are called *edges* and are ordered pairs of vertices; for example, if there is an edge going from vertex v_i to vertex v_j of G, we indicate such an edge as (v_i, v_j). Note that *self-loops* (edges from a vertex to itself) are possible. The function $w : E \mapsto R$, is called *weight function*, and associates a weight to each edge in E.

If (v_i, v_j) is an edge in a weighted directed graph $G = (V, E, w)$, then we write $v_i \rightarrow v_j$, and we say that vertex v_j is *adjacent* to vertex v_i. Note that the adjacency relation in a directed graph is not necessarily symmetric.

A *path*, p, from a vertex u to a vertex v in a weighted directed graph $G = (V, E, w)$ is a sequence $\langle v_0, v_1, \ldots, v_k \rangle$ of vertices such that $u = v_0$, $v = v_k$, and $(v_{i-1}, v_i) \in E$, for $i = 1, 2, \ldots, k$; p *contains* the vertices $v_0, v_1, \ldots, v_k$ and the edges (v_0, v_1), (v_1, v_2), ..., (v_{k-1}, v_k). If there exists a path p connecting vertex u to vertex v, then we write $u \stackrel{p}{\rightarrow} v$, and we say that v is *reachable* from u via p. If all the vertices of p are distinct, then p is *simple*. The *weight* of p is the sum of the weights of its constituent edges:

$$w(p) = \sum_{i=1}^{k} w_{i-1,i},$$

where $w_{i-1,i} = w(v_{i-1}, v_i)$ is the weight associated to edge $(v_{i-1}, v_i) \in E$.

Given two vertices $u, v \in V$ of G, the *shortest path weight* from u to v is defined as:

$$\delta(u, v) = \begin{cases} \min\{w(p): u \xrightarrow{p} v\} & \text{if there is a path } u \xrightarrow{p} v, \\ +\infty & \text{otherwise.} \end{cases}$$

A *shortest path* from vertex u to vertex v is then defined as any path p of weight $w(p) = \delta(u, v)$.

In a weighted directed graph, $G = (V, E, w)$, a path $p = \langle v_0, \ldots, v_k \rangle$ forms a *cycle* if $v_0 = v_k$; the cycle is simple if $v_1, \ldots, v_k$ are distinct. A self-loop is a cycle of length 1.

A weighted directed graph, $G = (V, E, w)$, can be represented as a square matrix, $A_G(a_{ij})$, called the *adjacency matrix* of G. A_G has as many rows and as many columns as the number of vertices of G, $|V|$; each entry, a_{ij}, of A_G is such that:

$$a_{ij} = \begin{cases} w_{ij} & \text{if } (v_i, v_j) \in E, \\ +\infty & \text{otherwise.} \end{cases}$$

4.2. Single-source shortest path

In the case of single-source shortest path, the `Bellman-Ford` algorithm (figure 8) takes, as inputs, the adjacency matrix, A_G, of the graph $G = (V, E, w)$, and the source vertex, $s \in V$, and returns the vector $S_s = (d_{s,v_i}), \forall\ v_i \in V$ of the shortest path weights between the source vertex s and every vertex v_i of G.

The procedure runs in time $O(|V|\,|E|)$ where $|V|$ is the number of vertices in the graph and $|E|$ is the number of edges (we make up to $|V|$ passes over all the edges). In practice, however, the run-time is usually much smaller, since the shortest path can often be found

```
Bellman_Ford (A_G,s) {
    S_s = Extract_Source (A_G,s);
    for (i = |V| − 1; i ≥ 0; i + +) {
        if (Negative_Else_Branch (S_s)) {
            Print ("Path Contains Negative Cycle");
            break;
        }
        w = SR_Matrix_Multiply (S,A_G);
        if (S_s == w) {
            Print ("Shortest Path From Source s Found.");
            break;
        }
        S_s = w;
    }
    return S_s;
}
```

Figure 8. The `Bellman_Ford` algorithm.

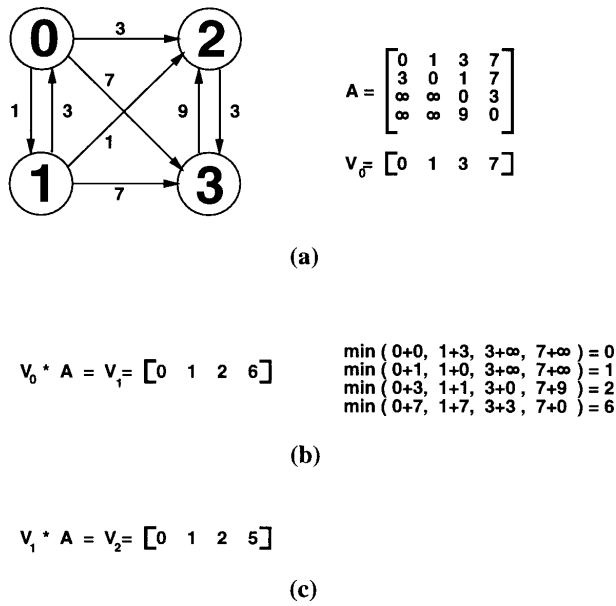

Figure 9. Application example of the `Bellman_Ford` algorithm.

in fewer than $|V|$ iterations. Furthermore, when the graph is represented symbolically, examining all edges does not necessarily require time proportional to their number. The procedure call to `SR_Matrix_Multiply` is the semi-ring multiplication procedure applied to $(\Re \cup \{+\infty, -\infty\}, \min, +, +\infty, 0)$.

The operation of this procedure is illustrated in figure 9(a). Node 0 is the source node. On the first iteration, we apply the min operation and we get the result of figure 9(b). After the second iteration, we get the final result of figure 9(c), which is the single-source shortest path for node 0.

4.3. All-pairs shortest path

Computation of all-pairs shortest path weights can be done using an extension of `Bellman-Ford` procedure to all vertices. This translates to repeatedly performing semi-ring matrix multiplication operations on the original graph using the ADD data structure, that is, we can compute the shortest path weights by extending shortest paths edge by edge. Each matrix multiplication extends the shortest path by one more edge. Such an algorithm will compute the all-pairs shortest path weights in $O(|V|^4)$. This is due to the fact that we are performing semi-ring matrix multiplication, an $O(|V|^3)$ operation, over all the vertices. This algorithm can be improved by applying `Repeated-Squaring`. Since we are not interested in intermediate results we can, instead of multiplying by the original matrix, multiply the resulting matrix by itself. The running time for this algorithm is $O(|V|^3 \log(|V|))$.

```
Floyd_Warshall (A_G) {
    S = A_G;
    for (k = 0; k < |V|; k++) {
        R = Extract_Row (S,k);
        C = Extract_Column (S,k);
        P = Out_Sum (S,R,C);
        S = P;
    }
    return S;
}
```

Figure 10. The `Floyd_Warshall` algorithm.

The second method we consider in computing the shortest path weights is based on the `Floyd-Warshall` algorithm. Instead of considering paths with increasing number of edges, the algorithm focuses on the intermediate vertices of a shortest path. For any pair of vertices $i, j \in V$, it considers all paths from i to j whose intermediate vertices are all drawn from a sub-set of vertices.

Procedure `Floyd-Warshall`, whose pseudo-code is shown in figure 10, computes the outer sum of a matrix, $A(x, y)$, and two vectors, $r(x)$ and $c(y)$. The recursive definition of outer sum is the following:

$$\min(A(x, y), c(x) + r(y)) = v \cdot \min(A_v(x, y), c_v(x) + r_v(y)) + v' \cdot \min(A_{v'}(x, y), c_{v'}(x) + r_{v'}(y)),$$

where v is the top variable between A, r, and c.

Input parameter to the procedure is the adjacency matrix, A_G, of the graph $G = (V, E, w)$, and the result of the computation is matrix $S = (d_{v_i,v_j})$, $\forall\ v_i, v_j \in V$ of the shortest paths between every pair of vertices v_i, v_j of G.

Procedure `Out_Sum` computes the minimum of the outer sum and the original matrix. The run time for `Out_Sum` is $O(|V|^2)$, since we are considering all possible pairs of vertices (i, j) through some intermediate node k. Since we must consider all possible nodes as intermediate nodes, the algorithm calculates the outer sum $|V|$ times giving a total running time of $O(|V|^3)$.

At first glance, it seems that `Floyd_Warshall` performs better than `Repeated_Squaring`, since `Floyd_Warshall` has a better time bound. However, the $O(|V|^3 \log(|V|))$ time bound for `Repeated_Squaring` is worst case; in practice, all-pairs shortest paths can be found in fewer than $\log(|V|)$ iterations, and the cost of each iteration is not proportional to V^3 in the typical case. In contrast, `Floyd_Warshall` must iterate through all $|V|$ vertices in the graph before guaranteeing that the shortest path has been found for all pairs of vertices. In fact, `Repeated_Squaring` does out-perform `Floyd_Warshall` for large graphs.

4.4. Relation to FSM traversal algorithms

When applied to the Boolean semi-ring $(\{0,1\}, +, \cdot, 0, 1)$, `Bellman_Ford` reduces to symbolic breadth-first traversal, where at each traversal step there is a choice of the set of states whose out-going edges will be explored. If ρ is the set of all states reached in the first p stages, and ν is the set of states reached for the first time at the pth stage, then any φ satisfying $\nu \leq \varphi \leq \rho$ can be used as a starting point for the $(p+1)$st stage. One can see that `Bellman_Ford` always chooses $\varphi = \rho$. It is obviously possible to modify the algorithm to choose a more convenient φ, for example ν. We refer to this last option as to the *selective trace* method.

`Repeated_Squaring` corresponds to the iterative squaring algorithm for FSM traversal. As in the case of FSM traversal, `Repeated_Squaring` may be more or less time consuming than `Bellman_Ford`, depending on the diameter of the sub-graph containing the source node of `Bellman_Ford`. If the diameter is large, then solving the all-pairs shortest path problem may actually prove cheaper than solving the single-source problem.

When considering integer lengths for the arcs, `Bellman_Ford` gives the same information as breadth-first traversal, provided the sets of states reached for the first time at each stage are saved in the latter. By contrast, iterative squaring on a Boolean semi-ring only tells us if a path exists between two states, whereas `Repeated_Squaring` on the semi-ring of the non-negative integers also provides the path lengths and, therefore, the graph diameter.

As in the case of `Bellman_Ford`, we can modify the `Repeated_Squaring` algorithm to re-compute distances only for pairs of nodes that "got closer" in the previous stage.

4.5. Experimental results

The key of using ADDs to implement the shortest path algorithms is that we are able to take advantage of the structure of the given graph by recombining like structures and representing them in the same ADD. In our experiments we have found that even for graphs with more than 10^{27} vertices, their corresponding ADDs have been fairly manageable in size.

We have run shortest path algorithms on state transition graphs of large sequential benchmark circuits; these circuits represent currently the most readily available source of large graphs, and they have been extracted from the topological description of the circuit by building the corresponding transition relation with a symbolic finite state machine traversal algorithm [14].

It is also important to note the role of caching when implementing these algorithms using ADDs. When caching the result of the matrix multiplication using `Repeated_Squaring`, the hit ratio was almost 80%. This, of course, can have a tremendous impact on the running time of the algorithm. Also, the size of the cache could have a `major` influence on this hit ratio. Too small of a cache could increase the run time by orders of magnitude.

For single-source shortest path calculation, the source vertex has been chosen by randomly picking a minterm belonging to the set of the initial states of the circuit, specified during the traversal phase.

In Table 2 we report the results; times (in seconds) have been measured on a DEC-Station 5000/200 with 80 MB of memory.

Table 2. Results for shortest paths.

Graph	# of Nodes	# of Edges	Algorithm	Time (sec.)
s1488	64	133	BellmanFord	0.14
			Bellman_Ford_st	0.10
			Repeated_Squaring	15.94
			Repeated_Squaring_st	9.18
			Floyd_Warshall	0.93
s208	256	511	Bellman_Ford	0.08
			Bellman-Ford_st	0.04
			Repeated_Squaring	1.29
			Repeated_Squaring_st	0.78
			Floyd_Warshall	0.80
s298	$1.64 \cdot 10^4$	$8.14 \cdot 10^4$	Bellman_Ford	0.28
			Bellman_Ford_st	0.18
			Repeated_Squaring	24.48
			Repeated_Squaring_st	17.35
			Floyd_Warshall	102.96
s349	$3.28 \cdot 10^4$	$2.15 \cdot 10^6$	Bellman_Ford	1.17
			Bellman_Ford_st	1.45
			Repeated_Squaring	933.06
			Repeated_Squaring_st	741.73
			Floyd_Warshall	Not Run
s1238	$2.62 \cdot 10^5$	$3.56 \cdot 10^8$	Bellman_Ford	18.77
			Bellman_Ford_st	15.18
			Repeated_Squaring	159.61
			Repeated_Squaring_st	81.98
			Floyd_Warshall	35453.34
s400	$2.1 \cdot 10^6$	$1.05 \cdot 10^7$	Bellman_Ford	21.98
			Bellman_Ford_st	8.64
			Repeated_Squaring	41146.96
			Repeated_Squaring_st	7410.69
			Floyd_Warshall	Not Run
s953	$5.37 \cdot 10^8$	$5.98 \cdot 10^9$	Bellman_Ford	0.71
			Bellman_Ford_st	0.67
			Repeated_Squaring	214.84
			Repeated_Squaring_st	116.53
			Floyd_Warshall	Not Run
mm9	$1.34 \cdot 10^8$	$1.37 \cdot 10^{11}$	Bellman_Ford	0.23
			Bellman_Ford_st	0.25
			Repeated_Squaring	14.96
			Repeated_Squaring_st	13.56
			Floyd_Warshall	Not Run
mm30	$1.24 \cdot 10^{27}$	$2.66 \cdot 10^{36}$	Bellman_Ford	1.20
			Bellman_Ford_st	1.22
			Repeated_Squaring	56.94
			Repeated_Squaring_st	59.37
			Floyd_Warshall	Not Run

The algorithms listed with *_st*, i.e., `Bellman_Ford_st` and `Repeated_Squaring_st`, are modifications of the original algorithms that apply the selective trace technique. That is, distances are recomputed only for pairs of nodes that "got closer" in the previous stage. Tests were run using `Floyd_Warshall` for only the smaller graphs. Finding the all-pairs shortest paths using this algorithm becomes infeasible for large graphs since the algorithm requires iterating through all nodes in the graph. Clearly, this is prohibitive for several of the graphs listed in the table.

Applying selective trace techniques to our algorithms for computing shortest paths did not always show an improvement over its unmodified version. Depending on the graph, the number of new states reached, or more specifically, the number of nodes that got closer, can be much larger than the number of reached states obtained at the previous iteration ($\nu \approx \rho$). In such a case, there is little gain, if any, in applying selective trace. On the other hand, if the number of nodes that have changed since the previous iteration is relatively small compared to the previously reached states ($\nu \ll \rho$), then these techniques will pay off. These two cases are both illustrated in the examples given in the table. It should be noted there is little overhead needed in implementing selective trace, therefore even if $\nu \approx \rho$, these results have shown that the selective trace implementation will not take much longer to run.

5. Solution of systems of linear equations

In this section we summarize some ADD approaches to the problem of solving systems of linear equations. We first discuss conventional solution algorithms based on Gaussian elimination followed by back substitution, and then we present ADD-based implementation of those methods. We also compare results we have obtained with the ADD approach to the one produced by a state-of-the-art linear algebra package based on sparse matrix representation. Finally, we propose possible alternative approaches to the solution of systems of linear equations that are currently under investigation.

5.1. Gaussian elimination based on outer product

One can view the Gaussian elimination phase of the solution of a linear system $Ax = b$, where A is a $N \times N$ non-singular sparse matrix, as a sequence of N pivoting steps [16]. A sketch of the outer product form of Gaussian elimination is the following:

1. Form matrix $A \mid b$, that is, append b to A as an extra column (or columns, if there is more than one right hand side).
2. For each row, i, of A:
   (a) Pick a pivot element a_{ij}, utilizing a combination of sparsity and stability factors.
   (b) Normalize row i of $A \mid b$, that is, divide non-zeros of $A \mid b$ in unpivoted columns in row i by a_{ij}, and put the normalized elements into a row vector r.
   (c) Form a sub-diagonal pivot column, that is, put non-zeros of A in unpivoted rows of column j into a column vector c.
   (d) Form the outer product $p = c \circ r^T$.

(e) Fill in the remainder of $A \mid b$, that is, subtract p from the sub-matrix formed by the unpivoted rows and columns of $A \mid b$.

After pivoting, A has been transformed into its upper triangular factor U, and b to $y = L^{-1}b$. Therefore, the solution of system $Ax = b$ can be computed by solving the triangular system $Ux = y$ by a simple variant of matrix multiplication (see Section 5.2).

The procedure above employs the so called partial pivoting, which promotes numerical stability while maintaining some priority for sparse efficiency. If the whole matrix is searched for the most stable pivot, computation may get more time consuming.

Depending on one's ethnic background, one emphasizes or de-emphasizes the significance of pivoting techniques for preserving numerical stability during Gaussian elimination. For example, in the case of digital circuit simulation, pre-pivoting techniques, in which the pivoting order is determined *a priori* by a weighted combination of sparsity and numerical factors, is quite effective [20]. Similar evaluation in other technical domains is reported in [16].

5.2. *Back substitution*

Once the Gaussian elimination phase of $Ax = b$ has terminated, matrix A is in upper triangular form (U factor), and column vector b has become $y = L^{-1}b$. The system to be solved is then $Ux = y$. Solution x can be recursively computed by means of a back substitution step, carried out by procedure `Back_Solve` outlined below. Let us write system $Ux = y$ as:

$$\begin{bmatrix} U_{ee} & U_{et} \\ 0 & U_{tt} \end{bmatrix} \begin{bmatrix} x_e \\ x_t \end{bmatrix} = \begin{bmatrix} y_e \\ y_t \end{bmatrix}.$$

Then, x can be determined as follows:

$$\begin{aligned} x_t &= \texttt{Back\_Solve}(U_{tt}, y_t), \\ z &= y_e - U_{et} \cdot x_t, \\ x_e &= \texttt{Back\_solve}(U_{ee}, z). \end{aligned}$$

5.3. *ADD realization*

The pseudo-code for the ADD-based implementation of the Gaussian elimination algorithm sketched in Section 5.1 is shown in figure 11. The code assumes matrix A to be a $N \times N$ matrix; moreover, it assumes that the right hand side(s) have been appended to A as extra columns. Thus, b does not appear explicitly in the pseudo-code. In practice, one more column encoding variable is added, so up to N right hand sides may be accommodated simultaneously.

Auxiliary ADDs R and C are built for future use in screening out already completed rows and columns from the outer product matrix. On each pass through the **for** loop, processing begins by selecting an appropriate pivot element in row i (procedure `Select_Pivot`); the

```
Gauss_Elim (A(r,c)) {
    R(r) = 1;
    C(c) = 1;
    A^1(r,c) = A(r,c);
    for(i = 1,i < N,i++) {
        (i,j,Pivot)=Select_Pivot (A);
        (r_i,c_j)= RC_Cubes (i,j);
        R = R ∨ r_j;
        C = C ∨ c_j;
        L^j(r) = (A^i(r,c)_{c_j} · R'(r));
        row(c) = (A^i(r,c)_{r_i} · C'(c));
        U^i(c) = Apply (row(c),-1/Pivot,*);
        Δ(r,c) = Apply (L^j(r),U^i(c),*);
        A^{i+1}(r,c) = Apply (A^i(r,c),Δ(r,c),+);
        A^{i+1}(r,c) = ITE (r_i,U^i(c),A^{i+1}(r,c));
    }
    return A^{N+1}(r,c);
}
```

Figure 11. The `Gauss_Elim` algorithm.

criteria used for selecting the pivot take into account either sparsity or numerical stability or both. Then, cubes r_i and c_j are formed from the binary encodings of i and j (procedure `RC_Cubes`). These logic functions are disjoined with the set representations R and C, whose complements are used to select rows and columns and shape the outer product update matrix, Δ. Then, matrix A is cofactored with respect to cube c_j and intersected with the updated complement C to form the column L^i of L. A row U^i of U is similarly computed, but also requires a call to `Apply` for normalization purposes. A second call to `Apply` is used to form the outer product matrix Δ, and a third call adds this to A^i to determine the matrix A^{i+1}. Finally, before completion, we need to replace the ith row of A^{i+1} by U^i. This is done by the call to `ITE`. The cumulative effect of these substitutions is that $A^{n+1} = U$, which is returned on exiting the **for** loop.

As we said in Section 5.2, after Gaussian elimination the system to be solved is $Ux = y$, where $y = L^{-1}b$, and the solution x can be computed by means of the recursive procedure `Back_Solve`, whose pseudo-code in ADD form is shown in figure 12.

We summarize the operation of this code as follows. First, note that the procedure considers only a single right hand side. In the actual implementation we use extra implicit coding variables to handle up to $N = 2^n$ right hand sides. Also, if the actual specified upper triangular matrix is UT, we then define $UT(r, c) = I(r, c) + U(r, c)$. Only the off-diagonal (upper triangle) portion U of UT is explicitly stored, and the input argument of `Back_Solve` is U, not UT. Therefore (because of triangularity), U cannot be:

- Constant other than 0;
- Independent of r;
- Independent of c.

```
Back_Solve (U(r,c),y(r)) {
    if (U(r,c) == 0 or y(r) == 0)
        return y(r);
    if (Table_Lookup (cache,(Back_Solve,U,y),R)
        return R;
    top_r = Top_Row_Var(U(r,c),y(r));
    top_c = Top_Column_Var(U(r,c),y(r));
    U_ee(r,c) = U_top_r'top_c'(r,c);
    U_et(r,c) = U_top_r'top_c(r,c);
    U_tt(r,c) = U_top_r top_c(r,c);
    y_e(r) = y_top_r'(r);
    y_t(r) = y_top_r(r);
    x_t(c) = Back_Solve (U_tt(r,c),y_t(r));
    p(r) = Matrix_Multiply (U_et(r,c),x_t(c));
    d(r) = Apply(y_e(r),p(r),-);
    x_e(c) = Back_Solve (U_ee(r,c),d(r));
    x(c) = ITE (top_c,x_t(c),x_e(c));
    R = x(c);
    Table_Insert (cache,(Back_Solve,U,y),R);
    return R;
}
```

Figure 12. The `Back_Solve` algorithm.

Consequently, the only terminal case is $U(r, c) = 0$, in which case we return the right hand side y. After checking the cache, the three cofactors of U and the two of y are computed. Note that although two-way splitting would normally produce four sub-matrices, in the present case we always have $U_{te} = 0$. Then the lower right system is solved for $x_t(c)$, and the matrix product $U_{et}(r, c) \times x_t(c)$ is subtracted from y_e. This forms $d(r)$, the modified right hand side for the upper right system of equations, which are then solved by the second recursive call to `Back_Solve`. We then conclude by using the top column variable to merge the $x_e(c)$ and $x_t(c)$ parts of the solution with a call to `ITE`, by inserting the solution into the cache, and by returning it.

We remark that we have chosen to represent only the off-diagonal part of U as an ADD, implicitly accounting for the unit diagonal of U. We experimented with another terminal case, in which the off-diagonal part of U was independent of the row variables r, and in which each component of the returned solution would have the same value. This test turned out to be too expensive to pay-off on the benchmarks we studied. However, we note that this variant (in contrast to the case for Gaussian elimination) enables us to escape with less than $N = 2^n$ recursive calls in at least the special cases in which $U = 0$ on input. Because U represents only the off-diagonal part of UT, this case corresponds to solving a system whose coefficient matrix is the $N \times N$ identity matrix.

5.4. Recursive formulation of Gaussian elimination

Procedure `Gauss_Elim` of Section 5.3 essentially computes the LU factors of a matrix using pivoting techniques; however, there are many other schemes for efficiently accomplishing

the same job. For example, the so called Crout algorithm iteratively computes a column of L, followed by a row of U, for each new element adding to the original element an inner product of a partial row of L and a partial column of U. The Doolittle, or row-wise, form adds multiples of previous rows of U to the pivot row, thus forming a new row of L and U, repeating this sequence for each pivot row.

Another approach [1], called 4-way decomposition, computes L and U by recursively breaking down the $N \times N$ matrix A into 4 $N/2 \times N/2$ sub-matrices. McGeer et al. [11] have already discussed an ADD algorithm for this approach to Gaussian elimination.

The idea behind this approach is that matrix $A = LU$ can be decomposed as:

$$A = \begin{bmatrix} A_{ee} & A_{et} \\ A_{te} & A_{tt} \end{bmatrix} = \begin{bmatrix} L_{ee} & 0 \\ L_{te} & L_{tt} \end{bmatrix} \begin{bmatrix} U_{ee} & U_{et} \\ 0 & U_{tt} \end{bmatrix} = LU,$$

and therefore the LU factors can be recursively determined by computing (using procedure `Four_Way_LU`):

$$\begin{aligned} (L_{ee}, U_{ee}) &= \texttt{Four\_Way\_LU}(A_{ee}), \\ U_{et} &= \texttt{Forward\_Solve}(L_{ee}, A_{et}), \\ L_{te} &= \texttt{Right\_Solve}(U_{ee}, A_{te}), \\ M &= A_{tt} - L_{te} \cdot U_{et}, \\ (L_{tt}, U_{tt}) &= \texttt{Four\_Way\_LU}(M), \end{aligned}$$

and by appropriately merging the sub-factors L_{ee}, L_{te}, L_{tt} and U_{ee}, U_{et}, U_{tt}.

5.5. ADD-based LDU factorization

We have formulated the 4-way decomposition algorithm in terms of LDU factorization, and we have implemented it using ADDs. This approach can be more naturally realized in a symbolic fashion than the outer product approach of Section 5.3, due to the recursive formulation of the algorithm.

Notice that the difference between LU factorization presented in Section 5.4 and LDU factorization used by algorithm `Four_Way_LDU` is that in the LU approach either L or U is normalized (1's on the main diagonal), while in the LDU approach, both L and U are normalized, and the pivot elements are kept in a separate diagonal matrix D. This incurs some extra arithmetic for the normalization, but this can be gained back in the symmetric case, that is, when $L = U^T$. A final variant for the symmetric case is Cholesky factorization, where U is partially normalized by dividing the pivot row by the square root of the pivot.

The pseudo-code of procedure `Four_Way_LDU` is presented in figure 13.

The procedure computes the LDU factorization of matrix A, which is supposed to be $2^n \times 2^n$. Notice that, if necessary, matrix A has been padded by appending an identity to the lower right corner. The first step, after checking for the terminal case (no more variables to split on, the pivot has been found) and looking up the cache for possibly existing result, is to compute the four cofactors of A (A_{ee}, A_{et}, A_{te}, and A_{tt}). Then, the first recursive

```
Four_Way_LDU (A, n) {
    if (n == 0) return (Value (A));
    if (Table_Lookup (cache,(Four_Way_LDU,A),R))
        return R;
    top_A = Top_Var(A);
    A_e = A_top_A; A_t = A_top_A;
    top = Top_Var(A_e,A_t);
    A_ee = A_e_top; A_et = A_e_top; A_te = A_t_top; A_tt = A_t_top;
    R_ee = Four_Way_LDU (A_ee,n − 1);
    L_ee = Lower_Triang (R_ee);
    y = Forward_Solve (L_ee,A_et);
    D_e = Diag (R_ee);
    U_et = Apply (y,D_e,/);
    U_ee = Upper_Triang (R_ee);
    y = Right_Solve (U_ee,A_te);
    L_te = Apply (y,D_e,/);
    tmp = Matrix_Multiply (y,U_et);
    M = Apply (A_tt,tmp,−);
    R_tt = Four_Way_LDU (M,n − 1);
    R_t = ITE (top_column_var,R_tt,L_te);
    R_e = ITE (top_column_var,R_ee,U_et);
    R = ITE (top_row_var,R_t,R_e);
    Table_Insert (cache,(Four_Way_LDU,A),R);
    return R;
}
```

Figure 13. The Four_Way_LDU algorithm.

call to Four_Way_LDU determines the upper left solution of the factorization, R_{ee}. The lower off-diagonal, L_{ee}, and the diagonal, D_e, are extracted from R_{ee}, and used to compute, through procedure Forward_Solve, U_{et}. Similarly, the upper off-diagonal U_{ee} of R_{ee} is extracted and it is used to compute, through procedure Right_Solve, L_{te}. Factorization proceeds then by taking the product of U_{et} and the out-come of Right_Solve, and by subtracting this result from A_{tt} to obtain M, which is used to get R_{tt} through the second recursive call to Four_Way_LDU. Finally, R_{tt} and L_{te} are combined to get R_t by means of procedure ITE. Similarly, R_{ee} and U_{et} are put together to form R_e. The top row variable is then used to build the final solution, R, which is stored in the cache and then returned. Notice that procedures Forward_Solve and Right_Solve are very similar to procedure Back_Solve of Section 5.3, so they are not discussed in detail here.

It is interesting to observe that, although the outer product form of Gaussian elimination (procedure Gauss_Elim) requires a loop over the N pivots and 4-way decomposition (procedure Four_Way_LDU) is based on implicit enumeration, both compute exactly N pivots, in accordance with a theoretical result discussed in detail in [21]. Therefore, as terminal condition for the 4-way decomposition algorithm, we simply need to check if the depth of the recursion is $n = \log(N)$. Notice also that, when we apply the LDU form of 4-way decomposition to asymmetric matrices, we do a little extra arithmetic in normalizing L as well as U.

5.6. Experimental results

We have compared the performance of the ADD-based solution methods to the one of a state-of-the-art linear algebra package. We have chosen the Harwell package *ma28*, a FORTRAN package for solving sparse systems of linear equations using the row-wise approach. It uses the data structure discussed in Section 3.5. The sparse arithmetic processing for both the elimination and back substitution phases is also quite similar. The theory for the *ma28* package is given in [15]. We will refer to the *ma28* elimination routine as `Row_Elim`.

Results on some selected benchmarks have been quite disappointing. In particular, the ADD versions of both `Gauss_Elim` and `Four_Way_LDU` are much slower (factors of 50–500) than the *ma28* implementation of `Row_Elim`. In this case, caching and recombination are barely appreciable factors. Furthermore, the ADD versions of `Back_Solve` is somewhat slower than the corresponding *ma28* code. Here, caching and recombination are appreciable but not dominant.

5.7. Alternative approaches

Results of Section 5.6 paint a rather negative picture for ADD solutions of linear algebraic equations. In this respect, we have investigated two modifications to the ADD approach:

1. Iterative methods, where elimination and *LU* factorization are replaced by matrix multiplication, in which one of the matrix factors is constant, thereby eliminating the need for canonicity management.
2. Direct methods based on a variant of ADDs, the Structural Decision Diagrams (SDDs), which do not allow recombination; absence of recombination helps in making computation faster, but destroys canonicity.

In the following sections, we comment briefly on these alternatives before leaving the subject of solving linear equations.

5.7.1. Iterative methods. Many iterative methods have been used for solving linear algebraic equations, and among these, the Gauss-Seidel algorithm is probably the most well known and has favorable convergence properties. However, since it involves what amounts to a back substitution, it requires considering each equation individually, which is not so promising for symbolic methods. However, we might at a later time consider a block Gauss-Seidel approach, with the equations partitioned to maximize the probability of the identity matrix special case.

An intuitively more attractive approach in the symbolic setting is the Gauss-Jacobi method, which is based on straight-forward matrix multiplication:

$$Ax = b \Rightarrow (D + R)x = b$$
$$x^{k+1} = D^{-1}(b - Rx^k) \Rightarrow x^{k+1} = -Rx^k + D^{-1}b$$

where D is the diagonal of A and R is the rest of A, that is, the off-diagonal part. Convergence is guaranteed if A is diagonally dominant. However, this is unfortunately not the case in many applications.

An alternative is to use the modified iteration:

$$x^{k+1} = b - \tilde{R}x^k,$$

where $A = I + \tilde{R}$, which is based on iterative refinement. This corresponds to a partial extraction of the diagonal, that may induce a reduction of the size of the eigenvalues of the update matrices. Also, there exist various known techniques for pre-conditioning the update matrix to promote diagonal dominance and/or lower the size of the update matrix eigenvalues.

We have implemented ADD procedures for iterative solution of systems of linear equations in the context of probabilistic analysis of large finite state machines [21, 22].

5.7.2. SDDs: Structural Decision Diagrams. The strength of the ADD lies in its ability to combine all non-zeros of a sparse matrix which have the same numerical value into a single terminal value in the binary decision diagram. It is this feature that enables ADD multiplication to be done, subject to the specified ADD ordering, with the absolute minimum number of multiplications. More specifically, suppose we count the number of times that the subroutine is called that handles the special case of actually performing a scalar multiplication. (For example, when the recursion has hit a minterm of both operands.) In the absence of recombination, this would count the actual number of multiplications required by the traditional method. However, the ADD algorithms cache intermediate results, and if there are no cache collisions, no multiplications are ever repeated. With these provisos, there is a unique number for each ADD variable ordering, and the minimum of these counts over all the orderings is the minimum number of multiplications for computing the given product. It thus appears that the ADD multiplication algorithm is implicitly building a sort of factored form for the matrix multiplication.

At first glance, this sounds very interesting. However, as we have discussed in Section 5.6, the canonicity management overhead completely dominates the actual accumulated multiplication time. Further, it is expensive to recompute the ADDs for the operands if any of their values change. As an example of this fact, suppose (as in the case of circuit simulation) that we are repeatedly solving the following equations:

$$\begin{aligned} J(x^k)\Delta x &= b(x^k), \\ x^{k+1} &= x^k + \Delta x, \\ k &= k + 1. \end{aligned}$$

Here, $J(x)$ represents the Jacobian of the non-linear algebraic system being solved at a given time step of the simulation, and $b(x)$ the associated right hand side. Sparse matrix technology deals with these equations with representations of the structural pattern of non-zeros in $J(x)$, independent of their values. In contrast, an ADD changes its pattern when one of its values changes. Thus, the cost of changing the value of the first non-zero element of a row of J is $O(1)$ for sparse matrix representations, and $O(|J|)$ for an ADD. Of

course, in practice the worst case seldom occurs. This effect is one of the major causes of the relatively poor performance of ADD matrix multiplication on existing sparse matrix benchmark problems. One tool for overcoming this difficulty is the Structural Decision Diagram (SDD).

Briefly, an SDD is just an ADD in which recombination is prohibited. However, all other properties of ADDs do persist, except canonicity. Therefore, it should suffice to introduce SDDs by means of a small example. The SDD $J_M(x, y)$ for a simple matrix M is illustrated in figure 14.

This SDD is built on top of the traditional data structure for the simple matrix of figure 7. The SDD data structure consists of the usual ADD (without recombination), plus row

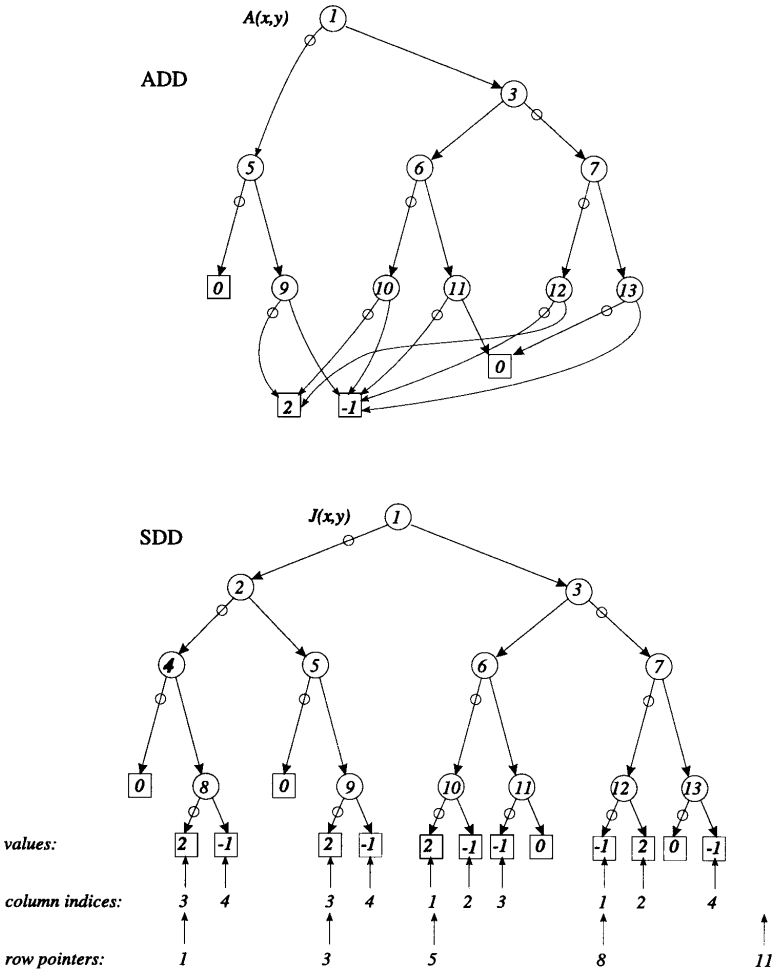

Figure 14. The SDD for a simple sparse matrix.

pointers (one per row) and column indices (one per non-zero). The SDD J is accompanied by a values array vJ, which has one entry for each of the terminal nodes of the SDD. An SDD, $J(x, y)$, can be seen as an ADD whose set of constant values P are *pointers*, rather than real numbers.

Although we have said that SDDs have no recombination, we actually allow a restricted form of recombination, in which identical rows can recombine. (But not identical elements in a given row.) Thus, for a matrix whose elements change but whose row identities are invariant, the SDD will also be invariant.

6. Conclusions and future work

In this paper we have presented Algebraic Decision Diagrams (ADDs), an efficient data structure for the manipulation of functions from $\{0, 1\}^n$ to S, where S is the finite carrier of the algebraic structure over which the functions are defined.

We have proposed and discussed a number of ADD-based algorithms for general purpose applications, such as matrix multiplication, shortest-path computation in graphs, and numerical linear algebra.

Experimental results have been presented in each area, which indicate varying degrees of success for our approach. We conclude this paper with a summary of the current status in each of the aforementioned areas, and with implications for future research.

Matrix multiplication. We have described three algorithms for ADD based matrix multiplication. Of these, one is original, and two are modifications of algorithms sketched very briefly in [10] and [11]. Comparison to sparse matrix technology was made on benchmarks deriving from economic models, PDEs for shale oil analysis and chemical plant modeling.

The ADD-based multiplication was slower (factors of 20–100) than classical sparse matrix methods. We believe this is due to the fact that recombination in product matrices for the benchmarks used in the comparisons was not sufficiently massive to overcome the burden of canonicity management overhead. Although this is disappointing, the new methods do provide a basis for shortest path computations, and performed well on cases such as the Walsh matrices, where the problem size was beyond the range of conventional methods.

By implementing algorithms for the closed semi-ring and quasi-ring, we tried to achieve algebraic generality. This was reflected in the pervasiveness of multiplication as work routines in applications as diverse as solving linear algebraic equations and solving the all pairs shortest path problem. Further, although we discussed primarily multiplication of real numbers, the quasi-ring formulation means that our methods would work equally well on ADDs with complex constants, or constants from any algebraic field.

Shortest-path computation. We have developed three algorithms for shortest-path computations based on our ADD package: Bellman-Ford, Repeated Squaring, and Floyd-Warshall. Bellman-Ford and Repeated Squaring were able to perform reasonably well on some large (but shallow) graphs, as well as on some fairly deep graphs derived from symbolic descriptions of FSMs. The latter graphs were of medium complexity, having over one million

states. It is nevertheless noteworthy that the ADD of a graph with so many possible edge weights remained fairly small. The performance of Floyd-Warshall was severely limited on such graphs by the mandatory loop over the N vertices of the graph.

Solution of linear algebraic equations. Results were disappointing with regard to traditional methods of Gaussian elimination. Therefore, we are devoting our ongoing research to the investigation of alternative data structures, and to techniques that exploit the structure of the problem. In these methods, the original structure of the coefficient matrix, as well as matrix multiplication itself, play a more dominant role. Recombination might be further promoted in this regard by truncating the non-zero elements of a given matrix into a restricted sub-set, and then using iterative refinement based on multiplication to correct a possibly inaccurate solution.

References

1. A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison Wesley, 1974.
2. G. Boole, *The Mathematical Analysis of Logic*, Macmillan, 1847, Reprinted by B. Blackwell, Oxford, UK, 1951.
3. K.S. Brace, R. Rudell, and R. Bryant, "Efficient implementation of a BDD package," *DAC-27: ACM/IEEE Design Automation Conference*, Orlando, FL, June 1990, pp. 40–45.
4. F.M. Brown, *Boolean Reasoning: The Logic of Boolean Equations*, Kluwer Academic Publishers, 1990.
5. R. Bryant, "Graph-Based Algorithms for Boolean function manipulation," *IEEE Transactions on Computers*, Vol. C-35, No. 8, pp. 79–85, Aug. 1986.
6. J.R. Burch, E.M. Clarke, K.L. McMillan, and D.L. Dill, "Sequential circuit verification using symbolic model checking," *DAC-27: ACM/IEEE Design Automation Conference*, Orlando, FL, June 1990, pp. 46–51.
7. J.R. Burch, E.M. Clarke, and D.E. Long, "Representing circuits more efficiently in symbolic model checking," *DAC-28: ACM/IEEE Design Automation Conference*, San Francisco, CA, June 1991, pp. 403–407.
8. H. Cho, G.D. Hachtel, S.W. Jeong, B. Plessier, E. Schwarz, and F. Somenzi, "ATPG aspects of FSM verification," *ICCAD-90: IEEE International Conference on Computer Aided Design*, Santa Clara, CA, Nov. 1990, pp. 134–137.
9. H. Cho, G.D. Hachtel, E. Macii, B. Plessier, and F. Somenzi, "Algorithms for approximate FSM traversal," *DAC-30: ACM/IEEE Design Automation Conference*, Dallas, TX, June 1993, pp. 25–30.
10. E.M. Clarke, K.L. McMillan, X. Zhao, M. Fujita, and J. Yang, "Spectral transforms for large Boolean functions with applications to technology mapping," *DAC-30: ACM/IEEE Design Automation Conference*, Dallas, TX, June 1993, pp. 54–60.
11. E.M. Clarke, M. Fujita, P.C. McGeer, K. McMillan, and J. Yang, "Multi-terminal binary decision diagrams: An efficient data structure for matrix representation," *IWLS'93: International Workshop on Logic Synthesis*, Lake Tahoe, CA, May 1993, pp. 6a:1–15.
12. T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *An Introduction to Algorithms*, McGraw-Hill, 1990.
13. O. Coudert, C. Berthet, and J.C. Madre, "Verification of sequential machines based on symbolic execution," *Automatic Verification Methods for Finite State Systems*, Lecture Notes in Computer Science, Vol. 407, pp. 365–373, 1989.
14. O. Coudert, C. Berthet, and J.C. Madre, "Verification of sequential machines using boolean functional vectors," *IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, Leuven, Belgium, Nov. 1989, pp. 111–128.
15. I.S. Duff, "Harwell Subroutine Library," AERE Report R.8730, Atomic Energy Research Establishment, Oxon, England, 1977.
16. I.S. Duff, A.M. Erisman, and J.K. Reid, *Direct Methods for Sparse Matrices*, Clarendon Press, 1986.

17. I.S. Duff, R.G. Grimes, and J.G. Lewis, "Sparse matrix test problems," *ACM Transactions on Mathematical Software*, Vol. 15, pp. 1–14, 1989.
18. S. Even, *Graph Algorithms*, Computer Science Press, 1979.
19. F.G. Gustavson, "Efficient algorithm to perform sparse matrix multiplication," *IBM Technical Disclosure Bulletin*, Vol. 20, No. 3, pp. 1262–1264, Aug. 1977.
20. G.D. Hachtel, R.K. Brayton, and F.G. Gustavson, "The sparse tableau approach to network analysis and design," *IEEE Transactions on Circuit Theory*, Vol. CT-18, No. 1, pp. 101–113, Jan. 1971.
21. G.D. Hachtel, E. Macii, A. Pardo, and F. Somenzi, "Symbolic algorithms to calculate steady-state probabilities of a finite state machine," *EDAC-94: IEEE European Conference on Design Automation*, Paris, France, Feb. 1994, pp. 214–218.
22. G.D. Hachtel, E. Macii, A. Pardo, and F. Somenzi, "Probabilistic analysis of large finite state machines," *DAC-31: ACM/IEEE Design Automation Conference*, San Diego, CA, June 1994, pp. 270–275.
23. H. Touati, H. Savoj, B. Lin, R. Brayton, and A. Sangiovanni-Vincentelli, "Implicit enumeration of finite state machines using BDDs," *ICCAD-90: IEEE International Conference on Computer Aided Design*, Santa Clara, CA, Nov. 1990, pp. 130–133.