Annals of Operations Research 64(1996)237–259

Solving linear programs with multiple right-hand sides: Pricing and ordering schemes

Horand I. Gassmann
School of Business Administration, Dalhousie University, Halifax, Nova Scotia, Canada B3H 1Z5
E-mail: gassmann@earth.sba.dal.ca

Stein W. Wallace
Department of Managerial Economics and Operations Research, Norwegian University of Science and Technology, N-7034 Trondheim, Norway
E-mail: sww@iok.unit.no

This paper discusses the use of different pricing and ordering schemes when solving many linear programs that differ only in the right-hand sides. This is done in a setting of what has become known as bunching or trickling down. The idea is to collect (bunch) all right-hand sides that have the same optimal basis, and to organize the search for these bases efficiently. We demonstrate that the choice of pricing rule is indeed very important, but we were not able to make conclusions regarding ordering schemes. Numerical results are given.

1 Introduction

The need to solve many linear programs that differ only in the right-hand sides can show up in different contexts. For example, a planning problem might be solved repeatedly, say, each morning, and the only change from one day to the next is the right-hand side of the LP describing the problem. The right-hand side might, for example, represent the demand for certain goods. In other contexts, the many right-hand sides show up within one single problem. If decisions have to be made before the actual values of some parameters are known, we might, for example, wish to minimize the expected value of the objective function. Project scheduling with random activity durations is an example. There exists a large collection of papers dealing with the calculation of the expected project duration time in PERT networks, see for example Kleindorfer [14], Shogan [20], Dodin [6] and Kamburowski [13].

One simple, but not always very efficient, way of solving many problems that differ only in the right-hand sides is to utilize the fact that the optimal basis for one right-hand side will probably be quite good for any other right-hand side that is similar to the first. Since the basis will be dual feasible, the dual simplex method seems most appropriate to reach all optimal bases starting from the optimal basis of a particular right-hand side.

Wets [21] introduced the idea of bunching. The principle here is to try to collect all possible right-hand sides that have the same optimal basis. In its simplest form, the setup is as follows. Pick some right-hand side, and solve the corresponding problem to optimality. Then, using the optimal basis, check for primal feasibility (and hence optimality) for all other right-hand sides. All right-hand sides having the same optimal basis are then "bunched" together. Next, pick one right-hand side not yet bunched, preferably the one with the lowest number of primal infeasible rows, and bring that problem to optimality. Then bunch for the new basis, and continue until all right-hand sides are checked.

This was later developed into what has become known as "trickling down", see Wets [22]. As before, pick some right-hand side and solve the corresponding problem to optimality, obtaining an optimal basis. Store this basis in the root of a tree, the "trickling down tree". Next, pick another right-hand side. Enter the tree at its root, and check for primal feasibility using the basis stored in that node of the tree. Since the basis is dual feasible for all right-hand sides, primal feasibility is equivalent to optimality. If at least one row has a negative entry, select such a row and pivot to obtain a new basis. This new basis becomes a new node in the trickling down tree, and the arc connecting it to its parent node is associated with the index of the row where pivoting took place. This process of pivoting and creating nodes in the tree is continued until the new problem is brought to optimality. The advantage of this approach is that when a right-hand side is being treated, we can avoid pivoting and the basis update if the present node in the trickling down tree has a child node corresponding to the index of an infeasible primal variable. We simply move to the child node and continue there.

This approach was tested by Haugland and Wallace [11] for generalized networks. In their paper they discuss the importance of picking right-hand sides in an appropriate order. Generally speaking, the result is that one should start with the problem representing the expected value of all random variables (or some other problem in the "middle"). Then problems should be picked in increasing distance from this first problem. The reason is that if one started with the extreme problems, the paths in the tree would also be extreme (in a certain sense), and later problems (right-hand sides) that were really only a few pivots away from the optimal basis were sent along long and time-consuming simplex paths.

This happens because it is generally preferable to follow an existing arc (pivot) in the trickling down tree, rather than create a new branch; i.e. whenever the current primal solution has a negative entry that coincides with the pivot row of a child of the current node, this will be an attractive row to pivot on. Of course this selection scheme

is myopic and may not be globally optimal. In particular, it may result in both large trickling down trees and a large overall number of "pivots". A large tree means increased storage, while the number of pivots indicates the number of infeasible primal solutions that have to be checked.

Stochastic decomposition, developed by Higle and Sen [12], has a related construct that we will mention briefly. During the run of their algorithm, two classes of optimizations take place. Once per main iteration, a problem is solved all the way to optimality. The dual optimal solution corresponding to this problem is stored for later use. All other problems considered in the same main iteration are solved approximately. This is done by checking which of the available dual solutions is best. Since this amounts to a restricted optimization, we end up with a bound. But more importantly for our purposes, it can be viewed as bunching the right-hand sides that use the same dual solution.

Gassmann's code MSLiP [8] uses another setup, which, in contrast to the other methods we have discussed so far, works within a primal setting. In principle, he builds up a trickling down tree, but the tree is never stored explicitly.

The method amounts to a pre-order (or depth-first) traversal of the tree of optimal bases and was designed to minimize the amount of storage. A "paradigm problem" is chosen in the beginning, and all problems are considered to be at the same level as the paradigm. Each iteration consists of one pricing step (for the paradigm problem), the ratio test to determine the leaving variable, and the basis update. Each right-hand side still in the running is then considered in turn. Since the incoming variable has already been identified – it must be the same as in the paradigm problem – a ratio test suffices to determine if the pivot step applies for the current problem or not. If the pivot step can be taken, the problem is carried along to the next level, otherwise it is left behind, along with information to reconstruct the basis that was active at this point. We propose the name "tamping" for this method.

Once the paradigm problem has been brought to optimality (or has been determined to be infeasible), the search for a new paradigm begins. The first group of candidates are those that followed the paradigm all the way to the final basis. At that point, all of those problems are either optimal or infeasible. If their infeasibilities lie in exactly the same rows as for the paradigm, then these problems have been dealt with or "bunched", otherwise a candidate for the new paradigm has been identified. Some thought could be expended on how to select the paradigm from among the possible candidates; at present we simply use the first candidate encountered.

Eventually there will be no more problems of this type, and one traverses the tree (backwards) to identify problems which have been left off at earlier stages. Whenever such a problem is encountered, the corresponding basis is recovered and all problems which were left behind at the same level are then considered together.

The advantage of this scheme is that only part of the basis tree needs to be in storage at any one time, namely the part that leads from the starting basis of the first paradigm problem to the current basis of the current paradigm.

2 Storage requirements

Both in trickling down trees and in Gassmann's implicit bunching tree, there is a question of what to store at each node (or what to leave behind together with a group of unbunched right-hand sides). The possibilities are in principle many. One could simply store the entering and leaving variables which permits one to reconstruct the relevant basis whenever needed; one could store the entire inverse basic matrix (which of course can only be done for very small problems), or one could store the basis updates in some incremental fashion. In all cases one would avoid both pricing and pivoting, although the computational burden is slightly different. The discussion of which method to use will depend both on the classical trade-off between time and storage, and on the overall setup, trickling down or implicit bunching.

We will look at three update schemes in particular and how they impact on the tamping method: the traditional product form using eta-vectors, Schur-complement updates, and the method of Bartels and Golub.

In the product form, an initial basis B_0 is explicitly factored by LU-decomposition, both factors being stored as a sequence of eta-columns. At each pivot step, another eta-vector is appended to that sequence. Retrieving the basis corresponding to an earlier pivot in the current sequence is therefore trivial, all that is required is to count back the required number of eta-vectors and to discard the tail of the sequence. No new data structures or storage locations are needed and there is no further loss of accuracy beyond the roundoff errors inherent in the product form update.

Schur-complement updates [4] also factor and store the initial basis B_0, then record the changes leading from B_0 to B_i as a square dense matrix of dimension $k_i \leq i$. Update formulas exist which allow converting S_i to S_{i+1} or S_{i-1}, but there is some processing required, and the effect of accumulated roundoff errors cannot be ignored. The alternative is to store each of the Schur-complements. The added storage requirement is moderate – and independent of the problem size – if pivot sequences are short, but grows rapidly with the length of the sequence.

The situation is worse for Bartels–Golub updates [5], which explicitly modify the LU-factorization at each step. Here, one has the choice between undoing the modifications, and storing all the inverses explicitly – surely a daunting prospect for large problems. On balance then, the product form, simple as it is, appears best suited for the needs of the tamping algorithm.

3 Pricing schemes

Our first step was to check if performance of the tamping method depended on which pricing scheme was used. There are many reasons why it might. First, a pricing method should preferably give few pivots, even if that comes at a price of higher costs per pricing operation. The reason is simple enough. For each new basis encountered, all non-bunched right-hand sides must be checked to see if they should be carried

along or left behind. The fewer pivots, the fewer times this must be done. Moreover, few pivots probably means fewer groups of right-hand sides left behind, and it might even mean fewer bunches. (The latter can happen because of degeneracy or because the same optimal basis is found several times, reached along different pivot paths.)

We studied the following possibilities:

- Most negative reduced cost. This is the standard setup.
- Random eligible column. This is the natural alternative to compare with if we want to test whether the choice of column matters at all.
- Biggest objective drop. Whenever the reduced cost is negative, we explicitly calculate the step length, and we pick the column that maximizes the absolute value of the reduced cost times the step length.
- Devex pricing. This method was suggested by Harris [10], and is also discussed in Murtagh [18]. The idea is to use a consistent framework in which to measure the distance moved during each pivot step. In the standard pricing scheme one can think of distances measured in the space of the nonbasic variables, but since the nonnegative variables change from iteration to iteration, so too does the framework in which one operates. The key is to scale the reduced costs by suitable correction factors which approximate the correct distance measure and to perform simple updates to the weights at each iteration.
- Steepest edge. The idea is to follow the steepest edge of the polytope of feasible solutions, where steepness is measured not in the space of non-basic columns, but in the space of all columns. Similar to devex pricing, one uses scale factors which multiply the reduced costs. Goldfarb and Reid [9] developed a recursion formula for the weights which makes this approach practicable.

We shall discuss numerical results in a later section. It turns out that it does indeed matter which rule is used. Which scheme to prefer depends on two measures of size. If the problem is large in terms of the number of possible right-hand sides, but small to moderate in terms of LP size (less than 100 rows), the biggest objective drop seemed to be best. If the problem is large in terms of LP size, the most negative column and steepest edge pricings came out best.

4 Ordering schemes

The paper by Haugland and Wallace mentioned earlier [11] is mostly concerned about the order in which right-hand sides are picked. That turns out to be very important when using trickling down trees. Not very surprisingly, it is of less importance for the tamping method. We chose to test the orderings listed below. All norms refer to the distance between the given right-hand side and the mean. In all cases we tested both increasing and decreasing distances.

- Lexicographic ordering.
- The L_1 norm.
- The L_2 norm.
- The L_∞ norm.
- Random order.
- "Generalized Gray codes". Gray codes are a method to order binary codes in such a way that two consecutive elements differ in exactly one bit. A similar scheme was used by Berland [1] in arranging the extreme points of an n-dimensional rectangle. In the current context, if the right-hand sides exhibit a lattice structure (for example, when they are generated from the realizations of independently distributed random variables), then one might order them such that consecutive problems differ in exactly one right-hand side element. This ought to imply that few pivots are needed to get from the optimal basis of one problem to the optimal basis of its successor, and thus might be a sensible way to order the right-hand sides. Bertsekas and Tsitsiklis [2, p. 50], give a construction of reflected Gray codes that can be easily adapted to define this order.

The results in terms of CPU and total number of pricings certainly vary quite a bit, often by several hundred percent, but we were not able to detect any patterns with respect to which scheme to use. Since the only actual effect of the chosen scheme is the picking of a paradigm problem when groups of right-hand sides left behind are later picked up, this is perhaps not very surprising. However, we must repeat that despite our inability to detect a pattern, there is a wide variation in performance.

5 Numerical results

The tests we performed fall into two broad categories. The first group of problems are ten two-stage stochastic linear programs, as outlined in table 1. In addition to these we tested seven (deterministic) problems from the netlib set [7]. Problem characteristics of these problems are collected in table 12.

The problems described in table 1 are well-known examples from stochastic programming, which is the setting that originally caused our interest in the questions asked in this paper. Further information about the problems can be found in Birge [3] or Gassmann [8]; problem 1 is also described in Louveaux and Smeers [15], problem 8 is from Mulvey and Vladimirou [17], problem 9 was taken from Sen et al. [19], and problem 10 appears in Mulvey and Ruszczyński [16].

We shall list two types of results in tables 2 to 11. The first, called "First bunching", refers to the first run through all the right-hand sides; the other, called "Overall", shows the results when adding up over all iterations needed to solve the two-stage recourse problem by Benders decomposition. (For an outline of the code used,

Table 1

Characteristics of the stochastic examples.

	Problem name	No. of rows	No. of cols	No. of RHS
1	louveaux	8	16	1280
2	sc205	23	22	800
3	scrs8	29	38	512
4	scsd8	21	140	216
5	scagr7	39	40	432
6	sctap1	61	96	216
7	scfxm1	149	225	96
8	mv3	108	303	80
9	ssn	176	706	210
10	storm	527	1260	200

Table 2

Results for example 1, louveaux.

Effects of pricing strategies				
	First bunching		Overall	
	Pricings	CPU	Pricings	CPU
Most negative	30	0.6	32306	65
Random	45	0.8	43250	92
Biggest drop	24	**0.4**	**13880**	**53**
Devex	30	0.6	29717	66
Steepest edge	**17**	0.5	26786	66
Effects of ordering schemes				
	First bunching		Overall	
	Pricings	CPU	Pricings	CPU
Lexicographical	30	0.6	32306	65
Random	31	0.7	14599	38
L2 norm, ascending	31	0.6	**8597**	**32**
L2 norm, descending		*not converged*		
Gray codes	30	0.6	32306	65

Table 3

Results for example 2, sc205.

Effects of pricing strategies				
	First bunching		Overall	
	Pricings	CPU	Pricings	CPU
Most negative	7	0.3	120	3.9
Random	7	0.3	156	4.0
Biggest drop	7	0.3	118	4.1
Devex	7	0.3	119	4.2
Steepest edge	7	0.3	126	4.2
Effects of ordering schemes				
	First bunching		Overall	
	Pricings	CPU	Pricings	CPU
Lexicographical	7	0.29	120	3.9
Random	7	0.33	117	3.8
L2 norm, ascending	7	0.29	100	**3.2**
L2 norm, descending	11	0.39	99	3.5
Gray codes	7	0.29	133	4.0

Table 4

Results for example 3, scrs8.

Effects of pricing strategies				
	First bunching		Overall	
	Pricings	CPU	Pricings	CPU
Most negative	235	1.1	301	1.9
Random	119	1.0	174	1.8
Biggest drop	**101**	**0.8**	**152**	**1.5**
Devex	143	1.1	209	1.9
Steepest edge	160	1.3	232	2.2
Effects of ordering schemes				
	First bunching		Overall	
	Pricings	CPU	Pricings	CPU
Lexicographical	235	1.1	301	1.9
Random	**116**	0.8	**162**	1.7
L2 norm, ascending	126	0.9	178	1.6
L2 norm, descending	150	0.8	186	1.7
Gray codes	194	1.1	266	1.9

Table 5

Results for example 4, scsd8.

Effects of pricing strategies				
	First bunching		Overall	
	Pricings	CPU	Pricings	CPU
Most negative	1831	6.0	1834	6.1
Random	5970	18.0	9679	29.8
Biggest drop	**115**	**0.6**	**177**	**1.3**
Devex	1342	6.4	1345	6.6
Steepest edge	270	1.5	457	3.0
Effects of ordering schemes				
	First bunching		Overall	
	Pricings	CPU	Pricings	CPU
Lexicographical	1831	6.0	1834	6.1
Random	1105	3.6	1573	5.6
L2 norm, ascending	**765**	**2.6**	1378	5.0
L2 norm, descending	848	2.9	**1178**	**4.4**
Gray codes	1188	3.9	1191	4.1

Table 6

Results for example 5, scagr7.

Effects of pricing strategies				
	First bunching		Overall	
	Pricings	CPU	Pricings	CPU
Most negative	30	1.0	343	8.3
Random	35	1.1	342	8.2
Biggest drop	30	1.0	206	**7.4**
Devex	**24**	0.9	335	8.4
Steepest edge	28	0.9	**197**	7.8
Effects of ordering schemes				
	First bunching		Overall	
	Pricings	CPU	Pricings	CPU
Lexicographical	30	1.0	343	8.3
Random	30	1.0	392	**7.8**
L2 norm, ascending	30	1.0	427	8.2
L2 norm, descending	30	1.0	348	8.3
Gray codes	30	1.0	347	8.3

Table 7

Results for example 6, sctap1.

Effects of pricing strategies				
	First bunching		Overall	
	Pricings	CPU	Pricings	CPU
Most negative	991	4.7	1061	5.9
Random	525	3.3	760	5.0
Biggest drop	**117**	**2.1**	**180**	3.3
Devex	1020	6.0	1059	7.1
Steepest edge	547	4.2	576	**3.0**
Effects of ordering schemes				
	First bunching		Overall	
	Pricings	CPU	Pricings	CPU
Lexicographical	991	4.7	1061	5.8
Random	1401	6.0	1455	7.2
L2 norm, ascending	1463	6.2	501	7.2
L2 norm, descending	**572**	**3.3**	**611**	**4.6**
Gray codes	991	4.6	1061	5.8

Table 8

Results for example 7, scfxm1.

Effects of pricing strategies				
	First bunching		Overall	
	Pricings	CPU	Pricings	CPU
Most negative	194	6.1	1734	49
Random	1344	45.9	11545	143
Biggest drop	163	9.7	**1113**	49
Devex	180	**5.1**	1434	**46**
Steepest edge	**152**	5.6	1223	49
Effects of ordering schemes				
	First bunching		Overall	
	Pricings	CPU	Pricings	CPU
Lexicographical	194	6.1	1734	49
Random	194	6.2	1703	48
L2 norm, ascending	194	6.1	1725	49
L2 norm, descending	194	6.2	1757	50
Gray codes	194	6.1	1734	49

Table 9

Results for example 8, mv3.

Effects of pricing strategies				
	First bunching		Overall	
	Pricings	CPU	Pricings	CPU
Most negative	5287	34.6	13739	**97**
Random	1245	8.6	15665	114
Biggest drop	**216**	**2.9**	**8610**	174
Devex	7756	65.9	14986	136
Steepest edge	7584	77.9	14316	177

Table 10

Results for example 9, ssn.

Effects of pricing strategies				
	First bunching		Overall	
	Pricings	CPU	Pricings	CPU
Most negative	34965	692	$> 10^{7}$ [1]	212,000
Random	89272	1775	[4]	n.a.
Biggest drop	8721	801	$> 2 \times 10^{6}$ [2]	> 350,000
Devex	7086	180	[4]	n.a.
Steepest edge	**4194**	**139**	$> 2 \times 10^{6}$ [3]	> 160,000

[1] Stopped after 10,000,000 pivots and 279 calls to the bunching routine.
[2] Stopped after over 2,000,000 pivots and 300 calls to the bunching routine. Total CPU time: over 100 hours.
[3] Stopped after over 2,000,000 pivots and 300 calls to the bunching routine. Total CPU time: over 45 hours.
[4] Stopped after first call to bunching routine.

see Gassmann [8].) We report both the number of pricing operations carried out and the number of CPU seconds used on a Sun SPARCstation 10 computer.

Of course, to the extent that the pricing and ordering schemes also affect the number of outer iterations needed, the Overall statistics are less informative than the First bunching results. Note, however, that this can only happen if we are faced with dual degeneracy. Therefore, in general, the Overall setting can be seen as just a larger example. In each column of the tables that follow, we have indicated with **bold** type whenever there is a clear winner. Some of the ordering schemes mentioned in section 4 are not reported, since orderings with respect to different norms gave very similar results to the L_2 norm.

Looking at tables 2 to 11, we see a few clear conclusions along with a number of question marks. While there are large differences in the performance of the ordering

Table 11

Results for example 10, storm.

Effects of pricing strategies				
	First bunching		Overall	
	Pricings	CPU	Pricings	CPU
Most negative	120543	**7048**	997662	64967
Random	n.a.	n.a.	n.a.	n.a.
Biggest drop	**90893**	69512	**413654**	156057
Devex	109060	8201	868357	66535
Steepest edge	106710	20351	470307	**63258**
Effects of ordering schemes				
	First bunching		Overall	
	Pricings	CPU	Pricings	CPU
Lexicographical	**120543**	**7048**	997662	64967
Random	n.a.	n.a.	n.a.	n.a.
L2 norm, ascending	122995	7098	981442	62336
L2 norm, descending	123460	7140	**945451**	**59831**
Gray codes	n.a.	n.a.	n.a.	n.a.

schemes, no obvious strategy suggests itself. On some problems, the lexicographical order is best, on others some reordering is useful. On some problems (especially problem scsd8) it is best to start near the centre and work one's way outward; on others (such as problem sctap1) it is best to move towards the centre. Even random ordering may be best in some instances (problem scagr7). The only definite conclusion regarding ordering schemes is that generalized Gray codes perform very similar to the natural lexicographical order, and so this option does not appear to be worth the cost of reordering.

Conclusions regarding pricing schemes are far more apparent. First, picking a random column does not seem useful. In no case was that option best. Devex pricing and steepest edge pricing are very similar in philosophy, and on numerical grounds, steepest edge pricing appears superior. This leaves a comparison between most negative, biggest drop and steepest edge.

Goldfarb and Reid found that steepest edge pricing is generally superior to most negative reduced cost in problems with one right-hand side, and this conclusion is upheld for the most part in our study as well. There are some exceptions, notably problems 2 and 8, both of which are somewhat anomalous. Problem 2 has very few rows and columns, while problem 8 has the fewest right-hand sides of all problems considered, so that bunching is not as effective here as on the other problems. In general, however, steepest edge wins out over most negative, particularly if the

number of right-hand sides is large. If the number of right-hand sides gets very large, the number of pivots is the deciding factor, and therefore biggest drop should win, followed by steepest edge pricing.

What came as somewhat of a surprise to us was the good showing of biggest drop on small problems, but in hindsight that should perhaps not have been so surprising after all. When the problem size is small, only few columns exist that can price out negatively, so the ratio test will not be triggered all that often. In addition, these problems are characterized by many right-hand sides, so that the number of pivots should be the prime determinant of speed.

The final question concerns the number of right-hand sides beyond which biggest drop will be superior to steepest edge in terms of CPU time. Our conjecture is that this limit depends on the number of rows and columns and will increase as the number of rows gets larger. An informal rationalization of this conjecture might go as follows. First, as the number of right-hand sides increases, the pricing operation will take a smaller and smaller fraction of the overall time. This is so because one pricing operation is sufficient to deal with an entire "bunch" of problems, but the ratio test must be performed separately for each right-hand side. Thus the strategy with the fewest iterations will win out in the long run. On the other hand, the effort in biggest drop grows faster than the effort in steepest edge if the problem dimensions are increased. Of course the work in steepest edge pricing increases with the problem size, but this is offset by the added work in each ratio test. The main point is that more columns means more eligible columns – on average – and thus a larger number of ratio tests that biggest drop must compute. Taken together, this means that for larger problems more right-hand sides are needed before biggest drop becomes cheaper than steepest edge. More computational testing is needed to make this statement more precise.

For further illustration we have included eight figures. In figures 1 to 6, we have shown CPU seconds and iterations (pricings) for examples 7, 9 and 10. The horizontal axis represents the iteration number in MSLiP. Therefore, the figures demonstrate the changing workload as the algorithm progresses. Figures 3 and 4 in particular show how the algorithm settles down as the first-stage variables approach their optimal values. Comparing these two figures we see clearly how both biggest drop and steepest edge pricing use far fewer pricings per bunching operation than most negative, but on the CPU seconds most negative comes out ahead of biggest drop, for the reasons explained in the previous paragraph.

Problem 9 appears to be extremely difficult, and we were not able to solve it to optimality. After nearly three days of elapsed time, a global iteration maximum was triggered in the most negative pricing run and the program was halted. Other pricing schemes apparently got us no further to the optimal solution and were also stopped prematurely. In addition, bunching did not seem to help as the right-hand sides are quite diverse. Figures 5 and 6 show large fluctuations from one iteration to the next. None of the methods seem to stabilize, and it is difficult to determine their relative

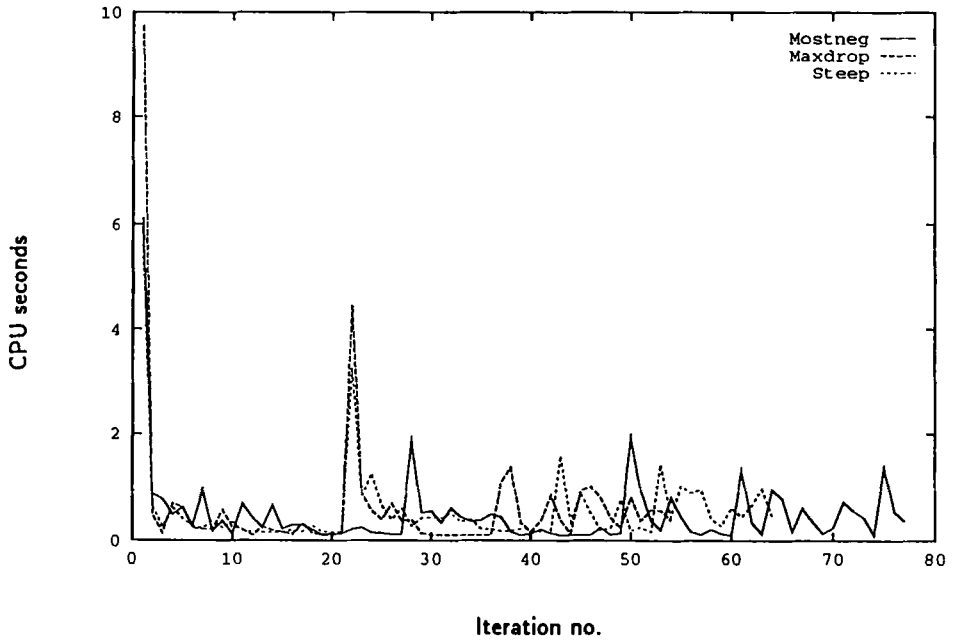

Figure 1. CPU seconds per bunching operation, problem 7 scfxm1.

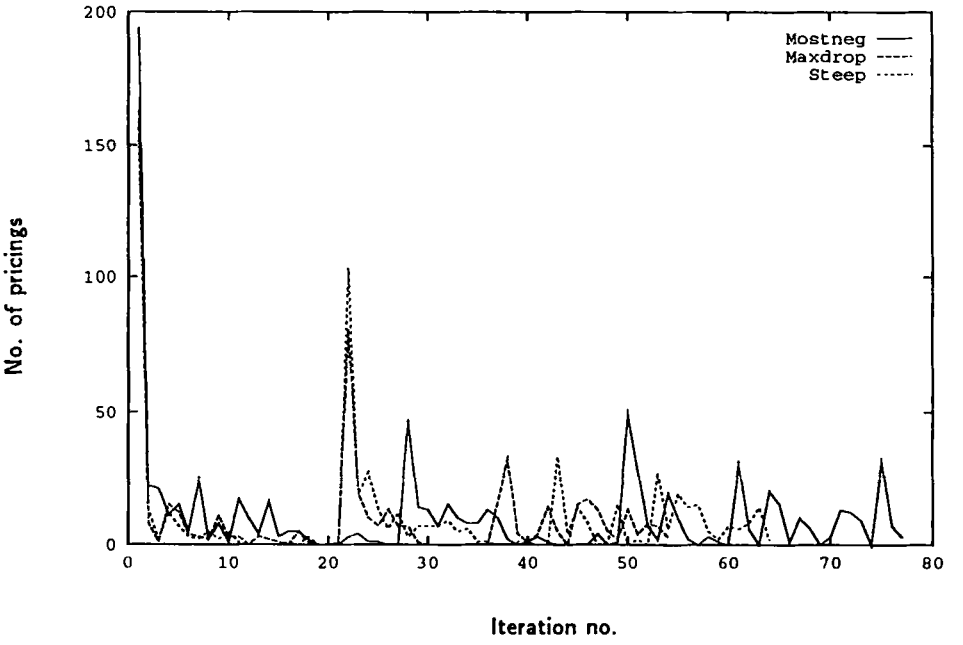

Figure 2. Number of pricings per bunching operation, problem 7 scfxm1.

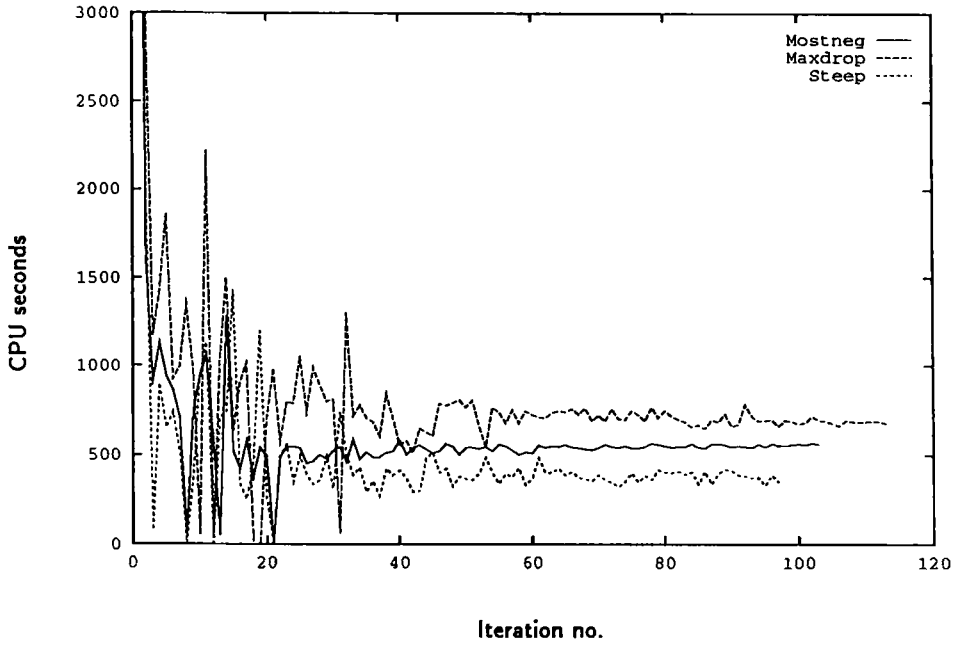

Figure 3. CPU seconds per bunching operation, problem 10 storm.

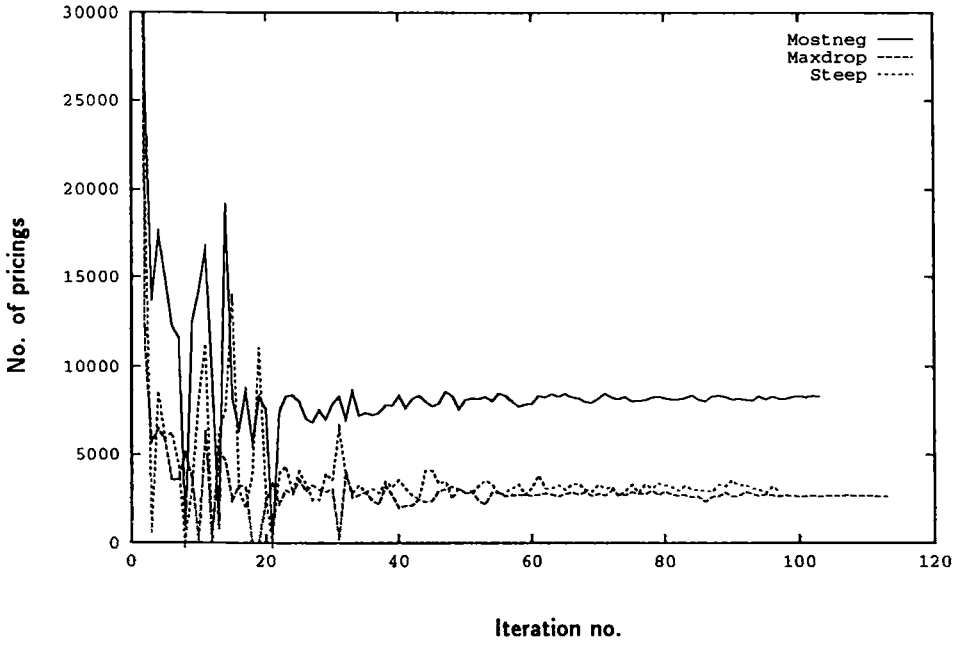

Figure 4. Number of pricings per bunching operation, problem 10 storm.

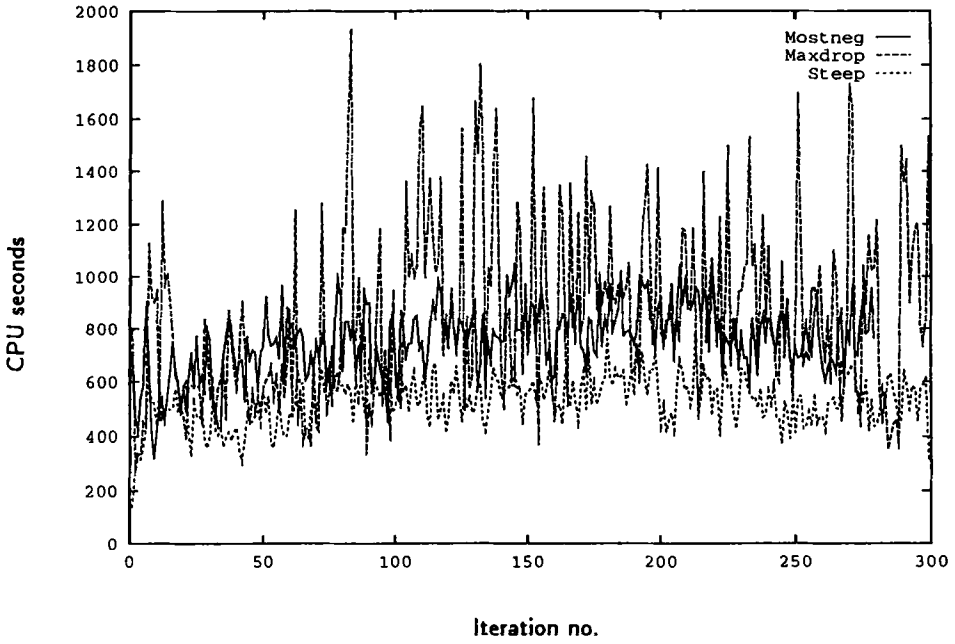

Figure 5. CPU seconds per bunching operation, problem 9 ssn.

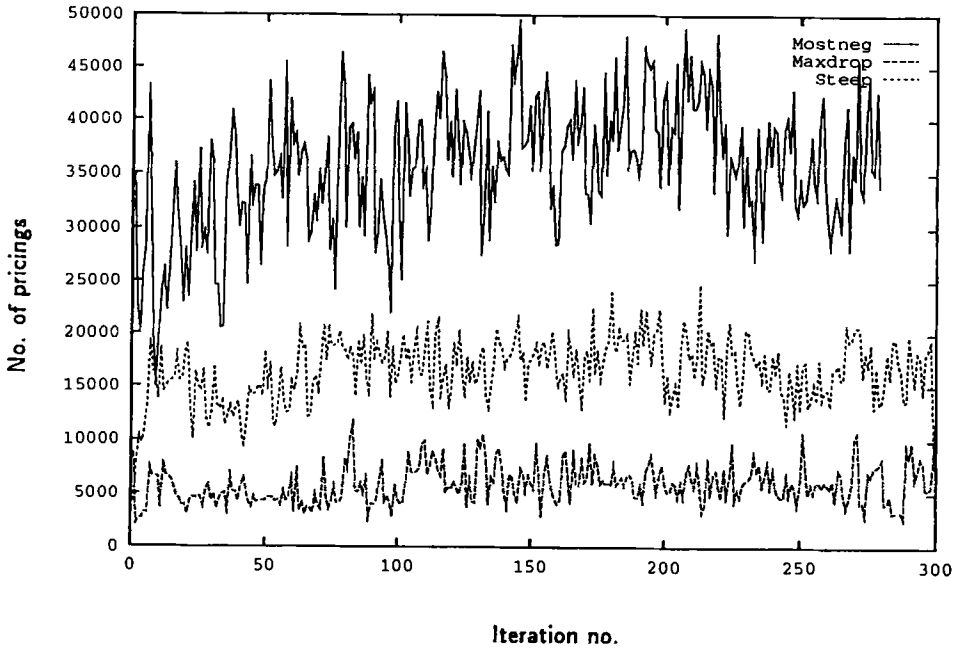

Figure 6. Number of pricings per bunching operation, problem 9 ssn.

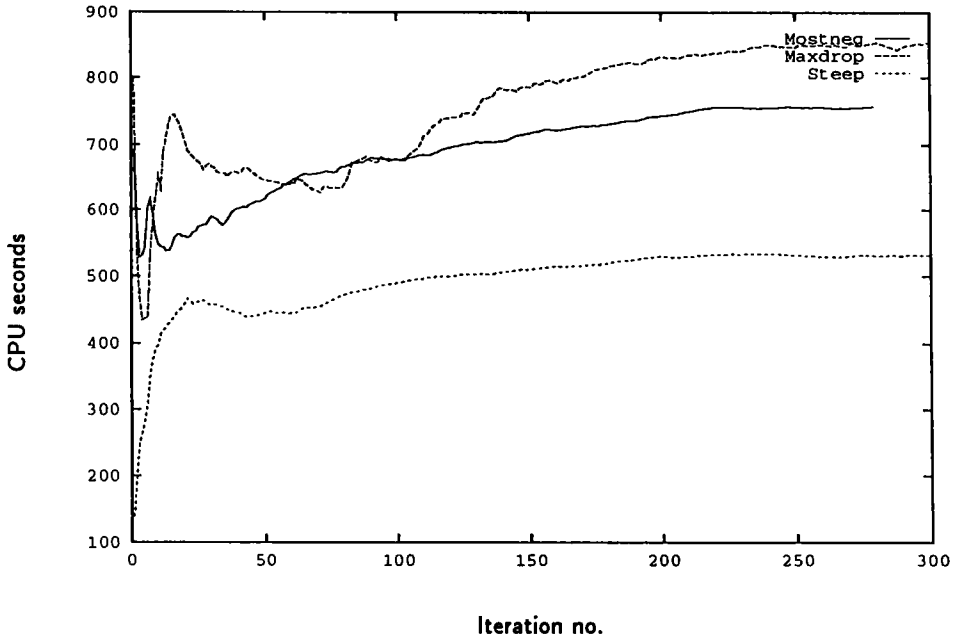

Figure 7. Cumulative average of CPU seconds, problem 9 ssn.

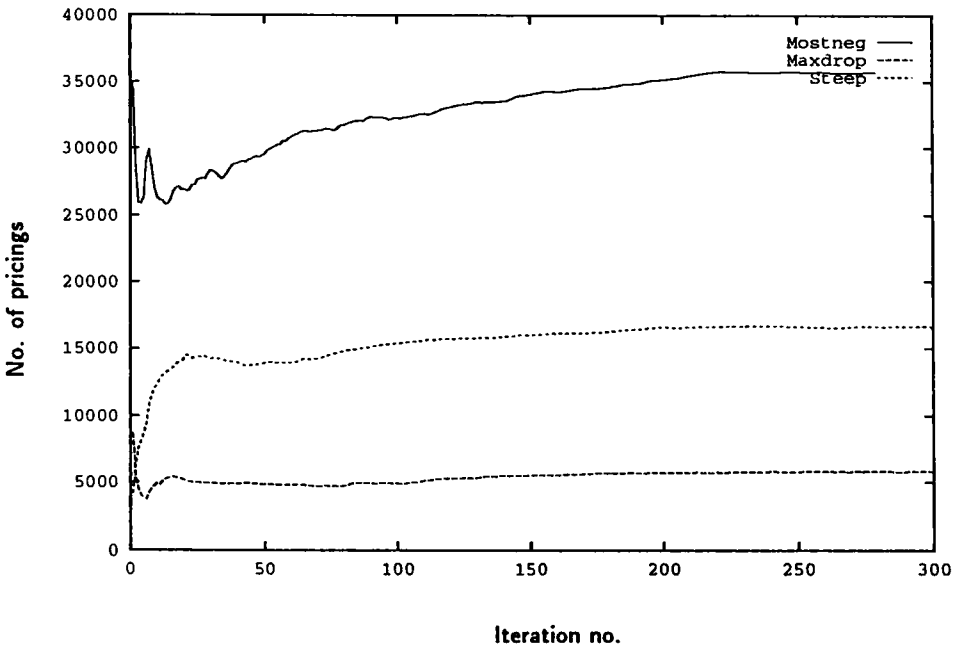

Figure 8. Cumulative average number of pricings, problem 9 ssn.

merits from the picture. For that reason we also included cumulative averages in figures 7 and 8 which make the relative rankings of the three pricing schemes much more apparent. Again, biggest drop is best in terms of the number of pricing operations, followed by steepest edge, but again biggest drop is the worst of the three in terms of CPU time used.

It bears pointing out that very little bunching takes place in problem 9, so essentially a large number of LPs are solved (from warm starts). Our findings in this case are directly comparable to Goldfarb and Reid's, and they show the same conclusions.

We ran a second set of tests using deterministic LPs from the netlib set and arbitrarily perturbing three of the right-hand sides. (These problems do not have time-staged structure, and no attempt was made to decompose them.) The problems were chosen to give a reasonable mix of sizes in the hope that additional information would

Table 12

Characteristics of the netlib problems.

	Problem name	No. of rows	No. of cols	Max. no. of RHS
11	afiro	28	32	1000
12	adlittle	57	97	1000
13	share2b	97	79	1000
14	brandy	221	249	1000
15	scorpion	389	358	1000
16	shell	537	1775	729
17	czprob	930	3523	512

be gathered concerning the performance of the pricing schemes in relation to the problem sizes. For each problem we chose four different randomizations of the right-hand sides, using 8, 27, 125 and 1000 realizations on most problems. (For the larger problems shell and czprob we had to show some restraint on the largest instances due to memory restrictions.)

The results are in general agreement with the findings for the two-stage stochastic problems, that is to say, biggest drop and steepest edge pricing tend to outperform most negative pricing on iteration counts, but the CPU times behave somewhat differently. For biggest drop pricing, the argument has been stated before. What came as a bit of a surprise was the performance of steepest edge pricing on the large problems shell and czprob. After some more checking, it was found that whenever the tamping algorithm switched to a new paradigm problem, the edge weights had to be recomputed. More work in this area might improve the picture.

The ordering schemes are again inconclusive. However, since the same ordering scheme seemed to work quite well for different numbers of realizations on the same problem, it may be possible to identify general rules to infer promising strategies

Table 13

Results for example 11, afiro.

	Effects of pricing strategies							
	8 scenarios		27 scenarios		125 scenarios		1000 scenarios	
	Pricings	CPU	Pricings	CPU	Pricings	CPU	Pricings	CPU
Most negative	16	0.09	17	0.12	20	0.31	23	2.16
Steepest edge	4	0.05	4	0.07	7	0.25	8	1.71
Biggest drop	**3**	**0.04**	**3**	**0.05**	**3**	**0.20**	**4**	**1.67**
	Effects of ordering schemes							
	8 scenarios		27 scenarios		125 scenarios		1000 scenarios	
	Pricings	CPU	Pricings	CPU	Pricings	CPU	Pricings	CPU
Lexicographical	16	0.09	**17**	**0.12**	20	0.31	**23**	**2.16**
L2 norm, ascending	16	0.08	47	0.23	48	0.53	81	2.77
L2 norm, descending	16	0.08	20	0.14	20	0.35	30	2.26
Random	**9**	**0.06**	47	0.22	20	0.34	40	2.85
Gray codes	16	0.09	20	0.13	23	0.35	71	2.82

Table 14

Results for example 12, adlittle.

	Effects of pricing strategies							
	8 scenarios		27 scenarios		125 scenarios		1000 scenarios	
	Pricings	CPU	Pricings	CPU	Pricings	CPU	Pricings	CPU
Most negative	210	2.2	608	6.5	2451	27.7	12171	159
Steepest edge	65	**1.6**	138	3.8	370	12.6	524	29
Biggest drop	**60**	1.8	**116**	3.7	**196**	**6.7**	**352**	**18**
	Effects of ordering schemes							
	8 scenarios		27 scenarios		125 scenarios		1000 scenarios	
	Pricings	CPU	Pricings	CPU	Pricings	CPU	Pricings	CPU
Lexicographical	210	2.2	608	6.5	2451	27.7	12171	159
L2 norm, ascending	210	2.2	**210**	**2.4**	**501**	**6.4**	**4076**	**60**
L2 norm, descending	210	2.2	553	6.1	2182	25.0	5792	83
Random	222	2.4	663	7.1	4089	25.2	4490	66
Gray codes	210	2.2	608	6.5	2451	27.5	12171	159

Table 15

Results for example 13, share2b.

	Effects of pricing strategies							
	8 scenarios		27 scenarios		125 scenarios		1000 scenarios	
	Pricings	CPU	Pricings	CPU	Pricings	CPU	Pricings	CPU
Most negative	144	2.3	434	6.9	1176	20.3	4794	98
Steepest edge	65	2.0	92	4.7	204	11.5	326	33
Biggest drop	**27**	**1.1**	**49**	**2.1**	**84**	**4.6**	**125**	**17**
	Effects of ordering schemes							
	8 scenarios		27 scenarios		125 scenarios		1000 scenarios	
	Pricings	CPU	Pricings	CPU	Pricings	CPU	Pricings	CPU
Lexicographical	144	2.3	434	6.9	1176	20.3	4794	98
L2 norm, ascending	144	2.4	200	3.6	752	13.8	**1344**	**37**
L2 norm, descending	144	2.4	**150**	**2.8**	1983	32.6	9624	180
Random	240	4.0	375	5.7	**518**	**10.5**	5543	109
Gray codes	**140**	**2.2**	424	6.7	1352	22.5	4815	98

Table 16

Results for example 14, brandy.

	Effects of pricing strategies							
	8 scenarios		27 scenarios		125 scenarios		1000 scenarios	
	Pricings	CPU	Pricings	CPU	Pricings	CPU	Pricings	CPU
Most negative	680	27	2522	101	11756	470	85222	3506
Steepest edge	329	31	1246	122	5129	503	31943	3518
Biggest drop	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.
	Effects of ordering schemes							
	8 scenarios		27 scenarios		125 scenarios		1000 scenarios	
	Pricings	CPU	Pricings	CPU	Pricings	CPU	Pricings	CPU
Lexicographical	680	27	2522	101	11756	470	85222	3506
L2 norm, ascending	680	27	2573	104	8170	347	65761	2724
L2 norm, descending	680	27	**1918**	**78**	8896	376	49554	2124
Random	**606**	**25**	2026	83	**7710**	**332**	**29186**	**1362**
Gray codes	680	25	2522	101	11756	471	85222	3500

Table 17

Results for example 15, scorpion.

	Effects of pricing strategies							
	8 scenarios		27 scenarios		125 scenarios		1000 scenarios	
	Pricings	CPU	Pricings	CPU	Pricings	CPU	Pricings	CPU
Most negative	164	9.6	420	26.6	1011	72.6	4492	400
Steepest edge	**44**	11.1	**117**	43.6	**252**	120.7	**372**	262
Biggest drop	53	**6.4**	149	**19.7**	348	**53.0**	631	**164**
	Effects of ordering schemes							
	8 scenarios		27 scenarios		125 scenarios		1000 scenarios	
	Pricings	CPU	Pricings	CPU	Pricings	CPU	Pricings	CPU
Lexicographical	164	9.6	420	26.6	1011	72.6	4492	400
L2 norm, ascending	164	9.6	**133**	**8.1**	190	16.7	257	**60**
L2 norm, descending	164	9.6	302	21.3	**123**	**15.8**	**137**	69
Random	**139**	**7.7**	223	13.8	586	46.1	225	73
Gray codes	164	9.7	420	26.8	1011	72.9	4492	398

Table 18

Results for example 16, shell.

	Effects of pricing strategies							
	8 scenarios		27 scenarios		125 scenarios		729 scenarios	
	Pricings	CPU	Pricings	CPU	Pricings	CPU	Pricings	CPU
Most negative	32	**4.0**	**69**	**9.5**	**102**	**21.1**	137	**68.1**
Steepest edge	**28**	21.0	71	83.6	110	197.6	**122**	272.6
Biggest drop	46	12.3	91	27.6	140	39.8	233	103.5

a priori. The present paper did not deal with this issue. (It should also be pointed out that problem shell has a slightly different structure from the other problems in that the randomness was introduced into the bounds of some of the variables. The tamping algorithm goes through unchanged for this kind of problem (since bounded variables are a special form of constraint), but the routine employed to sort the right-hand sides could not deal with it.)

Table 19

Results for example 17, czprob.

	Effects of pricing strategies							
	8 scenarios		27 scenarios		125 scenarios		512 scenarios	
	Pricings	CPU	Pricings	CPU	Pricings	CPU	Pricings	CPU
Most negative	2444	**513**	7635	**1611**	35126	**7439**	139683	**29668**
Steepest edge	1640	1897	5385	6439	**21803**	26488	**83872**	102,000
Biggest drop	**1567**	14025	**5446**	45631	n.a.	n.a.	n.a.	n.a.
	Effects of ordering schemes							
	8 scenarios		27 scenarios		125 scenarios		1000 scenarios	
	Pricings	CPU	Pricings	CPU	Pricings	CPU	Pricings	CPU
Lexicographical	2444	513	7635	1611	35126	7439	139683	29668
L2 norm, ascending	2444	513	**3598**	**772**	**13091**	**2833**	**57257**	**12256**
L2 norm, descending	2444	513	7734	1631	35735	7568	140801	30389
Random	**1209**	**257**	3757	799	13914	2994	251262	53153
Gray codes	2444	514	7635	1610	35126	7454	139683	29433

6 Concluding remarks

The purpose of this paper has been to discuss some computational aspects of solving many LPs that differ only in the right-hand sides. Our interest has been to solve stochastic programming problems, although the results apply to other contexts as well.

We have discussed a number of algorithmic principles for solving many LPs, and shown how advanced versions of warm starts, such as bunching, trickling down and implicit bunching can be used. We used Gassmann's code MSLiP and tested different pricing and ordering schemes. The latter gave us no clear results.

However, we found that it is indeed worthwhile considering pricing regimes other than the most negative reduced cost. The alternative approaches cost (substantially) more per pricing, but can still pay off because of savings related to the large number of problems that are to be solved. For almost all problems, the best idea was to take the column with the largest drop in the objective. Only for problems with a large number of rows did it not come out best in terms of CPU. In that case, the steepest edge approach seemed best.

Acknowledgements

The first author was supported in part by a grant from the Natural Sciences and Engineering Research Council of Canada (NSERC). Much of the work reported in

this paper was done while the first author spent a sabbatical leave at the University of Trondheim. The host institute's hospitality and use of computer equipment are gratefully acknowledged. Bjørn Nygren and Kjetil Haugen contributed valuable discussions and insight. We further thank the associate editor and two anonymous referees for helpful comments on an earlier draft of this paper.

References

[1] N.J. Berland, Stochastic optimization and parallel processing, Ph.D. Thesis, Department of Informatics, University of Bergen, February 1993.
[2] D.P. Bertsekas and J.M. Tsitsiklis, *Parallel and Distributed Computation*, Prentice-Hall, Englewood Cliffs, New Jersey, 1989.
[3] J.R. Birge, Decomposition and partitioning methods for multistage stochastic linear programs, Operations Research 33, 1985, 989-1007.
[4] J. Bisschop and A. Meeraus, Matrix augmentation and partitioning in the updating of the basis inverse, Mathematical Programming 13, 1977, 241-254.
[5] V. Chvátal, *Linear Programming*, W.H. Freeman, New York, 1980.
[6] B. Dodin, Bounding the project completion time in PERT networks, Operations Research 33, 1985, 862-881.
[7] J.J. Dongarra and E. Grosse, Distribution of mathematical software via electronic mail, Communications of the ACM 30, 1987, 403-407.
[8] H.I. Gassmann, MSLiP: A computer code for the multistage stochastic linear programming problem, Mathematical Programming 47, 1990, 407-423.
[9] D. Goldfarb and J.K. Reid, A practicable steepest-edge simplex algorithm, Mathematical Programming 12, 1977, 361-371.
[10] P.M.J. Harris, Pivot selection methods of the devex LP code, Mathematical Programming 5, 1973, 1-28.
[11] D. Haugland and S.W. Wallace, Solving many linear programs that differ only in the right-hand side, European Journal of Operational Research 37, 1988, 318-324.
[12] J.L. Higle and S. Sen, Stochastic decomposition: An algorithm for two-stage linear programs with recourse, Mathematics of Operations Research 16, 1991, 650-669.
[13] J. Kamburowski, Bounds in temporal analysis of stochastic networks, Foundation of Control Engineering 10, 1985, 177-185.
[14] G.B. Kleindorfer, Bounding distributions for a stochastic acyclic network, Operations Research 19, 1971, 1586-1601.
[15] F. Louveaux and Y. Smeers, Optimal investments for electricity generation: A stochastic model and a test problem, in *Numerical Techniques for Stochastic Optimization*, Yu. Ermoliev and R.J-B Wets, eds., Springer, 1988, pp. 445-454.
[16] J.M. Mulvey and A. Ruszczyński, A new scenario decomposition method for large-scale stochastic optimization, Technical Report, Department of Civil Engineering and Operations Research, Princeton University, May 1992.
[17] J.M. Mulvey and H. Vladimirou, Stochastic network optimization models for investment planning, Annals of Operations Research 20, 1989, 187-217.
[18] B.A. Murtagh, *Advanced Linear Programming: Computation and Practice*, McGraw-Hill, New York, 1981.
[19] S. Sen, R.D. Doverspike and S. Cosares, Network planning with random demand, Technical Report, SIE Department, University of Arizona, Tucson, AZ, December 1992.
[20] A.W. Dhogan, Bounding distributions for a stochastic PERT network, Networks 7, 1977, 359-381.
[21] R.J-B Wets, Stochastic programming: Solution techniques and approximation schemes, in *Mathematical Programming: The State of the Art*, A. Bachem, M. Grötschel and B. Korte, eds., Springer, Berlin, 1983, pp. 566-603.
[22] R.J-B Wets, Large scale linear programming techniques, in *Numerical Techniques for Stochastic Optimization*, Yu. Ermoliev and R.J-B Wets, eds., Springer, Berlin, 1988, pp. 65-94.