Models and Algorithms for Computing Minimum-Size Prime Implicants

Vasco M. Manquinho, Arlindo L. Oliveira and João P. Marques Silva
Cadence European Laboratories / Instituto Superior Técnico/ INESC
1000 Lisboa, Portugal
email: {vmm,aml,jpms}@algos.inesc.pt

Abstract

Minimum-size prime implicants of Boolean functions find application in many areas of Computer Science including, among others, Electronic Design Automation and Artificial Intelligence. The main purpose of this paper is to describe and evaluate two fundamentally different modeling and algorithmic solutions for the computation of minimum-size prime implicants. One is based on explicit search methods, and uses Integer Linear Programming models and algorithms, whereas the other is based on implicit techniques, and so it uses Binary Decision Diagrams. For the explicit approach we propose new dedicated ILP algorithms, specifically target at solving these types of problems. As shown by the experimental results, other well-known ILP algorithms are in general impractical for computing minimum-size prime implicants. Moreover, we experimentally evaluate the two proposed algorithmic strategies.

1 Introduction

Given a propositional formula φ in Conjunctive Normal Form (CNF), denoting a boolean function f, the problem of computing a minimum-size assignment (in the number of literals) that satisfies f is referred to as the ***minimum-size prime implicant problem***. Minimum-size prime implicants find several applications in Artificial Intelligence and in Electronic Design Automation (EDA). For example, the computation of minimum-size test patterns in testing is tightly related with the computation of the minimum size prime implicant of a Boolean function [15]. In Artificial Intelligence, the identification of minimum-size prime implicants (i.e. minimum-size satisfying assignments for propositional formulas) is commonly encountered in Automated Reasoning and Non-Monotonic Reasoning [7, 13].

In this paper we describe and empirically compare two algorithms for computing minimum-size prime implicants. One is an explicit algorithm, based on Integer Linear Programming (ILP), whereas the other utilizes implicit techniques, and so it is based on Binary Decision Diagrams (BDDs). The explicit approach merges in a single algorithm, *bsolo*, two commonly used search paradigms, namely branch and bound search and backtrack search, applying search-pruning techniques from both paradigms. The implicit approach, *min-bdd*, extends the prime implicant implicit representation of [3, 4]. An experimental comparison of the two approaches is provided in Section 5, which also provides empirical evidence validating the proposed ILP algorithm against state of the art ILP solvers [2].

The paper is organized as follows. A few brief definitions are provided in Section 2. The ILP model and algorithm for computing minimum-size prime implicants are described in Section 3. Section 4 is dedicated to the implicit BDD-based approach. The two strategies are compared in Section 5, and the paper concludes in Section 6.

2 Definitions

A propositional formula φ in conjunctive normal form (CNF) defined on n variables $\{x_1, \ldots, x_n\}$ is a conjunction of m clauses ($\omega_1, \ldots, \omega_m$). A clause $\omega_j = (l_1 + \ldots + l_k)$, $k \le n$, is a disjunction of literals, and a literal l is either a variable x_i or its complement $\bar{x}_i$. Each clause denotes a constraint which can also be viewed as a linear inequality, $l_1 + \ldots + l_k \ge 1$. We use this alternative interpretation when appropriate. Furthermore, since a literal $l = \bar{x}_i$ can also be defined by $l = 1 - x_i$, we shall in general use the latter definition when viewing clauses as linear inequalities. We can also interpret a CNF formula as denoting a boolean function $f: \{0, 1\}^n \rightarrow \{0, 1\}$.

A *product* is a conjunction of literals ($l_1 \ldots l_k$), $k \le n$, whose literals are built out of distinct variables [3]. A product p is defined as an implicant of f if and only if it evaluates f to 1 and it is defined as a prime implicant of f if and only if p is an implicant of f, and there is no other implicant q of f whose literals are a subset of the literals of p. The minimum-size prime implicant is the prime implicant of f with the least number of literals.

3 Explicit Methods

Existing explicit methods for the computation of minimum-size prime implicants use integer linear programming (ILP) formulations. In this section we develop an ILP model for the computation of minimum-size prime implicants which was proposed in [7] and [13]. In addition, we describe a SAT-based branch-and-bound ILP algorithm, *bsolo*, targeted at solving highly constrained ILPs. *bsolo* uses powerful pruning techniques from the SAT algorithm GRASP [14] as well as lower bounding procedures. We also address some of the differences between *bsolo* and other SAT-based ILP algorithms proposed in [1] and [15].

3.1 Prime Implicant Computation Using Integer Programming

In order to find the minimum-size prime implicant of a boolean function, given its description in CNF, we formulate the problem as an integer linear program [7, 13]. Given a CNF formula φ, which is defined on a set of variables $\{x_1, \ldots, x_n\}$, and which denotes a Boolean function $f: \{0, 1\}^n \rightarrow \{0, 1\}$, apply the following transformation:

1. Create a new set of boolean variables $\{y_1, \ldots, y_{2n}\}$, where y_{2i-1} is associated with literal x_i, and y_{2i} is associated with literal $\bar{x}_i$.
2. For each clause $\omega = (l_1 + \ldots + l_m)$, replace each literal l_j with y_{2k-1} if $l_j = x_k$, or with y_{2k} if $l_j = \bar{x}_i$.
3. For each pair of variables, y_{2i-1} and y_{2i}, require that at most one is set to one. Hence, $y_{2i-1} + y_{2i} \le 1$.
4. The set of inequalities obtained from steps 2. and 3. can be viewed as a single set of inequalities $\boldsymbol{A} \cdot \boldsymbol{y} \ge \boldsymbol{b}$. Finally, define the cost function to be,

$$\min \sum_j y_j \tag{1}$$

5. The complete ILP formulation is thus defined as follows:

$$\begin{aligned} \min \quad & \sum_j y_j \\ \text{s.t.} \quad & \boldsymbol{A} \cdot \boldsymbol{y} \geq \boldsymbol{b} \end{aligned} \tag{2}$$

Examples illustrating the application of this ILP formulation can be found in [9, 13]. Moreover, it is clear that the minimum value of (1) denotes a minimum-size prime implicant of the original CNF formula φ, and from [13] we have,

Proposition 1. Given a CNF formula φ and associated boolean function *f*, the solution of the optimization problem (1) is a minimum-size prime implicant of *f*.

We can easily see that the worst-case search space for this ILP model is $2^{2n} = 4^n$. We should note that for this model, and for a search-based ILP algorithm, a straightforward arrangement of the order of the decision variables leads to a worst-case search space of 3^n (since only 3 assignments are possible for each of the n pairs of variables), but unfortunately this information cannot in general be made available to the ILP solver.

3.2 Search Algorithms for Solving ILPs

The idea of solving ILPs using a propositional satisfiability (SAT) algorithm was originally proposed in [1]. However, the algorithm described in [1] is based on the Davis-Putnam [6] SAT procedure, which has been shown to be particularly inefficient for a large number of instances of SAT [14]. In [15], instead of the Davis-Putnam procedure, the GRASP SAT algorithm [14] is used with significantly better results. However, other algorithmic organizations can be envisioned. Next, we propose *bsolo*, a branch-and-bound SAT-based ILP algorithm which, as the experimental results of Section 5 indicate, is an improvement over the algorithms of [1, 15].

In this section we describe *bsolo* which is an algorithm developed to solve highly constrained ILPs. This algorithm builds a branch and bound procedure on top of a SAT algorithm. It uses the GRASP SAT algorithm, which includes several powerful pruning techniques for reducing the amount of search associated with instances of SAT. Among the pruning techniques included in GRASP, the following have been shown to be particularly significant:

- GRASP implements a ***non-chronological backtracking*** search strategy. This backtracking strategy potentially permits skipping over large portions of the decision tree for some instances of SAT.
- GRASP utilizes selective ***clause recording*** techniques. During the search process, and as conflicts are diagnosed, new clauses are created from the causes of the conflicts. These new clauses are then used for pruning the subsequent search. Moreover, bounds on the size of recorded clauses can be imposed for preventing an excessive growth of the resulting CNF formula.
- In most practical situations, instances of SAT can have highly structured CNF representations. The intrinsic structure of these representations can be exploited by GRASP, after diagnosing the causes of conflicts, by identifying ***necessary assignments*** required to prevent conflicts from being identified during the search.
- In addition, other pruning techniques, as for example the ones commonly used in covering problems [5], can be straightforwardly applied to SAT algorithms.

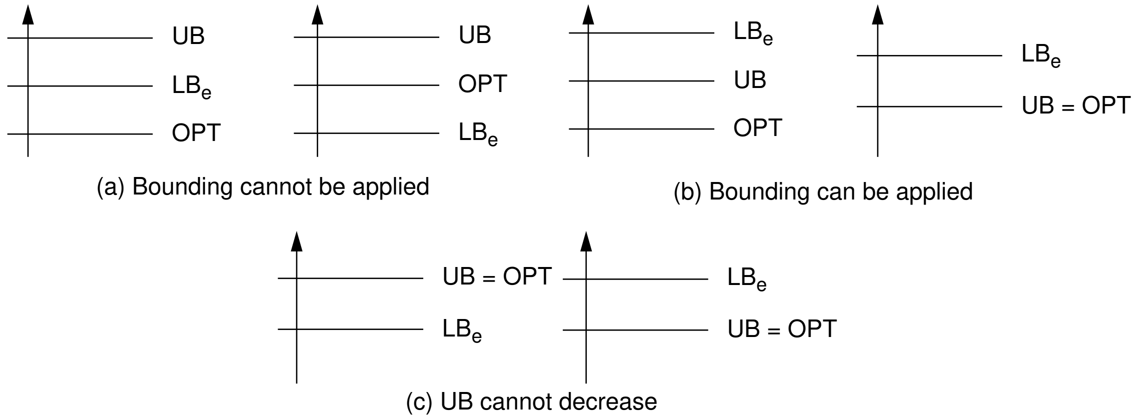

Figure 1: Using bounding in the ILP algorithm

3.3 SAT-Based Branch and Bound Algorithm

Our algorithm follows a different organization than [1] consisting in the use of a variation of the branch and bound procedure, where upper bounds to the cost function (1) are identified and lower bounds to the current set of variable assignments are estimated. In our implementation, we have used the lower bounding estimation procedures described in [5].

The operation of bounding for the proposed procedure is illustrated in Figure 1. Let UB denote the lowest *computed* upper bound on the solution of (2), $\mathrm{LB_e}$ denote an *estimated* lower bound on the solution of (2) and OPT denote the solution of (2). If the estimated lower bound is less than the already computed upper bound (as shown in Figure 1-(a)), then the search cannot be bound since it may still be possible to reduce the value of the upper bound. Clearly, the search can be bound whenever the estimated lower bound to the value of (1) is larger than or equal to the existing upper bound on the value of the cost function, as illustrated in Figure 1-(b). Finally, observe that Figure 1-(c) denotes the conditions after which the upper bound will no longer be updated during the search.

Moreover, since the branch and bound procedure is embedded in the SAT algorithm, every pruning technique used by the SAT algorithm can also be used in solving the ILP. This is particularly useful whenever a constraint of (2) becomes unsatisfied. Consequently, the branch and bound procedure consists of the following steps:

1. Initialize the upper bound to the highest possible value of the cost function.
2. If no decision can be made (i.e. a solution to the constraints has been identified), then compute an upper bound on the minimum value of the cost function of the ILP formulation. Update current upper bound and issue a conflict to guarantee that the search is bound. Otherwise, branch on a given decision variable (i.e. make decision assignment).
3. Apply boolean constraint propagation [16]. If a conflict is reached, then diagnose conflict, record relevant clauses, and proceed with the search process or backtrack if required.
4. Estimate lower bound. If this value is larger than or equal to the current upper bound, then issue a conflict, diagnose the conflict, backtrack, and continue the search from step 2.

The pseudo-code for the algorithm is shown in Figure 2. We should note that the proposed branch and bound SAT-based ILP algorithm has the following main differences with respect to the linear search ILP algorithm from [1, 15]:

- No clauses involving the cost function are created. In the algorithm described in [15], the ILP layer around the SAT algorithm creates additional clauses involving all the variables in the cost function. As

```
int bsolo(φ)
{
    _UB = ∞;
    while (TRUE) {
        if (Solution_found() || Decide() != DECISION) {
            _UB = Update_UB();
            Issue_UB_based_conflict();
        }
        while (Deduce() == CONFLICT) {
            if (Diagnose() == CONFLICT) {
                return _UB;
            }
        }
        while (Estimate_LB() ≥ _UB) {
            Issue_LB_based_conflict();
            if (Diagnose() == CONFLICT) {
                return _UB;
            }
        }
    }
}
```

Figure 2: SAT-based branch and bound algorithm

a consequence, whenever these clauses are the basis of a conflict, the algorithm backtracks chronologically. In *bsolo*, the exception to this rule occurs when the estimated lower bound is no less than the computed upper bound. In this situation a clause involving some of the literals in the cost function is temporarily created for causing the search procedure to backtrack. (See [14] for details of the backtrack search SAT algorithm.)

- Lower bounding procedures are required. As mentioned earlier, the lower bounding procedures of [5] are used, but other procedures based on linear-programming can also be used. Clearly, the tightness of the lower bounding procedure is crucial for the efficiency of the branch and bound algorithm.

4 Implicit Methods

An orthogonal approach for computing minimum-size prime implicants is based on implicit techniques. In this section we describe *min-bdd*, an algorithm for computing minimum-size prime implicants using BDDs. We use the algorithm described in [4] in order to obtain a BDD with all prime implicants. Afterwards, the identification of one prime with minimum-size is straightforward.

4.1 Prime computation using BDDs

BDDs are widely known by their power of representation and manipulation of boolean functions. Their efficiency in terms of time and space encourages its use in several domains, including logic synthesis, combinational equivalence checking and formal verification.

In this section we describe an implicit algorithm, *min-bdd*, for computing minimum-size prime implicants using BDDs. We start by creating a BDD for the boolean function. Afterwards, the algorithm

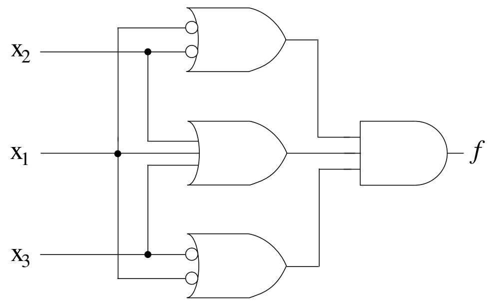

Figure 3: Example circuit for a given CNF formula

described in [4] is used for obtaining a BDD with all prime implicants. Finally, we obtain the minimum-size prime implicant through a straightforward breadth-first search on the BDD.

4.2 Variable Ordering

The variable ordering is crucial in order to be able to construct a BDD for a given boolean function. Different variable orderings may result in very different BDDs, therefore, in larger problems, only a good ordering allows us to build the BDD. It is known that the problem of finding the best order which minimizes the size of the BDD is NP-hard [10]. Consequently, the use of a heuristic method is vital.

With the purpose of finding a good ordering for the variables, we start by mapping the CNF formula into a two-level circuit. Afterwards, we apply the *Dynamic Weight Assignment Method* [10, 11] which uses the topological information of the circuit to find an approximation for the best variable ordering. The mapping process can be described as follows:

Given a CNF formula φ, defined on a set of variables $\{x_1, \ldots, x_n\}$, and which denotes a boolean function $f: \{0, 1\}^n \to \{0, 1\}$, we create a two-level circuit representing *f*. The circuit has *n* inputs (one for each variable) and one output which denotes the value of *f*. For each clause $\omega = \left(l_1 + \ldots + l_k\right)$ an OR-gate is created with *k* inputs. These gates are the first-level of the circuit, whereas the second-level consists of a single AND-gate whose inputs are the outputs of the first-level gates. Therefore, the output of this AND-gate represents the function *f*. Figure 4 shows an example circuit.

The *Dynamic Weight Assignment Method* heuristically identifies the input which most influences the output in a topological sense. In terms of the CNF formula, this means it identifies the variable with most control over the function. Since the variables with higher control of the function should be at higher position in the BDD, this seem to be a good method to order the variable set of the boolean function. This heuristic method was originally described as follows:

1. Each lead is assigned a weight, beginning with the assignment of a weight 1.0 to the primary outputs.
2. The weight is propagated toward the inputs as follows:
   a. At each gate, the weight on the output is equally distributed to the inputs.
   b. At each fan-out point, the weights of the fan-out branches are accumulated into the weight of the fan-out stem.
3. Choose the primary input with largest weight.
4. Delete the part of the circuit which can be *only* reached only from the chosen input. Go back to 1.

An example of this weight assignment is shown in Figure 4. However, our circuits always have the same topology and we may simplify this process, since we are trying to maximize the weight at the primary inputs independently of their logic value. Notice that the assigned weights at each OR-gate are always

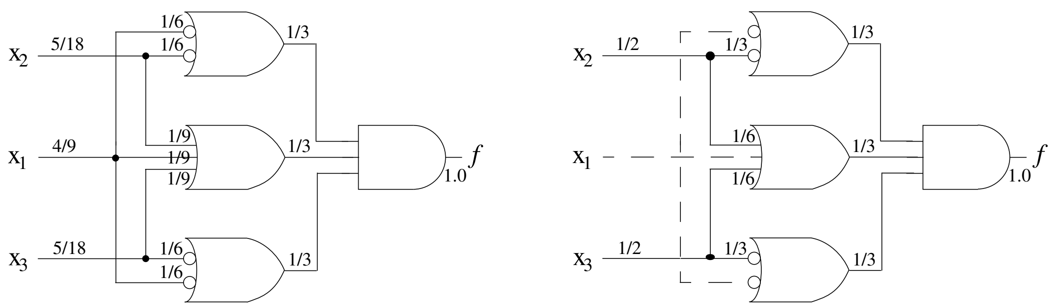

Figure 4: Example of Dynamic Weight Assignment Method

equal, therefore, there's no need to make this assignments every time we chose a primary input and we may fix them with value 1.0. So, the method can be simplified as follows:

1. Calculate the weight values of primary inputs $\{x_1, \ldots, x_n\}$ as,

$$w_{x_j} = \sum_i \frac{1}{inputs(g_i)} \tag{3}$$

   where g_i are those gates which have x_j as input and $inputs(g_i)$ denotes the number of inputs of g_i.
2. Choose the primary input with largest weight.
3. Delete the part of the circuit which can be reached only from the chosen input. Go back to 1.

4.3 Metaproducts

The metaproduct representation was first described in [3]. The purpose of metaproducts is to have a canonical representation of sets of products which can be efficiently represented with BDDs.

In the metaproduct representation we use two new sets of variables. From the initial set $\{x_1, \ldots, x_n\}$, we create $\{o_1, \ldots, o_n\}$ and $\{s_1, \ldots, s_n\}$ and call them the *occurrence* and *sign* variables, respectively. Consider *P* as the set of strings $\{x_1, \bar{x}_1, \varepsilon\} \times \ldots \times \{x_n, \bar{x}_n, \varepsilon\}$ where ε is the empty string. *P* is the set of products that can be built out of the initial set of variables. Let us also consider σ as being the mapping function from $\{0, 1\}^n \times \{0, 1\}^n$ to *P*, $\sigma(o, s) = l_1 \ldots l_n$, where

$$l_k = \varepsilon \text{ if } o_k = 0$$
$$l_k = x_k \text{ if } o_k = 1 \text{ and } s_k = 1$$
$$l_k = \bar{x}_k \text{ if } o_k = 1 \text{ and } s_k = 0$$

As an example, with $n = 3$ we would have $\sigma([101], [100]) = (x_1\bar{x}_3)$. It is straightforward that σ does not provide a one to one mapping since it maps a total of 4^n elements into 3^n elements. In [3] it is suggested that we should consider the canonical representation of a product to be the set of all couples that denote this product. In our example, the set $\{([101], [100]), ([101], [110])\}$ would be the canonical representation of $(x_1\bar{x}_3)$.

4.4 Computation of Minimum-Size Primes

In this section we present an algorithm to generate a BDD with all prime implicants of a boolean func-

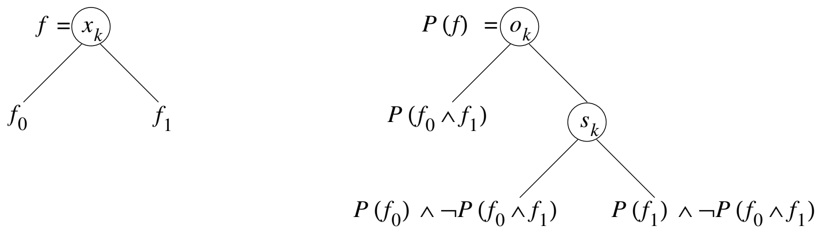

Figure 5: Computation of the prime set using metaproducts

tion, using the metaproduct representation introduced in Section 4.3. We assume that the BDD for the boolean function has already been built. Afterwards, the process of finding the minimum-size prime is straightforward.

Let *f* be a boolean function defined as $f : \{0, 1\}^n \rightarrow \{0, 1\}$ and $P(f)$ be the set of all its prime implicants. Consider $bdd(f) = bdd\left(x_k, f_0, f_1\right)$ in which x_k is the top node and, f_0 and f_1 are the functions pointed to by 0- and 1-edges. In [3, 4] is suggested that $o_{\Pi(1)} < s_{\Pi(1)} < \ldots < o_{\Pi(n)} < s_{\Pi(n)}$ is the best variable ordering to build the BDD, where Π is a permutation of the integers $\{1, \ldots, n\}$. This permutation is the same used to build the BDD of *f*, according to the heuristic procedure described in Section 4.2.

Consider a boolean function *f* represented in a BDD and let x_k be the top variable. The set of all prime implicants *P(f)* of *f* is the union of three sets of products, the ones where variable x_k doesn't occur and those where x_k occurs with positive and negative signs.

Let f_0 and f_1 be the function *f* where $x_k = 0$ and $x_k = 1$, respectively. $P(f_0)$ and $P(f_1)$ denote the primes of f_0 and f_1. Therefore, the intersection of these sets $P(f_0 \wedge f_1)$ denotes the primes of *f* where x_k doesn't occur. We can easily define the sets where variable x_k occurs with negative and positive sign as $\{\bar{x}_k\} \times P\left(f_0\right) \backslash P\left(f_0 \wedge f_1\right)$ and $\{x_k\} \times P\left(f_1\right) \backslash P\left(f_0 \wedge f_1\right)$. The set of primes of a boolean function *f* can be defined as [4]:

$$
\begin{aligned}
P(0) &= \varnothing \\
P(1) &= \{\varepsilon\} \\
P(f) &= P\left(f_0 \wedge f_1\right) U \ \{\bar{x}_k\} \times P\left(f_0\right) \backslash P\left(f_0 \wedge f_1\right) U \ \{x_k\} \times P\left(f_1\right) \backslash P\left(f_0 \wedge f_1\right)
\end{aligned}
$$

Given the prior definition, the BDD representing the set of prime implicants can be computed from the BDD of the function as shown in Figure 4, which illustrates how to build the BDD using the metaproduct representation.

As mentioned earlier, the minimum-size prime implicant can be obtained from this BDD through a breadth-first search. This can be achieved by finding the path from the top node to a 1-terminal node with the least number of 1-edges of the occurrence variables.

5 Experimental Results

In order to experimentally evaluate the different algorithms, some satisfiable instances of the DIMACS benchmarks [8] were mapped onto instances of the minimum-size prime implication computation problem. The experimental results for *lp_solve* [2], *opbdp* [1], *bsolo* and *min-bdd* are shown in Table 1. For each benchmark, 2,000 sec. of CPU time were allowed. As can be concluded, for most of the benchmarks, the explicit methods perform in most cases significantly better. (We should note, however, that for most examples the main difficulty with *min-bdd* is in the construction of the initial BDD and not in the derivation of

Benchmark	min	*lp-solve*	*opbdp*	*bsolo*	*min-bdd*
aim-100-1_6-y1-2	100	—	—	0.11	0.22
aim-100-2_0-y1-3	100	—	24.83	0.21	0.27
aim-100-3_4-y1-4	100	—	0.6	0.53	1.84
aim-200-1_6-y1-3	200	—	—	0.36	0.63
aim-200-3_4-y1-1	200	—	22.16	1.31	85.72
aim-200-6_0-y1-2	200	—	2.89	1.68	2.67
ii8a1	54	399.07	1.2	1.46	—
ii8b2	—	—	—	UB 369	—
ii8c2	—	—	—	UB 525	—
jnh1	92	—	1.51	6.06	—
jnh7	89	—	0.56	1.81	—
jnh12	94	—	0.34	0.67	—
par8-1-c	64	4.41	0.17	0.07	0.32
par8-2	350	58.02	0.87	0.34	37.22
par16-1-c	317	—	225.17	1237.5	424.36
par16-2	1015	—	433.47	—	—
par16-3-c	349	—	194.45	—	390.88
par16-4	1015	—	216.18	1454.9	—
ssa7552-038	—	—	UB 1452	UB 1448	—
ssa7552-159	—	—	UB 1327	UB 1327	—

Table 1: CPU times and computed upper bounds

the set of prime implicants.) The performance difference is particularly significant for practical instances derived from practical applications, as for example the *ssa* benchmarks from testing. These benchmarks are representative of practical applications of the algorithms described in the previous sections. Moreover, state of the art ILP algorithms, such as *lp_solve*, may perform poorly in specific ILP instances as is the case of the minimum-size prime implicant computation problem. Finally, we can also conclude that the techniques of GRASP [14] are crucial for some benchmarks, and so *bsolo* performs significantly better than *opbdp* for those cases. Finally, we note that for several example benchmarks no algorithm is able to find the optimal solution. For these examples only *bsolo*, and in some cases *opbdp*, are able to identify upper bounds to the optimum solution.

6 Conclusions

In this paper we describe two algorithms for computing minimum-size prime implicants. One is an explicit search algorithm and the other is based on implicit techniques. The algorithm based on explicit search performs in general better than the algorithm based on implicit techniques. Regarding the comparison of *bsolo* with *opbdp* [1], the overhead of some pruning techniques used in GRASP [14] and in *bsolo*, becomes apparent for some benchmarks. Nevertheless, *bsolo* performs in most cases better than *opbdp*. Furthermore, state-of-the-art ILP algorithms [2] are impractical for solving the ILPs associated with minimum-size prime implicant computation. From the results contained in this paper we can conclude that for solving these optimization problems, the explicit methods are in general preferable. However, all the evaluated algorithms can easily find extreme difficulties in solving instances of the minimum-size prime implicant problem. Hence, additional research work is necessary, trying to identify ever more effective search-pruning techniques for ILP algorithms.

References

[1] P. Barth, “A Davis-Putnam Based Enumeration Algorithm for Linear Pseudo-Boolean Optimization,” Technical Report MPI-I-95-2-003, Max-Planck-Institut für Informatik, January 1995.

[2] M.R.C.M. Berkelaar, UNIX^{TM} Manual Page of lp-solve. Eindhoven University of Technology, Design Automation Section, 1992.

[3] O. Coudert, J.C. Madre, “A New Method to Compute Prime and Essential Prime Implicants of Boolean Functions”, in *Advanced Research in VLSI and Parallel systems*, T. Knight and J. Savage Editors, The MIT Press, pp. 113-128, March 1992.

[4] O. Coudert, J.C. Madre, “A New Graph Based Prime Computation Technique”, in *Logic Synthesis and Optimization,* Kluwer Academic Pub., T. Sasao, pp. 34-57, 1993.

[5] O. Coudert, “On Solving Covering Problems,” in *Proceedings of the Design Automation Conference*, June 1996.

[6] M. Davis and H. Putnam, “A Computing Procedure for Quantification Theory,” *Journal of the Association for Computing Machinery*, vol. 7, pp. 201-215, 1960.

[7] B. Errico, F. Pirri, C. Pizzuti, “Finding Prime Implicants by Minimizing Integer Programming Problems” in *Eight Australian Joint Conference on Artificial Intelligence*, 1995.

[8] D. S. Johnson and M. A. Trick (eds.), *Second DIMACS Implementation Challenge*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 1993. DIMACS benchmarks available in ftp://Dimacs.Rutgers.EDU/pub/challenge/sat/benchmarks/cnf.

[9] V. M. Manquinho, P. F. Flores, J. P. Marques Silva and A. L. Oliveira, “Prime Implicant Computation Using Satisfiability Algorithms,” in *Proc. of the IEEE International Conference on Tools with Artificial Intelligence*, November 1997.

[10] S. Minato, “Graph-Based Representations of Discrete Functions” in *Representations of Discrete Functions*, pp. 2-28, Kluwer Academic Press, 1996.

[11] S. Minato, N. Ishiura, S. Yajima, “Fast Tautology Checking Using Shared Binary Decision Diagrams - Experimental Results” in *Proc. of the Workshop on Applied Formal Methods for Correct VLSI Design*, L. Claesen Ed., North Holland, 1989.

[12] T. Ngair, “A New Algorithm for Incremental Prime Implicant Generation,” in *Proceedings of the International Joint Conference on Artificial Intelligence*, 1993.

[13] C. Pizzuti, “Computing Prime Implicants by Integer Programming,” in *Proceedings of IEEE International Conference on Tools with Artificial Intelligence*, November 1996.

[14] J. P. M. Silva and K. A. Sakallah, “Conflict Analysis in Search Algorithms for Propositional Satisfiability,” in *Proceedings of the IEEE International Conference on Tools with Artificial Intelligence*, November 1996.

[15] J. P. M. Silva, “On Computing Minimum Size Prime Implicants,” in *International Workshop on Logic Synthesis*, May 1997.

[16] R. Zabih and D. A. McAllester, “A Rearrangement Search Strategy for Determining Propositional Satisfiability,” in *Proceedings of the National Conference on Artificial Intelligence*, pp. 155-160, 1988.