EVBDD-based Algorithms for Integer Linear Programming, Spectral Transformation, and Function Decomposition

Yung-Te Lai, Massoud Pedram and Sarma B.K. Vrudhula

Contents

1 Introduction **1**

1.1 Background 2

2 Integer Linear Programming **6**

2.1 Background 6

2.2 A Model Algorithm 8

2.3 The Operator *minimize* 12

2.4 Discussion 17

2.5 Experimental Results 18

3 Spectral Transformation **19**

3.1 Spectral EVBDD (SPBDD) 22

3.2 Boolean Operations in Spectral Domain 24

3.3 Experimental Results 25

4 Function Decomposition **25**

4.1 Definitions 26

4.2 Disjunctive Decomposition 27

4.3 Computing Cut_sets for All Possible Bound Sets 28

4.4 Experimental Results 32

5 Conclusions **33**

List of Figures

1 Two examples. 37

2 An example of flattened EVBDD. 38

3 A simple example (using flattened EVBDDs and OBDDs). 39

4 An example for conjoining constraints. 40

5 An example for the *minimize* operator. 41

6 An example of cut_set in EVBDD. 42

7 An example of disjunctive decomposition in EVBDD. 43

8 Representation of multiple-output functions. 44

9 Operations *include* and *exclude* in the decomposition chart. 45

10 Operations *include* and *exclude* in the EVBDD representation. 46

11 An example of the application of *cut_set_all*. 47

12 Example continued. 48

13 Example continued. 49

List of Tables

1 Experimental results of ILP problems. 50
2 Experimental results of SPBDDs. 51
3 Finding all decomposable forms with bound set size ≤ 4. 52

Abstract

Edge-Valued Binary-Decision Diagrams (EVBDD)s are directed acyclic graphs which can represent and manipulate integer functions as effectively as Ordered Binary-Decision Diagrams (OBDDs) do for Boolean functions. They have been used in logic verification for showing the equivalence between Boolean functions and arithmetic functions. In this paper, we present EVBDD-based algorithms for solving integer linear programs, computing spectral coefficients of Boolean functions, and performing function decomposition. These algorithms have been implemented in C under the SIS environment and experimental results are provided.

1 Introduction

Edge-Valued Binary-Decision Diagram (EVBDD) [27] is an integer version of Ordered Binary-Decision Diagram (OBDD) [8]. EVBDDs not only preserve the canonical function and compact representation properties of OBDDs but also provide a new set of operators – arithmetic operators, relational operators and minimum/maximum operators [30]. EVBDDs have been used for logic verification [27] and Boolean function decomposition [29].

EVBDDs are directed acyclic graphs constructed in a similar way to OBDDs. As in OBDDs, each node either represents a constant function with no children or is associated with a binary variable having two children, and there is an input variable ordering imposed in every path from the root node to the terminal node. However, in EVBDDs there is an integer value associated with each edge. Furthermore, the semantics of these two graphs are quite different. In OBDDs, a node $\mathbf{v}$ associated with variable x denotes the **Boolean function** $(x \wedge f_l) \vee (\overline{x} \wedge f_r)$, where f_l and f_r are functions represented by the two children of $\mathbf{v}$. On the other hand, a node $\mathbf{v}$ in an EVBDD denotes the **arithmetic function** $x(v_l + f_l) + (1 - x)(v_r + f_r)$, where v_l and v_r are values associated with edges going from $\mathbf{v}$ to its children, and f_l and f_r are functions represented by the two children of $\mathbf{v}$. To achieve canonical property, we enforce v_r to be 0.

EVBDDs constructed in the above manner are more related to pseudo Boolean functions [21] which have the function type $\{0,1\}^n \rightarrow integer$. For example, $f(x,y,z) = 3x + 4y - 5xz$ with $x,y,z \in \{0,1\}$ is a pseudo Boolean function, and $f(1,1,0) = 7$ and $f(1,1,1) = 2$. However, for functions with integer variables, we must convert the integer variables to vectors of Boolean variables before using EVBDDs. In the above example, if $x \in \{0,\ldots,5\}$, then $f(x,y,z) = 3(4x_2+2x_1+x_0)+4y-5(4x_2+2x_1+x_0)z$ and $f(4,1,1) = -4$.

By treating Boolean values as integers 0 and 1, EVBDDs are capable of representing Boolean functions and perform Boolean operations. Furthermore, when Boolean functions are represented by OBDDs and EVBDDs, they have the same size and require the same time complexity for performing operations [30]. Thus, EVBDDs are particularly useful in applications which require both Boolean and integer operations. We present three applications of EVBDDs.

The first application is in solving integer linear programming (ILP) problems. An ILP problem is to find the maximum (or minimum) of a goal function subject to a set of linear inequality constraints. Each constraint defines a feasible subspace which can be represented as a Boolean function. The conjoining of these constraint (i.e., the conjunction of the corre-

sponding Boolean functions) defines the overall feasible subspace. The problem is then solved by finding the maximum (or minimum) of the goal function over the feasible subspace.

The second application is in computing the spectral coefficients of a Boolean function. The main purpose of spectral methods [47] is to transform Boolean functions from Boolean domain into spectral (integer) domain so that a number of useful properties can be more easily detected. When a Boolean function is represented in Boolean domain, the function value for each minterm precisely describes the behavior of the function at that point but says nothing about the behavior of the function for any other point. In contrast, spectral representation of a Boolean function gives information which is much more global in nature. For example, for function $f(x_0, \ldots, x_{n-1})$, the spectral coefficient of $\overline{x_0} \ldots \overline{x_{n-1}}$ corresponds to the number of onset points of $f(x_0 \ldots x_{n-1})$. Since EVBDDs can represent functions in both Boolean and spectral domains, we are able to use arithmetic operations in EVBDDs to efficiently carry out the spectral transformations.

The third application is in the representation of multiple output Boolean functions. Clearly, we can use the OBDD representation to solve integer problems through the binary encoding of integer variables. Similarly, we can also use the EVBDD representation to perform multiple output Boolean function operations through integer interpretation of the functions (e.g., a multiple output function $f_0, \ldots, f_{m-1}$ can be represented by an integer function $2^{m-1}f_0 + \ldots + 2^0 f_{m-1}$). We will present an EVBDD-based function decomposition algorithm as an example. When this algorithm is applied to an EVBDD representing a Boolean function, it performs single-output function decomposition; when it is applied to an EVBDD representing an integer function representing a multiple-output Boolean function, it performs multiple-output function decomposition.

1.1 Background

The following definitions describe the syntax and semantics of EVBDDs. More details can be found in [27, 30].

Definition 1.1 An EVBDD is a tuple $\langle c, \mathbf{f} \rangle$ where c is a constant value and $\mathbf{f}$ is a directed acyclic graph consisting of two types of nodes:

1. There is a single *terminal node* with value 0 (denoted by $\mathbf{0}$).

2. A *nonterminal node* $\mathbf{v}$ is a 4-tuple $\langle variable(\mathbf{v}), child_l(\mathbf{v}), child_r(\mathbf{v}), value \rangle$, where $variable(\mathbf{v})$ is a binary variable $x \in \{x_0, \ldots, x_{n-1}\}$.

An EVBDD is ordered if there exists an index function $index(x) \in \{0, \ldots, n-1\}$ such that for every nonterminal node $\mathbf{v}$, either $child_l(\mathbf{v})$ is a terminal node or $index(variable(\mathbf{v})) < index(variable(child_l(\mathbf{v})))$, and either $child_r(\mathbf{v})$ is a terminal node or $index(variable(\mathbf{v})) < index(variable(child_r(\mathbf{v})))$. If $\mathbf{v}$ is the terminal node $\mathbf{0}$, then $index(\mathbf{v}) = n$. An EVBDD is reduced if there is no nonterminal node $\mathbf{v}$ with $child_l(\mathbf{v}) = child_r(\mathbf{v})$ and $value = 0$, and there are no two nonterminal nodes $\mathbf{u}$ and $\mathbf{v}$ such that $\mathbf{u} = \mathbf{v}$.

Definition 1.2 An EVBDD $\langle c, \mathbf{f} \rangle$ denotes the arithmetic function $c+f$ where f is the function denoted by $\mathbf{f}$. $\mathbf{0}$ denotes the constant function 0, and $\langle x, \mathbf{l}, \mathbf{r}, v \rangle$ denotes the arithmetic function $x(v+l) + (1-x)r$.

In this paper, we consider only reduced, ordered EVBDDs. In the graphical representation of an EVBDD $\langle c, \mathbf{f} \rangle$, $\mathbf{f}$ is represented by a rooted, directed, acyclic graph and c by a dangling incoming edge to the root node of $\mathbf{f}$. The terminal node is depicted by a rectangular node labelled 0. A nonterminal node is a quadruple $\langle x, \mathbf{l}, \mathbf{r}, v \rangle$, where x is the node label, $\mathbf{l}$ and $\mathbf{r}$ are the two subgraphs rooted at x, and v is the label assigned to the left edge of x.

Example 1.1 Fig. 1 shows two arithmetic functions $f_0 = 3 - 4x + 4xy + xz - 2y + yz$ and $f_1 = 4x_0 + 2x_1 + x_2$ represented in EVBDDs. The second function is derived as follows:

$$
\begin{aligned}
f_1 &= 0 + f_{x_0} \\
f_{x_0} &= x_0(4 + f_{x_1}) + (1 - x_0)f_{x_1} = 4x_0 + 2x_1 + x_2, \\
f_{x_1} &= x_1(2 + f_{x_2}) + (1 - x_1)f_{x_2} = 2x_1 + x_2, \\
f_{x_2} &= x_2(1 + 0) + (1 - x_2)0 \qquad = x_2.
\end{aligned}
$$

□

Figure 1 goes here.

A generic (EVBDD) *apply* operator is described in [27]. This operator takes $\langle c_f, \mathbf{f} \rangle, \langle c_g, \mathbf{g} \rangle$ and *op* as arguments and returns $\langle c_h, \mathbf{h} \rangle$ such that $c_h + h \equiv (c_f + f)\ op\ (c_g + g)$ where *op* can be any operator which is closed over the integers.

EVBDD representation enjoys a distinct feature, called the *additive* property, which is not seen in the OBDD representation. For example, consider the following operation:

$$(c_f + f) - (c_g + g) = (c_f - c_g) + (f - g).$$

Because the values c_f and c_g can be separated from the functions f and g, the key for this entry in *comp_table* is $\langle\langle 0, \mathbf{f}\rangle, \langle 0, \mathbf{g}\rangle, -\rangle$. After the computation of $\langle\langle 0, \mathbf{f}\rangle, \langle 0, \mathbf{g}\rangle, -\rangle$ resulting in $\langle c_h, \mathbf{h}\rangle$, we then add $c_f - c_g$ to c_h to have the complete result of $\langle\langle c_f, \mathbf{f}\rangle, \langle c_g, \mathbf{g}\rangle, -\rangle$. Hence, every operation $\langle\langle c'_f, \mathbf{f}\rangle, \langle c'_g, \mathbf{g}\rangle, -\rangle$ can share the computation result of $\langle\langle 0, \mathbf{f}\rangle, \langle 0, \mathbf{g}\rangle, -\rangle$. This then will increase the hit ratio for caching the computation results.

Another property of the EVBDD representation, called the *bounding* property, is the following: when the maximum or minimum of a function exceeds a boundary value, then the result can be determined without further computation. As an example, the following pseudo code $leq0(\langle c_f, \mathbf{f}\rangle)$ performs operation $(c_f + f) \leq 0$:

$leq0(\langle c_f, \mathbf{f}\rangle)$ {
1 if $((c_f + max(\mathbf{f})) \leq 0)$ return$(\langle 1, \mathbf{0}\rangle)$;
2 if $((c_f + min(\mathbf{f})) > 0)$ return$(\langle 0, \mathbf{0}\rangle)$;
3 if $(comp_table_lookup(\langle c_f, \mathbf{f}\rangle, leq0, ans))$ return(ans);
4 $\langle c_{h_l}, \mathbf{h_l}\rangle = leq0(\langle c_f + value(\mathbf{f}), child_l(\mathbf{f})\rangle)$;
5 $\langle c_{h_r}, \mathbf{h_r}\rangle = leq0(\langle c_f, child_r(\mathbf{f})\rangle)$;
6 if $(\langle c_{h_l}, \mathbf{h_l}\rangle == \langle c_{h_r}, \mathbf{h_r}\rangle)$ return $(\langle c_{h_l}, \mathbf{h_l}\rangle)$;
7 $\mathbf{h} = find_or_add(variable(\mathbf{f}), \mathbf{h_l}, \mathbf{h_r}, c_{h_l} - c_{h_r})$;
8 $comp_table_insert(\langle c_f, \mathbf{f}\rangle, leq0, \langle c_{h_r}, \mathbf{h}\rangle)$;
9 return $(\langle c_{h_r}, \mathbf{h}\rangle)$; }

A *comp_table* storing previously computed results is used to achieve computation efficiency. The entries of *comp_table* are used in line 3 and stored in line 8. After the left and right children have been computed resulting in $\langle c_{h_l}, \mathbf{h_l}\rangle$ and $\langle c_{h_r}, \mathbf{h_r}\rangle$ (lines 4 and 5), if $\langle c_{h_l}, \mathbf{h_l}\rangle = \langle c_{h_r}, \mathbf{h_r}\rangle$, the algorithm returns $\langle c_{h_l}, \mathbf{h_l}\rangle$ to ensure that the case of $\langle x, \mathbf{k}, \mathbf{k}, 0\rangle$ will not occur; otherwise, it returns $\langle c_{h_r}, \langle var, \mathbf{h_l}, \mathbf{h_r}, c_{h_l} - c_{h_r}\rangle\rangle$ to preserve the property of right edge value being 0. There is another table (*uniq_table*) used for the uniqueness property of EVBDD nodes. Before *leq0* returns its result, it checks this table through operation *find_or_add* which either adds a new node to the table or returns the node found in the table.

To speed up the relational operators and the *minimize* (Sec. 2.3) operation, we include the minimum and maximum function values in each EVBDD node. For the sake of readability,

we also use the *flattened* EVBDD as defined below.

Definition 1.3 A *flattened* EVBDD is a directed acyclic graph consisting of two types of nodes. A *nonterminal* node $\mathbf{v}$ is represented by a 3-tuple $\langle variable(\mathbf{v}), child_l(\mathbf{v}), child_r(\mathbf{v})\rangle$ where $variable(\mathbf{v}) \in \{x_0, \ldots, x_{n-1}\}$. A *terminal* node $\mathbf{v}$ is associated with an integer v. *Reduced*, *ordered*, flattened EVBDDs are defined in the same way as OBDDs.

Example 1.2 The flattened EVBDD for the function in Fig. 1 (b) is shown in Fig. 2. □

Figure 2 goes here.

Multi-terminal binary decision diagram (MTBDD) which was recently proposed in [11] is the same as flattened EVBDD. In general, in functions where the number of distinct terminal values is large, MTBDD will require larger number of nodes than EVBDD. However, in functions where the number of distinct terminal values is small, MTBDD may require less storage space depending on the number of nodes in the corresponding graphs.

From Example 1.1 (Fig. 1(b)) and Example 1.2 (Fig. 2), we see that EVBDD requires $n+1$ nodes to represent $2^{n-1}x_0 + \ldots + 2^0 x_{n-1}$ while flattened EVBDD (or MTBDD) require $2^{n+1}-1$ to represent the same function. When there are only two different terminal nodes (e.g., 0 and 1), EVBDD, MTBDD, and OBDD are equivalent in terms of the number of nodes and the topology of the graph [32]. In this case, EVBDD will require more space to represent the the edge-values.

The worst case time complexity for performing operations on EVBDDs is the same as that for MTBDDs. However, due to the *additive*, *bounding*, and *domain-reducing* properties of EVBDDs, many operations on EVBDDs are much more efficient the corresponding ones on MTBDDs. Details can be found in [32].

The remainder of this paper is organized as follows. Three applications of EVBDDs: solving ILP problems, computing spectral coefficients, and performing multiple output Boolean function decomposition are presented in sections 2, 3, and 4, respectively. Conclusions are given in section 5.

2 Integer Linear Programming

Integer Linear Programming (ILP) is an NP-hard problem [18] that appears in many applications. Most of existing techniques for solving ILP such as branch and bound [33, 13, 38] and cutting plane methods [19] are based on the linear programming (LP) method. While they may sometimes solve hundreds of variables, they cannot guarantee to find an optimal solution for problems with more than , say, 50 variables. It is believed that an effective ILP solver should incorporate integer or combinatorial programming theory into the linear programming method [4].

Jeong et al. [25] describe an OBDD-based approach for solving the 0-1 programming problems. This approach does not, however, use OBDDs for integer related operations such as conversion from linear inequality form of constraints into Boolean functions and optimization of nonbinary goal functions. Consequently, the caching of computation results is limited to only Boolean operations (i.e., for constraint conjunction).

Our approach for solving the ILP is to combine benefits of the EVBDD data structure (in terms of subgraph sharing and caching of computation results) with the state-of-the-art ILP solving techniques. We have developed a minimization operator in EVBDD which computes the optimal solution to a given goal function subject to a constraint function. In addition, the construction and conjunction of constraints in terms of EVBDDs are carried out in a divide and conquer manner in order to manage the space complexity.

2.1 Background

An ILP problem can be formulated as follows:

$$
\begin{aligned}
\text{minimize} \quad & \sum_{i=1}^{n} c_i x_i, \\
\text{subject to} \quad & \sum_{i=1}^{n} a_{i,j} x_i \leq b_j, \ 1 \leq j \leq m, \\
& x_i \ \text{ integer.}
\end{aligned}
$$

The first equation is referred as the *goal function* and the second equation is referred as *constraint functions*. Throughout this section we will assume the problem to be solved is a *minimization* problem. A *maximization* problem can be converted to a minimization problem by changing the sign of coefficients in the goal function.

There are three classes of algorithms for solving ILP problems [45]. The first class is

known as the branch and bound method [33, 13, 38]. This method usually starts with an optimum continuous LP solution which forms the first *node* of a search tree. If the initial solution satisfies the integer constraints, it is the optimum solution and the procedure is terminated. Otherwise, we split on variable x (with value x^* from the initial solution) and create two new subproblems: one with the additional constraint $x \leq \lfloor x^* \rfloor$ and the other with the additional constraint $x \geq \lfloor x^* \rfloor + 1$. Each subproblem is then solved using the LP methods, e.g., the simplex method [14] or the interior point method [26]. A subproblem is pruned if there are no feasible solutions, the feasible solution is inferior to the best one found, or all variables satisfy the integer constraints. In the last case, the feasible solution becomes the new best solution. The problem is solved when all subproblems are processed. Most commercial programs use this approach [34].

The second class is known as the implicit enumeration technique which deals with 0-1 programming [2, 3, 44]. Initially, all variables are *free*. Then, a sequence of *partial solutions* is generated by successively *fixing* free variables, i.e., setting free variables to 0 or 1. A *completion* of a partial solution is a solution obtained by fixing all free variables in the partial solution. The algorithm ends when all partial solutions are completions or are discarded. The procedure proceeds similar to the branch and bound except that it solves a subproblem using the *logical tests* instead of the LP. A logical test is carried out by inserting values corresponding to a given (partial or complete) solution in the constraints. A complete solution is feasible if it satisfies all constraints. A partial solution is pruned if it cannot reach a feasible solution or could only produce an inferior feasible solution (compared to the current best solution). One advantage of this approach is that we can use *partial order* relations among variables to prune the solution space. For example, if it is established that $x \leq y$, then portions of the solution space which correspond to $x = 1$ and $y = 0$ can be immediately pruned [7, 22].

In the early days, these two methods were considered to be sharply different. The branch and bound method is based on solving a linear program at every node in the search space and uses a breadth first strategy. The implicit enumeration method is based on logical tests requiring only additions and comparisons and employs a depth first strategy. However, successively versions of both approaches have borrowed substantially from each other [3]. The two terms branch and bound and implicit enumeration are now used interchangeably.

The third class is known as the cutting-plane method [19]. Here, the integer variable constraint is initially dropped and an optimum continuous variable solution is obtained.

The solution is then used to chop off the solution space while ensuring that no feasible integer solutions are deleted. A new continuous solution is computed in the reduced solution space and the process is repeated until the continuous solution indeed becomes an integer solution. Due to the machine round-off error, only the first few cuts are effective in reducing the solution space [45].

2.2 A Model Algorithm

In this section, we first show a straightforward method to solve the ILP problem using EVBDDs. We then describe how to improve this method in this and the following sections.

Example 2.1 We illustrate how to solve the ILP problems using EVBDDs through a simple example. For the sake of readability, we use flattened EVBDDs.

The following is a 0-1 ILP problem:

$$\begin{array}{lll} \text{minimize} & 3x + 4y, & \\ \text{subject to} & 6x + 4y \leq 8, & (1) \\ & 3x - 2y \leq 1, & (2) \\ & x, y \in \{0, 1\}. & \end{array}$$

We first construct an EVBDD for the goal as shown in Fig. 3 (a). We then construct the constraints. The left hand side of constraint (1) represented by an EVBDD is shown in Fig. 3 (b). After the relational operator $\leq$ has been applied on constraint (1), the resulting EVBDD is shown in Fig. 3 (c). Similarly, EVBDDs for constraint (2) are shown in Fig. 3 (d) and (e). The conjunction of two constraints, Fig. 3 (c) and (e), results in the EVBDD in Fig. 3 (f) which represents the solution space of this problem. A feasible solution corresponds to a path from the root to 1.

We then *project* the constraint function c onto the goal function g such that for a given input assignment X, if $c(X) = 1$ (feasible) then $p(X) = g(X)$; otherwise $p(X) = infeasible_value$. For minimization problems, the $infeasible_value$ is any value which is greater than the maximum of g, and for maximization problems, the $infeasible_value$ is any value which is smaller than the minimum of g. In our example, we use 8 as the $infeasible_value$. Thus, in Fig. 3 (g), the two leftmost terminal values have been replaced by value 8. The last step in solving the above ILP problem is to find the minimum in Fig. 3 (g) which is 0. □

Figure 3 goes here.

The above approach has three problems:

1. Converting a constraint from inequality form to a Boolean function may require exponential number of nodes;
2. Even if all constraints can be constructed without using excessive amounts of memory, conjoining them altogether at once may create too big an EVBDD; and
3. The operator *projection* is useful when we want to find all optimal solutions. However, in many situations, we are interested in finding *any* optimal solution. Thus, full construction of the final EVBDD (e.g., Fig. 3 (g)) is unnecessary.

In the remainder of this section, we will show how to overcome the first two problems by divide and conquer methods. In the next section, we will present an operator *minimize* which combines the benefits of computation sharing and branch and bound techniques to compute any optimal solution.

In our ILP solver, called FGILP, every constraint is converted to the form $AX - b \leq 0$. Thus, we only need one operator $leq0$ to perform the conversion. $AX < b$ is converted to $AX - b + 1 \leq 0$ (since all coefficients are integer); $AX \geq b$ is converted to $-AX + b \leq 0$; and $AX = b$ is converted to two constraints $AX - b \leq 0$ and $-AX + b \leq 0$.

Initially, every constraint is an EVBDD representing the left hand side of an inequality (i.e., $AX - b$) which requires n nonterminal nodes for an n-variable function. FGILP provides users with an n_supp parameter such that only if a constraint has less than n_supp supporting (dependent) variables, then it will be converted to a Boolean function. FGILP allows users to set another parameter c_size to control the size of EVBDDs. Only if constraints, in Boolean function form, are smaller in size than this parameter, they will be conjoined.

The function performed by $leq0$ is the same as $LI_to_BDD(I)$ of [25] where I is some representation of $w_1x_1 + \ldots + w_nx_n \geq T$. Our EVBDD representation of a linear inequality is more efficient than their representation, because $leq0$ can cache computation results as in all EVBDD operations while LI_to_BDD cannot. This is very important because the efficiency of OBDD operations heavily depends on the extent by which this property is exploited. In [25], the authors suggest that it is not advisable to replace an equality by two inequalities because the cost of testing terminal cases (lines 1 and 2) are the same for equality and inequality

relations. Our experiments, however, show a completely different result. Performing two inequalities followed by one conjunction (all in terms of EVBDD operations) is much faster than carrying out one equality. We think that the difference is due to our computation caching capability.

Parameters *n_supp* and *c_size* provide two advantages. First, they provide FGILP with a space-time tradeoff capability. The more memory FGILP has, the faster it runs. Second, combined with the branch and bound technique, some subproblems may be pruned before the conversion to the Boolean functions or the conjunction of constraints are carried out.

When there is only one constraint and it is in Boolean form, then the problem is solved through *minimize*. Otherwise, the problem is divided into two subproblems and is solved recursively. Since both the goal and constraint functions are represented by EVBDDs. The new goal and constraint functions for the first subproblem are the left children of the root nodes of the current goal and constraints. Similarly, the new goal and constraint functions for the second subproblem are the right children of the root nodes of the current goal and constraints.

Our main algorithm, *ilp_min*, employs a branch and bound technique. In addition to goal and constraint functions, *n_supp*, and *c_size*, there are two parameters which are used as bounding condition: *Lower bound* is either given by the user or computed through linear relaxation or Lagrangian relaxation methods; *Upper bound* represents the best feasible solution found so far. The initial value of the upper bound is the maximum of the goal function plus 1.

If the maximum of goal function is less than the lower bound (LB) or the minimum of goal function is greater than or equal to the upper bound (UB), the problem is pruned. Furthermore, if there exists a constraint whose minimum feasible solution is greater than or equal to the current best solution (upper bound), then again the problem is pruned.

```
ilp_min(goal, constr, LB, UB, n_supp, c_size)
{
1     if (max(goal) < LB) return;
2     if (min(goal) ≥ UB) return;
3     if (∃c ∈ constr : minimize(goal, c, UB) == 0) return;
4     new_constr = conjunction_constr(constr, c_size);
5     if (new_constr has only one element) {
6           minimize(goal, new_constr, UB);
7     }
8     else {
9     ⟨⟨goal_l, new_constr_l⟩, ⟨goal_r, new_constr_r⟩⟩ =
            divide_problem(goal, new_constr, n_supp);
10    ilp_min(goal_l, new_constr_l, LB, UB, n_supp, c_size);
11    ilp_min(goal_r, new_constr_r, LB, UB, n_supp, c_size);
12    }
}
```

Example 2.2 We want to solve the following problem:

$$\begin{array}{ll} \text{minimize} & -4x + 5y + z + 2w, \\ \text{subject to} & 3x + 2y - 4z - w \leq 0, \\ & 2x + y + 3z - 4w \leq 0, \\ & x, y, z, w \in \{0, 1\}. \end{array}$$

Figure 4 goes here.

1. The initial goal and constraint EVBDDs are shown in Fig. 4 (a). Suppose both parameters *n_supp* and *c_size* are set to 4.
2. Since the number of supporting variables in the constraint EVBDDs is not less than 4, we divide the problem into two subproblems: one with $x = 1$ (Fig. 4 (b)) and the other with $x = 0$ (Fig. 4 (c)). The final solution is the minimum of solutions to these two subproblems.
3. Next, we want to solve the subproblem with $x = 1$. Since the number of supporting variables in constraint EVBDDs is smaller than *n_supp*, we convert the constraint EVBDDs into Boolean functions by carrying out operation *leq*0 (Fig. 4 (d)).

4. Since the size of constraint EVBDDs are not less than *c_size*, we divide the problem into two subproblems: one with $y = 1$ (Fig. 4 (e)) and the other with $y = 0$ (Fig. 4 (f)).
5. Now, we want to solve the subproblem with $y = 1$. Since the size of both constraint EVBDDs are less than *c_size*, we conjoin them together and then solve this subproblem using the *minimize* operator (Sec. 2.3).
6. The remaining subproblems are solved in the same way. Note that the solution found from a subproblem can be used as an upper bound for the subproblems which follow. □

2.3 The Operator *minimize*

This operator is another key distinction between our approach and the one in [25]. Operator *minimize* takes advantage of the additive and bounding properties of EVBDDs to achieve much more computation sharing and pruning of the search space. These properties lead to big savings in the memory requirement and run time of the ILP solver because of the way the computed table entries are stored and updated as explained below.

Operator *minimize* is similar to the *apply* operator with one additional parameter b. Given a goal function g, a constraint function c, and an upper bound b, *minimize* returns 1 if it finds a minimum feasible solution $v < b$ of g subject to c; otherwise, *minimize* returns 0. If v is found, b is replaced by v; otherwise, b is unchanged.

Note that when *minimize* returns 0, it does not imply that there are no feasible solutions with respect to g and c. This is because *minimize* only searches for feasible solutions that are smaller than b. Those feasible solutions which are greater than or equal to b are pruned because of the branch and bound procedure.

The parameter b serves two purposes: it increases the hit ratio for computation caching and is a bounding condition for pruning the problem space. To achieve the first goal, an entry of the computed table used by *minimize* has the form $\langle g, c, \langle b, v\rangle\rangle$ where v is set to the minimum of g which satisfies c and is less than b. If there is no feasible solution (with respect to g and c) which is less than b, then v is set to b.

The following pseudo code implements *minimize*. Lines 1-8 test for terminal conditions. In line 1, if the constraint function is the constant function 0, there is no feasible solution. In line 2, if the minimum of the goal function is greater than or equal to the current best solution, the whole process is pruned. If the goal function is a constant function, it must be

less than *bound*; otherwise, the test in line 2 would have been true. Thus, a new minimum is found in line 3. In line 6, if the constraint function is constant 1, then the minimum of the goal function is the new optimum. Again, this must be true, otherwise, the condition tested in line 2 would have been true.

Lines 9-17 perform the table lookup operation. If the lookup succeeds, no further computation is required; otherwise, we traverse down the graph in lines 19-26 in the same way as *apply*. Since *minimize* satisfies the additive property, we subtract c_g from *bound* to obtain a new local bound (*local_bound*) in line 9. c_g will be added back to *bound* in lines 13 or 32 if a new solution is found.

Suppose we want to compute the minimum of g subject to c with current local upper bound *local_bound*. We look up the computed table with key $\langle g, c \rangle$. If an entry $\langle g, c, \langle entry.bound, entry.value \rangle \rangle$ is found, then there are the following possibilities:

1. $entry.value < entry.bound$, i.e., a smaller value v was previously found with respect to g, c, and *entry.bound* (i.e., the minimization of g with respect to c has been solved and the result is *entry.value*).

   (a) If $entry.value < local_bound$, then *entry.value* is the solution we wanted.

   (b) Otherwise, the best we can find under g and c is *entry.value* which is inferior to *local_bound*, so we return with no success.

2. $entry.bound = entry.value$, i.e., there was no feasible solution with respect to g, c, and *entry.bound* (i.e., there is no stored result for the minimization of g with respect to c and *entry.bound*).

   (a) If $local_bound \leq entry.bound$, then we cannot possibly find a solution better than *entry.bound* for g under c. Therefore, we return with no success.

   (b) Otherwise, no conclusion can be drawn and further computation is required. Although there is no better feasible solution than *entry.bound*, it does not imply that there will be no better solution than *local_bound*.

In cases 1.b and 2.a pruning takes place (also computation caching), in case 1.a, computation caching is a success, while in case 2.b both operations fail. Note that there is no need for updating an entry (of the computed table) except in case 2.b.

In lines 25-30, the branch whose minimum value is smaller is traversed first since this increases chances for pruning the other branch. Finally, we update computed table and return the computed results in lines 31-39.

$minimize(\langle c_g, \mathbf{g} \rangle, \langle c_c, \mathbf{c} \rangle, bound)$

{

/ test for terminal conditions */*

1 if $(\langle c_c, \mathbf{c} \rangle == \langle 0, \mathbf{0} \rangle)$ return 0;

2 if $(min(\langle c_g, \mathbf{g} \rangle) \geq bound)$ return 0;

3 if $(\langle c_g, \mathbf{g} \rangle == \langle c_g, \mathbf{0} \rangle)$ {

4 $bound = c_g$;

5 return 1; }

6 if $(\langle c_c, \mathbf{c} \rangle == \langle 1, \mathbf{0} \rangle)$ {

7 $bound = min(\langle c_g, \mathbf{g} \rangle)$;

8 return 1; }

/ use the additive property */*

9 $local_bound = bound - c_g$;

/ look up the computed table */*

10 if $(comp_table_lookup(\langle 0, \mathbf{g} \rangle, \langle c_c, \mathbf{c} \rangle, entry))$ {

11 if $(entry.value < entry.bound)$ {

12 if $(entry.value < local_bound)$ {

13 $bound = entry.value + c_g$;

14 return 1; }

15 else return 0; }

16 else {

17 if $(local_bound \leq entry.bound)$ return 0; } }

18 $entry.bound = local_bound$;

/ create two subproblems by traversing down g and c */*

19 $\langle c_{g_l}, \mathbf{g_l} \rangle = \langle value(\mathbf{g}), child_l(\mathbf{g}) \rangle$;

20 $\langle c_{g_r}, \mathbf{g_r} \rangle = \langle 0, child_r(\mathbf{g}) \rangle$;

21 if $(index(variable(\mathbf{c})) \leq index(variable(\mathbf{g})))$ {

22 $\langle c_{c_l}, \mathbf{c_l} \rangle = \langle c_c + value(\mathbf{c}), child_l(\mathbf{c}) \rangle$;

23 $\langle c_{c_r}, \mathbf{c_r} \rangle = \langle c_c, child_r(\mathbf{c}) \rangle$; }

24 else { $\langle c_{c_l}, \mathbf{c_l} \rangle = \langle c_{c_r}, \mathbf{c_r} \rangle = \langle c_c, \mathbf{c} \rangle$; }

/ solve the subproblem with lower minimum first */*

25 if $(min(\mathbf{g_l}) \leq min(\mathbf{g_r}))$ {

26 $t_ret = minimize(\langle c_{g_l}, \mathbf{g_l} \rangle, \langle c_{c_l}, \mathbf{c_l} \rangle, local_bound)$;

27 $e_ret = minimize(\langle c_{g_r}, \mathbf{g_r} \rangle, \langle c_{c_r}, \mathbf{c_r} \rangle, local_bound)$; }

28 else {

29 $e_ret = minimize(\langle c_{g_r}, \mathbf{g_r} \rangle, \langle c_{c_r}, \mathbf{c_r} \rangle, local_bound)$;

30 $t_ret = minimize(\langle c_{g_l}, \mathbf{g_l} \rangle, \langle c_{c_l}, \mathbf{c_l} \rangle, local_bound)$; }

/ a new minimum is found */*

31 if $(t_ret \,||\, e_ret)$ {

32 $bound = local_bound + c_g$;

33 $entry.value = local_bound$;

34 $comp_table_insert(\langle 0, \mathbf{g} \rangle, \langle c_c, \mathbf{c} \rangle, entry)$;

35 return 1; }

/ no new minimum is found */*

36 else {

37 $entry.value = entry.bound$;

38 $comp_table_insert(\langle 0, \mathbf{g} \rangle, \langle c_c, \mathbf{c} \rangle, entry)$;

39 return 0; }

}

Example 2.3 We want to minimize the goal function $-4x + 5y + z + 2w$ subject to the constraint $(xz\bar{w} \vee \bar{x}yz\bar{w} \vee \bar{x}\bar{y}z \vee \bar{x}\bar{y}\bar{z}w = 1)$ shown in Fig. 5. For the sake of readability, the goal function is represented in EVBDD while the constraint function is represented in OBDD. The initial *upper bound* is $max(goal) + 1 = 0 + 5 + 1 + 2 + 1 = 9$. The reason for plus 1 is to recognize the case when there are no feasible solutions.

(a) We traverse down to nodes **a** and **b** through path $x = 1$ and $y = 1$. By subtracting the coefficients of x and y from *upper bound*, we have $9 - (-4) - 5 = 8$ which is the *local upper bound* with respect to nodes **a** and **b**. That is, we look for a minimum of **a** subject to **b** such that it is smaller than 8. It is easy to see that the best feasible solution of **a** subject to **b** is 1 which corresponds the assignments of $z = 1$ and $w = 0$. Thus, we insert $\langle \mathbf{a}, \mathbf{b}, \langle 8, 1\rangle\rangle$ as an entry into the computed table and recalculate the *upper bound* as $-4 + 5 + 1 + 0 = 2$.

(b) We traverse down to nodes **a** and **b** this time through path $x = 1$ and $y = 0$. The new local upper bound is $2 - (-4) - 0 = 6$, i.e., we look for a feasible solution which is smaller than 6. From computed table look up, we find that 1 is the best solution with respect to **a** and **b** and it is smaller than 6. Thus, the new upper bound is $-4 + 0 + 1 = -3$.

(c) We reach **a** and **b** through path $x = 0$ and $y = 1$. The local upper bound is $-3 - 0 - 5 = -8$. Again, from the computed table, we know 1 is the best solution which is larger than -8. Thus, no better solution can be found under **a** and **b** with respect to bound -8 and the current best solution remains -3.

(d) We reach nodes **a** and **c** through path $x = 0$ and $y = 0$. The local upper bound is $-3 - 0 - 0 = -3$. The minimum of the goal function **a** is 0 which is greater than -3.

The optimal solution is -3 with $x = 1, y = 0, z = 1$, and $w = 0$. □

Figure 5 goes here.

2.4 Discussion

A branch and bound/implicit enumeration based ILP solver can be characterized by the way it handles *search strategies*, *branching rules*, *bounding procedures* and *logical tests*. We will discuss these parameters in turn to analyze and explore possible improvements to FGILP.

Search Strategy

Search strategy refers to the selection of next node (subproblem) to process. There are two extreme search strategies. The first one is known as breadth first which always chooses nodes with best lower bound first. This approach tends to generate fewer nodes. The second one is depth first which chooses a best successor of the current node, if available, otherwise backtracks to the predecessor of the current node and continues the search. This strategy requires less storage space. FGILP uses the depth first strategy.

Branching Rule

This parameter refers to the selection of next variable to branch. Various selection criteria which have been proposed use *priorities* [37], *penalties* [15], *pseudo-cost* [5], and *integer infeasibility* [3] conditions. Currently, FGILP uses the same variable ordering as the one used to create EVBDDs because it simplifies the implementation. When the variable selected does not correspond to the variable ordering of EVBDD, operation *cofactor* (instead of $child_l$ and $child_r$) should be used.

Bounding Procedure

The most important component of a branch and bound method is the bounding procedure. The better the bound, the more pruning of the search space. The most frequently used bounding procedure is to use the linear programming method. Other procedures which can generate better bounds, but are more difficult to implement include the cutting planes, Lagrangian relaxation [43], and disjunctive programming [4]. The bounding procedure used in FGILP is similar to the one proposed in [2]. In our experience, the most pruning takes place at line 3 of the code for *ilp_min*. This pruning rule however has two weak points. First, it is carried out on each constraint one at a time. Thus, it is only a 'local' method. Second, it can only be applied to a constraint which is in the Boolean form. The other bounding procedures described above are 'global' methods which are directly applicable to the inequality form.

Logical Tests

It is believed that logical tests may be as important as the bounding procedure [39]. In addition to partial ordering of variables, a particularly useful class of tests, when available,

are those based on *dominance* [24, 25]. Currently, FGILP employs no logical tests. We believe that the inclusion of logical tests in FGILP will improve its performance.

Despite the fact that there are many improvements which can be made to FGILP, the performance of our ILP solver, as it is now, is already comparable to that of LINDO [42] which is one of the most widely used commercial tools [39] for solving ILP problems.

2.5 Experimental Results

FGILP has been implemented in C under the SIS environment. Table 1 shows our experimental results on ILP problems from MIPLIB [36]. It also shows the results of LINDO [42] (a commercial tool) on the same set of benchmarks. FGILP was run under SPARC station 2 (28.5 MIPS) with 64 MB memory while LINDO was run under SPARC station 10 (101.6 MIPS) with 384 MB memory. In Table 1, column 'Problem' lists the name of problems, columns 'Inputs' and 'Constraints' indicate the number of input variables and constraints, and columns 'FGILP' and 'LINDO' are the running time in seconds for obtaining the optimal solution shown in the last column.

FGILP provides three options for the order in which constraints are conjoined together. When all constraints are conjoined together, the order of conjunction will not affect the size of final EVBDD, but it does affect sizes of the intermediate EVBDDs. It is possible that an intermediate EVBDD has size much larger than the the final one. Our motivation for this ordering is to control the required memory space and save computation time. These three options are:

1. Based on the order of constraints in the input file. This provides users with direct control of the order.
2. EVBDDs with smallest size are conjoined first.
3. Constraints with the highest probability of not being satisfied are conjoined first.

The parameters used for the problems in Table 1 are summarized below:

1. Constraint conjunction order. Using the third option in problem 'p0201' led to much less space and computation time than the other two options. The same option led to more time in other problems due to the overhead of computing the probability of function values being 0. For consistency, results are reported for this option only.

2. EVBDD size of constraints. Without setting c_size, 'bm23' failed to finish and 'stein27' required 71.56 seconds. The run time reported in Table 1 for the above two problems were obtained by setting $c_size = 8000$ while others were run under no limitation of c_size. In general, this parameter has a significant impact on the run time. We believe that the correct value for c_size is dependent on the size of available memory for the machine.

3. Size of supporting variables. There was no limitation on the size of n_supp.

As results indicate, the performance of FGILP is comparable to that of LINDO. Since ILP is an NP-complete problem, it is quite normal that one solver outperforms the other solver in some problems while performs poorly in others.

Since SPARC station 10 is about 5X faster than SPARC station 2, FGILP runs faster than LINDO on the examples reported above. LINDO aborted on 'bm23' with the following message: "Fatal out-of-space in invert, Re-installing best solution ..., Search aborted for numerical reasons".

FGILP, however, requires much more space than LINDO. As technology improves, memory is expected to become cheaper in cost and smaller in size. Increasing the available memory size will improve the speed of FGILP while will not benefit LINDO as much.

The amount of space needed by FGILP is a function of not only the number of variables and constraints but also the structure of the constraint space in relation to the goal function. In some cases, FGILP handles problems with thousands of constraints, in other cases, it runs out of space in problems with a few hundred constraints.

Table 1 goes here.

3 Spectral Transformation

The main purpose of spectral methods [47] is to transform Boolean functions from Boolean domain into another domain so that the transformed functions have more compact implementations. It was conjectured that these methods would provide a unified approach to the

synthesis of analog and digital circuits [46]. Although spectral techniques have solid theoretical foundation, until recently they did not receive much attention due to their expensive computation times. With new applications in fault diagnosis, spectral techniques have recently invoked interest [23]. New computational methods have been proposed. In [17], a technique based on arrays of disjoint ON- and DC-cubes is proposed. In [46], a cube-based algorithm for linear decomposition in spectral domain is proposed.

Recently, [11] proposed two OBDD-based methods for computing spectral coefficients. The first method was to treat integers as bit vectors and integer operations as the corresponding Boolean operations. The main disadvantage of this representation is that arithmetic operations must be performed bit by bit which is very time consuming. The second method employed a variation of OBDD called Multi-Terminal Binary Decision Diagrams (MTBDDs) [10] which are exactly the same as the flattened form of EVBDDs. The major problem with using MTBDDs is the space requirement when the number of distinct coefficients is large.

We propose EVBDD-based algorithms for computing Hadamard (sometimes termed Walsh-Hadamard) spectrum [47]. In our approach, the matrix representing Boolean function values used in spectral methods is represented by EVBDDs. This takes advantage of compact representation through subgraph sharing. The transformation matrix and the transformation itself are carried out through EVBDD operations. Thus, the benefit of caching computational results is achieved.

In [11], the Walsh-Hadamard matrix and Boolean functions are represented by MTBDDs and the spectral transformation is carried out by matrix operations on MTBDDs. In our approach, no representation of this matrix is required as the transformation is carried out by addition and subtraction on EVBDDs. It remains to be seen that which implementation of the Walsh-Hadamard transformation is superior. As a footnote, if the number of distinct coefficients of a function is large, it is more advantageous to use the EVBDD implementation. However, due to the overhead of representing edge-values, for functions with small number of distinct coefficients, the MTBDD implementation may be better.

The reason we use the Hadamard transformation rather than other transformations (e.g., Walsh, Rademacher-Walsh, and Walsh-Paley [23]) is that the Hadamard transformation matrix has the recursive Kronecker product structure which perfectly matches the recursive structure of EVBDDs.

The algorithms presented here include both the transformation from Boolean domain to spectral domain and the operations within the spectral domain itself.

The Hadamard transformation is carried out in the following form:

$$T^n \, Z^n \;=\; R^n, \tag{1}$$

where T^n is a $2^n \times 2^n$ matrix called *transformation matrix*,

Z^n is a $2^n \times 1$ matrix which is the truth table representation ofa Boolean function,

R^n is a $2^n \times 1$ matrix which is the spectral coefficients of a Boolean function.

Different transformation matrices generate different spectra. Here, we use the Hadamard transformation matrix [47] which has a recursive structure as follows:

$$T^n \;=\; \begin{bmatrix} T^{n-1} & T^{n-1} \\ T^{n-1} & -T^{n-1} \end{bmatrix},$$

$$T^0 \;=\; 1.$$

Example 3.1 The spectrum of function $f(x, y) = x \oplus y$ is computed as

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

The *order* of each spectral coefficient r_i (i^{th} row of R^n) is the number of 1's in the binary representation of $i, 0 \leq i \leq 2^n - 1$. For example, r_{00} is the zeroth-order coefficient, r_{01} and r_{10} are the first-order coefficients, and r_{11} is the second-order coefficient. Let $R_i^n = \{$ multi-set of the absolute value of r_k's where r_k is an ith-order coefficient of $R^n\}$, $0 \leq i \leq n$. In Example 3.1, $R_0^2 = \{2\}$, $R_1^2 = \{0, 0\}$, and $R_2^2 = \{2\}$. An operation on f and its R^n which does not modify the sets R_i^n is referred as an *invariance* operation. Given a function $f(x_1, \ldots, x_n)$ with spectrum R^n, three invariance operations on f and R^n are as follows (formal proofs may be found in [16]):

1. Input negation invariance: if x_i is negated, then the new spectrum $R^{n'}$ is formed by $r'_k = -r_k$ where the ith bit of k is 1, and $r'_k = r_k$ otherwise.

2. Input permutation invariance: if input variables x_i and x_j are exchanged, then the new spectrum is formed by exchanging r_k's and r_l's where $k - 2^i = l - 2^j$. That is, the ith and j^{th} bits of k and l are $\langle 1, 0\rangle$ and $\langle 0, 1\rangle$, respectively, while all other bits of k and l are the same.

3. Output negation invariance: if f is negated, the $R^{n'}$ is formed by replacing all r_k by $-r_k$.

Lemma 3.1 Two Boolean functions are input-negation, input-permutation, and output-negation equivalent (NPN-equivalent) only if their R_i^n's are equivalent.

With this property, R_i^n can be used as a filter to improve performance in a Boolean matching algorithm as presented in [28].

3.1 Spectral EVBDD (SPBDD)

The major problem with Equation 1 is that all matrices involved are of size 2^n. Therefore, only functions with a small number of inputs can be computed. We overcome this difficulty by using EVBDDs to represent both Z^n and R^n. When an EVBDD is used to represent R^n, we refer to it as SPectral EVBDD, or SPBDD for short. The difference between EVBDDs and SPBDDs is in the semantics, not the syntax. A path in EVBDDs corresponds to a function value while a path in SPBDDs corresponds to a spectral coefficient. The matrix multiplication by T^n is implicitly carried out in the transformation from Z^n to R^n (i.e., from EVBDD to SPBDD).

We define Z^n and R^n recursively as follows:

$$Z^n = \begin{bmatrix} Z_0^{n-1} \\ Z_1^{n-1} \end{bmatrix},$$

$$R^n = \begin{bmatrix} R_0^{n-1} \\ R_1^{n-1} \end{bmatrix}.$$

Then, Equation 1 can be rewritten as:

$$\begin{bmatrix} T^{n-1}Z_0^{n-1} + T^{n-1}Z_1^{n-1} \\ T^{n-1}Z_0^{n-1} - T^{n-1}Z_1^{n-1} \end{bmatrix} = \begin{bmatrix} R_0^{n-1} \\ R_1^{n-1} \end{bmatrix}. \quad (2)$$

Equation 2 then is implemented through EVBDD [1] as :

$$\begin{aligned} \tau(\langle x, Z_1^{n-1}, Z_0^{n-1} \rangle) = & \quad (3) \\ & \langle x, \tau(Z_0^{n-1}) - \tau(Z_1^{n-1}), \tau(Z_0^{n-1}) + \tau(Z_1^{n-1}) \rangle, \\ \tau(1) = & \ 1, \\ \tau(0) = & \ 0. \end{aligned}$$

[1]For the sake of readability, we use flattened EVBDD in this section.

where τ is the transformation function which converts an EVBDD representing a Boolean function to an SPBDD. To show the above equations correctly implement Equation 2, we prove the following lemma.

Lemma 3.2 Let τ : EVBDD $\rightarrow$ SPBDD as defined in Equations 3-4, then τ implements T, that is, $\tau(f^n) = T^n f^n$, where f^n is an n-input function. Or, equivalently, $\tau(\langle x_n, Z_1^{n-1}, Z_0^{n-1}\rangle) = \langle x_n, R_1^{n-1}, R_0^{n-1}\rangle$.

Example 3.2 The exclusive-or function in Example 3.1 is redone in terms of EVBDD representation.
$\tau(\langle x, \langle y, 0, 1\rangle, \langle y, 1, 0\rangle\rangle) =$
$\langle x, \tau(\langle y, 1, 0\rangle) - \tau(\langle y, 0, 1\rangle), \tau(\langle y, 1, 0\rangle) + \tau(\langle y, 0, 1\rangle)\rangle =$
$\langle x, \langle y, -1, 1\rangle - \langle y, 1, 1\rangle, \langle y, -1, 1\rangle + \langle y, 1, 1\rangle\rangle =$
$\langle x, \langle y, -2, 0\rangle, \langle y, 0, 2\rangle\rangle$

Pseudo code *evbdd_to_spbdd*$(ev, level, n)$ is the implementation of Equation 2. Because of the following situation, this procedure requires $level$ and n as parameters:

$$\begin{aligned} \tau(\langle x, z, z\rangle) &= \langle x, \tau(z) - \tau(z), \tau(z) + \tau(z)\rangle, \\ &= \langle x, 0, 2 \times \tau(z)\rangle. \end{aligned}$$

In reduced EVBDD, $\langle x, z, z\rangle$ will be reduced to z while $\langle x, 0, 2 \times \tau(z)\rangle$ cannot be reduced in SPBDD. We need to keep track of the current $level$ so that when the index of the root node ev is greater than $level$, we generate $\langle level, 0, 2 \times \tau(ev)\rangle$ (lines 3-8).

```
evbdd_to_spbdd(ev, level, n)
{
1       if (level == n) return ev;
2       if (ev == 0) return 0;
3       if (index(ev) > level) {
4           sp = evbdd_to_spbdd(ev, level + 1, n);
5           left = 0;
6           right = evbdd_add(sp, sp)
7           return new_evbdd(level, left, right);
8       }
9       sp_l = evbdd_to_spbdd(child_l(ev), level + 1, n);
10      sp_r = evbdd_to_spbdd(child_r(ev), level + 1, n);
11      left = evbdd_sub(sp_r, sp_l);
12      right = evbdd_add(sp_r, sp_l);
13      return new_evbdd(level, left, right);
}
```

3.2 Boolean Operations in Spectral Domain

In this section, we show how to perform Boolean operations in SPBDDs. We first present the algorithm for performing Boolean conjunction in SPBDDs by the following definition.

Definition 3.1 Given two SPBDDs f and g, the operator $\wedge$ is carried out in the following way. If f and g are terminal nodes, then

$$f \wedge g = f \times g.$$

Otherwise,

$$\langle x, f_l, f_r \rangle \wedge \langle x, g_l, g_r \rangle = \langle x, (f_l \wedge g_r + f_r \wedge g_l)/2, (f_l \wedge g_l + f_r \wedge g_r)/2 \rangle.$$

The following lemma and theorem prove that the above definition carries out the Boolean conjunction in SPBDDs.

Lemma 3.3 $(f+g) \wedge (i+j) = f \wedge i + f \wedge j + g \wedge i + g \wedge j$, where $f, g, i, j \in$ SPBDD. (Note that $+$'s may be replaced by $-$'s.)

Theorem 3.1 Given two Boolean functions f and g represented in EVBDDs, $\tau(f \cdot g) = \tau(f) \wedge \tau(g)$, where $\cdot$ is the conjunction operator in Boolean domain.

Other Boolean operations in SPBDDs are carried out by the following equations:

$$f \vee g = f + g - f \wedge g, \tag{4}$$

$$f \oplus g = f + g - 2 \times (f \wedge g), \tag{5}$$

$$\bar{f} = J^n - f, \tag{6}$$

where = $\vee, \oplus$, and $^-$, are *or*, *xor*, and *not* in spectral domain (SPBDD), $J^n = [2^n, 0, \ldots, 0]^t$. These operations $\vee, \oplus$, and $^-$ are from [23] with minor modification to match the τ operation. Operations $+$, $-$, and $\times$ are carried out in the same way as in EVBDDs (e.g., *apply* in [27]).

3.3 Experimental Results

Table 2 shows the results of some benchmarks represented in both EVBDD and SPBDD forms. Column 'EVBDD' depicts the size and time required for representing and constructing a circuit using EVBDDs while column 'SPBDD' depicts the size of SPBDDs and the time required for converting from EVBDDs to SPBDDs. In average, the ratio of the number of nodes required for representing SPBDDs over that of EVBDDs is 6.8, and the ratio of the conversion time for SPBDDs over the construction time of EVBDDs is 41.

One application of spectral coefficients is that they can be used as a filter for pruning search space in the process of Boolean matching [11, 28]. The performance of a filter depends on its capability of pruning (effectiveness) and its computation time (cost). Experimental results of [11] show that this filter is quite good because this filter rejected all unmatchable functions that were encountered. However, according to results of Table 2, this filter is relatively expensive to compute when comparing with other filters [28]. We believe that this filter should be used only after other filters have failed to prune.

Table 2 goes here.

4 Function Decomposition

While a problem with finite domain can be solved by conversion to Boolean functions, a problem related to multiple-output Boolean functions can also be solved by interpreting them as the bit representation of an integer function. For example, a multiple-output Boolean function $\langle f_0, \ldots, f_{m-1} \rangle$ can be transformed to an integer function F by $F = 2^{m-1} f_0 + \ldots + 2^0 f_{m-1}$. Based on this formulation, we present the application of EVBDDs to performing function decomposition of multiple-output Boolean functions.

The motivation for using function decomposition in logic synthesis is to reduce the complexity of the problem by a divide-and-conquer paradigm: A function is decomposed into a set of smaller functions such that each of them is easier to synthesize.

The function decomposition theory was studied by Ashenhurst [1], Curtis [12], and Roth and Karp [40]. In Ashenhurst-Curtis method, functions are represented by Karnaugh maps

and the decomposability of functions are determined from the number of distinct columns in the map. In Roth-Karp method, functions are represented by cubes and the decomposability of functions are determined from the cardinality of compatible classes. Recently, researchers [6, 9, 29, 41] have used OBDDs to determine decomposability of functions. However, most of these works only consider single-output Boolean functions.

In this section, we start with definitions of function decomposition and cut_sets in EVBDD representation. Based on the concept of cut_sets, we develop an EVBDD-based disjunctive decomposition algorithm.

4.1 Definitions

Definition 4.1 A pseudo Boolean function $f(x_0, \ldots, x_{n-1})$ is said to be *decomposable* under *bound set* $\{x_0, \ldots, x_{i-1}\}$ and *free set* $\{x_i, \ldots x_{n-1}\}$, $0 < i < n$, if f can be transformed to $f'(g_0(x_0, \ldots, x_{i-1}), \ldots, g_j(x_0, \ldots, x_{i-1}), x_i, \ldots, x_{n-1})$ such that the number of inputs to f' is smaller than that of f. If j equals 1, then it is *simple decomposable*.

Note that since inputs to a pseudo Boolean function are Boolean variables, function g_k's are Boolean functions. Here, we consider only disjunctive decomposition (the intersection of bound set and free set is empty).

Definition 4.2 Given an EVBDD $\langle c, \mathbf{v} \rangle$ representing $f(x_0, \ldots, x_{n-1})$ with variable ordering $x_0 < \ldots < x_{n-1}$ and bound set $B = \{x_0, \ldots, x_i\}$, we define

$$cut_set(\langle c, \mathbf{v} \rangle, B) = \{\langle c', \mathbf{v}' \rangle \mid \langle c', \mathbf{v}' \rangle = eval(\langle c, \mathbf{v} \rangle, j), 0 \leq j < 2^i\}.$$

For the sake of readability, we use the flattened form of EVBDDs in this section.

Example 4.1 Given a function f as shown in Fig. 6 with bound set $B = \{x_0, x_1, x_2\}$, $cut_set(f, B) = \{\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}\}$. □

Figure 6 goes here.

If an EVBDD is used to represent a Boolean function, then each node in the cut_set corresponds to a distinct column in the Ashenhurst-Curtis method [1, 12] and a compatible class in the Roth-Karp decomposition algorithm [40].

4.2 Disjunctive Decomposition

[9, 29, 41] describe algorithms for disjunctive decomposition of single-output Boolean functions based on the OBDD representation of these functions. However the concept of communication links used in [9, 41] cannot be directly applied to multiple output functions. Here, we present an EVBDD-based disjunctive decomposition algorithm which is an extension of the algorithm presented in [29].

Algorithm D: Given a function f represented in an EVBDD $\mathbf{v_f}$ and a bound set B, a disjunctive decomposition with respect to B is carried out in the following steps:

1. Compute the cut_set with respect to B. Let $cut_set(\mathbf{v}, B) = \{\mathbf{u_0}, \ldots, \mathbf{u_{k-1}}\}$.
2. Encode each node in the cut_set by $\lceil \log_2 k \rceil = j$ bits.
3. Construct $\mathbf{v_m}$'s to represent g_m's, $0 \le m < j$:
   Replace each node $\mathbf{u}$ with encoding $b_0, \ldots, b_{j-1}$ in the cut_set by terminal node $\mathbf{b_m}$.
4. Construct $\mathbf{v_{f'}}$ to represent function f':
   Replace the top part of $\mathbf{v_f}$ by a new top on variables $g_0, \ldots, g_{j-1}$ such that $eval(\mathbf{v_{f'}}, l) = \mathbf{u_l}$ for $0 \le l < k-1$, $eval(\mathbf{v_{f'}}, l) = \mathbf{u_{k-1}}$ for $k-1 \le l < 2^j$.

The correctness of this algorithm can be intuitively argued as follows: For any input pattern m in the bound set, the evaluation of m in function f will result at a node in the cut_set with encoding e. The evaluation of m on the g_l functions should thus produce the function values e. The evaluation of e in function f' should also end at the same node in the cut_set. Thus, the composition of f' and g_l's becomes equivalent to f.

In step 2 of Algorithm D, we use an arbitrary input encoding which is not unique. Different encodings will result in different decompositions. Furthermore, when $k < 2^j$, not every j-bit pattern is used in the encoding of the cut_set; Function g_l's can never generate function values which correspond to the patterns absent from the encoding, thus we can assign these patterns to any node in the cut_set. In step 4, we assign them to the last node in the cut_set ($\mathbf{u_{k-1}}$). Alternatively, we could have made them into explicit don't-cares.

Lemma 4.1 Given an EVBDD $\mathbf{v_f}$ with variable ordering $x_0 < \ldots < x_{n-1}$ representing $f(x_0, \ldots, x_{n-1})$, a bound set $B = \{x_0, \ldots, x_{i-1}\}$ and $cut_set(\mathbf{v_f}, B) = \{\mathbf{u_0}, \ldots, \mathbf{u_{k-1}}\}$, if Algorithm D returns EVBDDs $\mathbf{v_{f'}}, \mathbf{v_{g_0}}, \ldots, \mathbf{v_{g_{j-1}}}$, then

$$f(x_0, \ldots, x_{n-1}) = f'(g_0(x_0, \ldots, x_{i-1}), \ldots, g_{j-1}(x_0, \ldots, x_{i-1}), x_i, \ldots, x_{n-1})$$

where $f', g_0, \ldots, g_{j-1}$ are the functions denoted by $\mathbf{v_{f'}}, \mathbf{v_{g_0}}, \ldots, \mathbf{v_{g_{j-1}}}$, respectively.

Example 4.2 Fig. 7 shows an example of disjunctive decomposition in EVBDDs. The evaluation of the input pattern $x_0 = 1$, $x_1 = 0$, and $x_2 = 1$ in function F will end at the leftmost x_4-node which has encoding 10. The evaluation of the same input pattern in functions g_0 and g_1 would produce function values 1 and 0. Then, with g_0 being 1 and g_1 being 0 in function F', it would also end at the leftmost x_4-node. □

Figure 7 goes here.

When an EVBDD is used to represent a Boolean function, Algorithm D corresponds to a disjunctive decomposition algorithm for Boolean functions; when an EVBDD represents an integer function, then Algorithm D can be used as a disjunctive decomposition algorithm for multiple-output Boolean functions as shown in the following example.

Example 4.3 A 3-output Boolean function as shown in Fig 8 (a) can be converted into an integer function as shown in Fig. 8 (b) through $F = 4f_0 + 2f_1 + f_2$. The application of Algorithm D on F is the one shown in the previous example. After decomposition, we can convert F' back to a 3-output Boolean function f'_0, f'_1, and f'_2. □

Figure 8 goes here.

4.3 Computing Cut_sets for All Possible Bound Sets

In the previous section, we showed how to perform function decomposition directly on EVBDDs when a bound set is given. In this section, we show how to compute the *cut_sets* for all bound sets of (single-output) Boolean functions.

Our method is based on the encoding of columns of the decomposition chart where free variables define the rows and bound variables define the columns [1, 12]. Decomposability is determined by the number of distinct columns (i.e., bit vectors). By encoding these bit

vectors as integers, we transform the problem to that of computing the cardinality of a set of integers.

Initially, every variable is in the free set. For each variable x_i, we perform the following two operations:

1. *include*: include x_i in the bound set to derive a new cut_set, and
2. *exclude*: partially encode the columns such that distinct columns are given unique codes and variable x_i is permanently excluded from the bound set.

Example 4.4 Fig. 9 (a) shows a decomposition chart where variable x is in the free set and a, b, c, d, e, f, and g are Boolean values. To perform the *include* operation, we move the bottom two rows to the left of the top two rows such that x now is in the bound set (Fig. 9 (b)). To perform the *exclude* operation, we encode bit vectors $\langle c, a\rangle$, $\langle d, b\rangle$, $\langle g, e\rangle$, and $\langle h, f\rangle$ as $2c + a$, $2d + b$, $2g + e$, and $2h + f$, respectively (Fig. 9 (c)). The coded decomposition chart preserves the distinctness of columns, that is, column $\langle a, b, c, d\rangle$ is distinct from column $\langle e, f, g, h\rangle$ if and only if column $\langle 2c + a, 2d + b\rangle$ is distinct from column $\langle 2g + e, 2h + f\rangle$. Furthermore, variable x is absent from the encoded decomposition chart and will never be included in the bound set. □

Figure 9 goes here.

Given an EVBDD $\mathbf{v}$ with the top variable x_i, the right and left children of $\mathbf{v}$ correspond to the top and bottom halves of rows in the decomposition chart. Thus, operations *include* and *exclude* in the EVBDD representation are carried out in the following way:

1. *include*: construct the set $\{child_l(\mathbf{v}), child_r(\mathbf{v})\}$, and
2. *exclude*: construct an EVBDD representing $2^{2^i} \times child_l(\mathbf{v}) + child_r(\mathbf{v})$ where 2^{2^i} is to ensure that the resulting EVBDD has a unique encoded representation.

Example 4.5 Fig. 10 (a) is the EVBDD representation of the decomposition chart in Fig. 9 (a). The corresponding operations *include* and *exclude* are shown in Fig. 10 (b) and (c), respectively. □

Figure 10 goes here.

Pseudo code *cut_set_all* computes the cardinality of the cut_set for every possible bound set of a given function. The routine returns the set $\{\langle b, k\rangle \mid b$ is a bound set and k is the cardinality of the cut_set of $b\}$. Initially, we have $i = 0$ and $node_set = \{\mathbf{v}\}$ where $\mathbf{v}$ is the EVBDD representing the given function. This corresponds to the bound set $B = \phi$ and free set X where X is the set of input variables. If $i = n$, then we reach the terminal case and *node_set* is the (encoded) cut_set for bound set B (line 1); otherwise, we perform *include* and *exclude* operations with respect to variable x_i (lines 2 and 3). We repeat the process for variable x_{i+1} in lines 4 and 5. In line 6, the union of $\langle b, k\rangle$'s from lines 4 and 5 is returned. Pseudo code *include* and *exclude* perform the include and exclude operations for a set of EVBDD nodes.

```
cut_set_all(node_set, B, i)
{
1    if (i == n) return({⟨B, | node_set |⟩});
2    inc_set = include(node_set, i);
3    exc_set = exclude(node_set, i);
4    inc = cut_set_all(inc_set, B ∪ {x_i}, i + 1);
5    exc = cut_set_all(exc_set, B, i + 1);
6    return(inc ∪ exc);
}
```

```
include(node_set, i)
{
1    new_set = φ;
2    for each node u ∈ node_set {
3        if (index(variable(u)) == i)
4            new_set = new_set ∪ {child_l(u), child_r(u)};
5        else /* index(u) > i */
6            new_set = new_set ∪ {u};
7    }
8    return new_set;
}
```

```
exclude(node_set, i)
{
1    new_set = φ;
2    for each node u ∈ node_set {
3        if (index(variable(u)) == i)
4            new_set = new_set ∪ {2^(2^i) × child_l(u) + child_r(u)};
5        else /* index(u) > i */
6            new_set = new_set ∪ {2^(2^i) × u + u};
7    }
8    return new_set;
}
```

Example 4.6 Fig. 11 (a) shows a function represented by both a truth table and a flattened EVBDD. Initially, the bound set is empty. The applications of *include* and *exclude* with respect to variable x_0 are shown in Fig. 11 (b) and (c), respectively. In Fig. 11 (b), the bound set is $\{x_0\}$ and the cardinality of the cut_set is 2; In Fig. 11 (c), the bound set is ϕ with cut_set size 1.

The application of *include* and *exclude* on Fig. 11 (b) with respect to variable x_1 results in Fig. 12 (a) and (b) with bound sets $\{x_0, x_1\}$ and $\{x_0\}$ and cut_set sizes 2 and 2, respectively. The application of *include* and *exclude* on Fig. 11 (c) with respect to variable x_1 results in Fig. 12 (c) and (d) with bound sets $\{x_1\}$ and ϕ and cut_set sizes 2 and 1, respectively.

The application of *include* and *exclude* on Fig. 12 (b) with respect to variable x_2 results in Fig. 13 (a) and (b) with bound sets $\{x_0, x_2\}$ and $\{x_0\}$ and encoded cut_sets $\{\mathbf{0}, \mathbf{1}, \mathbf{5}, \mathbf{4}\}$ and **5, 12**, respectively. In Fig. 13, the top row shows the encoded decomposition charts, the second row shows the encoded cut_sets, and the third row shows the decomposition charts. The encodings used for the bottom row in Fig. 13 (a) and (b) are $1 \times row1 + 4 \times row2$ and $1 \times row1 + 2 \times row2 + 4 \times row3 + 8 \times row4$, respectively. □

Figure 11 goes here.

Figure 12 goes here.

Figure 13 goes here.

Since there are 2^n different bound sets for an n variable function, the computation of the *cut_set* for every bound set is very expensive. If we replace line 1 in *cut_set_all* by

1 if $(level == n \mid\mid \mid var_set \mid \leq k)$ return($\{\langle B, \mid node_set \mid \rangle\}$),

then *cut_set_all* becomes a routine for computing the cardinality of the cut_set for every bound set whose size is less than or equal to k which is useful for the technology mapping of k-input look-up table field programmable gate arrays [31].

A naive way to compute the *cut_set* for every bound set is to move the bound variables to the top of the EVBDD. Compared to this approach, our approach has the following advantages. Firstly, it is well known that the size of OBDD (and EVBDD) is very sensitive to the variable ordering (at least in many practical applications [35]). Moving the bound variables to the top will change the variable ordering and hence may cause storage problems. Our method will not change the variable ordering. Secondly, the number of variables in the direct variable exchange approach is never reduced. In contrast, in our approach, after the *include* and *exclude* operations, the number of variables will be decreased by 1.

4.4 Experimental Results

In order to evaluate the effectiveness of *cut_set_all*, we compared our program with the Roth-Karp decomposition algorithm implemented in SIS. In particular, we used the following command on a number of mcnc91 benchmark sets:

"xl_k_decomp[2] -n 4 -e -d -f 100" which for every node in the Boolean network, finds the best bound set of size ≤ 4 that reduces the node's variable support after decomposition, and then decomposes the node, and modifies the network to reflect the change.

We provided equivalent EVBDD-based implementation of this SIS command.

To assign a unique encoding for each EVBDD node, we need integers with 2^i bits where i is the number of variables considered so far. This is clearly very expensive. One way to overcome this difficulty is to relax the uniqueness condition (e.g, use 2 instead of 2^{2^i}). Then, two different EVBDD nodes representing different functions may be assigned the same encoding. As a result, the size of the cut_set for a given bound set may be underestimated.

[2] *xl_k_decomp* does not process circuits with ≥ 32 inputs.

This scheme may be used as a filter. For example, to find the bound set which has the smallest cut_set, we first perform *cut_set_all* to find the best ones, and then check for the real cut_sets by moving the bound variables to the top of the EVBDD.

Results shown in Table 3 used 2 as the weight in *exclude* operation. We stopped the processes which took more that 5000 cpu seconds on a Sun Sparc-Station II with 64 MB of memory. We obtain significant speed-ups (by an average factor of 35.4).

Table 3 goes here.

5 Conclusions

Because of the compactness and canonical properties, OBDDs and EVBDDs have been shown effective for handling verification problems [8, 27]; because of the additive property, EVBDDs are also useful for solving integer linear programming problems (e.g., Sec. 2). Boolean values are a subdomain of integer values and Boolean operations are special cases of arithmetic operations. With this interpretation, EVBDDs are particularly useful for applications which require both Boolean and integer operations. Examples are shown in performing spectral transformations (e.g., Sec. 3) and representing multiple output Boolean functions (e.g., Sec. 4).

EVBDDs could be used for other applications. For example, [10] uses MTBDDs to represent general matrices and to perform matrix operation such as standard and Strassen matrix multiplication, and LU factorization; [20] uses OBDDs to implement a symbolic algorithm for maximum flow in 0-1 network. Equivalent EVBDD-based algorithms can be developed to solve these problems.

References

[1] R. L. Ashenhurst, "The decomposition of switching functions," *Ann. Computation Lab.*, Harvard University, vol. 29, pp. 74-116, 1959.

[2] E. Balas, "An additive algorithm for solving linear programs with zero-one variables," *Operations Research*, 13 (4), pp. 517-546, 1965.

[3] E. Balas, "Bivalent programming by implicit enumeration," *Encyclopedia of Computer Science and Technology* Vol.2, J. Belzer, A.G. Holzman and Kent, eds., M. Dekker, New York, pp. 479-494, 1975.

[4] E. Balas, "Disjunctive programming," *Annals of Discrete Mathematics 5*, North-Holland, pp. 3-51, 1979.

[5] M. Benichou, J. M. Gauthier, P. Girodet, G. Hentges, G. Ribiere and O. Vincent, "Experiments in mixed-integer linear programming," *Math. Programming 1*, pp. 76-94, 1971.

[6] M. Beardslee, B. Lin, and A. Sangiovanni-Vincentelli, "Communication based logic partitioning," *Proc. of the European Design Automation Conf.*, pp. 32-37, 1992.

[7] V. J. Bowman, Jr. and J. H. Starr, "Partial orderings in implicit enumeration," *Annals of Discrete Mathematics*, 1, North-Holland, pp. 99-116, 1977.

[8] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Transactions on Computers*, C-35(8): 677-691, August 1986.

[9] S-C. Chang and M. Marek-Sadowska, "Technology mapping via transformations of function graph," *Proc. International Conf. on Computer Design*, pp. 159-162, 1992.

[10] E. M. Clarke, M. Fujita, P. C. McGeer, K. L. McMillan, and J. C.-Y. Yang, "Multi-terminal binary decision diagrams: An efficient data structure for matrix representation," *International Workshop on Logic Synthesis*, pp. 6a:1-15, May 1993.

[11] E. M. Clarke, K. L. McMillan, X. Zhao, M. Fujita, and J. C.-Y. Yang, "Spectral transforms for large Boolean functions with applications to technology mapping," *Proc. of the 30th Design Automation Conference*, pp. 54-60, 1993.

[12] H. A. Curtis, "A new approach to the design of switching circuits," Princeton, N.J., Van Nostrand, 1962.

[13] R. J. Dakin, "A tree-search algorithm for mixed integer programming problems," *Comput. J. 9*, pp. 250-255, 1965.

[14] G. B. Dantzig, *Linear Programming and Extensions*, Princeton, N. J.: Princeton University Press, 1963.

[15] N. J. Driebeck, "An algorithm of the solution of mixed integer programming problems," *Management Science 12*, pp. 576-587, 1966.

[16] C. R. Edwards, "The application of the Rademacher-Walsh transform to Boolean function classification and threshold-logic synthesis," *IEEE Transactions on Computers*, C-24, pp. 48-62, 1975.

[17] B. J. Falkowski, I. Schafer and M. A. Perkowski, "Effective computer methods for the calculation of Rademacher-Walsh spectrum for completely and incompletely specified Boolean functions," *IEEE Transaction on Computer-Aided Design*, Vol. 11, No. 10, pp. 1207-1226, Oct. 1992.

[18] M. R. Garey and D. S. Johnson, "Computers and Intractability: A Guide to the Theory of NP-Completeness," Freeman, San Francisco, 1979.

[19] R. E. Gomory, "Solving linear programming problems in integers," in *Combinatorial Analysis*, R.E. Bellman and M. Hall, Jr., eds., American Mathematical Society, pp. 211-216, 1960.

[20] G. D. Hachtel and F. Somenzi, "A symbolic algorithm for maximum flow in 0-1 networks," *International Workshop on Logic Synthesis*, pp. 6b:1-6, May 1993.

[21] P. L. Hammer and S. Rudeanu, *Boolean Methods in Operations Research and Related Areas*, Heidelberg, Springer Verlag, 1968.

[22] P. L. Hammer and B. Simeone, "Order relations of variables in 0-1 programming," *Annals of Discrete Mathematics*, 31, North-Holland, pp. 83-112, 1987.

[23] S. L. Hurst, D. M. Miller and J. C. Muzio, *Spectral Techniques in Digital Logic,* London, U.K. : Academic, 1985.

[24] T. Ibaraki, "The power of dominance relations in branch and bound algorithm," *J. Assoc. Comput. Mach. 24*, pp. 264-279, 1977.

[25] S-W. Jeong and F. Somenzi, "A new algorithm for 0-1 programming based on binary decision diagrams," Logic Synthesis Workshop, in Japan, pp. 177-184, 1992.

[26] N. Karmarkar, "A new polynomial-time algorithm for linear programming," *Combinatorica*, 4:373-395, 1984.

[27] Y-T. Lai and S. Sastry, "Edge-Valued binary decision diagrams for multi-level hierarchical verification," *Proc. of 29th Design Automation Conf.*, pp. 608-613, 1992.

[28] Y-T. Lai, S. Sastry and M. Pedram, "Boolean matching using binary decision diagrams with applications to logic synthesis and verification," *Proc. International Conf. on Computer Design*, pp.452-458, 1992.

[29] Y-T. Lai, M. Pedram and S. Sastry, "BDD based decomposition of logic functions with application to FPGA synthesis," *Proc. of 30th Design Automation Conf*, pp. 642-647, 1993.

[30] Y-T. Lai, M. Pedram and S. Vrudhula, "Edge-Valued Binary-Decision Diagrams: Theory and Applications," *Technical Report 93-31*, University of Southern California, Aug. 1993.

[31] Y-T. Lai, K-R. Pan, M. Pedram, and S. Vrudhula, "FGMap: A Technology Mapping Algorithm for Look-Up Table Type FPGAs based on Function Graphs", *International Workshop on Logic Synthesis*, 1993.

[32] Y-T. Lai, "Logic Verification and Synthesis using Function Graphs," PhD Dissertation, Computer Engineering, Univ. of Southern Calif., December 1993.

[33] A. H. Land and A. G. Doig, "An automatic method for solving discrete programming problems," *Econometrica* 28, pp. 497-520, 1960.

[34] A. Land and S. Powell, "Computer codes for problems of integer programming," *Annals of Discrete Mathematics* 5, North-Holland, pp. 221-269, 1979.

[35] H-T. Liaw and C-S Lin, "On the OBDD-representation of general Boolean functions," *IEEE Trans. on Computers*, C-41(6): 661-664, June 1992.

[36] Department of Mathematical Sciences, Rice University, Houston, TX 77251.

[37] G. Mitra, "Investigation of some branch and bound strategies for the solution of mixed integer linear programs," *Math. Programming 4*, pp. 155-170, 1973.

[38] L. G. Mitten, "Branch-and-bound methods: General formulation and properties," *Operations Research*, 18, pp. 24-34, 1970.

[39] G. L. Nemhauser and L. A. Wolsey, *Integer and Combinatorial Optimization*, Wiley, New York, 1988.

[40] J. P. Roth and R. M. Karp, "Minimization over Boolean graphs," *IBM Journal*, pp. 227-238, April 1962.

[41] T. Sasao, "FPGA design by generalized functional decomposition," in *Logic Synthesis and Optimization*, Sasao ed., Kluwer Academic Publisher, pp. 233-258, 1993.

[42] L. Schrage, Linear, Integer and Quadratic Programming with LINDO, Scientific Press, 1986.

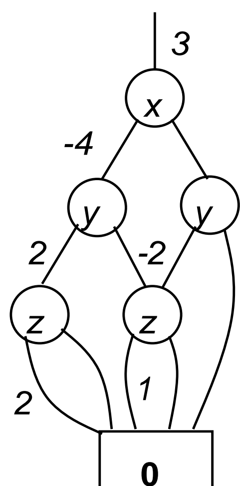
[43] J. F. Shapiro, "A survey of lagrangian techniques for discrete optimization," *Annals of Discrete Mathematics 5*, North-Holland, pp. 113-138, 1979.

[44] K. Spielberg, "Enumerative methods in integer programming," *Annals of Discrete Mathematics 5*, North-Holland, pp. 139-183, 1979.

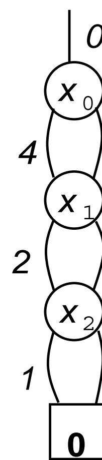
[45] H. A. Taha, "Integer programming," in *Mathematical Programming for Operations Researchers and Computer Scientists*, ed. A.G. Holzman, Marcel Derrer, pp.41-69, 1981.

[46] D. Varma and E. A. Trachtenberg, "Design automation tools for efficient implementation of logic functions by decomposition," *IEEE Transactions of Computer-Aided Design*, Vol. 8, No. 8, Aug. 1989, pp. 901-916.

[47] J. S. Wallis, "Hadamard matrices," *Lecture Notes No. 292,* Springer-Verlag, New York, 1972.



(a)



(b)

Figure 1: Two examples.

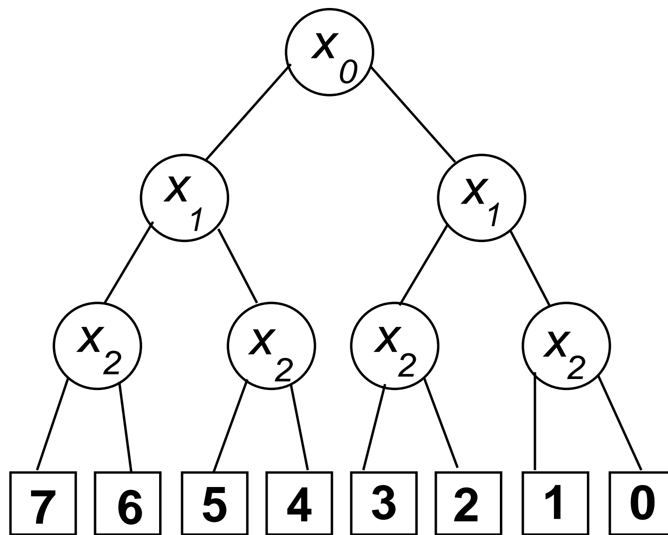


Figure 2: An example of flattened EVBDD.

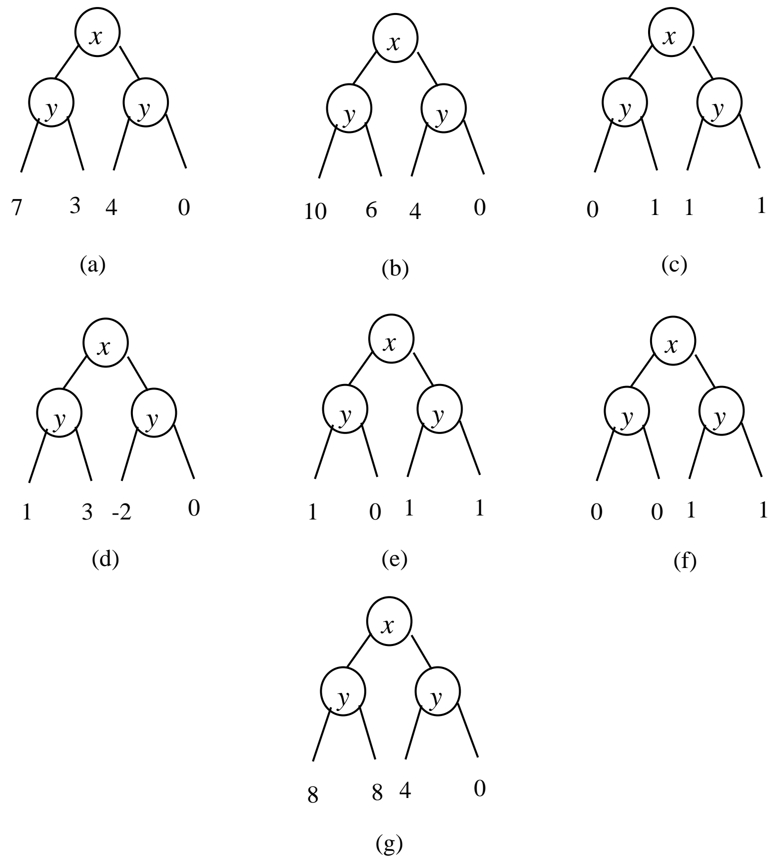


Figure 3: A simple example (using flattened EVBDDs and OBDDs).

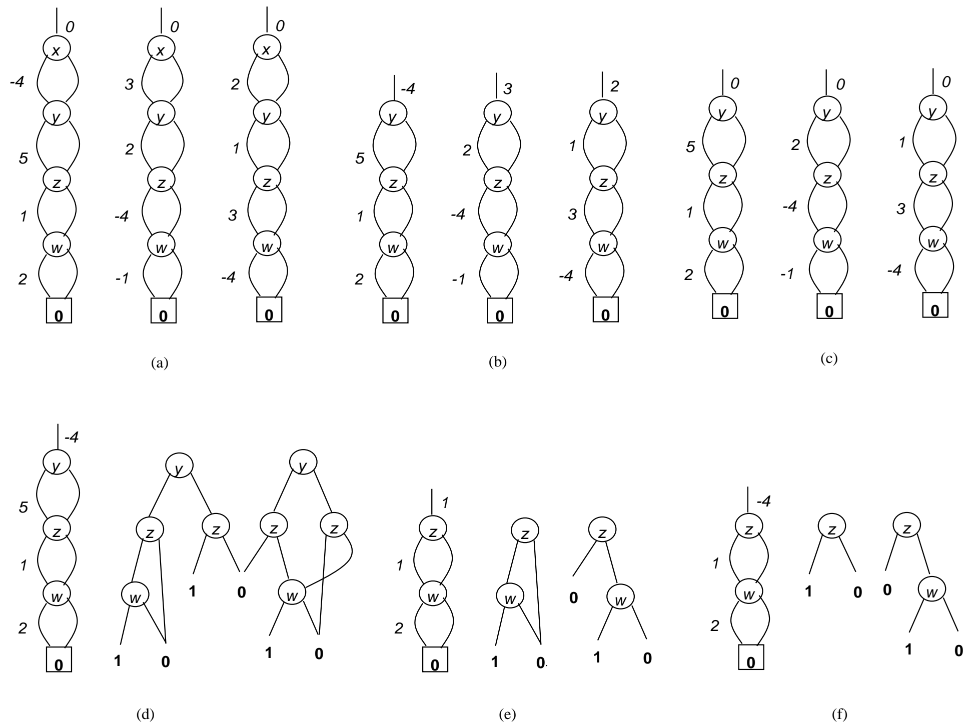


Figure 4: An example for conjoining constraints.

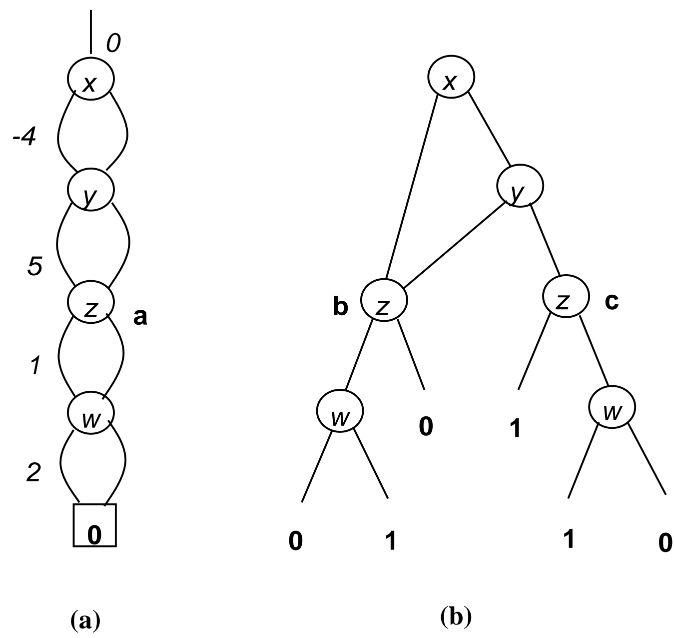


Figure 5: An example for the *minimize* operator.

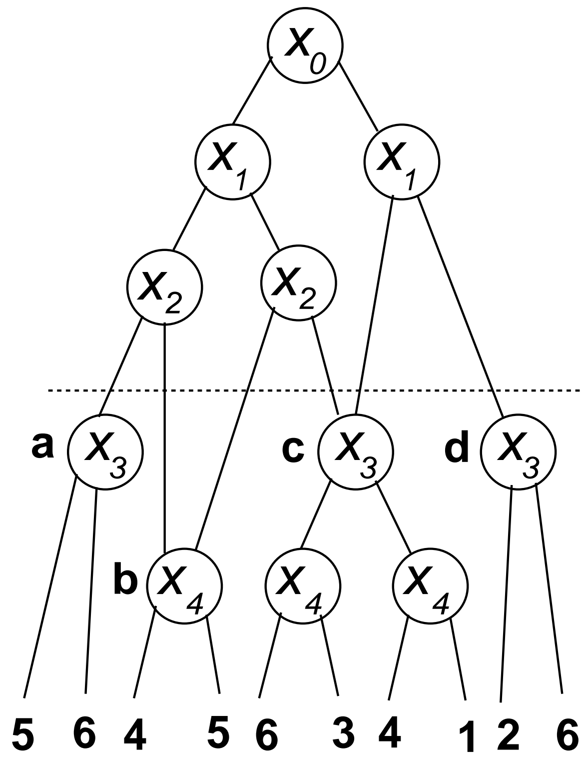


Figure 6: An example of cut_set in EVBDD.

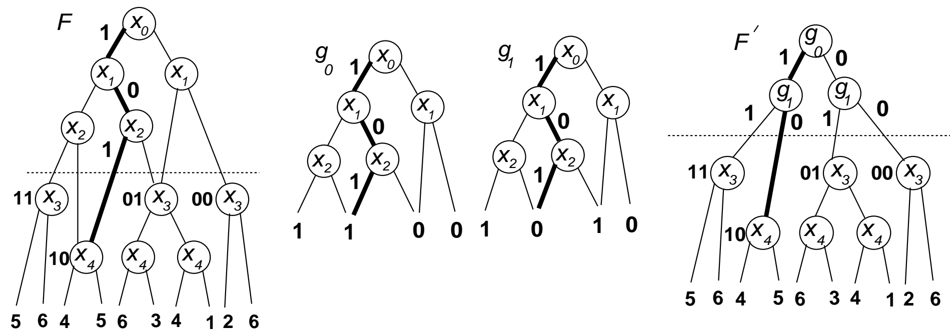


Figure 7: An example of disjunctive decomposition in EVBDD.

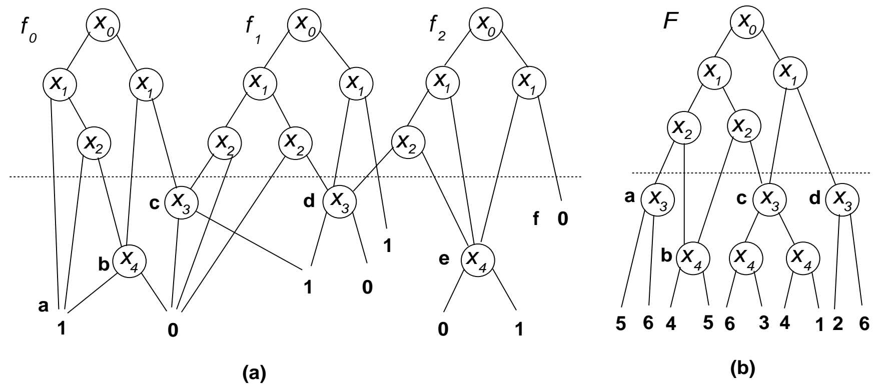


Figure 8: Representation of multiple-output functions.

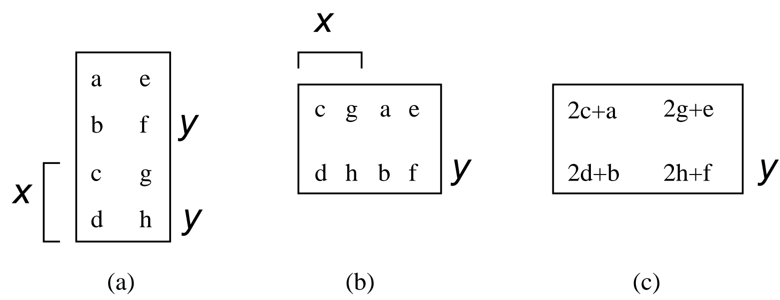


Figure 9: Operations *include* and *exclude* in the decomposition chart.

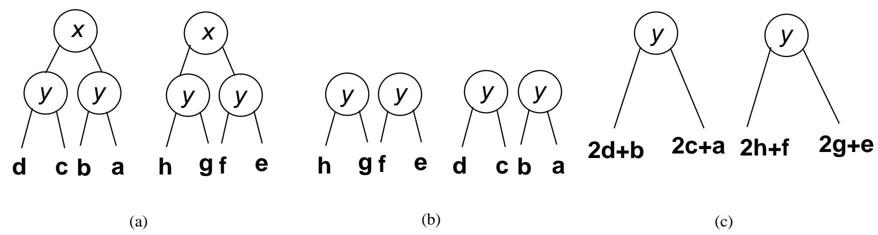


Figure 10: Operations *include* and *exclude* in the EVBDD representation.

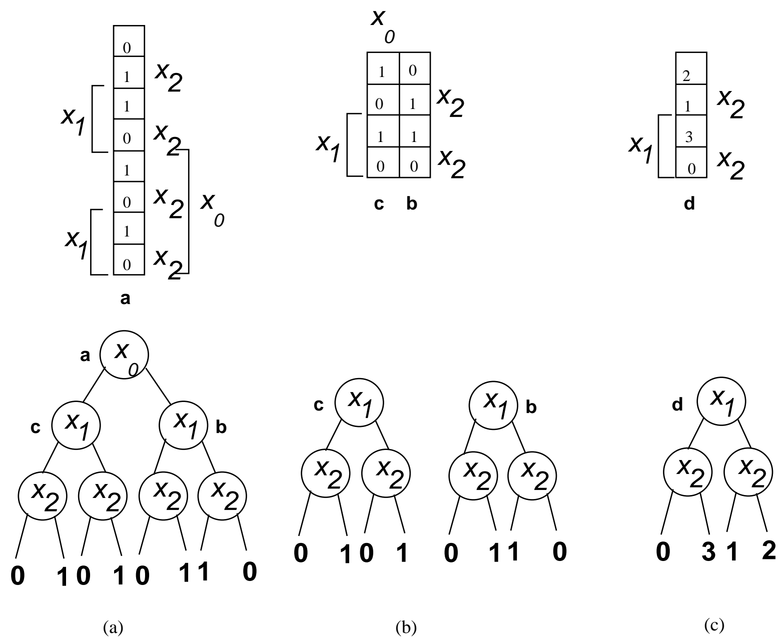


Figure 11: An example of the application of *cut_set_all*.

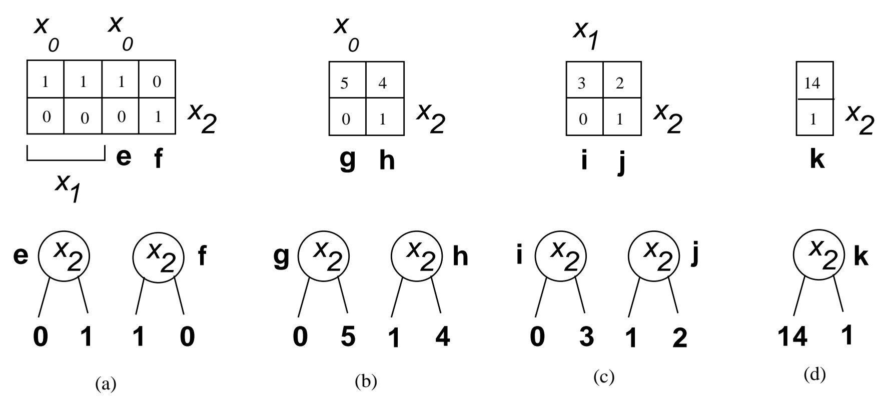


Figure 12: Example continued.

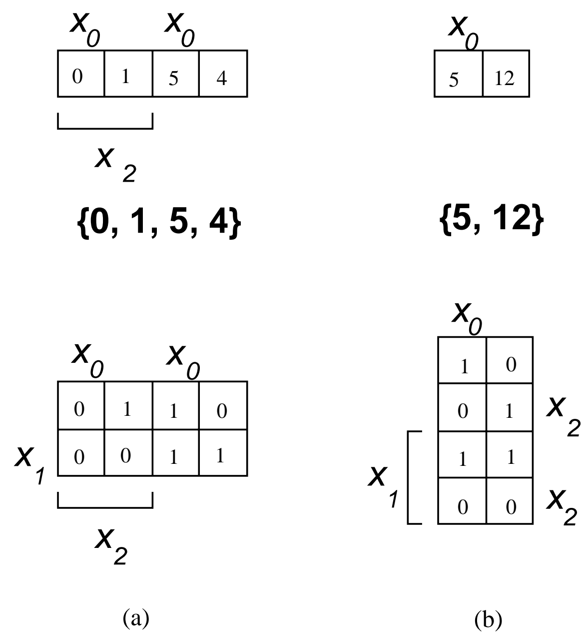


Figure 13: Example continued.

Problem	Inputs	Constraints	FGILP (sec)	LINDO (sec)	Optimal
bm23	27	20	1509.07	Error	34
lseu	89	28	Unable	186.44	1120
p0033	33	16	2.91	4.31	3089
p0040	40	23	0.98	0.37	62027
p0201	201	133	765.48	529.46	7615
stein15	15	36	1.44	1.66	9
stein27	27	118	51.24	120.03	18
stein9	9	13	0.13	0.31	5

Table 1: Experimental results of ILP problems.

	In	Out	EVBDD		SPBDD	
			Size	Time	Size	Time
9symml	9	1	24	0.17	39	0.15
c8	28	18	142	0.09	1310	4.79
cc	21	20	76	0.02	951	1.59
cmb	16	4	35	0.04	88	0.17
comp	32	3	145	0.10	809	35.31
cordic	23	2	84	0.09	208	1.35
count	35	16	233	0.14	1197	8.39
cu	14	11	66	0.04	266	0.67
f51m	8	8	65	0.08	295	0.34
lal	26	19	99	0.07	1111	2.99
mux	21	1	86	0.11	144	0.99
my-adder	33	17	456	0.62	5043	14.1
parity	16	1	16	0.02	30	0.14
pcle	19	9	94	0.03	640	1.18
pcler8	27	17	139	0.05	1617	4.53
pm1	16	13	57	0.01	465	0.74
sct	19	15	101	0.07	1295	2.42
ttt2	24	21	173	0.23	2046	5.35
unreg	36	16	134	0.04	816	5.78
x2	10	7	41	0.01	305	0.42
z4ml	7	4	36	0.09	83	0.13
b9	41	21	212	0.10	1809	12.08
alu2	10	6	248	0.69	889	2.69
alu4	14	8	1166	3.78	5913	66.12
term1	34	10	614	1.39	5643	104.35
apex7	49	37	665	0.35	3017	52.51
cht	47	36	133	0.08	452	5.19
example2	85	66	752	0.22	5163	23.14

Table 2: Experimental results of SPBDDs.

Circuit	karp	EVBDD	Speed-up
5xp1	31.1	2.9	10.7
9sym	513.6	3.7	138.8
apex4	2544.8	49.8	51.1
misex3	> 5000	381.9	13.1
misex3c	> 5000	131.8	37.9
Z5xp1	31.4	4.3	7.3
misex2	416.8	87.7	4.8
sao2	337.9	23.6	14.3
xor5	2.0	0.3	6.7
b12	74.3	6.4	11.6
ex1010	> 5000	48.5	> 103.1
squar5	4.3	0.9	4.8
Z9sym	508.2	4.2	121.0
t481	> 5000	62.1	> 80.5
alu4	> 5000	124.5	> 40.2
table3	> 5000	318.6	> 15.7
table5	> 5000	1225.8	> 4.1
cordic	> 5000	1331.7	> 3.8
vg2	> 5000	2051.3	> 2.4
seq	NA	> 5000	—
apex1	NA	> 5000	—
apex2	NA	> 5000	—
apex3	NA	> 5000	—
e64	NA	> 5000	—
Average			35.4

Table 3: Finding all decomposable forms with bound set size ≤ 4.