

retroLP, An Implementation of the Standard Simplex Method

Gavriel Yarmish

Richard Van Slyke



**Department of Computer and Information
Science**

**Technical Report
TR-CIS-2001-05
08/13/2001**

retroLP, AN IMPLEMENTATION OF THE STANDARD SIMPLEX METHOD

Gavriel Yarmish, Brooklyn College (gyarmish@photon.poly.edu), and

Richard Van Slyke, Polytechnic University (rvslyke@poly.edu)

1. INTRODUCTION:

Dantzig's simplex algorithm for linear programming has two major variants: the original, or *standard* method, and the *revised* method. Today, virtually all serious implementations are based on the revised method because it is much faster for sparse LPs, which are most common. However, the standard method has advantages as well. First, the standard method is effective for dense problems. While dense problems are uncommon in general, they occur frequently in some important applications such as wavelet decomposition, digital filter design, text categorization, image processing and relaxations of scheduling problems. [Chen et al, 1998; Ekstein et al, 1995] Second, the standard method can be easily and effectively extended to a coarse grained, distributed algorithm. (There are no scalable distributed versions of the revised simplex method.) Finally, as we shall show here, simple and accurate models of iteration times are available for standard simplex method implementations.

We describe a full featured implementation of the standard method, *retroLP*, which is available for research or educational use from the authors. *retroLP* is written in C/C++. It takes input in the MPS format [Murtagh, & Saunders, 1998], Our implementation supports virtually all the options for linear programming implied by the format. To preserve numerical stability, our implementation uses full pivoting reinversion. *retroLP* also uses the EXPAND degeneracy procedure of Gill, Murray, Saunders, and Wright [1989] to improve numerical stability and to avoid stalling and degeneracy. *retroLP* is most effective for dense linear programs with moderate aspect ratio.

In Section 2, we review the standard and revised simplex method, and introduce some terminology. We then sketch the implementation of our algorithm. In Section 3 we describe the configuration used in our experiments, and in Section 4, we present empirical performance measurements based on practical (Netlib), and synthetic problems; we also discuss performance models for *retroLP*. Finally, we present some conclusions and related work.

2. THE SIMPLEX METHOD

2.1 The Simplex Method Using Dictionaries

We consider linear programs in the general form:

$$\begin{aligned} \underset{x}{\text{Max}} \ z &= cx \\ b^l &\leq Ax \leq b^u \\ l_j &\leq x_j \leq u_j \text{ for } j = 1, \dots, n \end{aligned} \tag{1}$$

Or with $y = Ax$ we have:

$$\begin{aligned} \underset{x}{\text{Maximize}} \ z &= \sum_{j=1}^n c_j x_j \\ \text{Subject to } y_i &= \sum_{j=1}^n a_{ij} x_j \ (i = 1, 2, \dots, m) \\ l_j &\leq x_j \leq u_j \text{ for } j = 1, \dots, n; \quad b_i^l \leq y_i \leq b_i^u \text{ for } i = 1, \dots, m \end{aligned} \tag{2}$$

$A = \{a_{ij}\}$ is a given $m \times n$ matrix, x is an n -vector of decision variables x_j , each with given lower bound l_j and upper bound u_j . The m -vectors b^l and b^u are given data that define constraints. The

lower bound, l_j , may take on the value $-\infty$ and the upper bound, u_j , may take on the value $+\infty$. Similarly, some or all of the components of b^l may be $-\infty$ and some or all of b^u may be $+\infty$.

Equation (2) together with an assignment of values to the non-basic variables x is a variant of the *dictionary* representation of Strum and Chvátal [Chvátal, 1983]. The dictionary is said to be *feasible* for given values of the independent (non-basic) variables x_1, \dots, x_n if the given values satisfy their bounds and if the resulting values for the dependent (basic) variables y_1, \dots, y_m satisfy theirs. If a dictionary, feasible or not, has the property that each non-basic variable is either at its upper bound or its lower, and the basic variables satisfy the **equations** of (2), then the dictionary is said to be *basic*. Suppose our dictionary besides being feasible has the *following optimality properties*, (i) for every non-basic variable x_j that is strictly below its upper bound we have $c_j \leq 0$, and (ii) for every non-basic x_j that is strictly above its lower bound we have $c_j \geq 0$. Such a dictionary is said to be *optimal*. It is easy to see that no change in the non-basic variables will increase z and hence the current solution is optimal.

Starting with a feasible dictionary, the standard simplex method involves a sequence of feasible dictionaries. Each iteration of the sequence consists of three steps:

1. Select Column:

Choose a non-basic variable, x_s , that violates one of the two optimality properties. Such a non-basic variable is said to be *eligible*. There may be many eligible columns. There are several criteria for choosing the non-basic variable from the eligible columns. We will discuss three shortly. If there is no such variable the current solution is optimal. In this latter case we stop with an optimal solution.

2. Select Row:

Increase the non-basic variable if the first optimality condition was violated (decrease the non-basic variable if the second optimality condition was violated) until the non-basic variable or one of the basic variables reaches its bound and is about to become infeasible. If there is no limit to the change you can make in the non-basic variable, x_s , the value of z is unbounded above and continuing to change x_s will result in ever increasing values of z . We then terminate with a class of feasible solutions with the objective unbounded above.

3. Pivot:

If the non-basic variable reaches its bound, then the next dictionary is determined by all the non-basics remaining the same except for x_s , which is set at the bound it reached. The basic variables are adjusted accordingly. This is called a *minor pivot*. If a basic variable y_r starts to exceed its bound before x_s reaches its bound then x_s and y_r exchange their roles; that is, y_r becomes non-basic and x_s becomes basic. This is accomplished by a *major pivot step*. The result is the next dictionary.

It can be demonstrated that with proper care in breaking ties in the select row step, this process terminates in a finite number of dictionary changes.

2.2 Phase I

We assumed that we start the simplex method with a feasible dictionary. To find this initial feasible dictionary, if necessary, we introduce an auxiliary problem called the *Phase I* problem, which is initially feasible. The Phase I problem is a slight generalization of a linear program (it actually has a piecewise linear objective) that also can be solved using the simplex algorithm. The optimal solution provides an initial feasible dictionary for the original problem if one exists. We then use the simplex method again to solve the resulting feasible dictionary; this is called *Phase II*. We may start Phase I with an arbitrary dictionary in the form (2) with an arbitrary assignment of values (that satisfy the bounds or not) of the non-basic variables. See [Bixby, 1992, Part II] for a more detailed view of a similar scheme.

2.3 The Revised Simplex Method

In the standard simplex method, most of the effort in moving from one dictionary, (2), to the next comes from calculating the new a_{ij} and c_j coefficients. In general, most of these coefficients change for each new dictionary. This is particularly onerous if the number of columns, n , is relatively large compared to the number of rows, m . Moreover, sparsity is lost. That is, if most of the data elements are zero in the original dictionary, they fill in very quickly with non-zero values in a few iterations of the standard method. Particularly frustrating is that only a small part of each dictionary is used or even looked at!

To perform an iteration of the simplex method, we require only the following from the dictionary (2) (assuming we are using the classical column choice rule):

- 1) The objective coefficients c_j $j=1, \dots, n$,
- 2) The constraint coefficients, a_{is} $i = 1, \dots, m$, for the pivot column, s , and
- 3) The current values of the basic variables, y_i .

Item 1) is used to determine the pivot column, and Items 2) and 3) are used to determine the pivot row. In summary, we only use two columns and one row from all the data in the dictionary.

By the early 1950's, George Dantzig and William Orchard-Hays [1954] realized that these three elements could be derived from one, fixed, original dictionary together with a changing, *auxiliary data structure* that requires less work to update than the work required to change dictionaries. For most linear programs found in practice, it is more efficient to represent the current dictionary implicitly in terms of the original system and the auxiliary data structure rather than explicitly updating the form (2). Such an approach is called a *revised simplex method* and is more efficient for linear programs that are sparse (low density) and have high *aspect ratio* (n/m).

To explain more about this, it is convenient to recast (2). We rename y_i as $-x_{n+i}$ for $i = 1, \dots, m$ and reconfigure (2) in matrix form as:

$$\begin{aligned} & \underset{x}{\text{Maximize}} \quad z = CX \\ & \text{Subject to : } AX = 0 \\ & L \leq X \leq U \end{aligned} \tag{3}$$

where $X = [x_1, x_2, \dots, x_n, x_{n+1}, \dots, x_{n+m}]$, $C = [c_1, c_2, \dots, c_n, 0, \dots, 0]$, $N = \{a_{ij}\}$, $A = [N \mid I]$, $L = [l_1, \dots, l_n, -b_1^l, \dots, -b_m^l]$, and $U = [u_1, \dots, u_n, -u_1^l, \dots, -b_m^l]$.

In this notation, each different dictionary (2) corresponds to a different basis, B , of A . By adding appropriate multiples of the constraints $AX=0$, to the objective we also maintain zero coefficients for the basic variables in the successive C vectors. These operations can be expressed directly in matrix terms. For example, if we are going to pivot in column s (making the non-basic variable x_s basic) and replace the basic variable corresponding to row r , we premultiply A by P where:

$$P = \begin{pmatrix} 1 & 0 & \dots & \frac{-a_{1s}}{a_{rs}} & \dots & 0 \\ 0 & 1 & \dots & \frac{-a_{2s}}{a_{rs}} & \dots & 0 \\ & & \ddots & \vdots & & \\ & & & \frac{1}{a_{rs}} & & \\ & & & \vdots & \ddots & \\ 0 & 0 & \dots & \frac{-a_{ms}}{a_{rs}} & \dots & 1 \end{pmatrix}$$

Modulo some renumbering of equations and variables, the matrix A is updated iteration by iteration by premultiplying by P matrices. So after some number of iterations, k , the new constraint matrix A' can be given in terms of the original one, A , by $A' = [P^k P^{k-1} \dots P^1]A$. Again within numbering of rows and columns $B^{-1} = P^k P^{k-1} \dots P^1$ is the inverse of the current basis, B .

B^{-1} suffices to obtain from the original matrix A all we need at an arbitrary iteration, k . So this is our first example of an auxiliary structure. This is called the revised simplex method using the *explicit inverse*.

Clearly, we can represent the basis inverse as a product of the individual P matrices (actually you need only save the column with non-trivial entries and its index) separately as another auxiliary structure. This is called the *product form of the inverse*.

More common today is the *LU decomposition* of B (see, for example, [Nash and Sofer, 1996, Sections 7.6.1 and A.5]); The LU decomposition offers better numerical stability. Heuristics are used for (i) accomplishing the initial LU decomposition for (ii) the updating of the decomposition, and (iii) determining the frequency of updating. They seek an optimal trade off between numerical stability and the maintenance of sparsity corresponding to that of the original matrix *B*. [Bartels and Golub, 1969; Reid, 1982]. In this context Step 3, "pivot," corresponds to the updating of the LU decomposition, and its periodic (usually at most every 100 iterations) reinitialization or *refactorization*.

Updating any of the representations is, at most, of order m^2 average work. On the other hand, pivoting in the standard method on the explicit representation of the dictionary takes order mn work. Thus for high aspect ratios, the standard method takes more work. Table 1 summarizes the qualitative differences between the standard and revised simplex method.

| Revised Simplex Method | Standard Simplex Method |
|--|--|
| Takes better advantage of sparsity in problems | Is more effective for dense problems |
| Is more efficient for problems with large aspect ratio (n/m) | Is more efficient for problems with low aspect ratio. |
| Can effectively use partial pricing | Can easily use steepest edge, or greatest change pricing in addition to the classic choice rule. |
| Is difficult to perform efficiently in parallel, especially, in loosely coupled systems. | Very easy to convert to a distributed version with a loosely coupled system. |
| Frequently, the representation of the basis inverse representation is recomputed both for numerical stability and for efficiency (e.g., maintaining sparsity). The work is modest. | Rarely, the dictionary has to be recomputed directly from the original data to maintain numerical stability (but not for efficiency). The work is substantial. |

Table 1: Comparison of Revised and Standard Forms of the Simplex Method

With ideal computation, the revised and standard simplex methods perform the same sequence of column, and row choices and take the same number of iterations. This allows us to compare performance of the two approaches by comparing the average time per iteration rather than the total running time. This is very convenient because performance models of the time per iteration are much easier to come by than for total time. In other cases, for example, in comparing performance for different column choice rules, total time must be compared since the number of iterations may be quite different.

2.4 Alternative Column Choice Rules

Any eligible non-basic variable may be chosen in the column choice step of the simplex method. We discuss three approaches to picking the particular non-basic variable.

2.4.1 The classical column choice rule

The original rule used by Dantzig was to choose the eligible c'_j in the current dictionary with the largest absolute value. This selects the non-basic variable that gives the largest improvement in the objective function per unit change of the non-basic variable. This criterion is very simple and straightforward to compute. In contrast to some of other methods, the column is chosen without looking at any of the coefficients, a'_{ij} . However, it has the undesirable feature that by rescaling the variables you can cause any eligible column to be chosen.

2.4.2 The steepest edge column choice rule

The dependence of the column choice on scaling of the classical method can be removed by normalizing the value of c'_j by the length of the column in the current dictionary corresponding to

the non-basic variable j . Applying the steepest edge rule requires more work per iteration for both standard and revised methods. In both cases, for each eligible column one has to compute the norm of the column **in terms of the current basis**. In addition, in revised methods, one does not have readily at hand the current representation a'_{ij} . This would seem to rule out the steepest edge rule for revised implementations; however, clever recursive computations can be used to implement the rule with modest cost. [Forest and Goldfarb, 1992] The *Devex* rule of Harris [1973] is another scheme for the revised method that approximates the steepest edge criterion efficiently. In any case, the standard method has the needed coefficients readily available.

2.4.3 The greatest change rule

For each eligible column, perform the **row** choice step of the simplex method and then compute the improvement in the objective that would result if the column were chosen, and use this as the column choice rule. This is called the *greatest change* criterion. It takes even more work than the steepest edge rule. The payoff seems no better than for the steepest edge rule, so it is rarely used (see Section 4.3). Nevertheless, this method can be implemented easily in standard implementations; it is rarely used in revised implementations.

2.5 Reinversions and Refactorizations

Revised methods frequently reinitialize their auxiliary data structure (typically every 30 to 70) iterations. There are three reasons for reinitializations: a) numerical stability b) to support some degeneracy procedures [Gill et al, 1987] and c) refactoring the data structures used in the revised method [Chvátal, 1983]. Standard simplex methods need only reinitialize for the first two reasons. Reinitializations for the first two reasons are required relatively infrequently whereas refactorizations are quite frequent. For our standard method, reinversions are carried out on the order of thousands of iterations. On the other hand, the reinitialization in the standard method is very expensive, especially for problems with high aspect ratio.

3. EXPERIMENTAL CONFIGURATION

We performed experiments on problems from the Netlib library, and synthetic problems. Both pose difficulties. The Netlib problems are not at all typical. Many of them have been submitted because of "nasty" features that make them thorough tests of linear programming codes. (See, for example, [Ben-Tal & Nemirovski, 2000] for a discussion of this.) Moreover, the problems are very sparse. Finally, we wished to determine the performance of retroLP as a function of problem parameters, particularly density. To be able to control for the problem parameters, synthetic problems are convenient, but they may have covert features that make them much easier (or much harder) than "typical" problems. Fortunately, this is usually revealed in the number of iterations, rather than the work per iteration. Since we mostly analyze time per iteration rather than total time, these considerations should not affect our results. We also used multiple generators to try to minimize this potential problem.

3.1 Test Sets

Netlib contains problems for testing linear programming codes [www.netlib.org/lp/data, 1996]. While our program successfully ran all the Netlib problems, we used as our test set the 30 densest problems. These include all problems with density above 2.5%

We used three synthetic program generators. Each takes as input, m = number of rows, n = number of columns, d = the density of the non-zero coefficients ($0 < d \leq 1$), and *seed* = the seed for the random number generator. All the constraints are of the less than or equal type. Whether a coefficient, a_{ij} , of the constraint matrix is non-zero (or zero) is determined randomly with probability d . For one of the generators, the value of a non-zero coefficient is chosen at random, uniformly between -1 and 1. The objective coefficients are generated randomly between -1 and 1 (the sparsity condition does not apply to the objective). The variables are constrained to be between $-m$ and m . The constraints are constrained to range between -1 and 1. Since, setting all variables to 0 is feasible, no Phase 1 is required. The other two generators are similar.

3.2 MINOS

We use MINOS 5.5 [Murtagh & Saunders, 1998] as a representative implementation of the revised simplex method. We installed it to run in the same environment as retroLP. This allowed us to make reasonable comparisons between the standard and revised simplex methods. The purpose of these comparisons is not so much to compare running times but to examine the relative behavior of these approaches as the parameters of interest, primarily density, are varied. In general we used the default parameters with MINOS with a few, significant exceptions designed to make MINOS more comparable with retroLP. Quite often, these settings make MINOS run at less than it's fastest. Most importantly we disabled partial pricing, scaling, and basis "crashing."

3.3 The Computers

retroLP runs on PCs and Unix workstations. Most experiments were run on a Dell 610MT Workstation with 384 MB RAM. It has a Pentium II processor running at 400MHz with a 16KB L1 instruction cache, a 16KB L1 data cache, and a 512KB integrated L2 cache. The code was compiled using Visual C++ 6.0 for the Windows NT operating system. Some experiments, including all the experiments for the distributed version, dpLP, were run on Sun Ultra 5 Workstations (270 MHz clock rate; 128MB RAM) running Solaris 5.7.

4. EXPERIMENTAL RESULTS

4.1 Eligible Columns

When using steepest edge, or greatest change column choice rules, the amount of work in "pricing out" a column differs dramatically depending on whether the column is eligible or not. To determine eligibility, basically two comparisons are needed; however, if the column is, in fact, eligible an additional order m computations are needed. So for accurate performance analysis it is useful to be able to estimate the fraction of columns that are eligible. retroLP collects this information. When using greatest change column choice rule on the 30 Netlib problems, the fraction of columns that are eligible ranges from 4.6% to 42.9%. A simple average of the percentages is 23.18% while a weighted average resulted in 42.9% (one problem ran for very many iterations). For steepest edge the range was 4.6% to 56.44%. The simple average was 26.15% and the weighted average 40.74%. So it is rare that one needs to even consider half the columns in detail.

4.2 Major and Minor Iterations

Since substantially more work is taken in a major pivot than a minor one, the amount of each of the two types might radically change performance estimates. We did **not** find this to be a problem, at least for the Netlib test set (the synthetic problems, because their variables are tightly constrained, have more minor iterations). If the problem has no variables with upper and lower bounds (on the initial non-basic variables) or ranges (bounds on initial basic variables) minor iterations cannot occur. Of the 30 problems in our Netlib test set, only 10 have bounds and/or ranges. Even in these cases there were very few minor iterations for any of the column choice rules. One problem, RECIPE, had only 80.36% iterations that were major, and another, BOEING1, had 89.06%. The remaining 8 problems were all above 95 %.

4.3 Iterations by Column Choice Rules

An important factor in performance is the column choice rule used. Generally, there is a tradeoff between the number of iterations using a rule and the computational effort it takes to apply the rule to the column. The number of iterations resulting from the use of a particular rule, depends only on the problem, while the computational effort to apply the rule depends on the specific implementation as well. Most dramatically the effort depends on whether a standard or revised method is used, but choices of programming languages, skill of coders, and the particular hardware used is important also. The ratio of the number of iterations using the greatest change rule to the number using the classical rule ranges from 0.378 to 5.465. The simple average of the

30 ratios is 1.140, and the average weighted by the number of iterations is 1.053. For the steepest edge, the range was 0.318 to 1.209. The simple and weighted averages were 0.800 and 0.620, respectively. The averages were computed considering only major iterations, but the results were essentially the same based on all iterations. Compared to steepest edge, rarely does the classical method result in fewer iterations, and then only slightly. See also [Forrest & Goldfarb, 1992]. The greatest change rule, on the other hand, seems to offer little benefit compared to the classical method so we did not consider it further.

4.4 Performance Models

For the standard simplex method with the classical column choice rule, the time spent in pivoting can be over 95%. Fortunately the pivot routine is rather simple. This makes performance analysis straightforward. Virtually, all the instructions in *pivot* are of the form:

$$a[i][j] -= a[i][j] * t; \quad (4)$$

In order for to get accurate measurements, it is necessary to consider the effect of memory caches, compiler optimizations, special instructions and the effect of zero coefficients, especially for sparse problems. [Goedecker & Hoisie, 2001] We wrote a simple timing routine to estimate the time in ns. for an operation of type (4). The routine updates an array that can be set to an arbitrary size; this allows us to evaluate the impact of processor caches. The routine also allows the user to vary the sparsity of the array. Figure 1 shows the result of the timing program applied to the PC Workstation described in Section 3.3. The *Unit Time for pivoting* for a run is calculated as the Pivot Time divided by both the number of pivots and $(m+2)n+1$, which is the number of multiply/divide operations in the pivot routine. The average Unit Time as a function of the array length has two flat portions joined by a linear transition region. The transition region starts roughly at array lengths of 50,000 doubles and ends at about 100,000 doubles. Each double takes 8 bytes so that the transition region in bytes is about 400,000 to 800,000 bytes. The timing routine enters random numbers into an array. The L2 cache for the workstation is 500,000 bytes. If the array size is significantly less than the cache size, the entire generated array remains in the cache. Then, when the timing run is made of the updates, (4) there are no cache misses. The new values also are written into the cache. The Unit Time for all this is about 28.3 ns. On the other hand if the array is larger than the cache size, the end of the array overwrites the beginning in the cache. When the array is larger than twice the cache size, an L2 cache miss is frequently incurred. The Unit Time for this operation is about 64.2 ns., over twice as long. In the transition region, only part of the array is overwritten. For the PC workstation that we used, these parameters were not affected by sparsity. The UNIX machines described in Section 3.3. exhibit a similar behavior except that the cache size is 250KB so that the transition region is earlier.

Observed retroLP times are consistent with the results of the timing tests. Figure 2 shows the pivot time per pivot as a function of the number of multiply/divides for the smaller problems of the Netlib test set when using the classical column choice rule. The upper straight line corresponds to the Unit Time for large problem (outside the cache), and the lower line corresponds to the Unit Time for problems that fit in the cache.

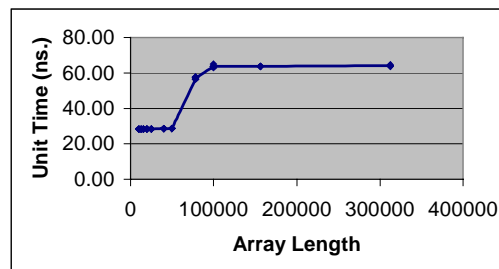


Figure 1: Unit Time vs. Array Length

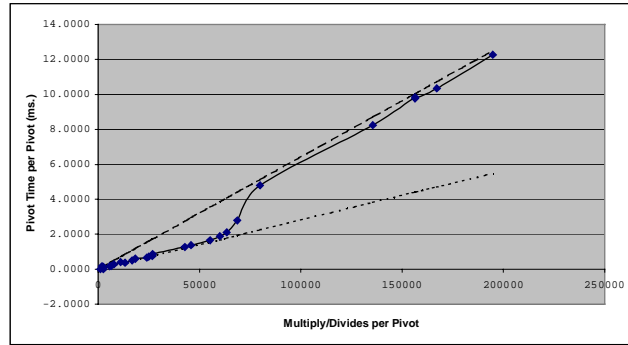


Figure 2: Pivot Time per Pivot vs. Number of Multiply/divides

These experiments were all based on the classical column choice rule. With the steepest edge column choice rule, the column choice time becomes significant. Typically about 75% of the time might be spent on pivoting and 25% on column choice. Usually the other parts of the program use little time. Because the dynamics of the column choice procedure is more complex than pivoting, the timing approach used in analyzing the classical column choice rule is difficult to apply. Instead, we estimated the Unit Times for pivots and steepest edge column choice directly from runs on the test problems. As before, there are pronounced cache effects so we made separate estimates for in-cache and out-of-cache regions. For the in-cache region, the estimate Unit Time for steepest edge column choice is 31.92 ns., and for pivoting 30.32 ns. In the out-of-cache range the column choice Unit Time estimate was 74.42 ns., and the pivoting estimate 62.58 ns.

We then end up with a performance model for retroLP of the following form:

$$T = p_m[(m+2)n+1]UT_p + c_e(m+1)UT_{se} \quad (5)$$

where p_m is the number of major pivots, c_e is the number of eligible columns evaluated using the steepest edge rule, UT_p is the unit time for major pivots, and UT_{se} is the unit time for the steepest edge evaluation for eligible columns. The Unit Times depend on whether the problem fits in cache or not. (5) is not defined in the transition region, although interpolation would not be difficult. Most accurately, T accounts for the column choice plus the major pivot time; however, the other contributions are generally quite small and T offers a good approximation to total time. When using the classical column choice rule, the last term of (5) is not used. Figure 3 shows how well the actual time spent in pivoting and column choice for the Netlib problems (excluding the three largest) using steepest edge column choice compared with the predicted time.

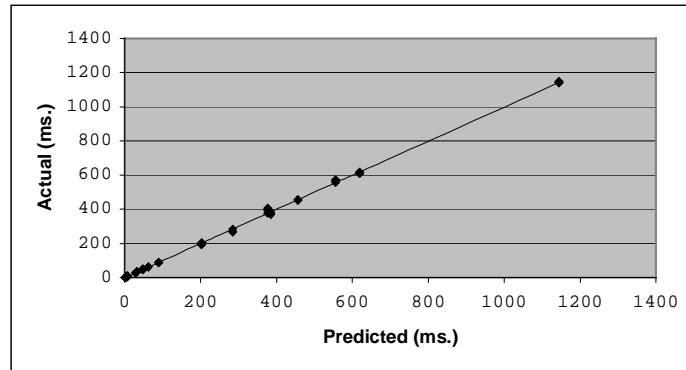


Figure 3: Pivot Plus Column Choice Time -- Predicted and Actual

4.5 Comparison of Revised and Standard Simplex Methods

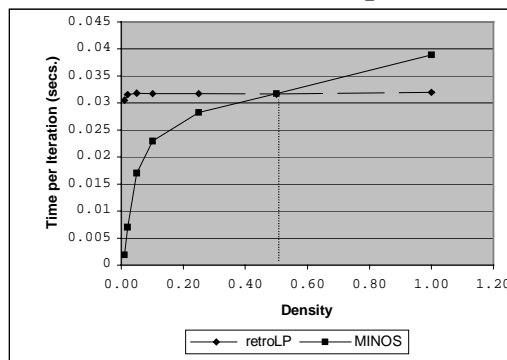


Figure 4: Comparison of retroLP and MINOS Iteration Time vs. Density

We first compare retroLP and MINOS when both use the classical column choice rule. Next we compare retroLP using steepest edge with MINOS using the classical rule (MINOS does not support steepest edge). In this latter case, for the first time, we must base our comparisons on total running time. These tests were on synthetic linear programs with $m=500$, and $n=1,000$. For each data point three different problem generators with three different seeds for a total of nine combinations were run.

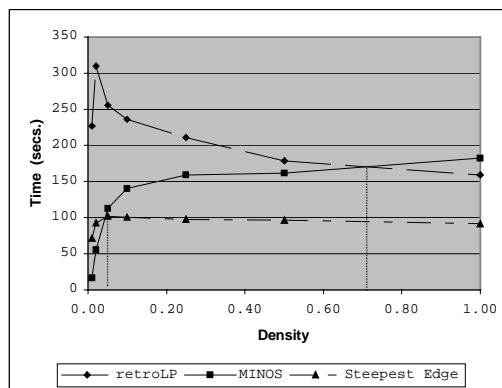


Figure 5: Comparison of Total Running Time

In Figure 4 we see that the time per iteration of retroLP is essentially independent of density, while the iteration time of MINOS goes up with density. The crossover point is about 0.5. The main objective of these studies is to show the **dependence** of the two algorithms on density.

Figure 5 is a comparison of total running time for retroLP using both classical column choice, and steepest edge, and MINOS using classical column choice. The breakeven for retroLP and MINOS both using classical column choice is at about 0.72 density. The breakeven for MINOS using classical column choice and retroLP using steepest edge is about 0.05 density.

5. SUMMARY AND CONCLUSIONS

In this paper we introduced a new implementation of an old algorithm, the standard simplex method. We provided performance models and experiments that can be used to estimate running time, and to compare retroLP with revised algorithms. While our few experiments comparing MINOS and retroLP tell us little about their relative performance, they do indicate that for moderate values of density the standard method can be competitive.

An implementation of the standard simplex makes possible a natural approach for a distributed simplex method. Partition the columns among a number of workstations. Each iteration, each workstation prices out its columns, and makes a "bid" to all the workstations. The winning bid defines a pivot column, then all the workstations pivot on their columns in parallel, and so on. [Yarmish, 2001] describes such a coarse grained distributed simplex method, dpLP. Models, based, in part, on the retroLP models given here, provide estimates of scalability, which turns out to be sizable. This is of interest because no scalable, coarse grained simplex methods are available.

REFERENCES

- Bartels, R. H. and Golub, G. H., "The simplex method of linear programming using the LU decomposition," *Communications of the ACM*, 12, pp. 266-268, 1969.
- Ben-Tal, A., and A. Nemirovski, "Robust solutions of Linear Programming problems contaminated with uncertain data, *Mathematical Programming, Series A*, pp. 411-424, September, 2000.
- Bixby, Robert E., "Implementing the Simplex Method: The Initial Basis," *ORSA J. on Computing*, Vol. 4, No. 3, pp. 267-284, Summer, 1992.
- Chen S.S., D.L. Donoho, and M.A. Saunders, Atomic Decomposition by Basis Pursuit," *SIAM J. on Scientific Computing*, 20, 1, pp. 33-61, 1998.
- Chvátal, Vasek, *Linear Programming*, Freeman, 1983.
- Dantzig, George, and William Orchard-Hays, "The product form for the inverse in the simplex method," *Mathematical Tables and Other Aids to Computation*, 8, pp. 64-67, 1954.
- Eckstein, J., I. Bodurglu, L. Polymenakos, and D. Goldfarb, "Data-Parallel Implementations of Dense Simplex Methods on the Connection Machine CM-2," *ORSA Journal on Computing*, v. 7, n. 4, pp. 402-416, Fall 1995.
- Forrest, John and Donald Goldfarb, "Steepest-edge simplex algorithms for linear programming," *Mathematical Programming*, vol. 57, pp. 137-150, 1992.
- Gill, P., W. Murray, M. Saunders, and M. Wright, "A Practical Anti-Cycling Procedure for Linearly Constrained Optimization," *Mathematical Programming*, 45, pp. 437-474, 1989.
- Goedecker, Stefan, and Adolfo Hoiste, *Performance Optimization of Numerically Intensive Codes*, SIAM, 2001.
- Harris, P. M. J., "Pivot selection methods of the Devex LP code," *Mathematical Programming*, 5, pp. 1-28, 1973.
- Murtagh, Bruce A., and Michael Saunders, "MINOS 5.5 Users Guide," Technical Report SOL 83-20R, Revised July, 1998.
- Nash, S. G., and A. Sofer, *Linear and Nonlinear Programming*, McGraw Hill, 1996.
- Reid, J. K., A sparsity-exploiting variant of the Bartels-Golub decomposition for linear programming bases," *Mathematical Programming*, 24, pp. 55-69, 1982.
- Yarmish, Gavriel, *A Distributed Implementation of the Simplex Method*, Ph.D. dissertation, Polytechnic University, Brooklyn, NY, March, 2001.