All-Integer Dual Simplex for Binate Cover Problems
Draft[*]

Matthias Stallmann [†]

October 4, 2006

Abstract

All-integer dual simplex, *int-dual*, first introduced around 1960, has not received much attention recently because, in most situations, it does not perform well when the goal is to obtain optimal solutions. This paper presents experimental results for int-dual in a limited context.

First instead of seeking optimality int-dual is only required to run long enough to match an upper bound obtained by stochastic local search. If both procedures are run simultaneously or concurrently, the maximum time for the two determines when they will "meet in the middle". Second, the problem domain is limited to *binate cover* problems, set cover problems with some side constraints, arising in design automation.

While initial results are encouraging, the algorithm exhibits extreme sensitivity to row and column permutations with one of the benchmark instances.

The paper gives important implementation details, experimental results, and a thorough analysis.

1 Introduction

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 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). The generalization is strict, even in the case of binate cover, because binate cover instances have very few negative literals in the formula.

See [15] for a further discussion of unate cover, binate cover, and min-cost sat. Applications of these problem formulations are presented in [5] and [11].

2 Integer programming formulation

An integer programming (IP) formulation of min-cost sat is

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

[*]An earlier draft was submitted to ALENEX07 — latest revision October 4, 2006.

[†]Department of Computer Science, North Carolina State University, matt_stallmann@ncsu.edu

where c is the cost vector, all ones in this setting; the *constraint matrix* A and bound vector b are determined as follows. A row, or constraint, in A, b is based on a clause. There is an entry of 1 in A corresponding to a positive occurrence of a variable and a -1 for a negative occurrence. The corresponding element of b is $1 - N$, where N is the number of negative entries in the clause.

When the instance is unate b is all 1's.

3 The dual simplex algorithm

The details and notation in this presentation are primarily from [6]. The tableau method used is referred to as the *Beale* tableau, after its originator, but its first known publication is in [7].

A dual simplex approach to solving this IP (or the LP relaxation) starts by expressing the constraints as equalities: $Ax - Is = b$, with $s \geq 0$ being a vector of *slack variables*.

Solution via a tableau method begins with $x = 0$ and $s = -b$ and $-I$ as the basis. This is dual-feasible and, unless $b = 0$, primal infeasible. The dual simplex algorithm does a sequence of *pivot* steps, changing the basis while maintaining dual feasibility.

y_{ij} with elements y_{0j} in the top (primal cost, dual constraint) row and y_{i0} in the left (dual cost) column. The initial tableau has (0,c) as row 0, (0,$-b$) as column 0 and $-A$ to fill out the rest. An example showing the vertex cover problem for K_3, a triangle, is shown in Table 1, the first tableau.

		$-x_1$	$-x_2$	$-x_3$
$-sol$	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 1: Initial tableau and sequence of pivots for the vertex cover problem of a triangle

A pivot changes the basis, removing a row and replacing it with a column. Dual feasibility is maintained as long as the top row of the tableau is ≥ 0. When $s \geq 0$ the current dual solution is optimal and the primal is feasible (all slack variables are non-negative). The process is simple when integrality is not an issue: a pivot row is chosen (any row r with $s_r < 0$), then a pivot column k with $y_{rk} < 0$ (in the absence of such k, the dual is unbounded and the primal infeasible). Except for the pivot row y_r and column C_k, the transformation of the pivot is described, from the point of view of an arbitrary column C_j, by $C'_j = C_j - (y_{rj}/y_{rk})C_k$. The pivot row y_r is replaced by y_r/y_{rk} and the pivot column C_k, exclusive of the pivot element,

by $-C_k/y_{rk}$.

The choice of row is arbitrary except as needed to ensure termination. The choice of column is determined by the need to maintain dual feasibility. A stronger condition is the invariant that all columns be lexicographically positive. The pivot column is therefore chosen to be lexicographically minimum among *eligible columns*, those that have $y_{rj} < 0$.

Table 1 shows the sequence of pivots for the vertex cover problem on a triangle. The optimal solution when there are no integrality constraints has each vertex with value 1/2. This would be found when doing the pivot on the element -2 in the third step of the example.

4 All-integer dual simplex

The int-dual algorithm avoids the fractional solution by ensuring that all pivots are -1 and thus all tableau entries will be integral throughout.

A *Gomory cut* [9, 10] is a new constraint of the form

$$\lfloor hy_{r0} \rfloor = \sum_{1 \le j \le n} \lfloor hy_{rj} \rfloor$$

for a given h and r. No integral solution satisfying the existing constraints is eliminated, but, given the right choice of h and r, an optimal fractional solutions is. In case of the triangle example, choosing $r = 2$ and $h = 2$ yields the constraint in the last row of the final tableau in Table 1, which leads, after another pivot, to the integer optimum of 2.

In the int-dual algorithm a cut is triggered when the choice of pivot is an integer < -1. The cut uses the pivot row and chooses h so that the new pivot will maintain lexicographically positive columns. Only the eligible columns matter. For the rest, $y_{rj} \ge 0$ and $y_{rk} = -1$ means a positive multiple of C_k will be added to C_j.

If j is an eligible column, let $\alpha_j =$ the maximum integer for which $C_j - \alpha_j C_k \ge 0$. Now h can be chosen as $\min_j\{\alpha_j/-y_{rj}\}$ – here y_{rj} is the pivot before the cut is applied. A summary of the algorithm appears in Fig.1

5 Implementation issues

Several important issues arise in the implementation of int-dual. A very early implementation with experimental results for a variety of smaller set cover problems is reported in [18].

- The fraction h, when computed in floating point arithmetic, leads to problems with accuracy and possible unpredictable results. The current implementation uses rational numbers represented as a sign and a pair of unsigned integers. This suffices in most cases.
- The all-integer approach merely trades floating point inaccuracy for the possible creation of large integer entries. This can result in overflow and unpredictable results if overflow is left undetected. This implementation catches possible overflow before the computation leading to it is performed. Execution time is increased by a factor of about 1.5, but the advantages of knowing of an overflow are worth it.
- Use of arbitrary precision integers (or fractions) was considered but abandoned. The increase in execution time, even when computations stayed within single word limits, amounted to a factor of 5.

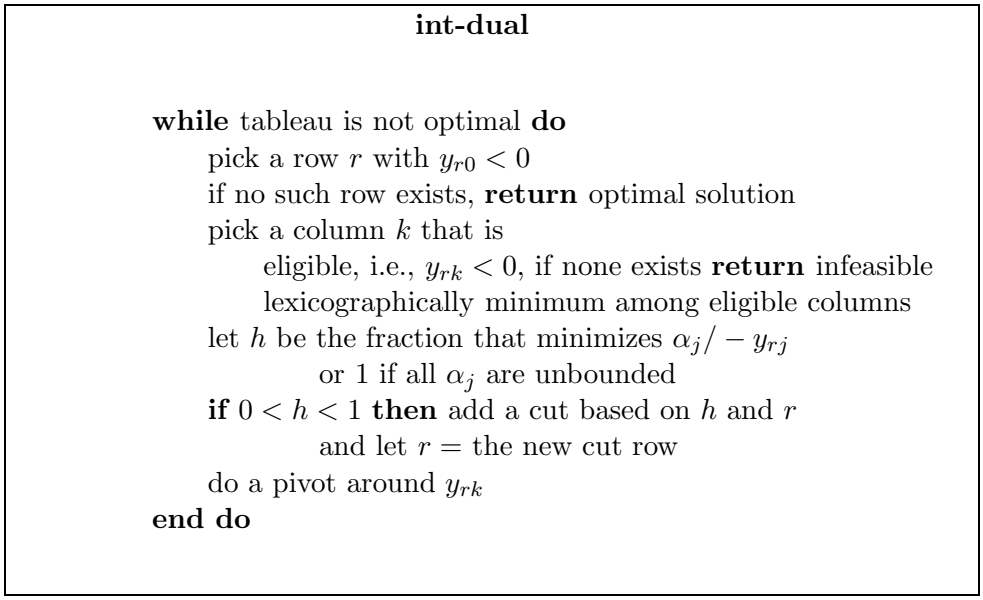

Figure 1: The all-integer dual simplex (int-dual) algorithm

- Choice of pivot row is an important consideration, as also discussed in [4]. The overall strategy in the implementation discussed here is to choose the row of least weight with weight defined as follows:
  - The primary consideration is to boost the dual cost whenever possible. The default strategy gives priority to rows in which all eligible columns have a positive entry in row 0. There is a major penalty for each eligible column that has a 0 in the first row.
  - A secondary consideration is efficiency. If rows are equal with respect to the previous consideration, the one with the smallest number of eligible columns is chosen.
  - Finally, to avoid overflow, it is desirable to avoid rows with large integers in them. Any row having an entry larger than the square root of the maximum representable integer is eliminated from consideration unless such rows are the only option.
- Although the problem instances start off being very sparse, they become dense after not too many iterations. The initial sparse linked-list implementation was quickly abandoned in favor of matrices whose rows are C++ STL vectors.
- The implementation is in C++ and now has a simple mechanism for adding options and statistics.

6 Benchmark results

Table 2 shows preliminary single-instance results on a set of benchmark problems based on logic synthesis. These were originally contributed to a 1991 workshop at MCNC in Research Triangle Park, NC – see [21] – and have been the subject of intense experimentation since, mostly with branch and bound algorithms. Up to a point, optimal solutions outweigh execution time considerations. Recent results with pointers to previous ones can be found in [14] and [16].

Instance size figures differ from those reported in other sources because these are *re-*

duced versions of the instances, modified by *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.

The good news is that, for all of the binate instances, int-dual was able to quickly prove the optimality of upper bounds obtained by stochastic local search – in particular, the one used in the branch and bound algorithm of [14]. In two cases, alu4 and e64.b, these optima had not been known prior to this work.

Execution times are at least as good as any obtained by cplex [12] on the same machine, and, as already suggested, cplex was unable to solve alu4 or e64.b even within a one-hour time limit, the standard used in most of our experiments[1]. Comparison with cplex, or any other algorithm, is not necessarily relevant here for two reasons.

1. These are single instance experiments and results, as will be shown later, are extremely sensitive to row and column permutations[2].

2. There is no branching involved here; int-dual is executed until it proves optimality for a stochastic local search solution.

The setup here is that two concurrent or parallel processes can find the optimum if one does local search while the other obtains a lower bound using int-dual. The program can halt when the two costs match each other, i.e., within the maximum of both execution times.

Originally, the optima were obtained via branch and bound, using int-dual for lower bounding.

7 Sensitivity to row/column permutations

Now the bad news, at least in the case of e64.b. The three most challenging instances were each subjected to 128 random row and column permutations, leading to 129 instances each, if we count the *reference instance* for which the result in Table 2 is reported. This procedure for assessing the repeatability and robustness of computational experiments is based on our earlier work with satisfiability [3].

All runs were done with a time out of one hour. Table 3 shows the number of time-outs and overflows, as well as information about the distribution on the remaining instances.

The alu4 results are typical of what we've experienced before — exponential distribution. The one outlier (time out) might be the result of a bug. Worthy of note is the fact that the single-instance result is at the high end of the range, so the real truth is much better than first appearances.

With rot.b things are more disturbing. Still only one outlier, but the distribution of the rest is decidedly worse than exponential, sometimes referred to as *heavy tail*.

The e64.b results, however, are unexpectedly bad. In the context of branch and bound, the time-outs and overflows were masked by the ability to branch after a fixed large number of iterations or after an overflow. Not so here.

The row weighting strategy discussed under implementation issues earlier makes sense but was, to some extent, guided by single instance experiments. Other strategies might work

[1]The cplex used was version 9.0, with default settings.

[2]This is true also for cplex but that's the subject of another paper.

equally well or better. To explore this issue, several other strategies were considered and subjected to the same 129-instance regimen:

- **Allow large numbers.** This strategy is identical to the default except that rows are not eliminated no matter how large their entries are.
- **Default with sorted rows.** Row weights are the same, but rows are sorted by the median index of their non-zero entries, a first step in the direction of approximating block diagonal form.
- **Minimum non-zeros.** This is a much simpler strategy in which the row weight is just the number of non-zeros in the row. Its main purpose is as a "control" to see if more sophisticated strategies make sense. Large numbers are allowed.
- **Break ties with minimum non-zeros.** Use minimum non-zeros as a tie breaking strategy in place of minimum number of eligible columns. Also allow large number.

When robustness within classes is involved, a this variety of row selection strategies appears to make little, if any, difference. See Table 4.

One might ask whether some of these permuted instances are significantly worse than others in that they foil most weighting strategies. Table 5 shows a list of instances, among the first 84, that had three failures (none had more than three). Most frequently represented in cases where the default strategy was foiled are those in the second, third, and fifth column, not too surprising given that these are all variations on the default.

The silver lining here is that, in the "difficult" instances, execution time of the best strategy is at most ten seconds. This is another situation where parallelism or concurrency has a big payoff.

8 Random restarts

A popular strategy in the constraint satisfaction community is to use random restarts to mitigate erratic behavior of an algorithm [8, 17, 20]. There, the algorithm is usually a stochastic search for a feasible solution (to, e.g., a satisfiability problem). Also there are arbitrary decisions involved about how long to run before restarting and how to tune the restart strategy for particular instances.

Here we can think in terms of a more systematic approach. Suppose we run int-dual for a baseline number of iterations N (time could be used as well but is trickier to implement and more sensitive to noise), long enough to justify doing a restart. If it does not find the desired solution (recall, that a stochastic search is also at working finding an upper bound), we run it again on another random permutation for N iterations. After that, each run doubles the number of iterations.

If p_k = the probability of (the original algorithm) finding the solution is $\leq N \cdot 2^k$ iterations, then r_k, the probability of the restart algorithm finding the solution within $N \cdot 2^k$ iterations is governed by $r_0 = p_0$ and $r_k = r_{k-1} + (1 - r_{k-1})p_{k-1}$. Figure 2 shows the relationship between p_k and r_k using the data from the 129 permutations of e64.b. The 20,000 iterations at which this starts to pay off represent an execution time of about 180 seconds. The relationship between iterations and time is roughly quadratic because many of the iterations add cuts and increase the size of the tableau.

The situation is even better when we take into account the fact that some unsuccessful executions stop early due to overflow. In this case they all stop within 20000 iterations.

iterations	p_k	r_k
2500	0.02325581	0.02325581
5000	0.37984496	0.0459708
10000	0.62015504	0.40835398
20000	0.73643411	0.77526624
40000	0.75193798	0.94076785
80000	0.75193798	0.98530675
160000	0.75193798	0.99635516
320000		0.99909585

Figure 2: Convergence with random restarts using e64.b data

9 Future work

Future work is in three main areas. One is to overcome the sensitivity of the algorithm to permutations, another is to take advantage of that sensitivity, and the third is to create a better test bed for this kind of analysis.

Overcoming sensitivity An ongoing concern is to try to understand the reasons for the erratic behavior and look for ways to mitigate it.

A step in that direction would be to find orderings of rows and columns that lead to better behavior. The author has tried approximating block diagonal form[1, 2][3] and crossing minimization [19] (of the bipartite graph of rows and columns, with non-zeros representing edges) with embarrassingly little success for what appear to be such good ideas.

A better approach is to analyze more carefully the sequence of events that lead to timeouts and overflows in order to craft avoidance strategies. This will be extremely difficult.

Finally, there is the possibility of the "best of all worlds" strategy suggested by the different row-weighting schemes. Unlike what was presented here, the strategies should be more radically different, so that known weaknesses of one can be overcome by another.

The pivot row choosing strategy of [4] has not been tried and it would guarantee a maximum increase in the dual cost whenever an increase is at all possible. Increases would also tend to occur close to the top left corner when the columns are sorted lexicographically. The possible disadvantages are: (a) there is significant extra computation involved in sorting the columns lexicographically at every iteration, and (b) the pattern of increases might make overflows more likely.

Large numbers Avoidance of large tableau entries is an interesting problem in its own right. The extreme brute-force strategy would require "trying out" every potential pivot in advance to see which minimizes the maximum resulting entry, a $\Theta(n^3)$ operation in the worst case. A simpler heuristic would be to record the number of times a non-zero value

[3] Minimization of deviation from block diagonal form, as reported in these papers, involves both a permutation of rows and columns and a partition into blocks. The partition aspect is motivated primarily by problem decomposition to take advantage of parallelism. Preliminary attempts by the author to minimize based on permutation alone, basing the measurement on an approximate partition that follows the permutation, are what is at issue here.

has been added to/subtracted from each entry since the last time that entry was 0; the current implementation would make this modification easy. Then potential pivot row/column combinations could be chosen so as to minimize the maximum of this quantity for the the respective row/column. This may also work as a tie-breaker for a strategy that seeks to increase dual cost whenever possible.

Exploiting sensitivity The use of random restarts is already a step in the direction of taking advantage of extremely erratic behavior, but it raises many questions about the statistical validity of any approach that relies on the unreliability of data. Is there a way to formalize this notion and still achieve results.

Another question is the degree to which an algorithm must be "random" in order for a restart approach to work. If the choice mechanism (as in choice of rows) is too sophisticated, the algorithm might actually get into a rut that cannot be overcome by random permutations.

Testbeds It is unsatisfying to have to work at the extremes when looking for problem instances as test cases for algorithms with this level of sophistication. Here, a single industrial instance has been analyzed to the hilt.

Random instances are not at all characteristic of the instances that need to be solved.

The ideal would be a set of instances whose behavior closely approximates that of the industrial benchmarks while still retaining some randomness. One idea along these lines is to do small perturbations of non-zeros, moving them up, down, left, or right. This can be done much more easily if the starting point is a randomly permuted benchmark — in the original ones, the non-zeros have lots of neighbors.

Acknowledgments

The setting in which this work arose is joint work with Franc Brglez and Xiao Yu Li. Franc's approach to experimental methodology, the use of classes for statistical significance, and development of experimental testbeds for thorough and reproducible findings, is much appreciated. Li provided the initial branch and bound algorithm and implementation on which this work is built, as well as a good enough stochastic search method to make it relevant.

Thanks also go to the staff of the NCSU High Performance Computing (HPC) facility for providing a hardware platform with fast dedicated processors. Eric Sills, in particular, dealt with many issues in a timely fashion.

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

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

[8] C.P. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *Proceedings of AAAI-98, Madison/WI, USA*, pages 431–437, 1998.

[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] 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.

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

[16] 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.

[17] A. Meier, C. Gomes, and E. Melis. Heavy-tailed behavior and randomization in proof planning. In *Proceedings of the Workshop on Model-Based Validation of Intelligence on AAAI Spring Symposium*, 2001.

[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] R. Williams, C.P. Gomes, and B. Selman. On the connections between backdoors, restarts, and heavy-tailedness in combinatorial search. In *Proceedings of SAT2003, Sixth International Symposium on the Theory and Applications of Satisfiability Testing, May 5-8 2003, S. Margherita Ligure - Portofino, Italy*, May 2003.

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

Table 2: Benchmark problems and single-instance results.

Optima were computed using a branch-and-bound algorithm that uses the all-integer dual simplex method to obtain lower bounds, setting a limit on the number of iterations performed before branching. With all of the binate benchmarks, no branching occurred. Time is on a dedicated Intel(R) Xeon(TM) 3.20GHz processor with 2048 MB cache.
Entries marked xxx indicate that either a timeout occurred after 900 seconds or an earlier arithmetic overflow.

Benchmark	Cols	Rows	Non-zeros	-1's	Opt	Time	Iter
Unate benchmarks							
maincont	61	50	868	0	7	< 0.1	13
mlp4	313	302	1283	0	87	< 0.1	137
lin_rom	578	544	2789	0	112	0.2	276
bench1	866	355	2821	0	113	116.2	7551
ex4inp	226	50	4709	0	5	< 0.1	31
max1024	904	916	5368	0	209	> 900	6311
exam	1113	458	9990	0	63	42.0	3624
prom2	1813	1524	10186	0	259	6.5	809
ex5	974	686	19025	0	36	> 900	33946
saucier	6203	116	340223	0	6	0.1	16
test4	5117	1435	99521	0	[80,92]	overflow	6343
Binate benchmarks							
f51m.b	175	187	2495	166	12	0.1	237
apex4.a	777	1271	3458	1066	204	0.5	562
clip.b	183	345	4197	314	14	< 0.1	49
5xp1.b	198	326	5758	307	10	5.3	3905
e64.b	571	920	6795	826	47	9.4	4459
sao2.b	276	416	7565	377	19	< 0.1	24
alu4	481	592	9866	526	49	25	5766
rot.b	887	1257	13742	1085	84	16.3	2652
count.b	421	618	14685	587	23	< 0.1	62
jac3	1280	929	17898	887	15	0.4	77
9sym	291	947	20770	932	5	< 0.1	19

Benchmark	t.o.'s	o.f.'s	min	med	mean	max	stdev
alu4	1	0	1	3.2	4.3	25.2	3.5
rot.b	1	0	2.4	10.8	18.8	267.4	29.0
e64.b	8	24	3.8	13.4	30.1	189.6	40.5

Table 3: Results of three benchmarks, each with 128 row/column permutations

Table 4: The effect of different row-weighting strategies on int-dual performance

row weight	t.o.'s	o.f.'s	min	med	mean	max	stdev
default	8	24	3.8	13.4	30.1	189.6	40.5
allow large numbers	8	22	3.1	12.5	65.1	1672.9	216.0
default w/ sorted rows	10	18	4.0	12.6	25.5	275.2	41.1
minimum non-zeros[a]	14	0	1.9	9.8	23.9	411.7	50.0
break ties w/ min non-zeros[b]	13	6	1.7	8.5	30.9	555.8	79.1

[a]Only 108 instances.Ran into wall-clock time limit because of many time-outs.
[b]Only 84 instances. Ran into wall-clock time limit because of many time-outs.

instance #	default	large numbers	sorted	min non-zeros	tie-breaking
4	time-out	time-out	12.4	3.2	time-out
11	11.3	9.8	overflow	time-out	time-out
17	overflow	overflow	27.2	5.4	time-out
18	overflow	overflow	7.9	6.9	time-out
19	overflow	1160.7	overflow	time-out	10.1
48	overflow	overflow	time-out	8.2	46.9
76	time-out	time-out	25.4	3.7	time-out

Table 5: Specific permuted instances that foil more than two row weighting strategies