Satisfiability-Based Algorithms for 0-1 Integer Programming

Vasco M. Manquinho, João P. Marques Silva, Arlindo L. Oliveira and Karem A. Sakallah

Cadence European Laboratories / INESC
Instituto Superior Técnico
R. Alves Redol, 9, 1
1000 Lisboa, Portugal
{vmm,jpms,aml}@algos.inesc.pt, karem@eecs.umich.edu

Abstract

In this paper we describe two Propositional Satisfiability-based algorithms for solving 0-1 integer linear programs (ILP). The algorithms are specifically targeted at ILP instances that are highly constrained, i.e. instances for which the constraints are hard to satisfy. The two algorithms are based on recent algorithms for solving instances of Propositional Satisfiability (SAT) which are also highly constrained. In particular we illustrate how the algorithms for solving ILPs can be improved with search pruning techniques commonly used in SAT algorithms. The usefulness of the proposed algorithms is illustrated on a practical application for which instances are in general highly constrained.

1 Introduction

The vast majority of practical and commercial algorithms for solving generic integer linear programs (ILPs) are based on branch-and-bound search using linear-programming relaxations (LPR) [9]. These algorithms perform in general better than other algorithmic solutions on a large number of applications. Nevertheless, other algorithmic solutions can often be preferred for specific applications [2, 7]. For example, for 01-ILPs[1], and in different application domains, one often finds highly constrained classes of instances, i.e. instances for which the constraints are hard to satisfy. In these cases, ILP algorithms based on branch-and-bound with LPR may be unable to even find, in a reasonable amount of time, an assignment to the variables that satisfies the constraints. This can happen, for example, whenever the search algorithm enters portions of the search space where an assignment satisfying the constraints cannot be found and the algorithm takes too long to figure that out. As a result, and for highly constrained ILPs, we would like to enable search-based ILP algorithms with the ability to quickly identifying portions of the search space where satisfying assignments to the constraints cannot be found. One possible approach for solving this problem is to use Propositional Satisfiability (SAT) algorithms, which in general face hard to satisfy constraints. We start by reviewing SAT-based ILP algorithms from [2, 12]. Afterwards, we describe how the usual organization of branch-and-bound search can be easily extended to handle highly constrained ILPs. The procedure we propose applies search pruning concepts to the basic branch-and-bound algorithm which have been developed for Propositional Satisfiability (SAT) in recent years [3, 11]. The proposed organization of the branch-and-bound algorithm implements both the commonly used bound-based pruning techniques as well as pruning techniques derived from SAT algorithms.

The paper is organized as follows. In Section 2 the notational framework used throughout the paper is introduced. Afterwards, we briefly review backtrack search algorithms for SAT and describe some of the most commonly used search pruning techniques. In Section 4 we describe two different organizations for SAT-based ILP algorithms, targeted at solving highly constrained integer programs. The next step consists of describing an application of these algorithms, i.e. the computation of minimum-size prime implicants of Boolean functions [10]. Preliminary results obtained on different benchmarks clearly justify using the proposed ILP algorithms and strongly suggest that commonly used ILP solvers may be inadequate for specific classes of instances. Finally, Section 6 concludes the paper by suggesting potential applications as well as improvements to the proposed algorithms.

2 Definitions

A 0-1 ILP is defined as follows,

$$\begin{array}{lll} \text{minimize} & \boldsymbol{c}^{T} \cdot \boldsymbol{x} & \\ \text{subject to} & \boldsymbol{A} \cdot \boldsymbol{x} \geq \boldsymbol{b} & \boldsymbol{x} \in \{0,1\}^{N} \end{array} \tag{1}$$

where $\boldsymbol{c}^{T} \cdot \boldsymbol{x}$ denotes the cost function and $\boldsymbol{A} \cdot \boldsymbol{x} \geq \boldsymbol{b}, \boldsymbol{x} \in \{0,1\}^{N}$ denote the set of linear constraints. Without loss of generality and for simplifying the description of the algorithms, we assume that all entries in $\boldsymbol{A}$ are defined in the set $\{-1, 0, 1\}$. Such restriction basically enables each constraint to be viewed as a propositional clause [2], thus

1. A restricted form of ILPs where variables assume binary values.

allowing SAT algorithms to be readily used for solving ILPs. For the more general case where the entries in $\boldsymbol{A}$ are integers, the SAT algorithms must be adapted to handle generalized clauses, as detailed in [2]. Moreover, the techniques described in the paper can also be applied to the more general case where the entries in $\boldsymbol{A}$ and $\boldsymbol{b}$ are real numbers.

A propositional formula φ in Conjunctive Normal Form (CNF) denotes a boolean function $f: \{0, 1\}^n \rightarrow \{0, 1\}$, φ consists of a product of clauses, where each clause ω is a sum of literals, and a literal l is either a variable x_i or its complement x_i'. For a backtrack search algorithm for SAT [3, 11], a ***conflict*** is said to be identified when all literals of a clause are assigned value 0. A clause $\omega = (l_1 + \ldots + l_k)$ denotes a constraint which can also be viewed as a linear inequality, $l_1 + \ldots + l_k \geq 1$ [2, 7]. We use this alternative representation when appropriate. Furthermore, since a literal $l = x_i'$ can also be defined by $l = 1 - x_i$, we shall in general use this latter representation when viewing clauses as linear inequalities.

3 Backtrack Search Satisfiability Algorithms

A significant number of algorithmic solutions for SAT are based on backtrack search [2, 3, 6-8, 11, 13]. In this section we briefly outline the most effective search pruning techniques developed for SAT algorithms in recent years [3, 11]. All these techniques result from ***conflict diagnosis***, and all basically exploit the information that can be derived from diagnosis the causes of conflicts identified during backtrack search. As empirically shown in [3, 11], conflict diagnosis techniques are crucial for solving instances of SAT from real-world applications. Among the different techniques, the following play a key role in pruning the amount of search:

- A ***non-chronological backtracking*** search strategy. This backtracking strategy potentially permits skipping over large portions of the decision tree for some instances of SAT, thus *proving* the CNF formula (i.e. the constraints) to be unsatisfied in those portions of the search space.
- Selective ***clause recording***. 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 SAT algorithms, after diagnosing the causes of conflicts, by identifying ***necessary assignments*** required for preventing 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.

As we show in the next section, with a suitable organization of the ILP algorithm, each of these search pruning techniques can also be applied in solving ILPs.

4 SAT-Based Search Algorithms for Solving 0-1 ILPs

One of the first SAT-based algorithms for solving 0-1 ILP is *opbdp*, described in [2]. This algorithm iterates, in decreasing order, through the possible values *k* of the cost function. At each stage a new clause is added to the set of constraints, which basically requires the cost function to be no greater than the current iteration value *k* of the cost function. The resulting set of constraints is solved as an instance of SAT, in which generalized clauses are assumed [2]. The process is iterated until an unsatisfied instance is reached, thus defining the solution to the ILP as the previous value of *k*. This algorithm will henceforth be referred to as the SAT-based *linear-search* ILP algorithm [12] (*ls_ilp*) and is further analyzed in Section 4.1 where some of its main drawbacks are summarized. Besides the linear-search ILP algorithm, we describe in Section 4.2 a *branch-and-bound* ILP algorithm (*bb_ilp*) also built around a SAT solver.

4.1 SAT-Based Linear Search Algorithm

Let us consider the cost function $\boldsymbol{c}^T \cdot \boldsymbol{x}$. The possible values assumed by the cost function for the different assignments to the variables are in the range $\{LLB, \ldots, HUB\}$, where,

$$LLB = \sum_{c_i \in \boldsymbol{c}^T \wedge c_i \leq 0} c_i \tag{2}$$

(LLB denotes the *lowest lower bound* on the value of the cost function.) The ILP algorithm consists of applying the following sequence of steps, starting from an upper bound of $k = HUB$ on the value of the cost function:

1. Create a new inequality $\boldsymbol{c}^T \cdot \boldsymbol{x} \leq k$.
2. Solve the resulting instance of satisfiability. (Note that the resulting instance of satisfiability assumes arithmetic operations, but updating most SAT algorithms for handling this generalization is straightforward.)
3. If the instance of SAT is satisfiable, then decrement *k* (i.e. iterate target value of the cost function) and go back to step 1. Otherwise, report that the solution to the ILP is $k + 1$.

Note that this ILP algorithm allows for any SAT algorithm to be used as the underlying SAT testing engine. The proposed ILP algorithm is illustrated in Figure 1, and follows the one in [12]. For the current implementation of *ls_ilp*, the `solve_sat()` function call invokes the GRASP SAT algorithm [11].

4.2 SAT-Based Branch and Bound Algorithm

The branch-and-bound algorithm for solving ILPs extends the general backtrack search algorithm for solving SAT described in Section 3. Besides implementing backtrack search, additional pruning can be achieved through *bounding*. This added pruning ability is illustrated in Figure 2. Let UB denote the lowest *computed* upper bound on the solution of (1), LB_e denote an *estimated* lower bound on the solution of (1) and OPT denote the solution of (1), i.e. the least feasible value of $\boldsymbol{c}^T \cdot \boldsymbol{x}$ when the variables assume binary values. If the estimated lower bound is less than the already computed upper bound (as shown in Figure 2-(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

```
int ls_ilp (φ)
{
     k = HUB;                                                    // Using (3)
     while ( k ≥ LLB ) {                                         // Using (2)
          φ = φ ∪ {c^T · x ≤ k};
          status = solve_sat (φ);                          // Invoke SAT solver
          φ = φ – {c^T · x ≤ k};
          if ( status == SATISFIABLE ) {
               k = Σ c_i;                                  // Get a tighter value for k
                   --k ; x_i = 1
          } else { ++k; break; }
     return k;
}
```

Figure 1: SAT-based linear search algorithm

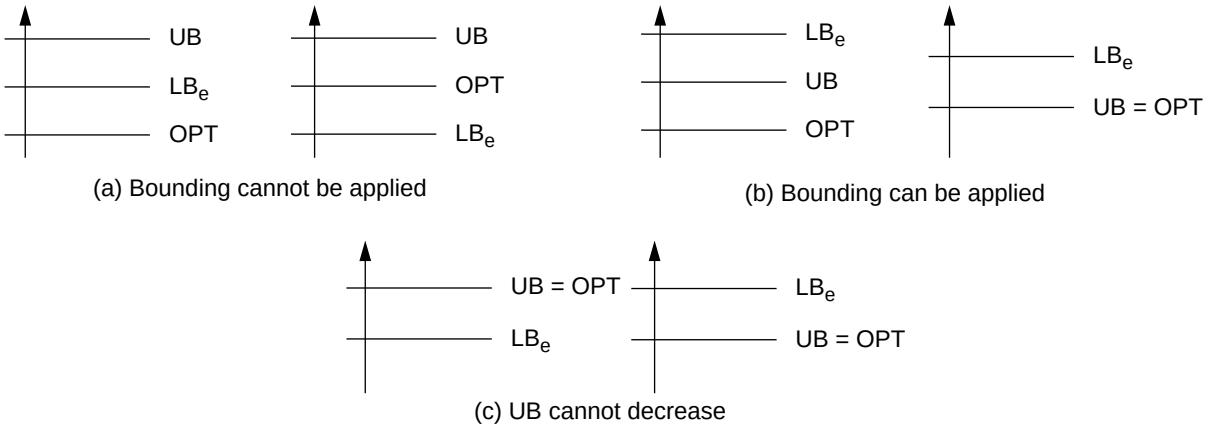

Figure 2: Using bounding in the ILP algorithm

the value of $\boldsymbol{c}^T \cdot \boldsymbol{x}$ is larger than or equal to the existing upper bound on the value of $\boldsymbol{c}^T \cdot \boldsymbol{x}$, as illustrated in Figure 2-(b). Finally, observe that Figure 2-(c) denotes the conditions after which the upper bound will no longer be updated during the search (i.e. UB = OPT).

Moreover, since the branch and bound procedure is embedded in the backtrack search 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 (1) becomes unsatisfied, since in this situation conflict diagnosis can be applied. The branch-and-bound algorithm is shown in Figure 3, and it consists of the following main steps:

1. Initialize the upper bound to highest possible value *HUB* (the *highest upper bound* on the cost function) plus 1. Observe that this value is given by,

```
int bb_ilp (φ)
{
     _UB = HUB + 1 ;                                          // Using (3)
     while (TRUE) {
          if (Solution_found() || Decide() != DECISION) { // Elect assignment
               Update_UB();
               Issue_UB_based_conflict();
          }
          while (Deduce() == CONFLICT) {            // Find necessary assignments
               if (Diagnose() == CONFLICT) { return _UB; }    // Diagnose conflict
          }
          while (Estimate_LB() ≥ _UB) {
               Issue_LB_based_conflict();
               if (Diagnose() == CONFLICT) { return _UB; }
          }
     }
}
```

Figure 3: SAT-based branch and bound ILP algorithm

$$HUB = \sum_{c_i \in \boldsymbol{c}^T \wedge c_i \geq 0} c_i . \tag{3}$$

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 $\boldsymbol{c}^T \cdot \boldsymbol{x}$. 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 [13], via `Deduce()`, for identifying necessary assignments. If a conflict is reached, then diagnose the conflict, record relevant clauses, and either 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.

Whereas upper bounds to the cost function $\boldsymbol{c}^T \cdot \boldsymbol{x}$ are updated as feasible assignments are identified, lower bounds to the current set of variable assignments are estimated. Different lower bound estimation procedures can be used, including ***linear programming relaxations*** and ***lagrangian relaxations*** [1, 9]. In our current implementation we have used the lower bound estimation procedures described in [5], since these procedures are the most suitable for the target application described in Section 5.

The two ILP algorithms, *ls_ilp* and *bb_ilp* have significantly different organizations. In general, we believe *bb_ilp* to be a better solution since clauses are not explicitly added to the original set of constraints. In general, clauses involving the cost function contain a large number of literals, which may affect negatively the effectiveness of the search pruning techniques described in Section 3. In the current implementation both algorithms, *bb_ilp* and *ls_ilp*, use the GRASP SAT solver [11] as the back-end SAT engine, but other SAT solvers could also be used [3]. Moreover,

we note that for *bb_ilp* the ILP algorithm is defined as a new layer on top of the SAT solver, whereas for *ls_ilp* the ILP algorithm explicitly invokes the SAT solver.

5 An Application: Computing Minimum-Size Prime Implicants

In this section we describe an application that in general yields highly constrained ILPs. 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 application in many areas including, among others, Automated Reasoning, Non-Monotonic Reasoning and Electronic Design Automation. As we describe below, the minimum-size prime implicant computation problem can be formulated as an ILP. We note, however, that instances of SAT, which can in general be hard to satisfy, yield ILPs that are accordingly highly constrained. As we show in Section 5.2, the proposed branch-and-bound ILP algorithm is extremely competitive in solving this problem.

5.1 The ILP Model

Given a description of a Boolean function in CNF, it is straightforward to formulate the computation of the minimum-size prime implicant as an integer linear program [10]. For this purpose we describe a simplified version of the ILP formulation introduced in [10]. 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 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 $l_j = x_i'$.
3. For each pair of variables, y_{2i-1} and y_{2i}, require that at most one is set to one. Hence, $y_{2i-1} + y_{2i} \leq 1$.
4. The set of inequalities obtained from steps 2. and 3. can be viewed as a single set of inequalities $\boldsymbol{A} \cdot \boldsymbol{y} \geq \boldsymbol{b}$. Finally, define the cost function to be,

$$\min \left[\sum_{j=1}^{2n} y_j \right] \tag{4}$$

It is clear that the minimum value of (4), that satisfies the given constraints, denotes a minimum-size prime implicant of the original CNF formula φ (see [10] for additional details).

5.2 Experimental Results

In this section we include experimental results of the two ILP algorithms, *bb_ilp* and *ls_ilp*, for computing minimum-size prime implicants of Boolean functions. We also compare these two SAT-based ILP algorithms with other ILP solvers, *lp-solve* [4], *opbdp* [2], and the commercial optimizer *CPLEX*. Moreover, the binate covering tool *scherzo* [5] is also evaluated, since minimum-size prime implicant computation can also be viewed as a restricted form of the binate covering problem (or alternatively as a 01-ILP). For this purpose we use a representative set of the

Benchmark	min	*CPLEX*		*lp-solve* [4]		*scherzo* [5]		*opbdp* [2]		*ls_ilp*		*bb_ilp*	
		UB	**T**	**UB**	**T**	**UB**	**T**	**UB**	**T**	**UB**	**T**	**UB**	**T**
aim-50-1_6-yes1-1	50	50	116.5	—	> 3,000	50	2.33	50	0.09	50	0.05	50	0.05
aim-50-2_0-yes1-2	50	50	109.5	—	> 3,000	50	5.65	50	0.64	50	0.02	50	0.03
aim-50-3_4-yes1-3	50	50	62.9	50	377.1	50	0.57	50	0.40	50	0.08	50	0.05
aim-50-6_0-yes1-4	50	50	26.9	50	96.8	50	0.73	50	0.48	50	0.07	50	0.04
aim-100-1_6-yes1-2	100	—	> 3,000	—	> 3,000	—	> 1,000	100	> 3,000	100	0.09	100	0.07
aim-100-2_0-yes1-3	100	—	> 3,000	—	> 3,000	100	691.57	100	42.45	100	0.17	100	0.11
aim-100-3_4-yes1-4	100	—	> 3,000	—	> 3,000	100	35.47	100	0.81	100	0.47	100	0.15
aim-100-6_0-yes1-1	100	100	294.3	—	> 3,000	100	2.78	100	0.18	100	0.32	100	0.11
aim-200-1_6-yes1-3	200	—	> 3,000	—	> 3,000	—	> 345	—	> 3,000	200	0.22	200	0.20
aim-200-2_0-yes1-4	200	—	> 3,000	—	> 3,000	—	> 1,705	—	> 3,000	200	0.83	200	0.71
aim-200-3_4-yes1-1	200	—	> 3,000	—	> 3,000	—	> 3,000	200	41.84	200	4.32	200	0.70
aim-200-6_0-yes1-2	200	—	> 3,000	—	> 3,000	200	619.38	200	4.59	200	3.58	200	5.18
ii8a1	54	54	63.3	54	786.90	54	0.98	54	1.93	54	861.53	54	9.34
ii8b2	—	388	> 3,000	474	> 3,000	—	> 3,000	—	> 3,000	379	> 3,000	379	> 3,000
ii8c2	—	629	> 3,000	668	> 3,000	—	> 3,000	—	> 3,000	525	> 3,000	525	> 3,000
ii8d2	—	588	> 3,000	—	> 3,000	—	> 3,000	—	> 3,000	540	> 3,000	540	> 3,000
ii8e2	—	653	> 3,000	—	> 3,000	—	> 3,000	—	> 3,000	494	> 3,000	494	> 3,000
jnh1	92	93	> 3,000	—	> 3,000	92	70.00	92	2.24	92	17.96	92	3.79
jnh7	89	90	> 3,000	—	> 3,000	89	5.35	89	0.45	89	9.06	89	0.91
jnh12	94	94	2,529	—	> 3,000	94	3.07	94	0.12	94	0.58	94	0.27
jnh17	95	95	873.9	—	> 3,000	95	17.28	95	0.30	95	2.53	95	0.77
ssa7552-038	—	1449	> 3,000	1450	> 3,000	—	> 223	1452	> 3,000	1448	> 1,205	1448	> 500

Table 1: Results on selected benchmarks

satisfiable instances of the DIMACS benchmarks [8], that are mapped to instances of the minimum size prime implicant problem. The experimental results, obtained on a SUN 5/85 machine with 64 MByte of physical memory, are shown in Table 1. For each benchmark and for each tool were allowed 3000 seconds of CPU time. Column **min** indicates the size of the minimum-size prime implicant, when this size is known. (Observe that for some of the benchmarks the minimum size prime implicant is still unknown.) In Table 1 and for each algorithm, column **T** denotes the CPU time and column **UB** denotes the computed upper bound on the minimum size prime implicant for each benchmark. When a given algorithm terminates, it reports the minimum size prime implicant if it was identified, otherwise the lowest computed upper bound is reported provided at least one upper bound was identified. For the results shown, whenever a tool quits earlier than 3000 sec, then the tool exceeded the available virtual memory (i.e. 64 MByte).

As can be concluded, general-purpose ILP solvers, such as *CPLEX* and *lp-solve*, are inadequate for computing minimum-size prime implicants. Similarly, despite the very promising results as an algorithm for solving binate cov-

ering problems [5], *scherzo* performs particularly poorly when computing minimum-size prime implicants. The three SAT-based ILP solvers can handle a large number of benchmarks and, in general, *ls_ilp* and *bb_ilp* perform better and are more robust than *opbdp*. For the JNH benchmarks, *opbdp* performs better because the amount of search is similar and the overhead of the underlying GRASP SAT algorithm is larger. One key drawback of *ls_ilp* derives from using an ILP layer around the SAT algorithm which creates large additional clauses. For the minimum-size prime implicant problem, these additional clauses involve *all* variables in the problem representation. Hence, conflicts involving this clause necessarily lead to chronological backtracking[2], and so the most useful features of GRASP [11] cannot be exploited. Finally, as we expected and as the experimental results confirm, *bb_ilp* tends to be a more efficient search algorithm than *ls_ilp*. From the obtained experimental results, it can also be concluded that the computation of the minimum-size prime implicant can be a particularly hard problem for specific sets of instances. This is the case, for example, with the ii8 and ssa7552 benchmarks.

6 Conclusions

In this paper we describe a new branch-and-bound algorithm, *bb_ilp*, for solving Integer Linear Programs (ILPs), that is based on backtrack search algorithms for Propositional Satisfiability (SAT) and that can incorporate powerful search pruning techniques commonly used in SAT algorithms. In addition, we describe another ILP algorithm, *ls_ilp*, which is based on the SAT-based ILP algorithm described in [2], but which implements more effective search pruning techniques. Both *bb_ilp* and *ls_ilp* are the first ILP algorithms that incorporate pruning techniques based on conflict diagnosis, thus implementing non-chronological backtracking, clause recording techniques and identification of necessary assignments. These pruning techniques are known to be effective in solving hard instances of SAT [3, 11], and consequently are targeted at highly constrained integer programs, for which the constraints are also hard to satisfy. One potential application of both algorithms is the computation of minimum-size prime implicants of Boolean functions, for which *bb_ilp*, *ls_ilp* and other SAT-based ILP algorithms are particularly well-suited for.

Despite the promising results given in the previous section for a specific application, SAT-based ILP algorithms still need to be significantly improved before becoming competitive with other ILP algorithms in more general application domains. Nevertheless, the preliminary results are promising, and are expected to continue to be so for application domains where most instances are highly constrained.

Future improvements to *bb_ilp* involve the incorporation of new search pruning techniques, including for example the ones commonly utilized in solving set covering problems [5]. Furthermore, improved lower bounding procedures, based on ***linear programming relaxations*** and ***lagrangian relaxations*** [1, 4, 9], are being developed.

References

[1] R. K. Ahuja, T. L. Magnanti and J. B. Orlin, *Network Flows: Theory, Algorithms, and Applications*, Prentice-Hall, 1993.

[2] 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. (source code for opbdp available from ftp://ftp.mpi-

2. In such a situation, each conflict involves *all* variables and so backtracking is necessarily chronological, to the most recent decision assignment [11].

sb.mpg.de/pub/guide/staff/barth/opbdp/opbdp.tar.Z.)

[3] R. Bayardo Jr. and R. Schrag, "Using CSP Look-Back Techniques to Solve Real-World SAT Instances," in P*roceedings of the National Conference on Artificial Intelligence (AAAI-97)*, 1997. (source code for rel_sat available from http://www.cs.utexas.edu/users/bayardo/bin/rel_sat.tar.Z.)

[4] M. R. C. M. Berkelaar, UNIX^{TM} Manual Page of lp-solve. Eindhoven University of Technology, Design Automation Section, 1992. (source code for lp_solve available from ftp://ftp.es.ele.tue.nl/pub/lp_solve/lp_solve_2.2.tar.gz.)

[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] J.N. Hooker, "Logic-Based Methods for Optimization," ORSA CSTS Newsletter, vol. 15, no. 2, pp. 4-11, 1994.

[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 from ftp://Dimacs.Rutgers.EDU/pub/challenge/sat/benchmarks/cnf/.)

[9] G. L. Nemhauser and L. A. Wolsey, *Integer and Combinatorial Optimization*, John Wiley & Sons, 1988.

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

[11] J. P. M. Silva and K. A. Sakallah, "GRASP—A New Search Algorithm for Satisfiability," in *Proceedings of the International Conference on Computer-Aided Design*, November 1996. (source code for GRASP available from http://algos.inesc.pt/pub/users/jpms/soft/grasp/grasp.tar.gz.)

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

[13] 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.