High-Contrast Algorithm Behavior: Observation, Hypothesis, 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.

We used crossing minimization to discover the right permutations that make the underlying structure, whether block diagonal or random, visible. Our conjecture leads to a testable hypothesis for a limited domain.

The hypothesis is based on earlier observations reported in an earlier version of this paper and subsequent exploratory experiments. We present a sample of the results validating the hypothesis and make additional observations outside its scope.

As far as we are aware the approach taken here is unique and, we hope, will inspire other research of its kind.

Categories and Subject Descriptors

G.4 [**Mathematical Software**]: Efficiency; G.2.2 [**Graph Theory**]: Graph algorithms; G.3 [**Probability and Statistics**]: Experimental design; F.2.2 [**Nonnumerical Algorithms and Problems**]: Computations on discrete structures; J.6 [**Computer-aided Engineering**]: Computer-aided design (CAD)

General Terms

Algorithm experimentation

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

ExpCS, 13-14 June 2007, San Diego, CA

Keywords

Optimization, Set cover, All-integer dual simplex, Crossing minimization, Block-diagonal form

Note: *A preliminary version of this paper using different ideas is available as a technical report [22]. The current paper differs substantially from its predecessor: a more precise hypothesis is stated, new experimental results are reported, and the proposed measure of deviation from block structure is omitted.*

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 10.1 [13][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 [8, 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 and a more focused hypothesis 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 hypothesis.

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 where we lead up to our conjecture and subsequent hypothesis. In Section 6 our experimental design is explained. Section 7

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

reports results for typical classes of instances. Some final thoughts conclude the paper in Section 8.

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* [14], 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 employees 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

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

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 [17] 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 [12].

This work began with an earlier branch and bound algorithm, *eclipse* [17, 16], 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 [20], 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.

[2]This was CPLEX 7.5 on a processor that was slower, by about a factor of two, than the one used here.

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* [10, 11], 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*.

All-integer dual simplex

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. Int-dual begins with a tableau based directly on the constraint matrix A. Every iteration 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 promising for set cover problems – see [20].

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,

[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 [9]. Any linear programming textbook has more details – see, e.g., [23].

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.

The pivot for each iteration (indicated by a boxed entry) is chosen from a row that has a negative entry in the leftmost column (if none exists, the current solution is optimal). Within that row a column with a negative entry must be chosen (if none exists, the instance has no feasible solution). To prevent *cycling* – an infinite loop – the column is chosen to be lexicographically minimum among those that have negative entries in the pivot row.

There are many strategies for choosing pivot rows: the simplest, choosing the uppermost eligible row, is usually not very effective. Our implementation uses a combination of choosing a row with the smallest number of non-zeros and other criteria.

In Table 1 the first pivot is from a lex-min column: it beats $-x_3$ by having a -1 where $-x_3$ has a 0 in the first row and $-x_2$ by having a -1 in the second row. The second pivot is the only one having a 0 in the top row (this "zero row" is also used to determine lex-min) where the others have a 1.

Success with all-integer dual simplex algorithms has only been reported a few times in the literature [20, 4], and, as is the case with our work, in a limited domain.

Branch and bound: cplex

The other algorithm, cplex, uses branch and bound – see, e.g. [15, 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 (*fathomed*) and no children are generated.
- 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 also 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 governing, among other factors, the choice of a variable at each node, the choice of a node to explore next, and how often to apply local search. See [17] for a discussion of these factors for eclipse, our earlier branch and bound algorithm.

4. 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 [24] – 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 and pointers to previous ones can be found in [16] and [18]. The chosen subset omits instances that are too easy – runtimes are less than a second for all recent algorithms – and one that is hard mainly because it is considerably larger than the others.[4] We retain two benchmarks (maincont and f51m.b) that are used as blocks to form larger instances in our experiments.

Various statistics give an idea of the size and *density* – the ratio of actual non-zeros to potential ones – of each instance and how far the optimum is from the linear programming lower bound. We always use *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 [12] for more details. In some cases the problem size decreases significantly after the reduction is performed. 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.

Experiments

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.

Observations

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.[5] 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

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

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

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]

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 [24]. Optima were computed using umbra, our branch-and-bound algorithm.

Unate benchmarks							
Benchmark	Cols	Rows	non-zeros	density	-1's	Opt	LP LB
maincont	67	105	2428	0.345	0	7	6.0
maincont reduced	61	50	868	0.285	0	7	6.0
bench1.pi	4676	398	9563	0.005	0	121	119.9
bench1 reduced	866	355	2821	0.009	0	113	111.9
max1024.pi	1278	1087	6974	0.005	0	259	256.8
max1024 reduced	904	916	5368	0.006	0	209	206.8
saucier	6207	171	500632	0.471	0	6	5.0
saucier reduced	6203	116	340223	0.473	0	6	5.0
Binate benchmarks							
Benchmark	Cols	Rows	non-zeros	density	-1's	Opt	LP LB
f51m.b	406	520	13397	0.063	476	18	15.8
f51m.b reduced	175	187	2495	0.076	166	12	9.8
e64.b	607	1022	8200	0.013	863	47	36.4
e64.b reduced	571	920	6795	0.013	826	47	36.4
alu4	807	1823	36259	0.025	1732	50	46.3
alu4 reduced	481	592	9866	0.035	526	32	28.3
rot.b	1451	2932	40755	0.010	2629	115	110.5
rot.b reduced	887	1257	13742	0.012	1085	84	79.5

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.

Benchmark (solver)	N (to,of)[a]	orig.	min	med.	mean	max	stdev.
Unate benchmarks							
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.

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

5. CONJECTURE AND HYPOTHESIS

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 marginal 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 [19], int-dual almost always times out while cplex finds optima quickly.

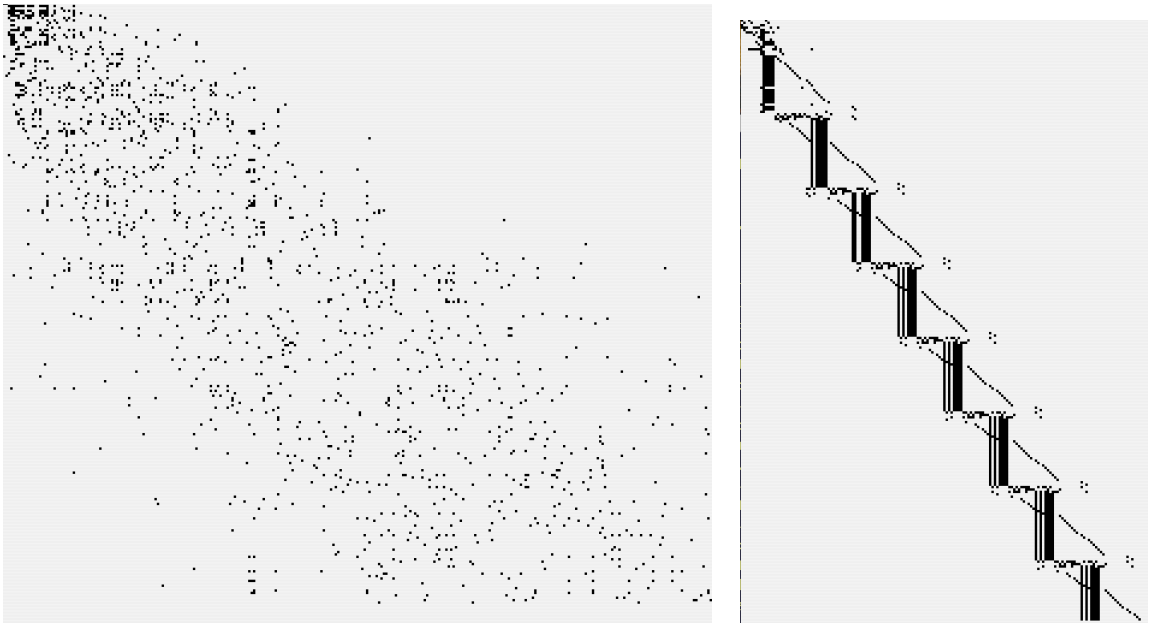
Saucier is a special case. Its shape is radically different – a large number of variables and very few constraints – and it is denser than the others by at least an order of magnitude if we ignore the two small instances maincont and f51m.b. These factors make block structure, the subject of our study, meaningless. We can easily explain the results for saucier: 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. [6]; 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 non-zeros 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 via 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.

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



(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.

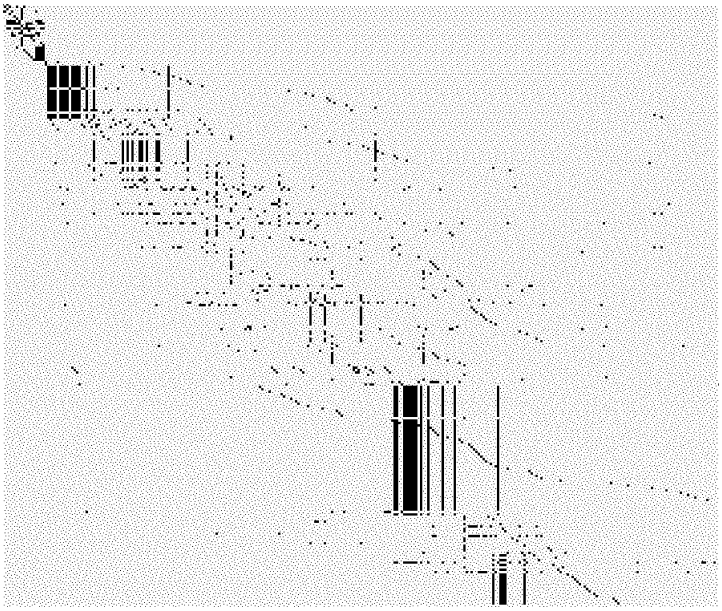


Figure 2: Plot of non-zero entries for the reduced version of rot.b after crossing minimization; first 250 rows.

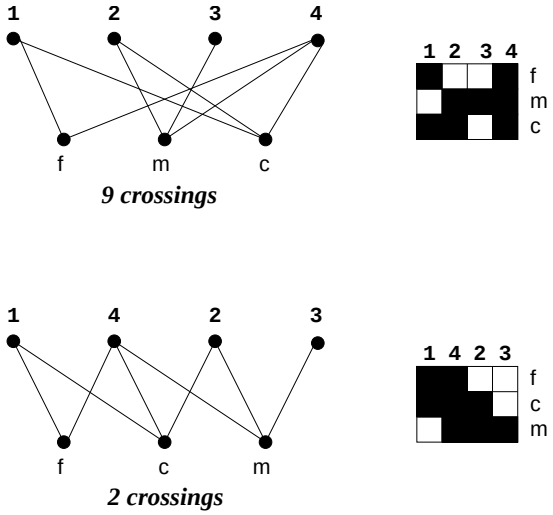


Figure 3: Crossing minimization illustrated using the employee/skills example.

Crossing minimization

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

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

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 [7], there are many good heuristics – see, e.g., [21]. 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 non-zeros 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.

Crossing minimization is a natural choice for revealing inherent block structure. If the block structure is perfect, all non-zeros occurring within distinct blocks, then the blocks form connected components of G. All $\pi_1(U)$ and $\pi_2(V)$ that minimize crossings keep the components contiguous and in the same sequence. The resulting display puts the blocks on the diagonal. This is true even if a heuristic is used – any reasonable heuristic is able to separate connected components.

Measures of inherent block structure are problematic, as shown by the attempt in [22]. The ones proposed in [2, 1], among others, are aimed at decomposing large matrices into smaller ones for the sake of parallelization. They therefore require that the matrix be partitioned into a fixed number of blocks of roughly equal size, rather than allowing the number of blocks and their size to be determined by the structure of the matrix when permuted.

Conjecture leading to hypothesis

The following vague conjecture, modified to eliminate any reference to s specific measure, is stated in [22] and supported by preliminary experiments.

Conjecture. *Int-dual performs better than cplex on problem instances which, if permuted using crossing minimization, exhibit visible block structure.*

In order to obtain a testable hypothesis we need to restrict the domain of the conjecture to a small subset of instances with which we can conduct experiments and quantify the notion "performs better" and "visible block structure".

Definition. An instance consists of *pure blocks* if it has a set of smaller instances arranged along the diagonal. The rows and columns of each smaller instance must be mutually disjoint. This term also describes any instance that is a permuted version of a pure blocks instance. Each smaller instance of a pure blocks instance is referred to as a *basic block*.

An instance with *added rows* is a pure blocks instance that has one or more rows added in such a way that the non-zeros of the added rows occur in columns of at least two of the smaller instances. A symmetric definition of added columns is also relevant, but not used in this paper.

It is difficult to ensure statistical significance when sampling the population of instances with added rows even when the number of rows, the number of elements per row, and the structure of the added rows with respect to the blocks are all fixed. The exact position of the added elements should not be fixed. Otherwise we are choosing single, possibly unrepresentative, instances and samples of their isomorphs.

The best we can do is observe trends and take note of classes for which runtime, the measure of merit, differs radically within the class (including the presence of time outs and overflows). The following definition addresses this issue.

Definition. A set of runtime data point is *erratic* if they are exponentially distributed or worse[7] or if there are timeouts/overflows present.

Since the timeout limit is arbitrary it might be the case that the runtimes are normally distributed around a value greater than the timeout limit. The reader will have to judge whether our experiments adequately allay this concern.

We shy away from the temptation to compare erratic distributions, to say that one distribution is worse than another if it has larger mean, median, or standard deviation, or more timeouts/overflows. Such comparisons must be left to future work in consultation with a statistician.

In the following three-part hypothesis we only address what happens in the case of added rows. The work of [22] suggests that it extends to added rows *and* columns and may extend to other methods of systematically introducing "randomness" to pure blocks.

Hypothesis.

[7]Data are exponentially distributed if their standard deviation is roughly the same as the mean and the median is about 60% of the mean. The usual distinction between exponential and heavy-tail distributions – see [3] – turns out not to be important to our hypothesis.

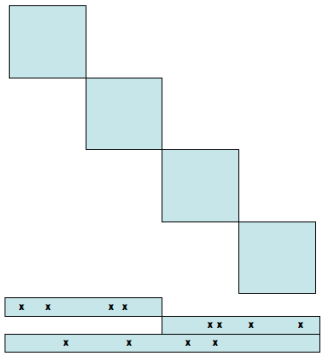


Figure 4: An abstract view of how four blocks are constructed with an added row at each iteration.

1. *The runtime of int-dual on pure blocks is predictable and grows polynomially with the number of blocks (roughly quadratically).*
2. *Cplex on pure blocks exhibits rapidly growing or erratic runtimes as soon as the number of blocks reaches a threshold that depends on the basic block.*
3. *As the number of added rows is increased, the runtime of int-dual grows rapidly or becomes erratic at a threshold that depends on the basic block.*

The behavior of cplex with increasing added rows is not predictable based on our preliminary experiments.

6. EXPERIMENTAL DESIGN

We experimented with a variety of basic blocks to confirm our hypothesis. The two reported here are typical and illustrate a range of possible outcomes. The logic synthesis instance *maincont*, in its reduced form, has 61 variables, 50 constraints, 868 non-zeros, an optimal solution with cost 7, and is easily solved by both algorithms within a small fraction of a second. The smallest (reduced) binate instance from the logic synthesis set, *f51m.b*, has 175 variables, 187 constraints, an optimal solution of 12, and is also solved easily by both algorithms but with larger execution times 0.2 seconds versus < 0.1.

Pure blocks were formed in the obvious way for both instances, by arranging the basic blocks along the diagonal. A class of 32 permuted versions of the result was then created. Added rows, as suggested by the hypothesis, were then added to pure-block instances. We settled on a specific strategy to reduce the number of variables, but we claim that the exact choice of method is not important to the overall results – any uniform reproducible strategy that generates increasing numbers of added rows could be used to confirm or disprove the hypothesis.

Two important issues *not* addressed in our experiments are: (a) whether adding -1's as well a 1's makes a difference, and (b) whether there is an observable effect when the number of non-zeros in each added row is increased. To reduce the number of variables, we chose to consider rows that have exactly four 1's, two aligned with each block when two blocks are combined.[8]

[8]The primary reason for this choice is to avoid row-dominance reductions with respect to rows with only three non-zeros; all rows of Steiner 15, an instance studied earlier, have exactly three non-zeros, for example.

Table 4: Performance of cplex and int-dual on increasing numbers of pure blocks (maincont).

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

An instance with k basic blocks and *row factor* r has $(k-1)r$ added rows and is created by the following round-robin algorithm.

put k basic blocks on a queue Q
while $|Q| > 1$ **do**
  let B_1 and B_2 be the first two blocks on Q
  remove B_1 and B_2 from Q
  create block B by
    putting B_1 and B_2 on the diagonal
    adding r rows, each with two random 1's
      overlapping each of B_1 and B_2
end do

Both the basic blocks and the final result are permuted randomly. Figure 4 illustrates how the extra rows are added at each iteration in the case where $k = 4$ and $r = 1$. Occasionally an added row will increase the cost of the optimal solution over what it would have been with no added rows. There appears to be no correlation between increased cost and runtime of either algorithm.

To get an idea of the range of possible outcomes for any given number of blocks and row factor we generate a *class* of 32 instances using the above algorithm with different random numbers. In contrast to the work of [3] our aim is not statistical significance. Rather, we want to ensure that we have a representative sample of the population of instances of each number of blocks and row factor.

Random choices of non-zero positions as well as random permutations are controlled by a sequence of random numbers generated using the IEEE standard[9]. The initial seed is a command-line argument to all scripts and programs that generate classes of instances or individual permuted instances. This guarantees that the identical class/instance will be created whenever the same seed is given, ensuring exact reproducibility of our experiments when desired. However, it is likely to be more useful to conduct the experiments with different random sequences to ensure that our results are typical and robust.

Our earlier experiments used one hour as the timeout limit. The results reported here use 10 minutes, i.e. 600 seconds. With the heavily skewed distributions observed earlier we doubt that this will make a difference: very few, if any, instances has runtimes between 600 seconds and one hour.

7. RESULTS OF CONTROLLED EXPERIMENTS

The first observation, easy to verify algebraically, is that pure-blocks instances with no permutations lead to a strictly

[9]`man rand48` on a Unix machine gives a description

Table 5: Runtimes for increasing numbers of pure maincont blocks.

blocks	statistic	int-dual	cplex
2	mean	< 0.1	< 0.1
2	stdev	< 0.1	< 0.1
4	mean	< 0.1	< 0.1
4	stdev	< 0.1	< 0.1
8	mean	0.1	0.2
8	stdev	< 0.1	< 0.1
16	mean	0.3	0.7[a]
16	stdev	< 0.1	0.6
32	mean	1.6	N/A
32	stdev	0.1	N/A

[a] $N = 29$, so this is clearly erratic

Table 6: Runtimes for increasing numbers of pure f51m blocks.

blocks	statistic	int-dual	cplex
1	mean	0.2	0.3
1	stdev	0.1	0.1
2	mean	0.5	85.3
2	stdev	0.2	75.4
3	mean	1.2	> 600
3	stdev	0.5	N/A
4	mean	2.2	> 600
4	stdev	0.7	N/A
8	mean	12.8	> 600
8	stdev	3.3	N/A
16	mean	64.0	> 600
16	stdev	11.3	N/A

Table 7: Runtimes for increasing numbers of added rows with eight maincont blocks.

row factor	statistic	int-dual	cplex
1	mean	0.1	0.4
1	stdev	< 0.1	1.1
1	N	32	32
2	mean	0.1	12.7
2	stdev	0.1	61.4
2	N	32	31
3	mean	0.3	1.5
3	stdev	0.1	4.6
3	N	32	32
4	mean	3.1	9.7
4	stdev	7.5	19.4
4	N	32	32
5	mean	14.6	59.9
5	stdev	33.1	140.5
5	N	27	32
6	mean	116.5	72.5
6	stdev	174.6	120.7
6	N	14	31

Table 8: Runtimes for increasing numbers of added row factor with three f51m blocks.

row factor	statistic	int-dual	cplex
1	mean	5.0	121.8
1	stdev	4.6	N/A
1	N	32	1
2	mean	74.6	223.3
2	stdev	122.4	164.0
2	N	23	9
3	mean	75.2	166.9
3	stdev	134.5	143.6
3	N	22	23
4	mean	81.5	105.2
4	stdev	93.4	118.2
4	N	19	31
5	mean	115.3	37.5
5	stdev	131.6	75.3
5	N	12	32
6	mean	95.0	15.1
6	stdev	74.8	13.7
6	N	15	32
7	mean	241.2	15.1
7	stdev	148.5	15.0
7	N	10	32
8	mean	225.4	8.8
8	stdev	139.4	6.6
8	N	7	32

linear increase in the number of simplex iterations of int-dual – see Table 4 for the case of maincont. The increase in runtime is also smooth but not linear – theory predicts an increase that is somewhere between quadratic and cubic.

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 pure-blocks instances[10]. 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.

Pure blocks and row factor increases

In Tables 5 through 8 we report only the mean, standard deviation, and N for each class, where N is the number of valid runtimes (ones that are not timeouts or overflows). When $N < 32$ the distribution is by definition erratic. For the first two tables, Tables 5 and 6, N is not reported. It is, except for one case – 16 blocks of maincont with cplex – either 32 or 0. When it is 0, the mean is more than 600 and standard deviation is unknown (N/A).

Tables 5 and 6 show the asymptotic behavior of both algorithms with an increasing number of pure blocks. As expected, the runtime of int-dual increases smoothly in both

[10] Similar results hold for our branch and bound solver.

cases. The standard deviation reflects the fact that the basic blocks are permuted and thus have (slightly) varying numbers of simplex iterations. Also expected is the fact that cplex becomes erratic on 16 basic blocks of maincont, only 2 of f51m. The latter is (probably) due to the fact that f51m is a larger and more difficult instance.

Results with increasing row factors – Tables 7 and 8 – reveal that the performance of int-dual degrades as expected in both situations. In the case of maincont the distribution is worse than exponential at row factor 4 and from then on N decreases below 32. With f51m the decrease in N already starts at row factor 1.

The performance of cplex is more complex. In the case of maincont it also degrades with increasing row factor, but one might argue that it improves *relative to int-dual* starting at row factors 5 and 6. This was also reported in [22], where the overarching concern was relative performance. However, both distributions are highly erratic, making statistical significance out of the question.

In the case of f51m, the performance of cplex actually improves with increasing row factor and overtakes that of int-dual at row factor 5, transitioning from a heavy-tail to an exponential distribution at 6 and with decreased runtime at row factor 8.

Experiments with other benchmarks and with different numbers of maincont and f51m blocks also validate the hypothesis and show varied behavior of cplex with respect to increasing row factor.

Visual impact of increasing row factor

To actually observe the difference in visual representation when row factor is increased one has to use a much larger row factor than the ones in the reported experiments. Figure 5 shows the differences between pure blocks, row factor 4 and row factor 16. This raises important questions about the reliability of visual inspection and the need for a precise indicative measurement. A first attempt, the diffusion measure proposed in [22], lacks both a cogent argument for direct relationship to visual observation and a way to calibrate it so that it correlates with observed algorithm behavior.

8. CONCLUSIONS AND FUTURE WORK

We have *observed* extreme contrast in the behavior of two algorithms on a set of well-known benchmark instances. We then *conjectured* that this contrast was based on the differences in structure underlying the constraint matrix, block diagonal versus random. This led to a *hypothesis*, primarily about the behavior of int-dual on pure blocks instances versus ones where additional elements outside of the blocks were added. Our *experimental results*, of which only a sample are reported here, have consistently *validated* our hypothesis.

This work is only preliminary. The goal of instance profiling has not been achieved nor can it be expected outside of a limited domain.

What follows is a list of loose ends that should stimulate future research.

1. We have completely side stepped the issue of statistical significance in our binary characterization of erratic versus non-erratic data. Statistically valid analyses of runtime distributions may be useful in assessing the *relative* performance of the algorithms. A new definition that addresses when one algorithm outperforms another is being explored and awaits rigorous statistical justification.
2. This work was motivated by visual inspection of plots of non-zeros after crossing minimization. Clearly this is unscientific – a measurement is needed that captures either the visual intuition or the predicted algorithm behavior or, ideally, both. We have already noted that *diffusion*, as described in the preliminary version, is ad hoc, and likely to fail on both counts.
3. Several variations on the experiments reported here have not been tried. All of our basic blocks are identical. What if we used different instances to compose pure blocks and then added rows? We used a particular strategy for adding rows. What if the rows had more overlap with each block? What if they contained -1's as well as 1's? And, finally, what if we added columns or both rows and columns?
4. The use of logic synthesis instances as basic blocks in our experiments is clearly limiting. We have also validated the hypothesis with a hard unate instance based on Steiner triples with 15 variables (see [6]). It would be interesting to see if the instances reported in [4] are nearly block structured when permuted using crossing minimization. That would at least suggest that an all-integer dual method succeeds in a variety of domains *if there is underlying block structure*. Given the paucity of successful uses of all-integer dual methods, this would be useful information indeed.

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, versions 9.0 (our preliminary work) and 10.1 (the work reported here). Eric Sills, in particular, dealt with many issues in a timely fashion.

Availability

C++ implementations of int-dual and a wrapper program for cplex, along with various scripts used in carrying out the experiments, will available as an alpha release at

`http://people.engr.ncsu.edu/mfms/Software`.

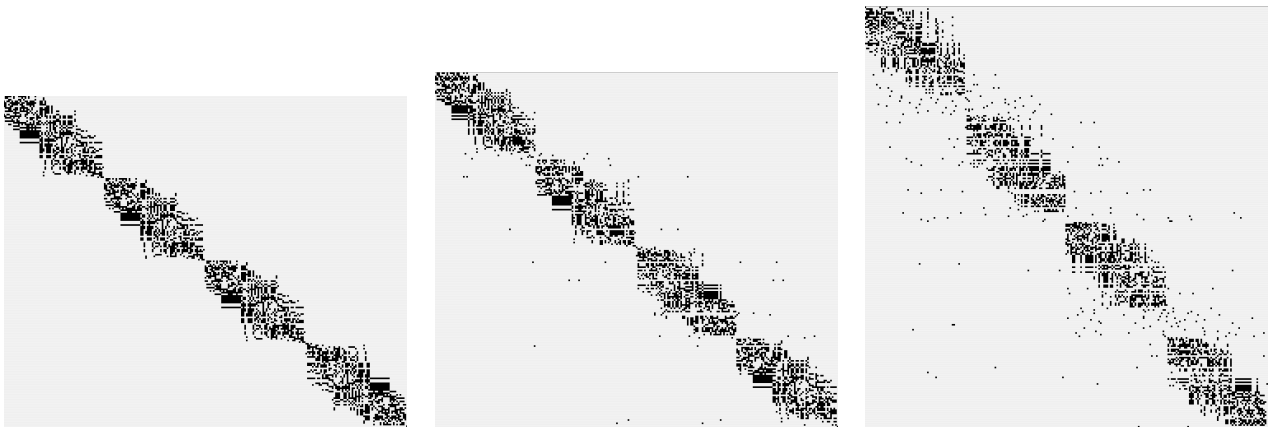
Also to be posted are instance classes and tables of results, not only for the experiments reported here, but others as well.

Target date for the first release is June 1, 2007.

9. 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.



(a) Pure maincont blocks (b) Row factor 4 (c) Row factor 16

Figure 5: Visual differences in 4 maincont blocks with added rows and columns.

[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] D.R. Fulkerson and G.L. Nemhauser. Two computationally difficult set covering problems that arise in computing the 1-width of incidence matrices of Steiner triple systems. *Mathematical Programming Study*, 2:72–81, 1974.

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

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

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

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

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

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

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

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

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

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

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

[22] M. F. Stallmann and F. Brglez. High-contrast algorithm behavior: Observation, conjecture, and experimental design. Technical Report TR-2007-14, Computer Science Department, NCSU, 2007. Available at `http://www.csc.ncsu.edu/research/tech/reports.php`.

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

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