Effective Bounding Techniques For Solving Unate and Binate Covering Problems

Xiao Yu Li
Amazon
Seattle WA, USA
xiaoli@amazon.com

Matthias F. Stallmann
NC State University
Raleigh NC, USA
matt_stallmann@ncsu.edu

Franc Brglez
NC State University
Raleigh NC, USA
brglez@ncsu.edu

ABSTRACT

Covering problems arise in many areas of electronic design automation such as logic minimization and technology mapping. An exact solution can critically impact both size and performance of the devices being designed. This paper introduces *eclipse*, a branch-and-bound solver that can solve many covering problems orders of magnitude faster than existing solvers. When used in place of the default covering engine of a well-known logic minimizer, *eclipse* makes it possible to find, in less than six minutes, true minima for three benchmark problems that have eluded exact solutions for more than a decade.

Categories and Subject Descriptors

B.6 [**Logic Design**]: Optimization; G.1.6 [**Optimization**]: Integer Programming; G.2.1 [**Combinatorics**]: Algorithms

General Terms

Algorithms, Performance, Experimentation

Keywords

covering, branch and bound, satisfiability, unate, binate

1. INTRODUCTION

Solutions to unate covering problems (UCPs) have traditionally been driven by problems in two-level optimization of logic functions [1]. Notable examples of binate covering problems (BCPs) are those related to the minimization of Boolean relations, state minimization for finite-state machines, and cell library bindings. Whereas both problems are intractable, binate covering is more difficult in practice than unate covering. A good introduction to these problems can be found in [2] and [3]. Significant advances have been made recently in both problem domains: UCP [4, 5, 6, 7, 8] and BCP [5, 6, 9, 10, 11]. However, despite these advances in branch-and-bound algorithms, many current benchmarks remain unsolvable by the leading solvers.

DAC 2005, June 13–17, 2005, Anaheim, California, USA.

Instances of unate/binate covering are special cases of a Boolean constraint satisfaction problem known as *Min Ones* [12] or *minimum-cost satisfiability* (*MinCostSat*) [2]. Given m constraints on n Boolean variables, the goal is to find an assignment that satisfies all constraints while minimizing $\sum_{i=1}^{n} w_i x_i$, where $w_i \geq 0$ is the weight of variable x_i. While the problem can be formulated and solved as an integer programming (*IP*) problem, special purpose, domain-specific solvers introduced in the articles cited earlier are reported to have outperformed *IP* solvers such as lp_solve [13] and *cplex* [14].

We introduce *eclipse*, a new branch-and-bound solver that improves the state of the art for solving *MinCostSat* problems in general, and unate/binate problems in particular. Our analysis and experiments demonstrate that *eclipse*'s primary advantage lies in its handling of lower and upper bounds.

Comprehensive treatment of optimization algorithms for *MinCostSat* and the topics in this paper can be found in [15]. Specifically, as shown in [15], the *MinCostSat* formulation provides an umbrella for classifying not only the traditional unate/binate covering problems but also design automation problems that were previously considered in relative isolation, including FPGA detailed routing and generation of minimum-size test patterns and minimum-length plans.

An unexpected outcome of this research is the experience with *cplex* [14], a general-purpose integer-programming engine. Until now *cplex* has not been considered competitive by the design automation community for solving covering problems. Our results suggest that sophisticated cutting-plane techniques, added recently to *cplex*'s *IP* solver, make it a prominent contender.

An even more general formulation, *pseudo-Boolean* constraints, is proposed in [16]. The authors use constraint propagation and conflict learning, generalizing these techniques from satisfiability solvers. Constraint propagation is implicit in the reduction techniques used by *eclipse*, but conflict learning is orthogonal to all of the techniques we use and hence may be worth considering in the future. We have yet to compare *eclipse* to any variant of the solver in [16].

We rely on textbooks such as [2, 3] to give details on notation, branch-and-bound basics, and *IP*. Section 2 outlines the principal features of *eclipse*. Section 3 analyzes the key performance factors of a branch-and-bound algorithm. Comparison of four solvers on a subset of the hardest logic minimization benchmarks is reported in Section 4. The end of that section illustrates the impact of eclipse on logic minimization: it is there that we present results for *latte*, the

name we give to our version of *espresso* [17], in which *mincov* is replaced by *eclipse*. *Latte* solves three problems that have eluded other solvers for more than a decade, each in less than six minutes. Conclusions and open problems are summarized in Section 5.

2. THE ALGORITHM: AN OUTLINE

Eclipse is a branch-and-bound solver. But, instead of recursion it uses a priority queue to decide which branch to explore next, allowing many different search patterns. The *search tree* is a tree whose root represents the original problem instance. The interior nodes represent sub-instances – the original instance with some variables already assigned. The leaves represent complete variable assignments. Each interior node has two children, sub-instances in which an unassigned variable is assigned 0 and 1, respectively. The outline of the basic algorithm follows.

Step 1. Initialize the root of the search-tree with the original instance and set its lower-bound to 0. Put the root node on a priority queue.

Step 2. If the priority queue is empty, then return the global upper-bound and terminate. Otherwise, choose the most promising node from the priority queue. If the lower-bound of this node is $\geq$ the current global upper-bound, then discard the node and repeat step 2.

Step 3. Choose a branching variable and generate the two children of the node, setting the variable to 0 and 1. For each child, (a) apply reduction techniques; (b) calculate a lower-bound for the child – if the lower-bound is $\geq$ the global upper-bound or the sub-instance is infeasible, then discard the child; (c) do a local search at the child – if a feasible solution costing less than the global upper-bound is found, decrease the upper-bound to the new cost; (d) put the child on the priority queue. Repeat step 2.

This is the *eclipse* algorithm – the two variants, *eclipse-lpr* and *eclipse-cp*, discussed later, differ only in how the lower bound is computed in step 3(b). We developed the details of *eclipse* by experimentation, but later analyzed the factors that explain its success. Seven of them, discussed in Section 3 and shown in Fig. 1, have led to the performance improvements reported. Lower- and upper-bounding techniques affect the number of nodes explored by the search – better bounds increase the likelihood that nodes will be ignored in Step 2 or that children will be discarded in Step 3(b).

The use of a priority queue allows us to consider both traditional depth-first search (recursion) and priority search – choosing a node according to its likely contribution to a quicker solution. Branching-variable selection influences not only the order of exploration, but also the structure of the search tree: the values of some variables lead more quickly to sub-instance resolution.

Reduction techniques and search-tree pruning are peculiar to covering problems. Reduction removes redundant rows and columns from the constraint matrix. Pruning deduces that both children of a node can be discarded after the lower bound for one of them is computed. Finally, the data structure for the (sparse) constraint matrix is critical – it allows frequent operations to be done in constant time.

Our experiments rely on benchmark instances whose names and descriptions are introduced in earlier publications [1, 4, 9, 5, 6, 7, 8, 11]. Unless specified otherwise, all our experiments in this work are done on a Pentium IV@2.0Ghz with 1GB of RAM under Linux.

3. PERFORMANCE FACTORS

Lower- and upper-bounding techniques dominate both the execution time per node and the total number of nodes explored. Traditional algorithms for covering problems use a maximal independent set of rows as a lower bound and some combination of greedy and local-search methods to obtain an initial upper bound. We improve lower bounds by introducing techniques from *IP*. For upper bounds, we refine local search and apply it at every interior node, not just at the root. Other factors have considerably less impact and are discussed briefly at the end of the section.

Factor 1: Lower-Bounding Technique. Our comparison begins with a standard greedy heuristic for maximum independent set [5] (MIS1) and an enhanced version (MIS2) that increments the bound whenever a row and its overlapping rows cannot all be covered by a single variable (recall that the MIS bound is based on the idea that two rows with non-overlapping positive variables need at least two distinct assignments of 1).

In *IP* solvers the usual lower bound heuristic relaxes the integrality constraint on variable values, solving an *LP* (by, e.g., the simplex method) instead. The *LP* relaxation (LPR) yields poor lower bounds for many *0-1 IP* problems, but covering appears to be an exception.

LPR is further enhanced by adding *cutting-planes* (CP): the *IP* problem is solved in its LPR form; instead of stopping there (if the solution has fractional values) CP systematically generates new constraints, called Gomory cuts [18, 19], and adds them to the tableau (in the simplex method). The new constraints eliminate the current fractional solution (and perhaps others) but they never exclude any optimal integer solutions. Therefore, the new *LP* problem created is guaranteed to contain the same optimal integer solution as the original one. Its optimal objective value is no smaller than that of the LPR and sometimes larger.

CP is more time-consuming than LPR. The extra work comes from generating the Gomory cuts and solving the

Factor 1: Lower bounding Techniques
- *a.* MIS1 (maximum independent set)
- *b.* MIS2 (maximum independent set improved)
- *c.* LPR (linear programming relaxation)
- *d.* CP (cutting planes)

Factor 2: Upper bounding Techniques
- *a.* No local search
- *b.* Local search only at the root
- *c.* Local search at each node
  - Initialization (Random/Using lower bound solution)
  - Amount of Search (Fixed/Dependent on tree depth)
- *d.* Ask the Oracle

Factor 3: Search-Tree Exploration Strategy
- *a.* Depth-First Search
- *b.* Priority Search

Factor 4: Branching Variable Selection Heuristics
Factor 5: Reduction Techniques
Factor 6: Search Pruning
Factor 7: Data Structures

Figure 1: The seven performance factors in *eclipse*.

Table 1: Comparison of four lower bounding techniques on unate and binate covering benchmarks.

For each group of benchmarks, the minimum cost cover (opt) is shown in the second column. The linear programming relaxation (LPR) and cutting planes (CP) techniques never perform worse than the maximal independent set techniques, MIS1 and MIS2. CP outperforms LPR for benchmarks such as *exam.pi* in the unate group and *apex4.a* in the binate group. The differences from left to right are particularly significant for *test4.pi* (unate) and *count.b*, *apex4.a*, *rot.b*, and *e64.b* (binate). In the c* group (binate), the lower-bounds at the root are far from the optimum, especially for MIS1 and MIS2, because all but a few rows contain -1's and rows with -1's cannot be added to the independent set.

unate covering benchmarks

benchmark	opt	MIS1	MIS2	LPR	CP
lin_rom	120	115	115	120	120
exam.pi	63	52	55	60	62
bench1.pi	121	116	116	120	121
prom2	278	264	264	278	278
prom2.pi	287	273	273	287	287
max1024	245	236	238	244	244
max1024.pi	259	250	252	258	258
ex5.pi	65	60	60	64	64
ex5	37	32	32	36	36
test4.pi	≤ 101	56	56	80	80
m100_100_10_30	11	4	4	7	7
m100_100_10_15	10	4	4	8	8
m100_100_10_10	12	5	5	10	10
m200_100_10_30	11	3	4	8	8
m200_100_30_50	6	1	2	3	3

binate covering benchmarks

benchmark	opt	MIS1	MIS2	LPR	CP
count.b	24	17	17	24	24
clip.b	15	10	10	14	14
jac3	15	12	12	15	15
f51m.b	18	14	15	16	17
sao2.b	25	24	25	25	25
5xp1.b	12	9	9	11	11
apex4.a	776	525	525	756	773
rot.b	115	95	98	111	114
alu4.b	50	38	39	47	47
e64.b	≤ 48	32	32	37	40
c432_F37gat@1	9	1	1	3	3
misex3_Fb@1	8	1	1	2	2
c1908_F469@0	11	1	1	4	4
c6288_F69gat@1	6	1	1	2	2
c3540_F20@1	6	1	1	3	3

new *LP* problem. But CP sometimes provides much better lower bounds than the other methods. For example, at the root of the search tree for the binate instance *apex4.a*, the four lower-bounding methods provide lower bounds of 525 (MIS1), 525 (MIS2), 756 (LPR) and 773 (CP), respectively. Table 1 shows lower bounds achieved by each of the four methods on the unate and binate benchmarks and the optimum solution, if known.

Factor 2: Upper-Bounding Technique. The effectiveness of branch and bound is regulated by two bounds: a local lower bound and a global upper bound. Whenever the lower bound at a node is $\geq$ the global upper bound, that node is discarded and the search subtree below it is pruned. *Eclipse* puts much effort into lower bounds, but equally important is the calculation of good upper bounds. The *upper bound* is defined as the cost of the best solution found so far. Eventually it is the cost of the optimal solution. A good upper-bounding strategy finds an optimal solution as quickly as possible, thus pruning as many nodes as possible. However, a near-optimal upper bound allows pruning of many nodes whose lower bound exceeds that of the root, and may be almost as good. We list four sources of upper bounds, in order of increasing effectiveness.

1. **Search-tree leaf:** This is the most naive method – there is no explicit search for an upper-bound. The only update of the upper-bound (initially ∞) occurs when a feasible solution is found at a leaf of the search tree. Performance of this approach is unreliable, as it depends heavily on where, with respect to the search order, the best solutions reside. In the worst case, the good solutions that cause pruning are not found until nearly all nodes have been searched.
2. **Local search at the root:** Many two-phase branch-and-bound solvers use this method – the first phase finds a good upper bound at the root of the search tree, and all future pruning relies on the relationship of lower bounds to this upper bound (or better ones encountered at the leaves). A solver using this method is more effective than one that only uses leaves. Good or even optimal solutions for many benchmarks can be found relatively quickly using local search.
3. **Local search at each node:** If local search works well at the root, why not use it at other interior nodes? The fact that some variables are already assigned often forces the search into previously unexplored territory. The benefits of this additional coverage can be quite dramatic (see, for example, max1024 in Table 2).
4. **Ask the oracle:** If the cost of the optimal solution is given a priori as an upper bound, we get the maximum possible pruning. We can use this method as a reference to see how well each of the other methods is doing.

Local search starts with an initial assignment, chosen randomly or by some heuristic. Variable values are changed one at a time, either to decrease the number of violated constraints or to decrease cost.[1] The search continues for a specified number of steps (instead of stopping at a 'local optimum'). Choosing this number is not easy.

- Too many steps increase processing time at each node and slow *eclipse* down.
- Too few steps risk overlooking an opportunity to improve the bound.

The search space is cut in half as each new variable is assigned, so it makes sense to halve the number of search steps for each branch taken relative to the root. In *eclipse* the number of search steps at any node is $(1/2)^d * c_f * S$, where d is the depth of the node (number of tree edges from the root), c_f is a constant,[2] and S is the size (number of nonzero coefficients) of the constraint matrix. Along any search-tree path from the root toward a leaf, the number of local-search steps decreases exponentially (with some additional decrease due to shrinking matrix size).

Table 2 summarizes the results of experiments with different upper-bounding methods used in combination with

[1] *Eclipse* uses techniques borrowed from *gsat* [20], *walksat* [21], and tabu search [22].

[2] We use $c_f = 5$, a value, arrived at by trial and error, that appears to work well across different problem instances.

Table 2: Comparison of four different upper-bounding methods with *eclipse*.

From left to right, results are shown (wrt local search) for (1) no search at all, (2) search at the root, (3) search at each node, and (4) using a known optimum (oracle). The mean and standard deviation for runtime and nodes are reported, based on 32 runs with different random seeds. Local search at every node is much better than search only at the root and not much worse than knowing the optimum ahead of time.

	none		root		each node		oracle	
benchmark	time	nodes	time	nodes	time	nodes	time	nodes
exam.pi	300/0*	2355/89	7.74/5.53	15/24	5.4/5.0	12/22	4.4/3.9	10/19
bench1.pi	300/0*	609/15	211.0/136.2	447/293	4.7/1.4	7/4	3.8/1.8	3/2
max1024	251.4/15.6	263/45	148.4/80.7	148/81	27.5/18.7	27/20	17.8/1.7	11/1
ex5	300/0*	97/3	25.3/33.4	6/11	15.1/7.7	5/3	3.7/0.2	1/0
5xp1.b	26.6/3.5	594/92	2.84/1.26	9/5	3.1/1.5	10/7	2.5/1.8	7/6
jac3	10.8/1.6	130/70	8.3/0.6	3/0	8.1/1.0	3/0	2.4/0.3	1/0

* times out at 300 seconds.

the best possible lower bound (CP). Mean and standard deviation are based on 32 runs of each instance/algorithm combination with different random seeds.

An important factor in local search is the starting assignment. While the search itself is randomized in the results reported in Table 2, the starting assignment is not. Linear programming solutions can be close to the integral optimum and, if rounded to the nearest integer (0 or 1 in our case), can yield much better than random starting assignments. When a CP lower bound is calculated, the dual simplex method yields a not-necessarily feasible integer 'solution' (a feasible solution to the dual problem) that can be used directly. In the table below, we see the benefit of choosing a starting assignment based on CP (the *heuristic init* columns) versus how a random initial assignment (*random-init*) would perform under the same circumstances (those of Table 2).

	random init		heuristic init	
benchmark	time	nodes	time	nodes
exam.pi	20.8/50.0	117/474	5.4/5.0	12/22
bench1.pi	300/0*	597/5	4.7/1.4	7/4
max1024	143.9/86.6	303/333	27.5/18.7	27/20
ex5	16.3/5.3	4/1	15.1/7.7	5/3
5xp1.b	3.0/1.5	10/7	3.1/1.5	10/7
jac3	8.7/1.5	3/1	8.1/1.0	3/0

Other Factors. Another factor for which we report comparisons is tree exploration. A well-chosen node exploration sequence can lead quickly to good upper bounds and pruning. In *eclipse*, all unexplored nodes are kept on a priority queue and the node to be explored next has the *minimum* lower bound. Intuitively, an optimal solution is more likely to reside below such a node in the search-tree. The table below compares *eclipse*'s strategy, priority search (best FS), with depth-first search (depth FS). Overall, priority search has better performance if all other factors are equal.

	depth FS		best FS	
benchmark	time	nodes	time	nodes
exam.pi	7.0/5.4	15/23	5.4/5.0	12/22
bench1.pi	7.91/12.0	12/21	4.7/1.4	7/4
max1024	74.1/27.0	68/43	27.5/18.7	27/20
ex5	16.2/6.7	4/2	15.1/7.7	5/3
5xp1.b	3.0/1.4	10/7	3.1/1.5	10/7
jac3	8.6/1.5	3/1	8.1/1.0	3/0

The remaining performance factors are branching variable selection, reduction, search pruning, and data structures. *Eclipse* uses a criterion similar to the one in [9], modified using the *LP* solution, for selecting a branching variable. For reductions, essentiality, row dominance and column dominance [3] are used. Search pruning in *eclipse* again borrows from [9] (the 'C_l lower bound'). Doubly-linked lists of both rows and columns implement the sparse matrix [17] so that reduction operations can be performed efficiently.

4. EXPERIMENTAL RESULTS

With every factor except for the lower bound we ended up with a clear-cut decision about what method to use in *eclipse*. The tradeoff between the LPR lower bound, which takes less time to compute, and the CP lower bound, which may be significantly better, is benchmark-dependent. We therefore include both *eclipse-lpr* and *eclipse-cp* in the results reported. Both search for an upper bound using a non-random starting assignment at every node. Both also use priority search and the other factor decisions mentioned at the end of the previous section. The only difference is the use of the LPR versus the CP lower bound, respectively.

We chose *scherzo* [5] to represent solvers designed specifically for covering problems – it was able to solve both unate and binate problems at least as well as solvers such as *aura*II [7] and *bsolo* [11] (except that *aura*II was able to solve ex5.pi and ex5 in about 7 minutes each, instead timing out at one hour). *Cplex* [14], the gold standard of general-purpose *IP* solvers, is version 7.5 using default settings. Recent experiments with version 9.0 and a variety of different settings did not yield significantly different results.

Unate/Binate Covering Results. Table 3 shows results on various logic minimization benchmarks. In each category, unate and binate, they are sorted by the number of non-zeros in the covering matrix. The unate table shows the percentage of variables whose value is 1 in an optimum solution – a larger value here puts *eclipse-lpr* at a disadvantage for two reasons: (a) priority search tends to favor exploration of the 0-branches in the search tree, and (b) *eclipse-lpr*'s starting assignment for local search, based on *LP* rounding, usually has fewer 1's than the dual-feasible solution used by *eclipse-cp*.

In the binate table we show the percentage of non-zero entries that are -1, i.e., the extent to which the instance deviates from being unate. More -1's favor *cplex* because they render useless most of the special-purpose techniques for covering problems.

Table 3: Comparison of five solvers on a logic minimization subset of unate/binate covering benchmarks.

The benchmarks used to compare solvers below are from logic minimization. *Scherzo* uses MIS-based lower bounds and is competitive primarily on small binate benchmarks that have nearly optimal MIS bounds, such as sao2.b and f51m.b (see Table 1). *Cplex*, *eclipse-lpr*, and *eclipse-cp*, have similar runtime. The results given for *eclipse-lpr* and *eclipse-cp* are for typical runs exhibiting average behavior. As seen in Table 2 and other factor comparisons, the standard deviation on 32 runs can be large. *Eclipse-cp* almost always has the smallest number of nodes and either the shortest or close to the shortest runtime. *Cplex*, however, gains the upper hand on sao2.b, jac3, rot.b, and apex4.a, which have the highest percentage of −1's. *Eclipse-lpr* manages to run faster than at least one of the other two when the LPR and CP bounds are identical and close to optimal; the exceptions to this are max1024 and max1024.pi, which are unusual in having an optimum solution with about 20% of the variables set to 1.

Unate benchmarks: runtime comparisons

benchmark	max1024.pi	max1024	bench1.pi	prom2	prom2.pi	exam.pi	ex5.pi	ex5	test4.pi
non-zeros	6974	7221	9563	15507	15545	25694	40681	41085	109318
% 1's in opt.	20.3	19.4	2.6	10.6	11.0	1.3	2.6	1.5	< 0.1
scherzo	1749.3	224.3	1349.8	376.9	650.5	3600*	3600*	3157.9	3600*
cplex	21.1	18.6	4.4	5.2	6.0	21.0	25.6	65.6	3600*
eclipse-lpr	53.0	39.3	3.8	2.2	2.4	162.2	7.7	18.2	3600*
eclipse-cp	18.0	15.9	2.2	8.0	6.1	3.1	9.0	11.3	3600*

Unate benchmarks: comparisons of number of nodes

benchmark	max1024.pi	max1024	bench1.pi	prom2	prom2.pi	exam.pi	ex5.pi	ex5	test4.pi
scherzo	414030	533635	2001438	25865	23585	–	–	615187	–
cplex	119	118	10	1	1	1379	104	550	–
eclipse-lpr	282	224	32	1	1	2768	9	29	–
eclipse-cp	16	11	4	1	1	2	3	3	–

Binate benchmarks: runtime comparisons

benchmark	sao2.b	f51m.b	count.b	jac3	5xp1.b	rot.b	apex4.a
non-zeros	12820	13397	17335	24011	29889	40755	57595
% −1's	5.6	3.6	3.6	9.2	2.7	6.5	19.1
scherzo	0.4	0.9	333.7	2.6	2.1	3600*	87.4
cplex	1.6	1.9	0.7	1.5	5.0	62.0	9.9
eclipse-lpr	3.1	3.0	0.7	2.6	2.3	3600*	3600*
eclipse-cp	3.5	2.7	1.8	7.8	4.1	370.0	29.5

Binate benchmarks: comparisons of number of nodes

benchmark	sao2.b	f51m.b	count.b	jac3	5xp1.b	rot.b	apex4.a
scherzo	285	1562	1429	294	2661	–	33185
cplex	236	199	20	7	258	3001	587
eclipse-lpr	64	140	1	3	128	–	–
eclipse-cp	14	37	1	3	36	735	77

* Solver times out at 3600 seconds.

As already noted, lower bounds are the most important factor in determining not only the number of nodes visited, but runtime as well. This causes *eclipse-cp* to do at least as well as *cplex* in most cases. Unfortunately *eclipse-cp* solves a new *LP* and recomputes cutting planes at every node, whereas *cplex* reuses cuts in the sub-instances and incorporates variable assignments incrementally. This is seen in the results on the larger binate instances.

The importance of upper bounds can also be seen in the table. The main advantage that *eclipse-cp* has over *cplex* is the use of local search at every node. Without the exponential decay based on search-tree depth, the time spent doing local search would overwhelm any advantage gained.[3]

Logic Minimization. To illustrate the importance of having an improved set-covering solver, we took *espresso* [17], a well-known two-level logic minimizer, and replaced its unate-covering solver, *mincov*, with *eclipse-cp*, calling the result *latte*. *Espresso* can run as a fast heuristic if an optimal solution is not required. Our experiment, however, used *espresso* in *exact* mode.[4] The table below shows that *latte* finds exact solutions of several logic minimization benchmarks that had not previously been solvable. In particular, each of the three runs of *latte* took less than six minutes.

	espresso (hours)		*latte* (secs)	
benchmark	cover	total	cover	total
ex5	–	12*	129.58	139.24
max1024	–	12*	329.11	329.5
prom2	–	12*	12.97	14.75

* times out at 12 hours.

The resulting solutions are also noticably better with respect to number of product terms than those produced by *espresso* in its *heuristic* mode. This suggests that the effort to find exact solutions is not wasted.

	orig		*espressoheur*		*latte*	
benchm	prod	lit	prod	lit	prod	lit
ex5	256	9668	74	1903	65	1193
max1024	1024	13472	274	2266	259	2207
prom2	287	5610	287	5526	287	5528

[3]In fact, *cplex* has an option to allow local search periodically, but for a fixed number of iterations that is independent of problem size or node depth. Experiments with this option did not yield any improvement.

[4]Specifically, `espresso -Dexact`, which minimizes the number of product terms as its primary objective, see, e.g., [3, p. 217]. *Latte* is identical to *espresso* except that *eclipse* is used in place of *mincov*.

5. CONCLUSIONS

We showed that by careful analysis of factors and experimental design, we can obtain a finely-tuned branch-and-bound covering solver that is able to yield significant performance improvement over previous special-purpose solvers. The role of *IP* solvers, such as *cplex* is more important than we initially realized. Future covering solvers will have to take advantage of *IP* techniques, covering-specific techniques, and perhaps techniques from satisfiability (e.g., those of [16]).

Future work on *eclipse* will focus on improving the performance of our lower bound engine and possibly applying it more selectively instead of at each node. We know from profiling that lower bound computation accounts for nearly all of the runtime of both *eclipse-lpr* and *eclipse-cp* on larger benchmarks. Soon to be released is a version that uses our own dual simplex engine instead of the one provided by *cplex*. Besides the obvious advantage of not being dependent on *cplex*, this also gives us finer control over the number of dual-simplex iterations performed at each node. Since intermediate data from the tableaux can yield both lower bounds and hints for variable selection, we need not wait for the *LP* to reach optimality.

Initial experiments with larger benchmarks suggest that space, as well as time, might be a limiting factor, given a large explosion in number of nodes on the queue (each node has its own copy of the constraint matrix). This can be mitigated by storing only the current assignment at each node instead of the whole matrix, but incurs an additional substantial penalty in time. A smooth tradeoff between space and time is clearly worth looking into.

Acknowledgments. Results of comparative experiments reported in this paper rely not only on benchmarks in the public domain but also on state-of-the-art unate/binate cover solvers that were genereously shared by several researchers cited in this paper. We shall follow suit in similar vein with our solvers *eclipse* and *latte* – once the experiments for the journal version of this paper have been completed and organized similarly to the schema described in [23]. We also thank the anonymous reviewers for constructive comments that helped us clarify the presentation in this paper.

Researchers interested in the status of current versions of these solvers and data sets can contact the authors directly.

6. REFERENCES

[1] R. Rudell and A.L. Sangiovanni-Vincentelli. Multiple-valued optimization for PLA optimization. *IEEE Transactions on CAD/ICAS*, CAD-6(5):727–750, September 1987.

[2] G. D. Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Publishers, 1994.

[3] G.D. Hachtel and F. Somenzi. *Logic Synthesis and Verification Algorithms*. Kluwer Academic Publishers, 1996.

[4] P.C. McGeer, J.V. Sanghavi, R.K. Brayton, and A.L. Sangiovanni-Vincentelli. Espresso-signature: A new exact minimizer for logic functions. In *Proceedings of the 30th Design Automation Conference*, pages 618–624, 1993.

[5] O. Coudert. On solving covering problems. In *Proceedings of the 33rd Design Automation Conference*, pages 197–202, 1996.

[6] S. Liao and S. Devadas. Solving covering problems using lpr-based lower bounds. In *Proceedings of the 34th Design Automation Conference*, pages 117–120, 1997.

[7] E.I. Goldberg, L.P. Carloni, T. Villa, R.K. Brayton, and A.L. Sangiovanni-Vincentelli. Negative thinking in branch-and-bound: the case of unate covering. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19:1–16, 2000.

[8] R. Cordone, F. Ferrandi, D. Sciuto, and R. Wolfler. An efficient heuristic approach to solve the unate covering problem. In *Proceedings of the Design, Automation and Test in Europe*, 2000.

[9] O. Coudert and J.C. Madre. New ideas for solving covering problems. In *Proceedings of the 31st Design Automation Conference*, 1995.

[10] T. Villa, T. Kam, R.K. Brayton, and A.L. Sangiovanni-Vincentelli. Explict and implicit algorithms for binate covering problems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16:677–691, 1997.

[11] V.M. Manquinho and J.P. Marques-Silva. Search pruning techniques in SAT-based branch-and-bound algorithms for the binate covering problem. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21:505–516, 2002.

[12] S. Khanna, M. Sudan, L. Trevisan, and D. P. Williamson. The approximability of constraint satisfaction problems. *SIAM J. Comput.*, 30(6):1863–1920, 2000.

[13] M. Berkelaar. FTP site for lp_solve, 2004. Source code is available at `ftp://ftp.es.ele.tue.nl/pub/lp-solve`.

[14] ILOG. CPLEX Homepage, 2004. Information on CPLEX is available at `http://www.ilog.com/products/cplex/`.

[15] X. Y. Li. *Optimization Algorithms for the Minimum-Cost Satisfiability Problem*. PhD thesis, Computer Science, North Carolina State University, Raleigh, N.C., August 2004. This thesis is accessible at `http://www.lib.ncsu.edu/theses/-available/etd-10072004-021218/`.

[16] D. Chai and A. Kuehlmann. A fast pseudo-Boolean constraint solver. In *Proceedings of the 40th Design Automation Conference*, pages 830–835, June 2003.

[17] R.L. Rudell. Logic synthesis for VLSI design. *Ph.D. Dissertation, Department of EECS, University of California at Berkeley*, 1989.

[18] R.E. Gomory. Outline of an algorithm for integer solution to linear programs. *Bulletin of the American Mathematical Society*, 64:275, 1958.

[19] R.E. Gomory. An algorithm for the mixed integer problem. *RM-2537. Santa Monica California: Rand Corporation*, 1960.

[20] B. Selman, H.J. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In P. Rosenbloom and P. Szolovits, editors, *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 440–446, Menlo Park, California, 1992. AAAI Press.

[21] D.A. McAllester, B. Selman, and H. Kautz. Evidence for invariants in local search. In *Proceedings of AAAI/IAAI*, pages 321–326, 1997.

[22] F. Glover. Future paths for integer programming and links to artificial intelligence. *Computers and Operations Research*, 5:533–549, 1986.

[23] F. Brglez, X. Y. Li, and M. F. M. Stallmann. On SAT instance classes and a method for reliable performance experiments with SAT solvers. *Ann. Math. Artif. Intell.*, 43(1):1–34, 2005.