Computational Results with Another Method of Solving Very Long and Thin LPs

Petra Mutzel

CRPC-TR90091
May, 1990

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

Computational Results with Another Method of Solving Very Long and Thin LPs

Petra Mutzel*

May, 1990

*Supported by the Center for Research on Parallel Computation, Rice University, Houston, Texas 77251 through National Science Foundation Cooperative Agreement #CCR-8809615.

Contents

1 Introduction

The initial goal was to parallelize the Simplex code CPLEX. A good approach seemed to be to study special types of linear programs. We decided to study LPs of the type

$$\begin{aligned} \min \quad & c^T x \\ \text{s.t.} \quad & Ax = b \\ & \ell_x \leq x \leq u_x \ , \end{aligned} \tag{1}$$

whereby the number of columns of the matrix A is much bigger than the number of rows. Such types occur in crew-scheduling problems of airlines. The examples which we studied had about 800 rows and up to five million columns.

If you try to solve such big problems, you will have a lot of difficulties: the needed storage space may be too big, high degeneracy occurs and it takes an enormous amount of time to try to solve. But if you cut the problem (for example down to 20,000 columns), solve that and then use the solution of this problem for a bigger one (say 100,000 columns), you will be able to solve it much faster.

To solve the 200,000 problem, we took the basics of the 100,000 solution, but that still took too long to solve. That observation lead to the basic idea of our approach.

The basic idea to solve (1): Select a smaller number of columns $J = \{i_1, \ldots, i_k\}$ of the matrix A. Construct the sub-LP:

$$\begin{aligned} \min \quad & c_J^T y \\ \text{s.t.} \quad & A_J y = b \\ & \ell_J \leq y \leq u_J \ , \\ \text{whereby} \quad & A_J \text{ is the matrix of all selected} \\ & \text{columns of } A \ . \end{aligned} \tag{2}$$

Solve this subproblem (2).

If the solution of the subproblem is optimal, then it is feasible for LP(1), but not necessarily optimal. Therefore, we have to price out all not selected columns of the matrix A. If we find columns with negative reduced costs, we choose new columns

out of A, generate a new subproblem with A_J, and solve that new subproblem. The objective function value improves with every solution of a subproblem. So, with some luck, we will eventually have positive reduced costs for every column of A.

2 The development of the algorithm

2.1 The basic algorithm

0. Choose the size of the subproblems (number of columns of SLP = smac)

1. Select the first "smac" columns (= J) out of A_{LP}

2. Generate the sub-LP:

$$\begin{aligned} \min \quad & c_J^T y \\ \text{s.t.} \quad & A_J y = b \\ & \ell_J \leq y \leq u_J \end{aligned}$$

3. Load the sub-LP into CPLEX (callable library)

4. Optimize the sub-LP with CPLEX

5. Get the solution and the basis B of the optimal solution

6. Price out the columns of the matrix A.
   If all reduced costs are positive → STOP!
   The optimal solution of the last SLP is also
   the optimal solution of LP(1).
   Otherwise: go to 7.

7. Select new columns J out of A_{LP} (choosing the x with the most negative reduced costs first, but with $B \subset J$).
   Go to 2.

2.2 Methods of variation

I decided to describe the development of the algorithm as a "history of growth." The basic idea was implemented, test runs were performed with various problems, and the behaviour was observed.

The problems fit1d and fit2d behaved very well. With aa6 the problems started: I observed a kind of "cycling", so something had to be changed (2.2.1). I also implemented the steepest-edge pricing for choosing the new columns, which had to be put into the new sub-LPs (2.2.3). There were also a lot of the parameters to change.

First, I will describe the "good ones," those which lead to better results than without using them.

2.2.1 Cycling

In problem aa6, I observed a kind of cycling: The objective function value was optimal, but the algorithm did not stop. I observed that the same subproblems were solved over and over again. For example, in every iteration with number $2n$ ($n = k, k+1, k+2, \ldots$) the columns I_1 entered the subproblem, and in iterations with numbers $2n + 1$ the columns I_2 entered the subproblem (for $n \geq k$, $I_1, I_2 \subset I$, $I =$ all columns of A). The idea was to mix these columns ($I_1 \cup I_2$) together in one subproblem.

Whenever "circling" occurs, do:

- Choose just 80% of the columns for the new subproblem out of the nonoptimal variables and "fill-up-rest" (the factor $0.8 * \text{smac}$ is called: fill-up-factor)
- Choose the other 20% out of these columns, which entered the current subproblem as nonoptimal columns.

If this did not help to stop the program or to get away from the sticking point, I put a restart in, which is: I keep just the basis of the last sub-problem and select randomly the rest of the variables. This gave much better results with aa6 and also with aa 20,000p. I ran the problems with various fill-up-factors. The best one seemed to be around (0.7 – 0.8) *smac. (smac is the number of variables in the subproblem).

2.2.2 Iteration-limit

Another idea was to not optimize each subproblem completely. CPLEX just performed "maxitcp" iterations for each subproblem. But the question was how to choose maxitcp.

With the very small problem "fit1d", the best results were with pricing = 0 and cutting down the iteration-limit to $(0.3 * \text{smac})$ (see Table 1). That's a small number in this case.

On the other problems I got the best results with maxitcp = smac/10 or smac/2.

2.2.3 Steepest Edge Pricing

The next idea was to use another kind of pricing: steepest edge. Before we chose the columns with minimum reduced costs. In steepest edge we choose the minimum normalized reduced cost columns, which corresponds to choosing the edge of steepest descent of the objective function.

But to compute the normalized reduced costs, you have to calculate the norms of each column. This takes a lot of time, but you can get a better criterion for how to choose the best ones out of the nonoptimal variables. In the following, I call this implemented procedure "ONENORM".

In Table 3.1.4 you can see that the time for calculating these norms increased very rapidly. Therefore, I tried one more idea ("SAVEPRICTIME"). Instead of calculating all the norms in every case I just calculated them if the number of nonoptimal variables was much greater than the number of columns to be chosen. If the number of nonoptimals is very high, then it suffices to compute the norms of just these variables, which have the best reduced costs, and then choose the best of them.

The algorithm is now:

- Sort all the variables in decreasing order by reduced costs.
- If the number of nonoptimal variables is very high, then:

  - Take the best (2 * smac) of them
  - Compute the norms of these variables
  - Sort them again
  - Take the best ones into the new subproblems

- Else
  - Take all of them into the new subproblem.

I got very good results in doing this. The time decreased, especially in subproblems with a smaller size (comp. aa50,000p, aa100,000p).

These were the "good" variations I tried. Now I describe the other ideas, which lead to worse results or lead to no significant change.

2.2.4 Number of Restarts

To change the number of restarts, either do more or less of them. If you do a restart too often, you can't see any improvements in the behaviour of the algorithm. If the algorithm notices that there was no improvement in the objective-value for several iterations, it will do a restart.

I tried to vary this parameter, MAXCIRC, but the initial value (= 4) seemed to be the best solution (but see 3.1.5).

2.2.5 Prefer old variables

The algorithm ended with better results if it did not fill up all free variables with new ones. That lead to the suggestion that it must be better to have the old variables in the subproblem if possible, and just choose the nonoptimal variables as new ones. This lead to an algorithm that was very badly behaved. The number of nonoptimal variables didn't decrease, but instead went up and down all the time.

2.2.6 Fill-up-rest

The best way to fill up the rest of the columns of the subproblem depends on the problem. Sometimes it seemed better to choose the first ones (fitld), or the last ones. I decided to take every tenth column and to start randomly on one of the beginning columns.

2.3 The Algorithm

2.3.1 Main

1. Read the problem (LP).
   Read the number of columns for the sub-LPs (= smac).

2. Select the first "smac" columns out of A_{LP} and initialize the subproblem
   (ARRAYS: select, boolsel, statvar)

3. Set parameters for the sub-LP
   (maxitcp, maxnopt=fill-up-factor, MAXCIRC, ...)

4. (Procedure Gensubprob)
   Generate the sub-LP for loading into CPLEX
   (copy the selected columns into one memory location)

5. Load sub-LP into CPLEX (callable library)

6. Optimize sub-LP with CPLEX

7. Get the solution and the basis B from CPLEX

8. (Procedure Selectcol)
   Pricing: Generate the reduced costs of all columns of A_{LP} not in
   sub-LP. If all reduced costs are positive $\rightarrow$ STOP!
   Optimal!

Otherwise: Select new columns out of A_{LP}: the most negative ones: If necessary, then generate the ("steepest edge") norms.

2.3.2 Selectcol

1. Select all columns which were in the basis of the last sub-LP
2. If "restart", then select the first columns $\rightarrow$ STOP.
3. (Pricing)
   Calculate the reduced costs of every nonselected column if the solution of the last sub-LP was "optimal."
   Otherwise (Case: Maximal iteration limit was reached before the solution was found), calculate the same for the selected columns also.
4. If there are more columns with negative reduced costs than needed, sort the columns in increasing order of reduced costs ("Bucketsort").
5. Take the best of them (at most $2 * \text{smac}$),calculate the norms of columns, and calculate the (steepest-edge) reduced normalized costs. Sort them again.
6. Select the columns out of the buckets, but at most either (fill-up-factor *smac)(if "circling" is active) or smac.
7. If "circling" is active, then select the saved columns ($\equiv$ nonoptimal columns of the last sub-LP).
8. If (#selected $<$ smac) then fill-up the rest "randomly".

2.3.3 Gensubprob

1. For all columns which are new selected ($\equiv$ were not selected in the last sub-LP), generate the right-hand side.
2. For all selected columns generate the arrays boolsel, statvar, scstat.

3. For all columns which were selected before, but not now, generate the right-hand-side.

4. For all selected columns generate the data to load into CPLEX for matrix $A_{\text{S-LP}}$ (smatbeg, smatcnt, smatind, smatval), the objective function coefficients (sobjx), and upper/lower bounds (sbdl, sbdu).

3 Results

3.1 The Test Problems and their Results

In the following I describe the various test problems. Here you can see the results obtained from trying the various ideas on every particular test problem.

3.1.1 fit1d

3 started with a small one:

# rows:	24
# columns:	1026

CPLEX needed 759 iterations to solve it (pricing = 1) in 50.05 seconds and 1082 iterations with pricing = 0 in 9.68 seconds.

The method did not lead to better results in most cases. The test problem is too small. But here you can see the differences with the various sizes of the maximum iteration limit for every subproblem. (see Table 1).

The best solution I got here was a user time of 6.77 seconds. I reached that by choosing the number of variables of the subproblem (smac) equal to 100, and a maximum iteration limit per subproblem of 30 (0.3 * smac) (pricing = 0). You can read the following out of the table: In this case there were 45 subproblems to solve. Twenty-four of them stopped before an optimal solution was found, and the average iteration count was 23. None of the subproblems did more than 30 iterations (That's

clear, because the maximum iteration limit was 30). To solve the test problem, 1035 iterations were needed, which took a total of 6.77 seconds.

3.1.2 aa6

# rows:	541
# columns:	4486

CPLEX needed 4581 (2137) iterations (with 2137 in Phase I) to solve it (with pricing = 6) in a time of 38.86 seconds on the CRAY-Y-MP. This is a very hard problem.

I got the best results by choosing a size of 2500 variables for the subproblem, and it took 51.77 seconds to solve. Before implementing the steepest-edge pricing, I got the following results:

smac	LP's to solve	Iterations	usertime (sec.)
1500	28	9099	90.73
2000	14	6133	59.55
2500	12	4764	51.77
3000	12	4939	57.67

3.1.3 aa20,000p

This is the smallest of the problems we are interested in.

# rows:	837
# columns:	20,000
# const. nonzeros:	149,371

CPLEX needed 4759 (1020 Phase I) iterations in 58.38 seconds to solve it with pricing = 0 and 1726 (513 Phase I) iterations in 32.19 seconds to solve it with pricing = 6 (steepest-edge pricing) on the CRAY-Y-MP.

Results:

	without ONENORM			with ONENORM/before SAVEPRICTIME		
smac	LP's	iterations	usertime/sec.	LP's	iterations	usertime/sec.
900	21	821	13.83	17	743	13.33
1000	16	800	12.15	18	853	16.97
1200	16	770	13.53	16	786	13.50
1400	18	1008	18.97	11	680	10.85
1600	14	712	14.79	12	749	14.62
1800	15	761	17.37	13	752	17.03
2000	13	711	16.38	15	760	20.08
3000	9	695	17.98	12	691	22.67
4000	13	840	33.48	11	848	30.56
5000	12	1004	41.14	13	1008	43.55

If you take a look at the distribution of the times (see Table 2), you can notice that they changed: the procedure CPLEX needs much less time than without ONENORM. The procedures Selectcol and Bucketsort now needed 40% of the whole usertime (especially if you choose smaller subproblems. In the bigger ones, there was no significant change.). So it is possible in this case to decrease the time by paralleling and vectorizing the algorithm.

3.1.4 aa50,000p

The first 20,000 columns of this problem are exactly that of the aa20,000 test problem.

# rows:	837
# columns:	50,000
# const. nonzeros:	380,535

The best way to solve this problem with CPLEX was first to solve the aa20,000p problem and then to start the aa50,000p problem with the optimal solution of that.

If you add both together you get a total iteration count of 2623, and a usertime of 99.45 seconds (pricing = 6) on the CRAY-Y-MP.

Some results with the new algorithm are in the tables below.

smac	LP's	iterations	time for optimizing the subproblems/sec.	whole usertime/sec.
1000	107	3396	85.74	578.40
1500	49	2581	62.61	209.76
2000	34	2374	59.66	102.95
2500	27	2606	63.02	104.45
3000	23	2121	62.98	79.01
3500	23	1886	67.89	76.90
4000	20	2431	71.71	88.35
4500	25	1969	93.87	104.69
5000	20	2047	85.97	95.57
5000	21	2109	103.13	113.46

If you look up the results with ONENORM, the time for optimizing the subproblems with CPLEX is relatively constant, whereas the whole usertime increases very much (especially for the smaller subproblems).

Especially in these cases, the implementation of SAVEPRICTIME is very efficient (see last table in this section 3.1.4).

I got these results without any iteration-limit and with a fill-up-factor of (0.8 * smac). I then tried an iteration-limit of (smac/10) and (smac/20). You can see the results in the following tables:

Results with ONENORM (before SAVEPRICTIME)

	Max. it-limit = smac/20				Max. it-limit = smac/10			
smac	LP's	iterat.	optim.	ut/sec.	LP's	iterat.	optim.	ut/sec.
1500	–	–	—	–	49	2581	62.81	210.40
2000	31	1916	50.87	115.15	31	2118	54.26	96.68
2500	32	2280	68.42	122.50	27	2587	63.02	104.59
3000	28	1808	69.33	90.68	19	2160	54.43	69.33
3500	12	1358	46.83	53.87	21	1260	61.91	70.19
4000	22	1951	74.47	98.64	20	2431	71.88	88.56
4500	18	1780	69.39	77.72	25	1969	94.11	104.94
5000	21	2205	94.09	104.55	20	2047	86.09	95.71
5500	21	2689	108.87	119.76	21	2109	103.30	113.62

The results with smac = 3500 and max. iteration-limit = smac/20 are fantastic, but it may be luck. Compared with the results with no iteration-limit you can see an improvement, especially for the small subproblems (smac $<$ 4000). So I decided to test runs with an iteration-limit of smac/10.

I also decided to try out another fill-up-factor: to fill 90% of the new subproblem with non-optimal variables if the algorithm notices that the objective value didn't change. The results were worse (before, this factor was 80%).

Results with ONENORM, Max. it-limit = smac/10 and
Fill-up Factor "maxnopt" = 0.9 * smac

smac	LP's	iterations	optim./sec.	ut/sec.
1500	62	2791	80.57	305.18
2000	49	2742	86.50	226.72
2500	31	2249	64.97	124.30
3000	28	2189	73.35	91.26
3500	21	1760	61.88	70.16
4000	27	2257	92.66	112.04
4500	25	1969	94.13	104.97
5000	20	2047	85.94	95.53
5500	21	2109	103.05	113.37

I tried it again with a fill-up factor of (0.7 * smac), but I had implemented SAVEPRICTIME by that time.

Results with SAVEPRICTIME, Max. it-limit = smac/10

	maxnopt = 0.7 * smac				maxnopt = 0.8 * smac			
1500	36	2567	51.56	91.57	35	2679	51.58	90.86
2000	35	2593	62.62	95.26	26	2312	47.71	78.66
2500	22	1851	47.03	63.85	25	2194	56.06	74.70
3000	25	2161	65.14	83.42	22	1887	58.00	75.45
3500	21	1760	?	70.23	21	1760	61.79	70.08
4000	24	2200	83.92	103.38	22	2290	77.84	96.50
4500	25	2137	98.84	109.73	25	2137	98.66	109.52
5000	17	1884	75.25	83.84	17	1884	75.03	83.61
5500	22	1971	105.38	116.15	22	1971	105.26	116.01

The differences are not very big between the two, except that the fill-up-factor (0.8 * smac) is a little better than (0.7 * smac). But you can see the big improvement of the concept "SAVEPRICTIME", especially on the smaller subproblems.

3.1.5 aa100,000p

# rows:	837
# columns:	100,000
# const. nonzeros:	770,645

The first 20,000 columns of this problem are exactly that of the aa20,000p test problem. Therefore, first solve the smaller problem and use the optimal solution of that as advanced basis to solve the bigger problem. The total iteration count was then 3956 and the test time 531 seconds on a CRAY-2.

First, I ran this problem without ONENORM. You can see the results with no iteration-limit and an iteration-limit of smac/10 in the table below. (see Table 3)

Cutting down the iteration-limit to smac/10 seemed to be a good solution. I tried the same with smac/20, but the results were worse.

TABLE 3
Results Without ONENORM

	No iteration-limit			Max it.-limit = smac/10			
smac	LP's	iter.	ut/sec.	LP's	iter.	optim./sec.	ut/sec.
3000	62	8261	576.30	–	–	–	–
3200	56	9372	578.97	53	6931	456	507.22
3400	36	10,289	454.29	48	6526	430	475.20
3600	40	5289	389.74	37	4891	322	356.62
3800	43	7087	490.35	41	6020	410	449.79
4000	39	5652	431.92	31	5169	?	357.79
4200	35	5715	414.55	40	5805	423	461.63
4400	36	6356	452.09	38	5300	411	448.43
4600	44	5424	512.89	14	4111	184	198.79
5000	39	5406	507.37	–	–	–	–
5500	35	6348	540.32	–	–	–	–

After I had implemented ONENORM, the iteration number went down, but not the user time.

Results with ONENORM,
Max. it-limit = smac/2, pricing = 6

smac	LP's	iter.	optim./sec.	ut/sec.
3400	42	5209	339.60	498.48
3600	40	5630	360.91	640.69
3800	33	4878	306.23	433.13
4000	32	4804	311.88	447.09
4200	39	6089	426.80	621.72

Here I seemed to get better results with an iteration-limit of smac/2 instead of smac/10.

The concept of SAVEPRICTIME got the best results. I tried this with a fill-up-factor of (0.7 * smac) and (0.8 * smac).

Results with ONENORM + SAVEPRICTIME, Max. it-limit = smac/2

	maxnopt = 0.7 * smac				maxnopt = 0.8 * smac			
smac	LP's	iter.	optim/sec.	ut/sec.	LP's	iter.	optim./sec	ut/sec.
3000	34	4375	244.10	336.03	58	11,742	553.01	894.77
3200	39	4450	297.14	385.69	31	4562	255.99	358.59
3400	32	4882	275.11	350.90	35	4816	292.77	395.46
3600	38	6029	355.18	471.69	39	5434	349.34	518.88
3800	29	4448	264.62	334.56	35	5029	320.71	407.19
4000	34	5109	329.74	420.10	38	5020	356.69	452.05
4200	31	4762	313.48	433.00	39	5048	384.82	543.40
4400	33	4749	351.89	428.01	29	5035	330.88	423.26
4600	32	4285	334.31	399.34	29	4497	323.52	385.69
4800	34	5155	391.46	493.01	38	5188	437.88	543.94
5000	29	5217	360.95	477.76	39	5653	481.43	620.92

You can see that the times decreased tremendously. The time of smac < 3000 in the case maxnopt = 0.7 * smac would be very interesting (I did not have more time to do test runs, unfortunately).

I also changed the number of restarts. Before, a restart was done every MAXCIRC = 4 times, if there was no change in the objective function value. Now I tried MAXCIRC = 10:

Results with ONENORM + SAVEPRICTIME, MAXCIRC = 10
Max. iter.-limit = smac/2; maxnopt = 0.8 * smac

smac	LP's	iteration	optim./sec.	ut/sec.
3400	35	5322	305.16	413.74
3600	33	5363	309.41	458.09
3800	30	5054	285.64	372.13
4000	29	5293	299.36	391.31
4200	37	5237	380.51	543.76

This seemed to lead to better results, but I did not have the time to continue research in this direction. However, it looks promising.

3.1.6 aa200,000p

# rows:	837
# columns:	200,000
# nonzeros:	1,535,412

Although I started the problem with an advanced basis (the solution of the aa100,000p problem), CPLEX did not reach a solution after 8 hours. I stopped it after 50,000 iterations without a solution on the CRAY-2.

Before I had implemented ONENORM, I started testing on the CRAY-Y-MP with smac = 6000. The behaviour looked nice, but the connection to the CRAY-Y-MP was closed several times before the program stopped.

The network was a big hurdle for that problem, because it broke down several times a day. I had to start the same program over and over again.

After I had implemented ONENORM and SAVEPRICTIME, I ran this problem with smac = 3000, 4000, 6000 and 7000 with different parameters. In all cases, the network broke down or it ran several hours without coming to an end.

So I tried it with 10,000 columns. There I got a solution: it took 56,114 iterations (7261 in Phase I) to solve 61 subproblems with a total usertime of 1834.38 seconds. The time for optimizing the subproblems was 1657.94. You can see the printout in Table 4.

I got another solution with smac = 8000. Therefore it took 68,345 iterations to solve 70 LP's in a usertime of 2137 seconds (out of that time, it took 1710.45 seconds to optimize the subproblems). With smac = 7000, it ran without ending.

These results are not very good because to solve subproblems with 10,000 variables takes a lot of time. The number smac = $10,000$ is too big for our purpose.

Our problem could be the huge number of nonoptimal variables. If that number was relatively small, then the method behaved very well. Unfortunately, this number is very high in our case. It starts with about 40,000 nonoptimal variables. If you look at this in relation to 5000 columns it is a factor of 8 (in the aa100,000 problem, it starts with $\approx 12,000$ nonoptimal variables, divided by 3000 gives a factor of 4). This suggests a change of the method such as: start with a bigger size of the subproblem. If the number of nonoptimal variables goes down, then reduce the size of the subproblems you are solving.

3.2 Comprehensive Results

The best results in the aa20,000p problem were with choosing the size of the subproblems around 900–3000 columns. Then I was able to solve it in about 12–19 seconds. The aa50,000p problem lead to the best results with smac $\approx$ 1500–5000, with a usertime of 70–110 seconds. The results on the aa100,000p problem were with smac $\approx$ 3000–4600 and needed a time of 336–400 seconds.

You may think that smac = 6000 columns would lead to good results in the aa200,000p test problem, but that's not right. The results with smac $\geq$ 8000 columns weren't very good. The time went up to 1800 seconds, about 3 CPU-hours. Maybe that's just because it is a hard problem to solve (think about aa6: it's not very big, but it's very hard to solve). But, maybe the method is not efficient enough if there is a very big number of nonoptimal variables at the beginning (see 3.1.6). If that number is relatively small, the method behaves very nicely.

Compared with CPLEX or other LP-solvers, this method seems to be more efficient.

Take a look at the aa100,000p problem: CPLEX needed 3956 iterations and a time of 531 seconds to solve it (with advanced basis!). The best solution I found was with choosing 3000 columns. I got 4375 iterations (with the smaller subproblem) and a time of just 336 seconds. That's about 37% better. If you take the average solution time between smac = 3000–4600 columns you will get the value of 395 seconds. That's still 26% better.

In the problem aa50,000p, the results lead to an average improvement of 15% (smac between 1500–5000, 84.8 seconds). CPLEX needed a time of 99 seconds to solve it (with advanced basis).

The best improvement I got was in the aa20,000p problem. If you run the algorithm with smac = 1000, you will get a time of just 12.15 seconds, which is 62.5% better than the time CPLEX needed (= 32.19 seconds). In the average case, the improvement is still 50%.

I think that these results are encouraging to continue the research in this method.

4 Proposals to Improve the Algorithm

Looking backward to our original goal, to solve much bigger LP's than the aa200,000p problem, I would suggest to vary the size of the subproblem we generate.

Start with a bigger subproblem, and when the number of nonoptimal variables (=

positive reduced costs) decreases, then decrease the size of the subproblems as well. I would suggest solving these big subproblems in the same way we solved the big LP's: to split them into-smaller subproblems.

Another possibility would be to solve a few subproblems in parallel, to do the pricing for all of them in the big LP, and to continue with the solution which leads to the minimum number of nonoptimal variables.

Most of the test problems were computed on a CRAY. Therefore, it's necessary to use the network. We had two problems: the CRAY is only available at certain times. If your program hasn't finished after that, all the computation you did on it was lost. The other problem was the network. On some days it went down several times. I had to start the same test problems over and over again. This suggests doing something like CPLEX does, to write out the basis at certain intervals during optimization. In this method, though, that's much more complicated than in CPLEX. The big problem is to start with an advanced basis, because you've got a lot more information to use. You have to write/read almost every array. It would be worth it to think about this, however.

Maybe the "circling" was a bad idea. One can try to run it without that. But, I think that it would probably lead to an improvement if "circling" was just changed: Instead of saving the "old" nonoptimal variables every second time, do it every time.

You could also run the program with a smaller number of restarts. The good results of 3.1.5 encourage that. But unfortunately, I didn't have the time to do it.

Last, but not least, it would be a big improvement to parallelize and vectorize the algorithm. I hope that efforts to try that will be rewarded.

Acknowledgement.

I first wish to thank Professor Robert E. Bixby for suggesting the problem and helping with the research. I also wish to thank Ronni Rae Indovina for reading and correcting the manuscript.