FGILP: An Integer Linear Program Solver based on Function Graphs

Yung-Te Lai, Massoud Pedram and Sarma Vrudhula

CENG Technical Report 93-11

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213)740-4458

April 1993

FGILP: An Integer Linear Program Solver based on Function Graphs

Yung-Te Lai, Massoud Pedram

Dept. of EE-Systems
University of Southern California
Los Angeles, CA 90089

Sarma B.K. Vrudhula
(a.k.a. Sarma Sastry)
Dept. of ECE
University of Arizona
Tuscon, AZ 85721

April 12, 1993

Contents

List of Figures

List of Tables

FGILP: An Integer Linear Program Solver based on Function Graphs

Abstract

Edge-Valued Binary Decision Diagrams (EVBDD)s extend BDDs from Boolean domain to integer domain and from Boolean operations to arithmetic operations. They have been used to perform logic verification and compute decomposability of Boolean functions. In this paper, we present a new EVBDD application for solving *Integer Linear Programs* (ILP), which is an NP hard problem that appears in many CAD problems. Our approach is to combine the benefits of EVBDD data structure (in terms of subgraph sharing and caching of computational results) with the state-of-the-art ILP solving techniques. Our program, called FGILP, has been implemented in C under the SIS environment. The preliminary results of FGILP are comparable to those of LINDO.

1 Introduction

Integer Linear Programming (ILP) is an NP-hard problem [14] that appears in many CAD applications. Most of existing techniques for solving ILP such as branch and bound [23, 10, 2, 26] and cutting plane methods [15, 16] 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 40 variables. It is believed that an effective ILP solver should incorporate integer or combinatorial programming theory into the linear programming method [4]. Several authors have proposed *logical test* based methods for 0-1 programming [7, 17, 18] for 0-1 programming.

Binary Decision Diagram (BDD) [9] is an effective data structure for representing Boolean functions and performing Boolean operations. By representing an integer as a set of binary variables, a linear inequality constraint function can be converted to a Boolean function. Similarly, a set of constraint functions can be represented by the conjunction of these Boolean functions.

Reference [19] reports on a BDD-based approach for solving the 0-1 programming problems. Because BDDs can only represent binary functions, for integer related operations such as conversion from linear inequality form of constraints into Boolean functions and optimization of nonbinary goal functions, BDDs are not directly applicable. This shortcoming limits the caching of computation results to only Boolean operations (i.e., for constraints conjunction only).

Edge-Valued Binary Decision Diagram (EVBDD) is an integer version of Bryant's Binary Decision Diagrams (BDDs) [9]. EVBDDs not only preserve the canonical function and compact representation properties of BDDs but also provide a new set of operators – arithmetic operators, relational operators and minimum/maximum operators. EVBDDs have been used for logic verification [20] and Boolean function decomposition [22]. In this paper, we present FGILP, an ILP solver based on EVBDDs.

Our approach for solving the ILP is to combine benefits of EVBDD data structure (in terms of subgraph sharing and caching of computational 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.

1.1 Review of ILP Solving Techniques

An ILP problem can be formulated as follows:

$$\text{minimize} \quad \sum_{i=1}^{n} c_i x_i \tag{1}$$

$$\text{subject to} \quad \sum_{i=1}^{n} a_{i,j} x_i \leq b_j, \ 1 \leq j \leq m \tag{2}$$

$$x_i \ \text{integer} \tag{3}$$

The first equation is referred as the *goal function* and the second equation is referred as *constraint functions.* Although the goal function can also be *maximize*, throughout this paper we will assume the problem to be solved is a *minimization* problem.

There are three classes of algorithms for solving ILP problems [31]. The first class is known as the branch-and-bound method [23, 10, 2, 26]. 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 method. 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 codes use this approach [24].

The second method is the implicit enumeration technique which deals with 0-1 programming [1, 3, 30]. 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, 17].

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, recent versions of both approaches have borrowed substantially from each other. The two terms branch and bound and implicit enumeration are now used interchangeablely.

The third method is the cutting-plane technique [15, 16]. In this method, 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 for cutting the solution space [31].

1.2 Overview

Our algorithm for solving the ILP employs a branch and bound technique. Initially, every constraint function is represented by an EVBDD in linear inequality form. The conversion of a constraint to a Boolean form and the conjunction of constraints are carried out in a divide and conquer fashion. When there is only one constraint which is in the Boolean form, we use an EVBDD operator *minimize* to solve the optimization problem. Otherwise, the problem is divided into two subproblems and is solved recursively. *minimize* employs the branch and bound and computation caching techniques.

The paper is organized as follows. In Section 2, we give basic definitions and operations of EVBDDs. In Section 3, we demonstrate how EVBDDs can be used to solve ILP problems through a simple example. The construction of constraints into the EVBDD form and conjunction of constraints are discussed in Section 4. A novel EVBDD based optimization operator and our basic algorithm are presented in Sections 5 and 6, respectively. The experimental results of FGILP and an analysis of our algorithm are given in Sections 7 and 8.

2 Edge-Valued Binary Decision Diagrams

This section gives an overview of EVBDDs.

Definition 2.1 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* $\mathbf{0}$.
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 $index(variable(\mathbf{v})) < index(variable(child_r(\mathbf{v})))$. An EVBDD is reduced if there is no nonterminal node $\mathbf{v}$ such that $child_l(\mathbf{v}) = child_r(\mathbf{v})$ and $value = 0$ exists at the same time, and there are no two nonterminal nodes $\mathbf{u}$ and $\mathbf{v}$ such that $\mathbf{u} = \mathbf{v}$.

Definition 2.2 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 EVBDD. For the sake of clarity, we also use the flattened form of EVBDDs in which a nonterminal node is represented by $\langle x, child_l, child_r \rangle$ and a terminal node is some integer.

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. For example, Figure 1 shows two arithmetic functions $3 - 4x + 4xy + xz - 2y + yz$ and $4x_2 + 2x_1 + x_0$ where all variables are binary variables.

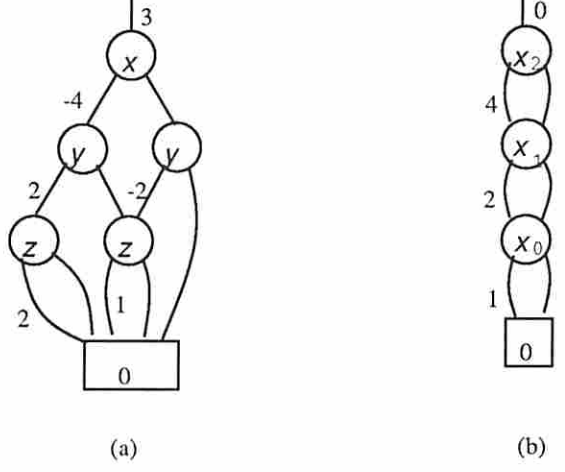

Figure 1: Two examples

Note that, it requires only n nonterminal nodes to represent an n variable linear function, for example, Figure 1 (b) is a linear function which can also be interpreted as a 3-bit interger.

The function value is computed by the summation of edge values (right edge values are 0) along the path associated with the input assignment. For example, in Figure 1 (a), the function value of $x = 1, y = 0$ and $z = 1$ is $3 + (-4) + 0 + 1 = 0$, and the function value of $x_2 = 1, x_1 = 0$ and $x_0 = 1$ in Figure 1 (b) is $0 + 4 + 0 + 1 = 5$.

Most of BDD and EVBDD operations follow paradigms of *ite* [8] and *apply* [20]. For readers who are interested in these paradigms, we put the pseudo code of *apply* and an example in the appendix. One of the main reasons for the success of BDD representation is the caching of computation results. This is carried out by using a *comp_table* in the above paradigms. An entry of *comp_table* has the form $\langle f, g, op, h\rangle$ which stands for $f\ op\ g = h$. To compute $f\ op\ g$, we first look up the *comp_table* with key $\langle f, g, op\rangle$, if an entry is found then the last element of the entry h is retrieved as the result; otherwise, we perform *op* on the subgraphs of f and g and store the result in *comp_table* after the completion of $f\ op\ g$.

EVBDD representation enjoys a distinct feature, called *additive* property, which is not seen in the BDD 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 $\langle\langle 0, \mathrm{f}\rangle, \langle 0, \mathrm{g}\rangle, -\rangle$. After the computation of $\langle\langle 0, \mathrm{f}\rangle, \langle 0, \mathrm{g}\rangle, -\rangle$ resulting in $\langle c_h, \mathrm{h}\rangle$, we then add $c_f - c_g$ to c_h to have the complete result of $\langle\langle c_f, \mathrm{f}\rangle, \langle c_g, \mathrm{g}\rangle, -\rangle$. Hence, every operation $\langle\langle c'_f, \mathrm{f}\rangle, \langle c'_g, \mathrm{g}\rangle, -\rangle$ can share the computation result of $\langle\langle 0, \mathrm{f}\rangle, \langle 0, \mathrm{g}\rangle, -\rangle$. This then will increase the hit ratio for caching the computation results. To speed up the relational operators and the *minimize* operation, we include the minimum and maximum function values in each EVBDD node.

3 An Example

This section illustrates how we solve the ILP problem using EVBDDs through a simple example. For the sake of readability, we use the flattened form of 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. 2 (a). We then construct the constraints. The left hand side of constraint (1) represented by an EVBDD is shown in Fig. 2 (b). After the relational operator $\leq$ has been applied on constraint (1), the resulting EVBDD is shown in Fig. 2 (c). Similarly, EVBDDs for constraint (2) are shown in Fig. 2 (d) and (e). The conjunction of two constraints, Fig. 2 (c) and (e), is the EVBDD in Fig. 2 (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. 2 (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. 2 (g) which is 0.

4 The Constraint Functions

Initially, every constraint is represented by an EVBDD in inequality form which requires n nodes for an n-variable function. When a constraint is converted to the Boolean function form, the number of nodes required may become exponential in n. To avoid this space complexity, the conversion of constraints are carried out *dynamically*.

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 $AX - b + 1 \leq 0$ (provided that 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 $-AX + b \leq 0$. Operations addition, subtraction and multiplication by -1 are carried out through the *apply* operator.

We describe how relational operation $leq0$ is carried out in EVBDDs:

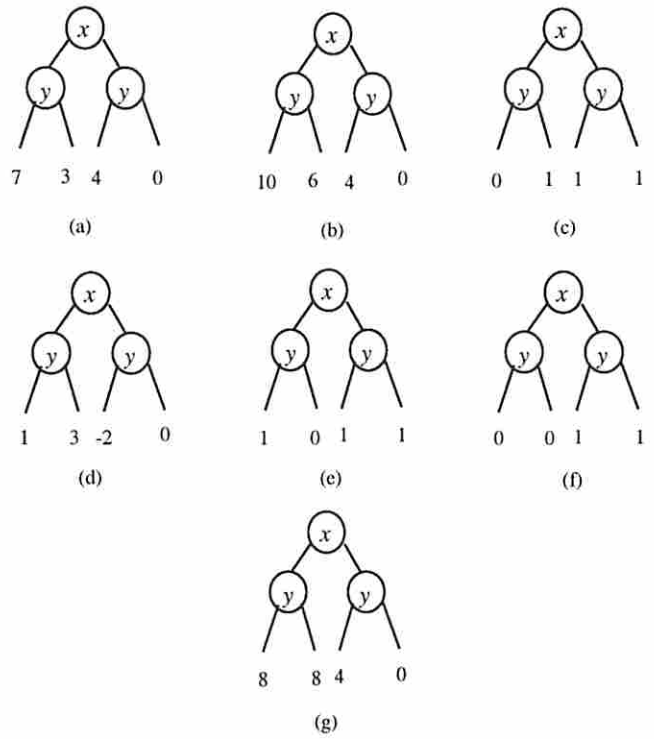

Figure 2: A simple example

```
leq0(⟨c_f, f⟩)
{
    if ($max(\langle c_f, \mathbf{f}\rangle) \leq 0$) return($\langle 1, \mathbf{0}\rangle$);
    if ($min(\langle c_f, \mathbf{f}\rangle) > 0$) return($\langle 0, \mathbf{0}\rangle$);
    if ($comp\_table\_lookup(\langle c_f, \mathbf{f}\rangle, LEQ0, ans)$) return($ans$);
    $\langle c_{gl}, \mathbf{g}_l\rangle = leq0(\langle c_{fl}, \mathbf{f}_l\rangle)$;
    $\langle c_{gr}, \mathbf{g}_r\rangle = leq0(\langle c_{fr}, \mathbf{f}_r\rangle)$;
    if ($\langle c_{gl}, \mathbf{g}_l\rangle == \langle c_{gr}, \mathbf{g}_r\rangle$) return ($\langle c_{gl}, \mathbf{g}_l\rangle$);
    $\mathbf{g} = find\_or\_add(index(\mathbf{f}), \mathbf{g}_l, \mathbf{g}_r, c_{gl} - c_{gr})$;
    $comp\_table\_insert(\langle c_f, \mathbf{f}\rangle, LEQ0, \langle c_{gr}, \mathbf{g}\rangle)$;
    return ($\langle c_{gr}, \mathbf{g}\rangle$);
}
```

Operator *leq*0 follows the paradigm of *apply*. To speed up *leq*0, we include the minimum and maximum function values in each EVBDD node. The procedure is recursive and terminates when any one of the following conditions holds: the maximum of function f is less than or equal to 0, the minimum of f is greater than 0, or it has been computed before. Otherwise, the procedure traverses down the graph.

The function performed by *leq*0 is the same as $LI\_to\_BDD(I)$ of [19] 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 *leq*0 can cache computation results as in all EVBDD operations while LI_to_BDD can not. This is very important because the efficiency of BDD operations heavily depends on the extent by which this property is exploited. In [19], 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 believe that the difference is due to our computation caching capability.

5 The Goal Function

The goal function represented by an EVBDD is very compact. An n-variable function requires only n nodes. For example, the EVBDD in Fig. 1 (b) can be used to represent the goal function *minimize* $4x_2 + 2x_1 + x_0$. The main advantage of this representation over the one in [19] is that our representation provides computation sharing on operations performed on the goal and constraint functions.

The operator projection (defined in Section 3 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. 2 (g)) is unnecessary. To address this issue, we have developed an operator called *minimize* which employs both the branch and bound and computation caching techniques. This operator 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 $b' < b$ of g subject to c; otherwise, *minimize* returns 0. If b' is found, b is replaced by b'; otherwise, b is unchanged.

Note that when *minimize* returns 0, it does not implies 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 process.

The parameter b serves two purposes: it increases the hit ratio for computation caching and it 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 are no feasible solutions (with respect to g and c) which are less than b, then v is set to $b + 1$.

Suppose we want to compute the minimum of g subject to c with current best solution b. We look up the computed table with key $\langle g, c\rangle$. If there is an entry $\langle g, c, \langle b', v\rangle\rangle$ matched, then there are three possibilities:

1. If $b > v$ then v is the solution we wanted.

2. If $b' \geq v \geq b$ or $v > b' \geq b$ then b is the optimum solution because the best we can find under g and c is v which is inferior to b.

3. If $b \geq v > b'$ then no conclusion can be inferred and further computation is required. Although there is no better feasible solution than b' (due to $v > b'$), it does not imply that there will be no better solution than b.

In case 1, computation caching is a success, in case 2 pruning takes place (also computation caching), but in case 3 both operations fail.

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 will be 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 will be 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.* In lines 27-32, 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 33-41.

```
$minimize(\langle c_g, \mathbf{g}\rangle, \langle c_c, \mathbf{c}\rangle, bound)$
{
    if ($\langle c_c, \mathbf{c}\rangle == \langle 0, \mathbf{0}\rangle$) return 0;
    if ($min(\langle c_g, \mathbf{g}\rangle) \geq bound$) return 0;
    if ($\langle c_g, \mathbf{g}\rangle == \langle c_g, \mathbf{0}\rangle$) {
        $bound = c_g$;
        return 1;   }
    if ($\langle c_c, \mathbf{c}\rangle == \langle 1, \mathbf{0}\rangle$) {
        $bound = min(\langle c_g, \mathbf{g}\rangle)$;
        return 1;   }
    $new\_bound = bound - c_g$;
    if ($comp\_table\_lookup(\langle 0, \mathbf{g}\rangle, \langle c_c, \mathbf{c}\rangle, entry)$) {
        if ($entry \rightarrow value \leq entry \rightarrow bound$) {
            if ($new\_bound > entry \rightarrow value$) {
                $bound = entry \rightarrow value + c_g$;
                return 1;   }
            else return 0;   }
        else {
            if ($new\_bound \leq entry \rightarrow bound$) return 0;   }   }
    $entry \rightarrow bound = new\_bound$;
    if ($index(\mathbf{c}) \geq index(\mathbf{g})$) {
        $\langle c_{gl}, \mathbf{g}_l\rangle = \langle value(\mathbf{g}), child_l(\mathbf{g})\rangle$;
        $\langle c_{gr}, \mathbf{g}_r\rangle = \langle 0, child_r(\mathbf{g})\rangle$;   }
    else {   $\langle c_{gl}, \mathbf{g}_l\rangle = \langle c_{gr}, \mathbf{g}_r\rangle = \langle 0, \mathbf{g}\rangle$;   }
    if ($index(\mathbf{c}) \leq index(\mathbf{g})$) {
        $\langle c_{cl}, \mathbf{c}_l\rangle = \langle c_c + value(\mathbf{c}), child_l(\mathbf{c})\rangle$;
        $\langle c_{cr}, \mathbf{c}_r\rangle = \langle c_c, child_r(\mathbf{c})\rangle$;   }
    else {   $\langle c_{cl}, \mathbf{c}_l\rangle = \langle c_{cr}, \mathbf{c}_r\rangle = \langle c_c, \mathbf{c}\rangle$;   }
    if ($min(\mathbf{g}_l) \leq min(\mathbf{g}_r)$) {
        $t\_ret = minimize(\langle c_{gl}, \mathbf{g}_l\rangle, \langle c_{cl}, \mathbf{c}_l\rangle, new\_bound)$;
        $e\_ret = minimize(\langle c_{gr}, \mathbf{g}_r\rangle, \langle c_{cr}, \mathbf{c}_r\rangle, new\_bound)$;   }
    else {
        $e\_ret = minimize(\langle c_{gr}, \mathbf{g}_r\rangle, \langle c_{cr}, \mathbf{c}_r\rangle, new\_bound)$;
        $t\_ret = minimize(\langle c_{gl}, \mathbf{g}_l\rangle, \langle c_{cl}, \mathbf{c}_l\rangle, new\_bound)$;   }
    if ($t\_ret \,||\, e\_ret$) {
        $entry \rightarrow value = new\_bound$;
        $comp\_update(\langle 0, \mathbf{g}\rangle, \langle c_c, \mathbf{c}\rangle, entry)$;
        $bound = new\_bound + c_g$;
        return 1;   }
    else {
        $entry \rightarrow value = entry \rightarrow bound + 1$;
        $comp\_update(\langle 0, \mathbf{g}\rangle, \langle c_c, \mathbf{c}\rangle, entry)$;
        return 0;   }
}
```

Example 5.1 We want to minimize the goal function $-4x + 5y + z + 2w$ subject to the constraint shown in Fig. 3 (b). For the sake of readability, the goal function (Fig. 3) is represented in EVBDD while the constraint function is represented in BDD. Note that in both representations, the left edges represent the true edges and the right edges represent the false edges. 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) Again, we traverse down to nodes **a** and **b** this time through path $x = 1$ and $y = 0$. The new local upper bound is now $2 - (-4) - 0 = 6$, 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 small than 6. Thus, a new solution is found $-4 + 0 + 1 = -3$.

(c) Through path $x = 0$ and $y = 1$, we reach **a** and **b** again. The local upper bound is $-3 - 0 - 5 = -8$. Again, from computed table, we know 1 is the best we can get which is larger than (inferior to) -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) Next, 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. Thus, this subproblem is pruned.

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

□

6 The Algorithm

The basic algorithm in FGILP, *ilp_minimize*, employs a branch and bound technique as shown in Fig. 4. In addition to goal and constraint functions, 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 upper bound is the maximum of the goal function plus 1.

If the maximum of goal function is less than the lower bound or the minimum of goal function is greater than or equal to the upper bound, the problem can be pruned. There is a third pruning process. If there exists a constraint whose minimum feasible solution is

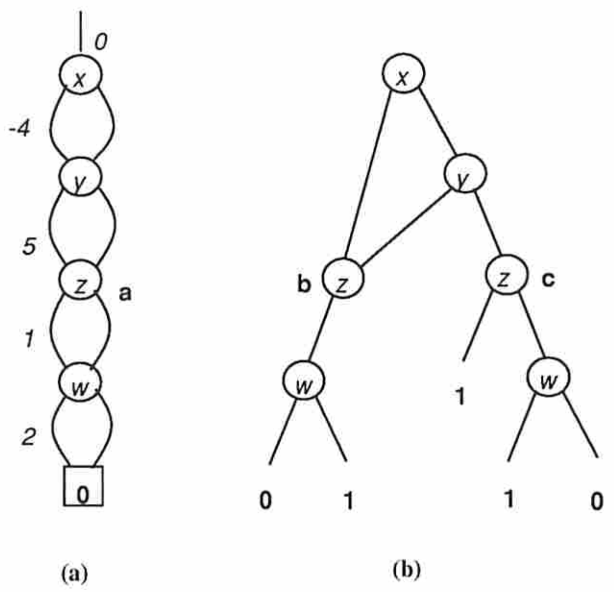

Figure 3: An example for the *minimize* operator.

```
$ilp\_minimize(goal, constraints, lower\_bound, upper\_bound)$
{
1    if ($max(goal) < lower\_bound$) return;
2    if ($min(goal) \geq upper\_bound$) return;
3    if ($\exists c \in constraints$ such that $minimize(goal, c, upper\_bound)$ == 0) return;
4    $new\_constraints$ = conjunction of $constraints$;
5    if ($new\_constraints$ has only one element and is in Boolean form) {
6        $minimize(goal, new\_constraints, upper\_bound)$;
7        return;
8    }
9    creates two new subproblems $\langle goal_l, new\_constraints_l\rangle$ and $\langle goal\_r, new\_constraints_r\rangle$
10   $ilp\_minimize(goal_l, new\_constraints_l, lower\_bound, upper\_bound)$;
11   $ilp\_minimize(goal_r, new\_constraints_r, lower\_bound, upper\_bound)$;
}
```

Figure 4: Pseudo code for *ilp_minimize*.

greater than or equal to the current best solution (upper bound), then again the problem can be pruned.

Initially, each constraint function is represented by an EVBDD in inequality form. The conversion of a constraint from the inequality form to a Boolean function and the conjunction of constraints are carried out dynamically in order to manage the space complexity. 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.

7 Experimental Results

FGILP has been implemented in C under the SIS environment. Table 1 shows our experimental results on ILP problems from MIPLIB provided by the Rice University. It also shows the results of LINDO [28] (a commercial tool) on the same set of benchmarks. FGILP was run under SPARC station 2 (28.5 MIPS) with 64 MB memory (this is also an important parameter with respect to FGILP) while LINDO was run under SPARC station 10 (101.6 MIPS). Both LINDO and FGILP produce the optimal solutions for the benchmarks.

FGILP provides users with an *n_supp* parameter such that if a constraint has less than *n_supp* supporting (dependent) variables, then it will be converted to a Boolean function; otherwise, it remains in inequality form. With this parameter, the conversion from inequalities to Boolean functions can be carried out in a divide and conquer manner.

Even if all constraints can be constructed without using excessive amounts of memory, conjoining them altogether at once may create too big an EVBDD. FGILP allows users to set another parameter *c_size* to control the size of EVBDDs. Only when constraints, in Boolean function forms, are smaller in size than this parameter, they are conjoined. Thus, conjoining constraints is carried out in a divide and conquer fashion.

These parameters provide two advantages. First, they provide FGILP with a space-time tradeoff capability. The more memory FGILP can have, the faster FGILP can run because of better performance of computation caching. On the other hand, users can set these parameters to control size of the program. Second, when combined with the branch and bound technique, some subproblems may be pruned before inequalities are converted to the Boolean form or constraints are conjoined together.

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 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. The smallest sizes of EVBDDs are conjoined first.

3. The constraints which have 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 timing 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. All problems reported here set no limitation for 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. LINDO could not solve 'bm23' benchmark.

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.

Problem	Inputs	Constraints	FGILP	LINDO	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

8 Discussions

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 less 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 depth first. A combination of these two approaches has been reported in [13].

Branching Rule

This parameter refers to the selection of next variable to branch. Various selection criteria which have been proposed use *priorities* [25], *penalties* [11, 32], *pseudo-cost* [6], and *integer infeasibility* [3] conditions. Currently, FGILP uses the same variable ordering as the one used to create EVBDDs because it simplifies the implementation. However, this is quite restrictive. Our future work includes the incorporation of more sophisticated selection criteria into FGILP [1].

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 [12, 29], and disjunctive programming [4]. The bounding procedure used in FGILP is similar to the one proposed in [1] (i.e., the additive method). In our experience, the most pruning takes place at line 3 of the code for *ilp_minimize*. 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 on 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. In future, we will incorporate the above methods into FGILP.

Logical Tests

It is believed that logical tests may be as important as the bounding procedure [5, 27]. In addition to partial ordering of variables, a particularly useful class of tests, when available, are those based on *dominance* [18, 19]. Currently, FGILP employs no logical tests. We believe that the inclusion of logical tests in FGILP will certainly 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 [28] which is one of the most widely used commercial tools [27] for solving ILP problems.

In conclusion, we presented an ILP solver based on EVBDD representation called FGILP. Our approach is to combine the benefits of EVBDD data structure with the state-of-the-art ILP solving technique. The distinct features of FGILP are the capabilities of constructing and conjuncting constraints dynamically and caching the computation results on both Boolean

[1]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.

operations and optimization operations. These features give rise to efficient memory management and fast execution time. The run-time of FGILP is comparable to that of LINDO.

APPENDIX A

The following algorithm describes the function *apply* which 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.

```
$apply(\langle c_f, \mathbf{f}\rangle, \langle c_g, \mathbf{g}\rangle, op)$
{
1    if ($terminal\_case(\langle c_f, \mathbf{f}\rangle, \langle c_g, \mathbf{g}\rangle, op)$ return($\langle c_f, \mathbf{f}\rangle\ op\ \langle c_g, \mathbf{g}\rangle$));
2    if ($comp\_table\_lookup(\langle c_f, \mathbf{f}\rangle, \langle c_g, \mathbf{g}\rangle, op, ans)$) return($ans$);
3    if ($index(\mathbf{f}) \geq index(\mathbf{g})$) {
4        $\langle c_{gl}, \mathbf{g}_l\rangle = \langle c_g + value(\mathbf{g}), child_l(\mathbf{g})\rangle$;
5        $\langle c_{gr}, \mathbf{g}_r\rangle = \langle c_g, child_r(\mathbf{g})\rangle$;
6        $var = variable(\mathbf{g})$;
7    }
8    else {
9        $\langle c_{gl}, \mathbf{g}_l\rangle = \langle c_{gr}, \mathbf{g}_r\rangle = \langle c_g, \mathbf{g}\rangle$;
10       $var = variable(\mathbf{f})$;
11   }
12   if ($index(\mathbf{f}) \leq index(\mathbf{g})$) {
13       $\langle c_{fl}, \mathbf{f}_l\rangle = \langle c_f + value(\mathbf{f}), child_l(\mathbf{f})\rangle$;
14       $\langle c_{fr}, \mathbf{f}_r\rangle = \langle c_f, child_r(\mathbf{f})\rangle$;
15   }
16   else { $\langle c_{fl}, \mathbf{f}_l\rangle = \langle c_{fr}, \mathbf{f}_r\rangle = \langle c_f, \mathbf{f}\rangle$;}
17   $\langle c_{hl}, \mathbf{h}_l\rangle = apply(\langle c_{fl}, \mathbf{f}_l\rangle, \langle c_{gl}, \mathbf{g}_l\rangle, op)$;
18   $\langle c_{hr}, \mathbf{h}_r\rangle = apply(\langle c_{fr}, \mathbf{f}_r\rangle, \langle c_{gr}, \mathbf{g}_r\rangle, op)$;
19   if ($\langle c_{hl}, \mathbf{h}_l\rangle == \langle c_{hr}, \mathbf{h}_r\rangle$) return ($\langle c_{hl}, \mathbf{h}_l\rangle$);
20   $\mathbf{h} = find\_or\_add(var, \mathbf{h}_l, \mathbf{h}_r, c_{hl} - c_{hr})$;
21   $comp\_table\_insert(\langle c_f, \mathbf{f}\rangle, \langle c_g, \mathbf{g}\rangle, op, \langle c_{hr}, \mathbf{h}\rangle)$;
22   return ($\langle c_{hr}, \mathbf{h}\rangle$);
}
```

Example .1 An example of $apply(\langle 0, \mathbf{f}\rangle, \langle 0, g\rangle, +)$ is shown in Figure 5. Let the variable ordering be $x_1 < x_0$. Figure 5 (a) shows the initial arguments of *apply*; (b) is the recursive call of *apply* on line 7 whose result is (c). Similarly, another call to *apply* on line 8 and its results are shown in (d) and (e) respectively. The final result is shown in (f) which is generated from either line 10 or 13. □

References

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

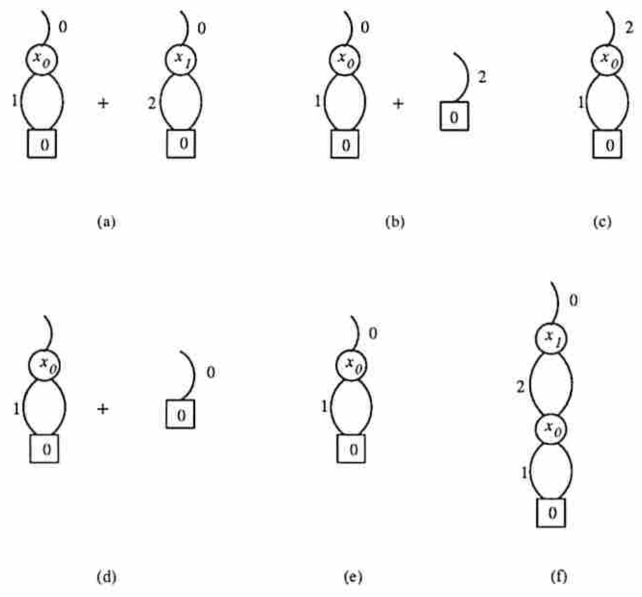

Figure 5: Snapshots of *apply*$(\langle 0, \mathbf{f}\rangle, \langle 0, \mathbf{g}\rangle, +)$

[2] E. Balas, "A note on the branch-and-bound principle," *Operations Research*, 16, (1968), pp. 442-445.

[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, 1975, pp. 479-494.

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

[5] E. Balas, "Report on branch and bound/implicit enumeration," *Annals of Discrete Mathematics 5*, North-Holland, 1979, pp. 185-191.

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

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

[8] K.S. Brace, R.L. Rudell, and R.E.Bryant, "Efficient implementation of a BDD package," *Proc. of the 27th Design Automation Conference*, 1990, pp. 40-45.

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

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

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

[12] M.L. Fisher, "The Lagrangian relaxation method for solving integer programming problems," *Management Science*, 1981, pp. 1-18.

[13] J.J.A. Forrest, J.P.H, Hirst and J.A. Tomlin, "Practical solution of large mixed integer programming problems with Umpire," *Management Science 20*, 1974, pp. 736-773.

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

[15] R.E. Gomory, "Outline of an algorithm for integer solutions to linear program," *Bulletin of the American Mathematical Society 64*, 1958, pp. 275-278.

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

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

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

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

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

[21] Y-T. Lai, S. Sastry and M. Pedram, "Boolean matching using binary decision diagrams with applications to logic synthesis and verification," *ICCD* 1992, pp.452-458.

[22] 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*, in press.

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

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

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

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

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

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

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

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

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

[32] J. A. Tomlin, "Branch and bound methods for integer and non-convex programming," in J. Abadie, ed., *Integer and non-linear programming*, North-Holland, Amsterdam, 1970.