High-Contrast Algorithm Behavior: Observation, Conjecture, and Experimental Design

Matthias F. Stallmann* Franc Brglez*

ABSTRACT

After extensive experiments with two algorithms, CPLEX and our implementation of all-integer dual simplex, we observed extreme differences between the two on a set of design automation benchmarks. In many cases one of the two would find an optimal solution within seconds while the other timed out at one hour.

We conjecture that this contrast is accounted for by the extent to which the constraint matrix *can be made* block diagonal via row/column permutations. The actual structure of the matrix without the permutations is not important.

Our conjecture is made more precise in two steps: (a) crossing minimization is used on a derived graph to achieve desirable permutations of rows and columns; and (b) the degree of randomness (lack of structure) is measured using *diffusion*, a measure that approximates what a human perceives as lack of structure.

Additional experiments on synthetic instances related to the benchmarks add validity to our conjecture. We observe unexpectedly sharp thresholds where, with only slight variation of our measure, the dominance of the algorithms reverses dramatically. The nature of and explanation for this threshold behavior is left for future research as are many other questions. As far as we are aware the approach taken here is unique and, we hope, will inspire other research of its kind.

1. INTRODUCTION

The objects of the study in this paper are two extremely different algorithms, each too complex to be treated as much more than a black box. The industrial benchmarks on which the behavior of these algorithms has been observed fall into three categories: (a) both algorithms find the optimal solution within a few seconds; (b) both fail to find the optimum after one hour or longer; and (c) one algorithm clearly dominates the other when runtime is used as a measure of merit. This discussion considers category (c) only.

The benchmarks represent generalized set cover problems arising in logic synthesis for design automation. One of the algorithms is CPLEX 9.0 [12][1], a general-purpose solver for integer programs, including set cover. The other, called *int-dual*, is our implementation of all-integer dual simplex — see [7, Sec. 5.9], specialized for this problem domain.

The focus is on methodology. Ideally, we could profile problem instances and choose the better algorithm. Our study is a limited step in that direction. Observations in the form of experimental data from the benchmark set — we only show those relevant to category (c) — lead to a conjecture about the structure of instances that work better for int-dual. We then design extensive experiments to build a case for our conjecture. Though there are several limitations to our results, both the experimental evidence and a rudimentary understanding of the algorithms support the conjecture.

Now that we have presented an outline of the story to be told, the remainder of the paper is organized as follows. Section 2 gives a brief description of the background leading to this study. In Section 3 we give an overview of both algorithms. Section 4 describes the benchmark instances and the results that make up our observations. Section 5 is the turning point where we lead up to our main conjecture. In Section 6 the details of additional experiments supporting our conjecture are explained. Some final thoughts conclude the paper in Section 7.

2. BACKGROUND

The *(unate) set cover* problem has input consisting of a family $\mathcal{C} = \{C_1, \ldots, C_n\}$ and a set $S \subseteq C_1 \cup \cdots \cup C_n$. The goal is to find a sub-family $\mathcal{C}' = \{C'_1, \ldots, C'_k\}$ of $\mathcal{C}$ so that $S \subseteq C'_1 \cup \cdots \cup C'_k$ and k is minimized. *Binate cover* is set cover with additional constraints, usually ones that ensure mutual exclusivity or inclusiveness among some of the C'_j.

Both are special cases of *minimum cost satisfiability* [5], also known as *min-ones satisfiability* [13], where a cnf formula is to be satisfied in such a way as to minimize the number of true variables (or, more generally, their cost).

As an example of a unate instance, consider four employees, each possessing a subset of three skills: financial (f), managerial (m), and computer (c). The skill set for employee 1, C_1, is $\{f, c\}$, financial and technical; $C_2 = \{m, c\}$; $C_3 = \{m\}$; and $C_4 = \{f, m, c\}$. The employer has to downsize the enterprise, retaining a minimum number of employ-

*Department of Computer Science, North Carolina State University, {matt_stallmann,brglez}@ncsu.edu

[1]We subsequently use the lower case "cplex" except when referring to specific releases.

ees while ensuring that each skill is still represented. The obvious solution with $cost = 1$ is to keep employee 4.

If x_i is a 0/1 variable that is 1 iff employee i is retained, the constraints of the problem are given by three inequalities, one for each skill:

$$\begin{aligned} x_1 + x_4 &\geq 1 \quad \text{(f)} \\ x_2 + x_3 + x_4 &\geq 1 \quad \text{(m)} \\ x_1 + x_2 + x_4 &\geq 1 \quad \text{(c)} \end{aligned}$$

A binate instance arises when employee 4 decides to stay only if the other employees are kept as well (not a smart decision, but noble). Now the best solution has $cost = 2$: the employer can retain employee 1 and either of 2 or 3. The additional constraints are

$$\begin{aligned} -x_4 + x_1 &\geq 0 \quad x_4 \rightarrow x_1 \quad \sim x_4 \vee x_1 \\ -x_4 + x_2 &\geq 0 \quad x_4 \rightarrow x_2 \quad \sim x_4 \vee x_2 \\ -x_4 + x_3 &\geq 0 \quad x_4 \rightarrow x_3 \quad \sim x_4 \vee x_3 \end{aligned}$$

The instance is called binate because the coefficients in the inequalities can be either $+1$ or -1. The connection with satisfiability is illustrated in the comments to the right of each constraint. Each constraint, whether unate or binate, has $1 - c_n$ on the right hand side, where c_n is the number of negative coefficients.

See [16] for a further discussion of unate cover, binate cover, and min-cost sat. Applications of these problem formulations to design automation (logic synthesis) are presented in [5] and [11].

This work began with an earlier branch and bound algorithm, *eclipse* [16, 15], which outperformed cplex on most of the unate benchmarks. The binate benchmarks, however, were more challenging: eclipse either did not do as well as cplex or neither algorithm was able to find a solution within an hour[2]. The biggest challenge, as we saw it, was to develop a better mechanism for obtaining lower bounds. An all-integer dual simplex algorithm seemed promising because (a) it had already been tried successfully on small set cover instances [18], and (b) any intermediate step provided a lower bound even if it did not prove optimality. To our surprise, consideration (b) turned out to be superfluous — int-dual, when used to compute a lower bound, actually proved optimality, and made branching unnecessary.

This surprising success applied only to the hard binate benchmarks. Int-dual performed very poorly on the unate ones. The many attempts to explain this dichotomy, and also why cplex performed poorly on benchmarks where int-dual did well, led to the current work. We proceed by giving a (slightly) more detailed description of the two algorithms.

3. INTEGER PROGRAMMING FORMULATION AND ALGORITHMS

The constraints of a unate or binate set cover instance have an integer programming (IP) formulation

$$\min c^T x \text{ subject to } Ax \geq b, \qquad x \in \{0,1\}$$

where c is the cost vector, all 1's in this setting; the *constraint matrix* A and bound vector b are determined as follows. Each row in A is the left side of a constraint, each entry of b the right side. When the instance is unate the A entries are either 0 or 1 and b is all 1's.

When the x_i are allowed to be fractional, the formulation is called the linear-program (LP) relaxation of the IP and can be solved using a simplex algorithm. Suppose, for example, that there are three variables and three constraints (this is the vertex cover instance for a triangle):

$$\begin{aligned} x_1 + x_2 &\geq 1 \\ x_2 + x_3 &\geq 1 \\ x_1 + x_3 &\geq 1 \end{aligned}$$

The optimal solution, allowing fractions, is $x_1 = x_2 = x_3 = 1/2$ with cost 3/2. However, the optimal integer solution must have two of the three variables = 1 for a cost of two. After the fractional solution is found by a simplex method, one approach is to apply a *Gomory cut* [9, 10], a new constraint that eliminates the fractional solution but keeps all optimal integer solutions of the original instance. Such a cut is also called a *fractional cut*.

The all-integer dual algorithm, int-dual, anticipates each cut before a fractional solution arises. All of the tableau entries [3] at each step are integers and the numerical issues associated with floating point computations are avoided.[4] Int-dual begins with a tableau based directly on the constraint matrix A. Every step of int-dual (and any simplex method) is a *pivot* — a variable is swapped for a constraint and some rows of the tableau are added to others. A pivot entry of the tableau determines the row and column to be swapped. The only changes to the tableau involve rows that have a nonzero in the pivot column.

The dual simplex method is used because it is easier to implement and appears to be more promising for set cover problems — see [18].

To give a rough idea of how the algorithm works, consider Table 1 which illustrates the solution of the three-variable instance above. The initial tableau has costs, all 1's here, across the top. Each row is the negative of a constraint, written with the b-vector entry on the left. The top left corner represents the negative of a lower bound on the solution.

Success with all-integer dual simplex algorithms has only been reported a few times in the literature [18, 4], and, as is the case with our work, in a limited domain.

The other algorithm, cplex, uses branch and bound — see, e.g. [14, Ch. 10]. The basic algorithm starts with a root node that represents the initial instance. Every node has two children, one representing the reduced instance that arises when a given variable is set to 0, the other representing a value of 1 for that variable. The implied exhaustive search of the solution space is mitigated in several ways.

- The solution to the LP-relaxation at a node may give a lower bound that is no better than the cost of an already-identified solution; the node can be discarded and no children are generated.

[2]This was CPLEX 7.5 on a slower processor (by about a factor of two).

[3]A tableau keeps track of the current step in the simplex algorithm; the idea is attributed to Beale, but its first known publication is in [8]. Any linear programming textbook has more details — see, e.g., [20].

[4]The C++ implementation is available by request and will be posted on the web as soon as it is properly documented. Scripts for running experiments, instances, instance generators and instance classes, and detailed results, are also available.

Table 1: Initial tableau and sequence of pivots for the vertex cover problem of a triangle

		$-x_1$	$-x_2$	$-x_3$
$-sol$	0	1	1	1
s_1	-1	[-1]	-1	0
s_2	-1	-1	0	-1
s_3	-1	0	-1	-1

$\Longrightarrow$

		$-s_1$	$-x_2$	$-x_3$
$-sol$	-1	1	0	1
x_1	1	-1	1	0
s_2	0	-1	1	-1
s_3	-1	0	[-1]	-1

$\Longrightarrow$

		$-s_1$	$-s_3$	$-x_3$
$-sol$	-1	1	0	1
x_1	0	-1	1	-1
s_2	-1	-1	1	[-2]
x_2	1	0	-1	1

$\Longrightarrow$

		$-s_1$	$-s_3$	$-x_3$
$-sol$	-1	1	0	1
x_1	0	-1	1	-1
s_2	-1	-1	1	-2
x_2	1	0	-1	1
s_{cut}	-1	-1	0	[-1]

- A local search algorithm applied to the instance at a node may give a solution that has cost no greater than lower bounds of nodes still under consideration. Those nodes can be eliminated.
- A fractional cut can be used to eliminate potential solutions for all nodes simultaneously.

Cplex uses all of these standard devices plus other heuristics determining the choice of a variable at each node, the choice of a node to explore next, how often to apply local search, etc.

4. OBSERVATIONS: BENCHMARKS AND RESULTS

Table 2 shows profile data for a subset of some well-known benchmark problems based on logic synthesis. These were originally contributed to a 1991 workshop at MCNC in Research Triangle Park, NC – see [22] – and have been the subject of intense experimentation since, mostly with branch and bound algorithms. Up to a point, optimal solutions outweigh runtime considerations. Recent results with pointers to previous ones can be found in [15] and [17]. The chosen subset omits instances that are too easy — runtimes are less than a second for most recent algorithms — and one that is hard mainly because it is considerably larger than the others.[5]

Instance sizes differ from those reported in other sources because these are *reduced* versions of the instances, modified using *essentiality* (eliminating unit constraints and assigning the appropriate variables), *row dominance* (removing redundant rows), and *column dominance* (removing redundant columns). See [11] for more details. Since our algorithms — including int-dual — preprocess instances to reduce them, the use of reduced instances levels the playing field with respect to other algorithms.

In some cases the reduced instance has a smaller-valued optimum solution because some variables are forced to have a value of 1 during the reduction. This accounts for the differences in optima. The LP lower bounds (relative to known optima) are not affected by reduction.

The columns marked *row-diff* and *col-diff* will be discussed later — they are an essential part of our main conjecture.

[5]The optimal solution cost for this benchmark, test4.pi, is known only to be ≤ 92 (using stochastic search) and ≥ 82 (using cplex).

Table 3 shows results for the two algorithms on the six benchmarks under consideration. Cplex is run using default settings and int-dual is as described earlier.

All reported results give runtime on a dedicated Intel(R) Xeon(TM) 3.20GHz processor with 2048 MB cache. Runs were terminated after one hour if the instance was not solved by then. For int-dual, there was also the possibility of an overflow when the integers in the tableau became too large..

In these tables we only report runtime because that is the only measure that can be compared meaningfully. Runtime for cplex usually correlates well with the total number of simplex iterations if normalized for the size of the instance but it may also be affected by the number of cuts. There are multiple factors influencing the time taken by int-dual, no one of which is a reliable indicator.

As we noted in [3] it can be drastically misleading to compare results of different algorithms on single benchmark instances. For statistically significant results, we ran each algorithm on 33 instances: the original benchmark as posted (and reduced) and 32 *isomorphs* obtained by randomly permuting rows and columns of the constraint matrix.

On five of the instances one of cplex or int-dual so clearly dominates the other that detailed statistics are hardly relevant — the maximum time taken by one is less the minimum time of the other.

Remark: The four overflows observed in the int-dual runs for e64.b can be discounted. The maximum time it takes to encounter an overflow is 497 seconds.[6] When an overflow occurs, int-dual can randomly permute the matrix and run again. Even in the worst case, at least seven such repeated runs can be accomplished in an hour — the probability that all seven encounter overflows is infinitesimal.

Cplex dominates on bench1 and max1024, while int-dual performs significantly better than cplex on saucier, e64.b, and alu4. The only controversial benchmark is rot.b where the distribution for cplex is exponential — standard deviation close to the mean — and that for int-dual is near exponential. Whether cplex dominates in this case or we regard the algorithms as indistinguishable is not important to the remainder of our discussion.

The fact that cplex and int-dual complement each other can be useful — running the two concurrently can greatly improve the chances of obtaining an optimal solution within reasonable time. Our purpose here, however, is to attempt

[6]The other three are 75, 146, and 167 seconds, respectively.

Table 2: Benchmark problems, original and reduced.

The benchmarks used here are *reduced* versions of ones that were originally contributed to a 1991 workshop at MCNC in Research Triangle Park, NC – see [22]. Optima were computed using umbra, our branch-and-bound algorithm that uses int-dual to obtain lower bounds, setting a limit on the number of iterations performed before branching.

Unate benchmarks								
Benchmark	Cols	Rows	nonzeros	−1's	Opt	LP LB	row-diff	col-diff
bench1	4676	398	9563	0	121	119.9		
bench1 reduced	866	355	2821	0	113	111.9	25.9	20.7
max1024.pi	1278	1087	6974	0	259	256.8		
max1024 reduced	904	916	5368	0	209	206.8	29.2	28.9
saucier	6207	171	500632	0	6	5.0		
saucier reduced	6203	116	340223	0	6	5.0		
Binate benchmarks								
Benchmark	Cols	Rows	nonzeros	−1's	Opt	LP LB	row-diff	col-diff
e64.b	607	1022	8200	863	47	36.4		
e64.b reduced	571	920	6795	826	47	36.4	1.4	4.8
alu4	807	1823	36259	1732	50	46.3		
alu4 reduced	481	592	9866	526	32	28.3	3.0	12.5
rot.b	1451	2932	40755	2629	115	110.5		
rot.b reduced	887	1257	13742	1085	84	79.5	9.7	22.5

Table 3: Runtimes for the benchmarks.

Time is on a dedicated Intel(R) Xeon(TM) 3.20GHz processor with 2048 MB cache. Runs were terminated after an hour if the algorithm had not achieved optimality by then. For int-dual, there is also the possibility of an overflow when the integers in the tableau get too large.
Statistics (min, median, mean, max, and standard deviation) are based on 33 runs with isomorphs. The second column indicates how many of the 33 instances ran to completion versus time outs and overflows. The third column, marked "orig.", refers to the original unpermuted benchmark. A 16 hour limit on wall-clock time meant that, in some cases, less than 33 instances were executed.

Unate benchmarks							
Benchmark (solver)	N (to,of)[a]	orig.	min	med.	mean	max	stdev.
bench1 (cplex)	33 (0,0)	1.7	0.2	1.9	1.8	2.5	0.7
bench1 (int-dual)	1 (12,9)	t.o.	2,872.2	2,872.2	2,872.2	2,872.2	n/a
max1024 (cplex)	33 (0,0)	2.9	2.5	3.6	3.5	6.4	0.9
max1024 (int-dual)	0 (5,28)	o.f.	n/a	n/a	n/a	n/a	n/a
saucier (cplex)	0 (33,0)	t.o.	n/a	n/a	n/a	n/a	n/a
saucier (int-dual)	33 (0,0)	0.8	0.2	1.0	243.1	3,549.9	845.2
Binate benchmarks							
e64.b (cplex)	0 (33,0)	t.o.	n/a	n/a	n/a	n/a	n/a
e64.b (int-dual)	29 (0,4)	27.5	2.6	15.9	28.4	168.0	33.0
e64.b (cm)[b]	31 (1,1)	55.0	1.7	19.4	34.5	105.0	34.8
alu4 (cplex)	33 (0,0)	38.6	23.1	93.6	190.6	2,166.2	369.4
alu4 (int-dual)	33 (0,0)	2.7	1.1	3.8	4.1	12.6	2.4
rot.b (cplex)	33 (0,0)	6.3	1.6	3.9	6.0	33.3	6.1
rot.b (int-dual)	33 (0,0)	67.4	3.0	14.6	21.1	67.4	15.1

[a] Number completed (time outs,overflows)

[b] Each isomorph is subjected to crossing minimization and int-dual is run on the minimized row/column permutation.

a characterization that accounts for the extreme difference in behavior between the two.

5. A CONJECTURE

There are some obvious contrasts here: cplex does better on the unate instances, except for the rather stark exception of saucier; int-dual does better on binate instances except for a slight advantage for cplex on rot.b.

The unate/binate distinction may not be the only reason for the differences in performance. There is no obvious reason why int-dual should have an advantage in the binate case. In fact, if the number of -1's in the constraint matrix becomes too large, as is the case with the test generation benchmarks of [21], int-dual almost always times out while cplex finds optima quickly.

Saucier is easy to dispense with as an out-lier. It is much denser than the others and has an extremely large number of variables with relatively few constraints. Cplex does poorly because (a) it runs a complete simplex algorithm at every node, and (b) it is very conservative about making cuts even when told to do so aggressively. [7] Int-dual, on the other hand, always reaches the optimal solution of 6 very quickly and then spends the remaining iterations doing cuts to prove optimality.

For the remaining benchmarks it is instructive to look at the nonzeros of the initial constraint matrix after these have been organized to be as "block structured" as possible. Figure 1(a) shows the first 250 rows of max1024 (reduced), one of the unate benchmarks on which int-dual performs poorly. Figure 1(b) shows the first 250 rows of e64.b. The contrast is visually obvious. Figure 2 shows an "in-between" situation — 250 rows of rot.b — corresponding to the situation where domination is less clear.

These pictures are generated using crossing minimization on the bipartite graph induced by the constraint matrix — more details on this below. Several important points must be made:

1. The block structure is not evident at all in the original benchmark — the reference e64.b, for example, looks pretty random.

2. The block structure is present, but hidden, regardless of how the rows and columns are permuted. What we are observing is that, when a block structure emerges as the result of crossing minimization, it may predict the salutary behavior of int-dual over a whole class of isomorphs.

3. Int-dual does not necessarily perform better if an instance is rearranged into block structure first. The line marked "e64.b (cm)" in Table 3 illustrates this point. In fact, the restructuring improves runtime in only 14 of the 33 instances, mostly by only a few seconds.

Figure 3 illustrates how crossing minimization works to make a matrix appear more block structured. Let $G = (U, V, E)$ be a bipartite graph where

$$\begin{aligned} U &= \text{the set of constraints} \\ V &= \text{the set of variables} \\ E &= \{uv \mid \text{variable } v \text{ appears in constraint } u\} \end{aligned}$$

[7] Only one cut was attempted in the first 4602 nodes, which, in turn required almost 130,000 simplex iterations.

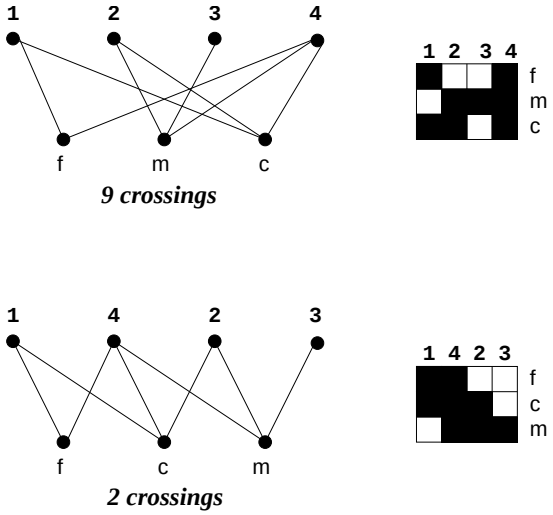


Figure 3: Crossing minimization illustrated using the employee/skills example.

Given a permutation $\pi_1(U)$ and another permutation $\pi_2(V)$, put the vertices of U on a horizontal line in the order defined by π_1; do the same for V with respect to π_2. If each edge is drawn as a straight line, the number of edge crossings is called the *crossing number* of G with respect to $\pi_1(U)$ and $\pi_2(V)$. While the problem of finding permutations that minimize crossing number is NP-hard [6], there are many good heuristics — see, e.g., [19]. The one we use is treatment 17 from that paper.

The permutations $\pi_1(U)$ and $\pi_2(V)$ are also permutations of the rows and columns of the matrix, respectively. Crossing minimization brings the nonzeros as close to the diagonal as possible (on average). The small employee skills example, before and after crossing minimization as shown in Figure 3, illustrates this dramatically.

The conjecture that underlying block structure favors int-dual needs to be quantified further. Obvious measures related to crossings are neither good indicators of the visual impact nor reliable predictors of algorithm performance on even this small set of benchmarks.

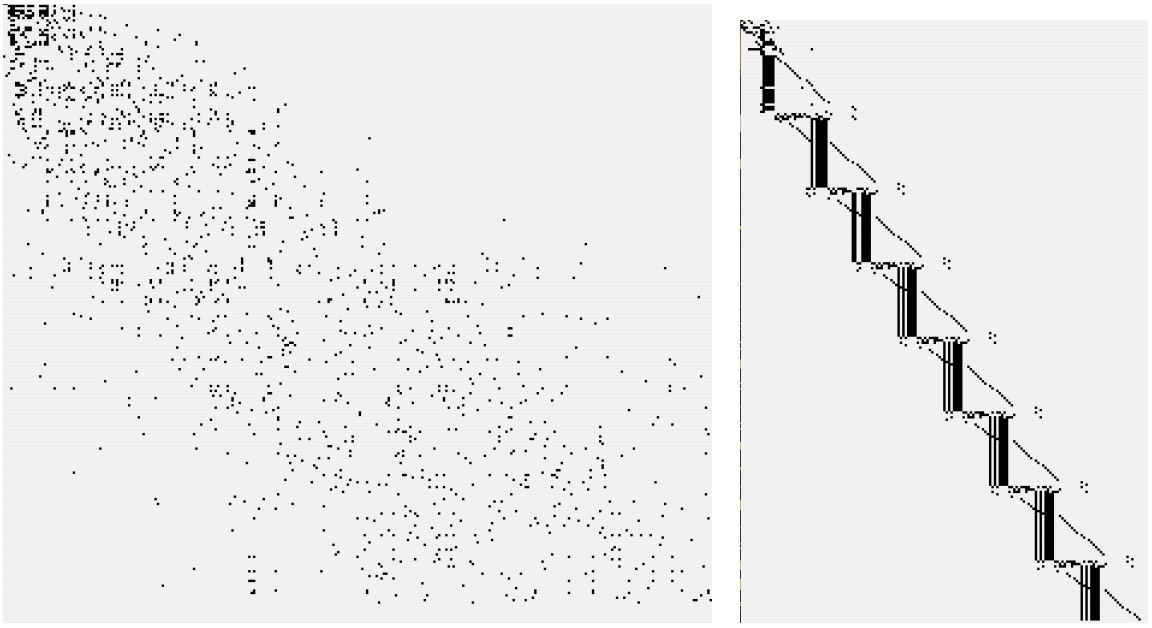
Measures of block structure, as proposed in [2, 1], are unsuitable for our purposes: they require that the matrix be partitioned into a fixed number of blocks of roughly equal size, rather than allowing the block number and size to be determined by the structure of the matrix when permuted.

We need a measure to capture the idea that an already crossing-minimized matrix "looks" block-diagonal.

Definition. The *diffusion* of row A_r of a matrix A, $dx(A_r)$, is the maximum distance between nonzeros divided by the number of nonzeros. More formally, $dx(A_r) = \max\{j - i + 1 \mid A_{ri} \neq 0, A_{rj} \neq 0\}/|A_r|$ — by abuse of notation, A_r represents the set of nonzeros in row A_r. A similar definition applies to columns of A. The *row diffusion* of A is $dx(A_r)$ averaged over all rows r of A. The parallel definition for *column diffusion* applies to the average diffusion over all columns of A. Diffusion is small when a matrix is structured, large when it is random.

Remarks.

1. In practice, neither row diffusion nor column diffusion,



(a) max1024.pi (b) e64.b

Figure 1: Plot of non-zero entries for reduced versions of max1024.pi and e64.b after crossing minimization; first 250 rows.

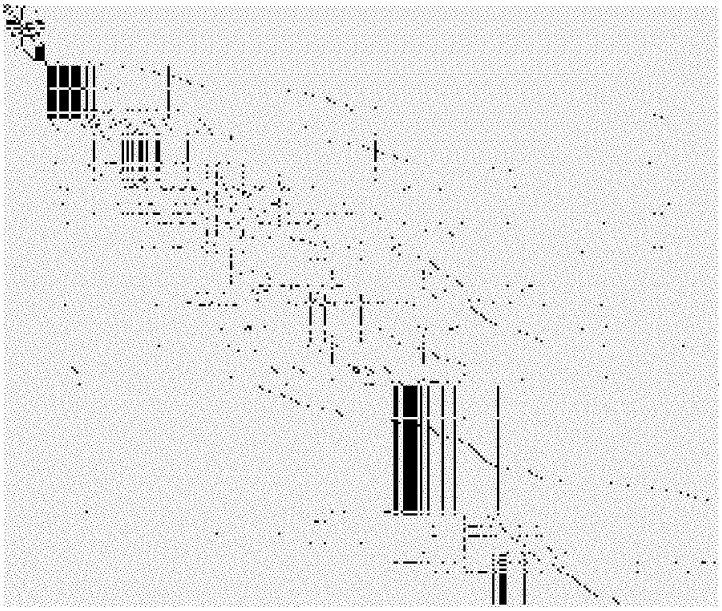


Figure 2: Plot of non-zero entries for the reduced version of rot.b after crossing minimization; first 250 rows.

row/col	min	mean	max	stdev
rot.b (row-diff)	9.7	10.6	11.1	0.6
rot.b (col-diff)	19.7	21.2	22.8	1.1

Table 4: Row and column diffusion for 33 different initial row/column permutations of rot.b.

as just defined, completely captures the structured versus random look of the pictures. A small number of outliers in rows or columns can add severe bias.

We adjust for this by taking the "middle half" of the nonzeros in a row/column (sorted by increasing column/row index), computing the corresponding diffusion and taking the smaller of that and the original.

2. The crossing-minimization preprocessing is essential for our purposes since we are looking for possibly hidden block structure.

3. The structure revealed after crossing minimization and the diffusion are both relatively immune to row and column permutations of the original instance — see Table 4 for an example.

4. Diffusion can be small even if nonzeros are far from the diagonal as long as they are contiguous. This is not a concern for us because of the crossing minimization.

5. Diffusion is 1.0 when all rows/columns are contiguous and can be much larger. Ideally it should be normalized to be a number between 0 and 1, but this appears difficult.

6. Row and column diffusion can differ from each other by wide margins. For now it does not matter whether the appropriate statistic is minimum, maximum, or average.

7. As will be more evident later, diffusion may need to be scaled to account for differences in instance size or density.

The last column of Table 2 shows row and column diffusion (after crossing minimization) for the five benchmarks where structure (or lack thereof) makes a difference. The correlation of these numbers with algorithm performance is striking, whether we take their minimum, maximum, or average.

We end this section with a conjecture, to be further explored in the next section.

Conjecture. *Int-dual performs better than cplex on problem instances which, if permuted using crossing minimization, exhibit small row and column diffusion.*

If this is an apt characterization of instances with respect to the performance of these algorithms, there should either be a threshold value above which cplex begins to dominate or a gradual transition from better performance by int-dual to better performance by cplex. The next section addresses this issue.

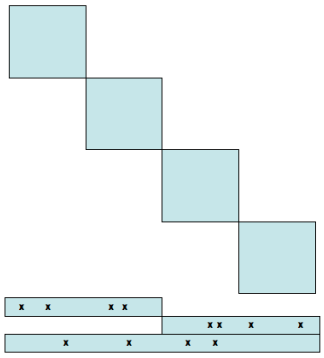


Figure 4: An abstract view of how four blocks are constructed with an added row at each level.

6. ADDITIONAL EXPERIMENTS

The design of additional experiments to explore the conjecture required many arbitrary choices. Other possibilities than those described here have also been explored, leading to similar conclusions. It must be emphasized that the choices described here, while apparently arbitrary, are not premeditated. We did not choose as we did to favor particular conclusions nor did we select among already-explored variations those that would best lend credence to our conjecture. The phenomena described here hold, as far as we can tell, for a variety of methods of adding "noise" to matrices that are purely block-diagonal.

We took a small unate instance from the same benchmark set. This instance, maincont, in its reduced form, has 61 variables, 50 constraints, 868 nonzeros. an optimal solution with cost 7, and is easily solved by both algorithms within a small fraction of a second.

Maincont was then used to form multiple blocks along the diagonal with the number of blocks being powers of 2: two blocks of maincont were combined for a 2-block instance, then two 2-block instances were combined into a 4-block instance and so on. This approach led to a sequence of pure block-structured instances.

To introduce increasing amounts of randomness, we added random rows or a combination of both rows and columns while combining instances at each step.

An instance with 2^k blocks and r added rows (and possibly columns) was created from two instances with 2^{k-1} blocks. At $k = 1$ both instances are the original maincont. At every step in the creation of 2, 4, ..., 2^k blocks, exactly the same number of rows (and possibly columns) were added. The choice of nonzeros in the rows and columns was completely random, but the number of nonzeros was fixed at four,[8] two lining up with each block. Figure 4 illustrates how the extra rows are added at each level in the case of one added row and four blocks created from two blocks of two.

Occasionally an added row will increase the cost of the optimal solution over what it would have been with no added rows. An added column can decrease the cost.

The first observation, easy to verify algebraically, is that

[8]There are technical reasons for this choice. Fewer than four entries in an added row give int-dual an advantage — it processes short rows first. With more than four we run the risk of introducing row dominance, causing the extra rows to be reduced away.

blocks →		8	16	32	64
algorithm	measure				
cplex	runtime	< 0.5	3.6	t.o.	t.o.
int-dual	runtime	< 0.5	< 0.5	1.0	5.8
int-dual	iterations	56	112	224	448

Table 5: Performance of cplex and int-dual on increasing numbers of pure blocks (maincont).

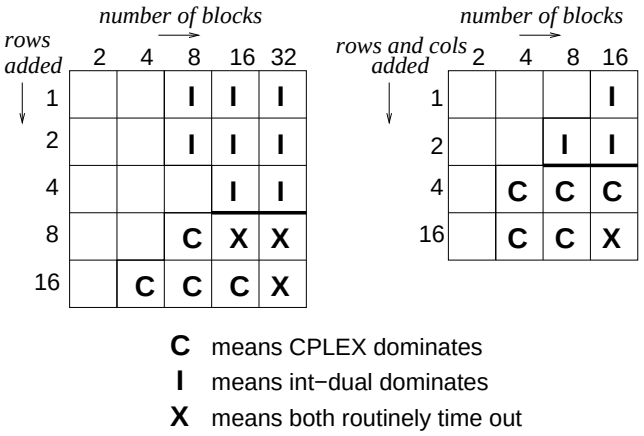


Figure 5: Dominance of int-dual versus cplex on various block structures with rows or both rows and columns added.

without added rows or columns, the asymptotic performance of int-dual is very smooth. Indeed the number of iterations, shown in the third row of the table, exactly doubles when the instance size is doubled.

Fact: Every step of int-dual is a pivot and a pivot element in a given block will only affect the entries corresponding to that block *no matter how much they are interleaved throughout the matrix via row/column permutations.*

We therefore have an underlying model for the better performance of int-dual on more general block-structured matrices.

There is also a straightforward explanation for the poor performance of cplex and other branch and bound solvers on larger instances[9]. Branch and bound generates two new nodes each time a branching variable is chosen, but fixing the value of a variable makes progress only within one block. Global information is needed to obtain good bounds and cuts.

What happens when we add rows and columns along the way? As predicted, and shown in Figure 5, there is a transition from dominance of int-dual to dominance for cplex. The blank entries are cases where both algorithms routinely find the optimum within a few seconds. The dominance is most often characterized by time-outs on the part of one algorithm and reasonable execution times on the other. This will be examined more closely later.

We narrow our search for a threshold, and address the applicability of diffusion, by looking at a single column of each rectangle, the one corresponding to 8 blocks. This is to avoid any issues that might be related to differences in instance size.

[9]Similar results were obtained by our branch and bound solver.

both	winner	row-diff	col-diff
0	easy	1.8	1.9
1	easy	2.3	2.2
2	int-dual	2.7	2.5
4	cplex	3.8	3.3
16	cplex	14.5	11.5

rows	winner	row-diff	col-diff
0	easy	1.8	1.9
1	int-dual	2.2	1.9
2	int-dual	2.8	1.9
4	int-dual	3.7	2.0
8	cplex	5.8	2.3
16	cplex	9.9	2.7

Table 6: Diffusion measures correlated with algorithm behavior on 8 blocks with added rows (and columns).

Table 6 makes the point that there may be a sharp threshold when both rows and columns are added and possibly a region of gradual transition when only rows are added. The entries marked "easy" are easy for both solvers because the number of blocks is too small for the asymptotic behavior of cplex to emerge.

Row diffusion but not column diffusion increases notably when only random rows are added. This is to be expected.

We have to look at the extremes — see Figure 6 — to see differences in the visual representations, and even then the difference is not nearly as pronounced as with some of the industrial benchmarks. The low diffusion numbers here are consistent with the pictures, but the crossover from int-dual to cplex is not in the same numerical range as with the benchmarks.

Now consider the results results from both the isomorphs and a random class (different random choices of nonzeros in added rows) based on 4 to 8 added rows with eight blocks, as shown in Table 7. Given the hopelessly skewed nature of the data distributions, we have to be cautious and regard our explorations as just that; no definite conclusions can be drawn.

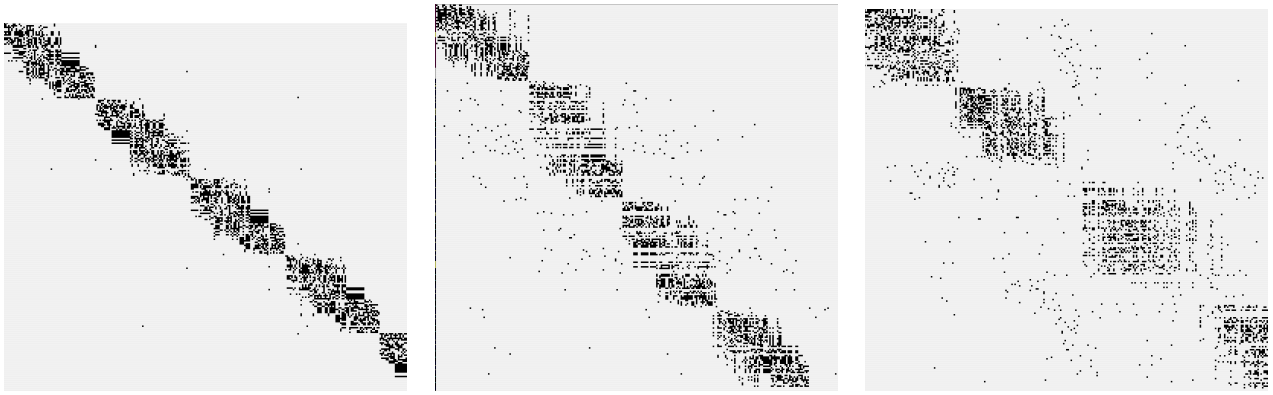
We would expect isomorphism classes to be better behaved than random classes. This is always true for cplex, but only true for int-dual in the absence of timeouts/overflows. Both classes are included in the table for the following reasons.

1. Among the random instances, we would not expect radical differences in underlying structure. Whatever behavior we conjecture should therefore hold, at least in the aggregate, for both isomorphic and random classes.

2. Nontrivial variation exists in the isomorphism classes just as it does for the benchmark instances.

3. We want to show that the original isomorph, in which the block structure is evident, does not necessarily lead to the best performance.

4. Using isomorphs on a single instance of the random class is not sufficient — the relative difficulty among random class instances varies a great deal.



(a) One row and column added (b) 16 rows added (c) 16 rows and columns added

Figure 6: Visual differences in 8 blocks with added rows and columns.

Table 7: Threshold behavior from 4 to 8 added rows for 8 blocks of maincont.

Benchmark (solver)	N (to,of)[a]	orig.[b]	min	med.	mean	max	stdev.
8 blocks, 4 rows, row-diff = 3.7, col-diff = 2.0							
cplex (iso)	33 (0,0)	0.5	0.4	1.5	2.0	6.7	1.3
int-dual (iso)	33 (0,0)	0.5	0.2	0.5	0.8	5.0	0.8
cplex (rnd)	32 (0,0)	n/a	0.3	1.5	9.4	130.5	24.5
int-dual (rnd)	32 (0,0)	n/a	0.1	0.4	0.4	1.2	0.2
8 blocks, 5 rows, row-diff = 4.1, col-diff = 2.1							
cplex (iso)	33 (0,0)	7.0	1.8	10.8	16.3	84.8	16.1
int-dual (iso)	27 (4,2)	5.1	1.1	14.0	31.6	321.5	61.3
cplex (rnd)	32 (0,0)	n/a	1.6	13.4	56.0	425.9	99.0
int-dual (rnd)	30 (1,1)	n/a	0.8	5.8	70.7	924.0	196.3
8 blocks, 6 rows, row-diff = 4.6, col-diff = 2.2							
cplex (iso)	33 (0,0)	2.4	0.8	3.9	6.4	37.2	7.2
int-dual (iso)	33 (0,0)	1.8	0.4	2.6	3.9	12.7	3.2
cplex (rnd)	32 (0,0)	n/a	0.9	29.3	121.1	1,695.2	303.7
int-dual (rnd)	31 (1,0)	n/a	0.6	5.0	10.3	79.8	15.4
8 blocks, 7 rows, row-diff = 4.9, col-diff = 2.2							
cplex (iso)	33 (0,0)	211.5	9.4	29.3	83.7	438.4	109.3
int-dual (iso)	6 (8,19)	63.2	11.0	56.6	85.9	203.3	70.0
cplex (rnd)	32 (0,0)	n/a	0.8	16.7	66.8	833.4	158.5
int-dual (rnd)	19 (7,6)	n/a	0.5	45.4	375.1	3,508.6	847.4
8 blocks, 8 rows, row-diff = 4.9, col-diff = 2.2							
cplex (iso)	33 (0,0)	585.6	45.7	220.7	281.3	746.9	199.5
int-dual (iso)	0 (6,27)	o.f.	n/a	n/a	n/a	n/a	n/a
cplex (rnd)	32 (0,0)	137.9	29.5	222.1	493.8	3,391.3	665.6
int-dual (rnd)	0 (2,30)	o.f.	n/a	n/a	n/a	n/a	n/a

[a]Number completed (time outs,overflows). The random classes have only 32 instances.
[b]Original reference instance (unpermuted): applies to isomorph class only.

Ideally we would have a class of isomorphs for each random-class instance, but this is impractical. The instance for which the isomorph class is constructed is no less random than an other.

Also important to point out is that timeouts/overflows on the part of int-dual can no longer be dismissed as easily. In an isomorph class we can run for a fixed length of time, stopping after that time and restarting with a different isomorph, and doubling the time until we find an optimum solution. But a random instance that times out may time out for all isomorphs or just a few.

Finally, it must be remembered that both algorithms are, at heart, exponential. Absent other circumstances their runtime will double with the addition of just a few rows.

A closer look at the data reveals the following points of interest.

- Int-dual exhibits two thresholds. The first, between four and five added rows, is a transition from runtimes less than a second to substantial runtimes accompanied by timeouts and overflows.

  The second threshold, between seven and eight rows, leads from optimal solutions within the time bound for most instances [10] to the complete absence of successful solutions.
- For cplex the transition is much smoother. The gradual increase in runtime with problem size (more rows mean larger instance) is briefly interrupted with the transition from 6 to 7 added rows. Figure 7 shows the behavior of several relevant statistics as number of rows increases. The explanation could be that 7 rows is "just random" enough for cplex not to be handicapped by the effects of block structure.
- Putting the previous two items together, it appears that threshold behavior with respect to moving from structured to random is primarily the fault of int-dual. This may no longer be trued with a larger number of blocks. Unfortunately, 16 is already too many, as it leads to timeouts for both algorithms in the region of most interest. A possible exploration could look at 12 blocks composed of 4 with 8.

As expected the data for random classes is more erratic than for isomorphs. These data are important for making our point, however.

In summary, our additional experiments have definitively given credence to our conjecture, modulo pinning down the exact interpretation of numerical values. Instances with low diffusion favor int-dual while a progression toward higher diffusion ultimately favors cplex. The key questions left open are (a) at what level does the transition occur — the numbers in the blocks experiments suggest a different scale than those of the original benchmarks — and (b) how sharp is the transition — again experiments with blocks point to an extreme contrast that does not appear to exist with the benchmarks.[11]

7. CONCLUSIONS AND FUTURE WORK

We have *observed* extreme contrast in the behavior of two algorithms on a set of well-known benchmark instances. Using a carefully defined and tractable measure of randomness versus structure underlying the constraint matrix, we then *conjectured* a plausible and quantifiable explanation for the difference. The explanation was validated with additional *experiments*, but there were some unexpected results that require further explanation. In the process we introduced a measure of underlying block-diagonal structure that is independent of how the rows and columns of a matrix are initially permuted. The measure correlates well with what the human observer perceives as structure *after* the matrix has been suitably permuted.

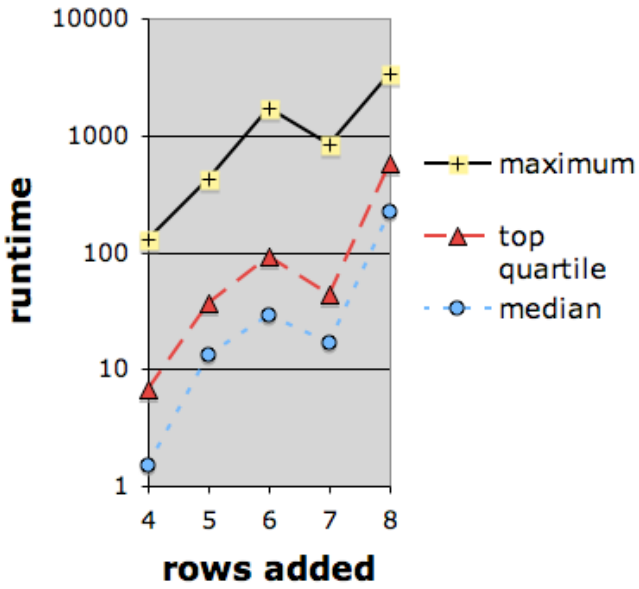
This work is only preliminary. The goal of instance profiling has not been achieved, primarily because our measure still needs to be properly scaled.

What follows is a list of cautionary remarks that should stimulate future research.

1. We have no easy way to explore the threshold versus gradual transition question with industrial instances. There are no in-between cases for a gradual morphing from e64.b to alu4. The transition from alu4 to rot.b appears smoother and is more difficult to interpret.

2. The threshold and/or gradual transition predicted by the diffusion measure may depend on the type and size of the instance. There is no question of the correspondence between visually identifiable structure and diffusion. The issue is whether either one can predict algorithm behavior.

   The number of optimal solutions and their relative po-



Each unit along the x-axis represents a total increase of seven rows: four when the 2-blocks are created, two for the 4-blocks, and one for the final 8-block instance. The y-axis has logarithmic scale — the difference between median and maximum is too great. Use of logarithmic scale also reveals the almost exponential increase in runtime with increasing instance size.

Figure 7: Behavior of cplex in the threshold region.

[10]The seven-row instance on which isomorphs are created appears to be harder than the median instance.

[11]It is also fair to say that we do not have a large enough collection of benchmarks to draw this conclusion.

sition in the space of all variable assignments both play a nontrivial role, albeit one that is difficult to verify in controlled experiments.

3. There is no "natural" example of a unate instance on which int-dual outperforms cplex (except saucier, which is anomalous). The unate instances in the logic synthesis domain, for whatever reason, are less structured than the binate ones. It would help if we could identify unate instances that favor int-dual to clarify whether a smooth transition, such as that between alu4 and rot.b, occurs only for binate instances, only for industrial instances, or only in special circumstances. More binate benchmarks in the interesting range of difficulty are needed as well.

All of these and many other issues need to be addressed. Characterizing the performance of complex algorithms, even in a limited domain, is challenging. We hope this case study inspires similar work.

Acknowledgments

The setting in which this work arose is joint work with Xiao Yu Li, whose earlier contribution to the umbra solver (then called *eclipse*) are much appreciated. Thanks also go to the staff of the NCSU High Performance Computing (HPC) facility for providing a hardware platform with fast dedicated processors and access to CPLEX, version 9.0. Eric Sills, in particular, dealt with many issues in a timely fashion.

8. REFERENCES

[1] C. Aykanat, A. Pinar, and Ü. V. Çatalyürek. Permuting sparse rectangular matrices into block-diagonal form. *SIAM J. Sci. Comput.*, 25:1860–1879, 2004.

[2] R. Borndörfer, C. E. Ferreira, and A. Martin. Decomposing matrices into blocks. *SIAM J. Optimization*, 9:236–269, 1998.

[3] 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.

[4] M. J. Brusko. Solving personnel tour scheduling problems using the dual all-integer cutting plane. *IIE Transactions*, 30:835–844, 1998.

[5] Giovanni De Micheli. *Synthesis and Optimatization of Digital Circuits*. McGraw-Hill, 1994.

[6] M. R. Garey and D. S. Johnson. Crossing number is NP-complete. *SIAM J. Algebraic and Discrete Methods*, pages 312–316, 1983.

[7] Robert S. Garfinkel and George L. Nemhauser. *Integer Programming*. John Wiley and Sons, 1972.

[8] Samuel I. Gass. *Linear Programming: Methods and Applications*. McGraw-Hill, 1958.

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

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

[11] Gary Hachtel and Fabio Somenzi. *Logic Synthesis and Verification Algorithms*. Kluwer Academic Publishers, 1996.

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

[13] S. Khanna, M. Sudan, L. Trevisan, and D. P. Williamson. The approximability of constraint satisfaction problems. *SIAM J. Computing*, 30:1863–1920, 2001.

[14] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys, editors. *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*. John Wiley, 1985.

[15] X. Y. Li, M. F. Stallmann, and F. Brglez. Effective Bounding Techniques For Solving Unate and Binate Covering Problems. In *Proceedings of the 42nd Design Automation Conference*, June 2005.

[16] Xiao Yu Li. *Optimization Algorithms for the Minimum-Cost Satisfiability Problem*. PhD thesis, Computer Science, North Carolina State University, Raleigh, N.C., August 2004.

[17] V.M. Manquinho and J. Marques-Silva. On using cutting planes in pseudo-boolean optimization. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:209–219, 2006.

[18] H. M. Salkin and R. D. Koncal. Set covering by an all integer algorithm: Computional experience. *JACM*, 20:189–193, 1973.

[19] M. Stallmann, F. Brglez, and D. Ghosh. Heuristics, Experimental Subjects, and Treatment Evaluation in Bigraph Crossing Minimization. *Journal on Experimental Algorithmics*, 6(8), 2001.

[20] Robert J. Vanderbei. *Linear Programming: Foundations and Extensions*. Kluwer Academic Publishers, 2nd edition, 2001.

[21] 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, 2002.

[22] S. Yang. Logic synthesis and optimization benchmarks user guide. Technical Report 1991-IWLS-UG-Saeyang, MCNC, Research Triangle Park, NC, January 1991.