A fast LU update for linear programming

Leena M. Suhl
Institut für Angewandte Informatik, Technische Universität Berlin, Franklinstrasse 28–29, D-1000 Berlin 10, Germany

and

Uwe H. Suhl
Institut für Wirtschaftsinformatik, Freie Universität Berlin, Garystrasse 21, D-1000 Berlin 33, Germany

Abstract

This paper discusses sparse matrix kernels of simplex-based linear programming software. State-of-the-art implementations of the simplex method maintain an LU factorization of the basis matrix which is updated at each iteration. The LU factorization is used to solve two sparse sets of linear equations at each iteration. We present new implementation techniques for a modified Forrest–Tomlin LU update which reduce the time complexity of the update and the solution of the associated sparse linear systems. We present numerical results on Netlib and other real-life LP models.

1. Introduction

The simplex method is currently used in most commercial linear programming (LP) codes. Although LP software and computers have become much faster, LP models increased in size. Furthermore, LP optimizers are used in interactive applications and in integer programming, where many LPs have to be solved. More efficient algorithms and improved implementation techniques are, therefore, still of great interest. Interior point methods have become very competitive to or even better than the best simplex codes. Nevertheless, it seems very unlikely that the simplex algorithm will be replaced for at least two reasons: efficient restarts from advanced starting bases and basic solutions. These aspects are very important for solving mixed-integer optimization problems (MIP). Basic solutions are required to take advantage of strong linear programming relaxations in MIP.

The speed of the simplex method to solve a given LP problem is determined by:

- The number of LP iterations. Most practical LP models are solved in $k \cdot m$ iterations, where m is the number of constraints of the LP model. The constant

© J.C. Baltzer AG, Science Publishers

k is determined by the pivot column and row selection rules, and the nature of the LP model.

- The time to factorize and update the LU factorization and to solve two sparse linear systems of equations at each pivot step.

This paper discusses only design and implementation of the sparse matrix kernels, i.e. aspects of algorithms, data structures and computer programs to perform the iterations of the simplex method as fast as possible. Pivot selection rules which are, for difficult problems, also very important are an independent topic not considered here.

The software discussed here is part of MOPS (Mathematical Optimization System). MOPS contains a simplex-based high-speed LP optimizer developed at the Freie Universität Berlin [9]. The system is fully written in FORTRAN and has been ported to a variety of platforms including PCs, workstations and mainframes. MOPS is used in industrial applications and as a research prototype to test new ideas in linear and mixed-integer programming.

2. The primal simplex algorithm

We briefly outline the primal simplex algorithm to clarify our notation. It is assumed that the reader is familiar with the simplex method. Using matrix notation, we may state the LP problem in the computational canonical form:

$$\text{(LP)} \quad \text{minimize} \quad \begin{aligned} & c'x! \\ & Ax = b, \\ & l \le x \le u. \end{aligned}$$

l, u, c, x are $m + n$ vectors, b is an m vector, A is a sparse $m \times (m+n)$ matrix with rank m. The vectors l and u may contain elements which are plus or minus infinity. Let B represent any starting basis; B is a nonsingular sparse $m \times m$ matrix, h the m vector of indices of basic variables, i.e. $h(i)$ is the index of a column vector of A which is at position i in the basis. To B corresponds an m vector of basic variables x_B and an n vector of nonbasic variables x_N. A *basic solution* is determined by fixing the nonbasic variables to one of its bounds (if they exist) and then solving the linear system to determine x_B. If a nonbasic variable is free, i.e. $l_j = -\infty$ and $u_j = +\infty$, it is set to zero. Let π, y and z be real m vectors.

Ignoring many details, the revised simplex method using an LU factorization of the basis can be stated as follows:

Step 0. (*Factor*) Compute $LU = PBQ'$, where L, respectively U, are lower, respectively upper, triangular matrices and P, respectively Q, are permutation matrices. The LU factorization of B is used to solve the linear systems in steps 1 and 4.

Step 1. (*BTRAN*) Compute the dual cost vector π: solve $B'\pi = f$, where f is a suitably chosen pricing form.

Step 2. (*Test of optimality*) If the solution is *dual feasible, terminate* (the problem may be primal infeasible). Dual feasibility can be tested by computing $d_j = \pi a_j$ (phase 1) or $d_j = c_j - \pi a_j$ in phase 2: If for all nonbasic variables j, $d_j \geq 0$ if variable j is at lower bound or $d_j \leq 0$ if variable j is at upper bound, stop.

Step 3. (*Price*) If the test in step 2 failed, select a nonbasic variable q not dual feasible to enter the basis according to a pivot column selection rule.

Step 4. (*FTRAN*) Update the entering column: Solve $By = a_q$ if $d_q < 0$ or $By = -a_q$ if $d_q > 0$ by using the LU factorization and retain the *spike* $L^{-1}a_q$.

Step 5. (*CHUZR*) Determine a blocking variable to leave the basis in position p and a step length Θ according to a pivot row selection rule or switch x_q to the other bound. If no such variables exist, the LP is unbounded; stop.

Step 6. (*UPDATE*) If the basis changed, replace column p of U by the spike $L^{-1}a_q$. Let r be the position of the last nonzero in the *permuted spike*. Move columns $p + 1$ to r one position to the left and column p to position r. This creates the matrix $H = [U_1, L^{-1}a_q, U_2]$. Restore upper triangularity of H. If there are numerical problems or too many elements in the new L or U, go to 0, otherwise update h, x_B and x_N to reflect the basis change and go to step 1.

The sparse matrix kernels in this algorithm are: initial and periodic LU factorizations, LU update, FTRAN and BTRAN. The time spent in these kernels is typically between 60–90% of the overall execution time (see also table 2).

Details of the LU factorization in MOPS are described elsewhere [11]. Since the LU factorization is closely related to our LU update, we briefly summarize our approach:

- Each pivot chosen in the active submatrix satisfies a threshold pivoting criterion [4]. As a consequence, we compute $LU = PBQ' + E$, where E is a perturbation matrix controlled by the threshold pivoting tolerance.
- The pivot selection is based on a modified Markowitz criterion determining the permutation matrices P and Q.
- Emphasis is placed on computing a sparse L^{-1} so that the spikes $L^{-1}a_q$ are sparse. This is important to reduce the work in LU update, FTRAN and BTRAN.

3. Overview of basis updating methods

Historically, the representation of the basis inverse in state-of-the-art simplex software has evolved from the explicitly stored form over the product form of inverse (PFI) to the LU factorization (LUF) computed by Gaussian elimination. As the names imply, the PFI is a representation of the inverse, whereas the LUF is a factorization of the matrix B. It was proven by Brayton et al. [3] that the LUF is in general sparser than the PFI. In LP software, the PFI was initially used both in reinversion and update. Markowitz introduced the LUF in reinversion, but used the PFI during simplex iterations. Bartels and Golub [1] found a technique of updating the LUF after each step of the simplex method. Forrest and Tomlin [5] presented an efficient implementation and showed experimentally for large-scale LP problems a smaller growth rate in the number of nonzeros than the PFI update.

Following the idea of the IBM simplex code MPSX/370 described in [2], we initially included in MOPS a PFI update for extremely sparse matrices, and an LU update for other cases. After optimizing the LU update, it became the fastest method for all problems and the PFI update became superfluous.

3.1. NOTATION

Factorization methods involve the transformation of the basis into a product of two triangular matrices: $PBQ' = LU$. L and U should be stored as a sequence of elementary row and column transformations to exploit their sparsity.

In the simplex method, basis changes are performed in such a way that one column is leaving and one is entering the basis at pivot position p. If a_q is the entering column, B the original basis and B the new basis, then we have

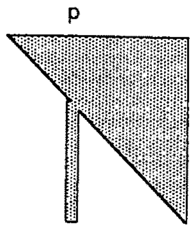
$$B = B + (a_q - Be_p)e_p', \tag{3.1}$$

where p is the pivot row index and e_p the pth unit column.

If (3.1) is multiplied from the left with L^{-1}, we obtain

$$L^{-1}B = U + (L^{-1}a_q - Ue_p)e_p', \tag{3.2}$$

where the permutations are left out for simplicitly; it is sufficient to keep track of the row and column order. In other words, column p in U is replaced by the transformed incoming column $g = L^{-1}a_q$ which we call the spike:

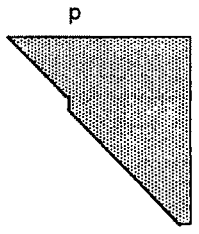
$L^{-1}B =$ 

(3.3)

3.2. BARTELS-GOLUB LU UPDATE

Bartels and Golub proposed the first LU update [1]. Permuting the columns so that g is last, and moving the other columns up one place, we obtain

$L^{-1}B =$ (3.4)



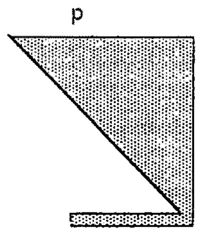
i.e. we have an upper Hessenberg matrix H. The permutation matrix is again left out for simplicity.

Now the Bartels-Golub method consists of a series of elementary transformations which reduce the subdiagonal elements of H to zero. For each column of H, we pivot either on the diagonal or on the subdiagonal element, depending on which of them has a larger absolute value. This is to improve the stability of the representation. If we pivot on a subdiagonal element, *a row interchange* occurs, and the permutation vectors have to be updated.

A main advantage of the Bartels-Golub update is that a stability bound can be computed a priori on each step. However, because row interchanges are possible on each elimination step, the Bartels-Golub update cannot be implemented as efficiently as other methods.

3.3. FORREST-TOMLIN LU UPDATE

The Forrest-Tomlin update [5] differs from the Bartels-Golub method in the way the spiked matrix (3.2) is handled. The same permutation is used, but it is applied to the rows as well as the columns to produce the matrix



(3.5)

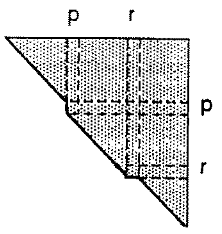
Next, multiples of rows are added to the last column to eliminate the row spike. Mathematically, the algorithm is equivalent to reducing the upper Hessenberg matrix H to a triangular form by eliminations between adjacent rows which *always involve an interchange*. This means that there is no fill in U, and all eliminations are stored in L. This was an essential advantage in 1972 when the algorithm was published, since U could be kept out-of-core.

In contrast to the Bartels–Golub update, the pivot selection ignores stability aspects. Nevertheless, it works well in practice. A stability check can be performed a posteriori at each iteration [10]. If it is not satisfied, the basis is refactorized.

The Forrest–Tomlin update can be implemented without permutation vectors, since the incoming column is always packed to the end and a row interchange is made for each row. Therefore, FTRAN and BTRAN can be performed in one work array.

3.4. REID UPDATE

Reid proposed an ingenious variant of the Bartels–Golub update, which favors a sparse L representation [8]. The sparse spike is placed in U at the row position of its last nonzero. Suppose the spike is in column p and extends to row r. Then, using the Bartels–Golub update, we obtain an upper Hessenberg matrix of the form:



(3.6)

where the submatrix consisting of rows and columns from p to r forms a *bump*. The Reid update first performs permutations within the bump to improve sparsity so that fewer eliminations are needed. The update proceeds in three major steps:

(1) Search column singletons within the bump; move each of them to the upper left corner of the bump, thus reducing the bump size.

(2) Search row singletons within the bump; move each of them to the lower right corner of the bump, thus reducing the bump size.

(3) Eliminate the subdiagonal elements of the reduced bump. If the diagonal element is zero, it will suffice to interchange rows. If the diagonal element is nonzero, the pivot is chosen among the diagonal and subdiagonal element inside a column. The element on the row which contains fewer entries is chosen, if it satisfies the threshold pivoting criterion, i.e. its absolute value is greater than a factor u $(0 < u \leq 1)$ times that of the other considered element.

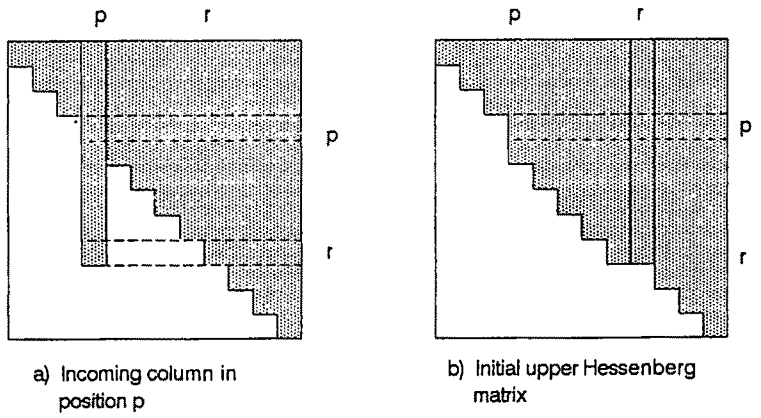
It should be noted that the final bump (if it exists) has no column singletons. It follows that if it is possible to permute H into an upper triangular form, this will be done in step 1. Furthermore, it is not possible for subsequent eliminations to create any column singletons. This means that every pivot will be in a column with minimal number of nonzeros and within this column in its row with least nonzeros – if the stability criterion is satisfied. The Reid update therefore uses a "minimal row count within minimal column count" pivoting strategy.

The Reid update pays special attention to sparsity and stability considerations. The main disadvantage is its complexity. In the worst case, it requires four passes through all bump rows, which may contain many thousand elements. Elaborate bookkeeping of the permutations is also required. Furthermore, eliminations may be slow, since the pivot row is chosen only for one elimination step at a time. This LU-update looked very promising. Therefore, we designed and implemented several versions of it. Compared to the update described below, we found only slight advantages at much higher update costs. Due to an additional permutation, FTRAN and BTRAN are also slower.

4. Algorithmic aspects of UPDATE, FTRAN and BTRAN

Our update is based on a modified Forrest–Tomlin LU updating method. This method allows a fast implementation, where numerical accuracy can be tested a posteriori on each iteration. The only reason to use the name Forrest–Tomlin for this update is that we always pivot on the diagonal. The organization and implementation of our method is fundamentally different from that presented in [5], since U is kept in memory and can, therefore, be continuously updated.

Like in other recent implementations, the original Forrest–Tomlin update has been modified so as to exploit the sparsity of the incoming vector $L^{-1}a_q$. Thus, the incoming vector is not permuted to the last position as in (3.5), but to the position corresponding to its last nonzero (the last position within the bump):


p
r
p
r
a) Incoming column in position p
p
r
p
r
b) Initial upper Hessenberg matrix

(4.1)

During the LU factorization, we compute a representation $PBQ' = LU$, where P is a row and Q' a column permutation. In the simplex method, we can freely choose the order in which the columns are placed in the basis matrix B. After completing the LU factorization, we permute the columns so that we always pivot on the diagonal. The pivot order is reflected by a permutation matrix R. During the LU updates, we perform symmetric row and column permutations in such a way that the pivots remain on the diagonal; it is sufficient to maintain only one permutation R. Note that the Reid updating method needs two permutation vectors throughout, since row and column permutations are performed independently.

We prefer using the original row indices in the L part, and having the permutations in the U part – thus the incoming spike $L^{-1}a_q$ can be computed and inserted without permutations. On each basis change, a further permutation matrix is applied to the "spiked" U to restore its triangularity. At iteration k, we have

$$L(k)^{-1}B(k) = R(k)U(k)R(k)', \tag{4.2}$$

where $B(k)$ is the basis matrix, $R(k)$ the permutation matrix, $L(k)$ and $U(k)$ lower and upper triangular matrices.

Assume that U is stored row- and columnwise in a sparse form, and L^{-1} columnwise in a sparse form. The algorithm proceeds in the following way:

ALGORITHM LU-UPDATE

Step 0. Suppose the spike $L^{-1}a_q$ is computed and pivot position p selected.

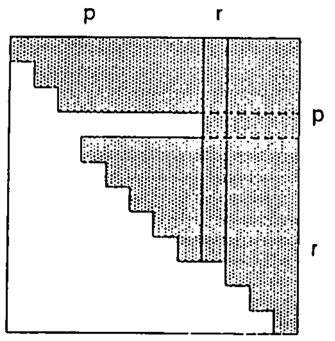
Step 1. Column permutation: Determine position r (see fig. 4.1); place the spike in position r; move columns $p+1, \ldots, r$ one position to the left (an upper Hessenberg matrix).

Step 2. Expand row p from its sparse representation into a full work array.

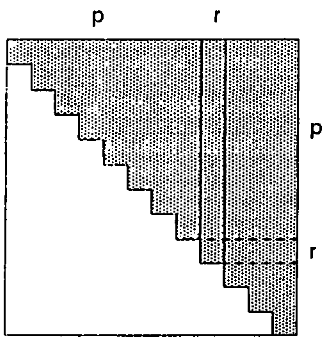
Step 3. Use rows $p+1, \ldots, r$ to eliminate the elements on row p in columns $p, \ldots, r-1$; store the eliminations in the L part.

Step 4. Row permutation: Place row p in position r; move rows $p+1, \ldots, r$ one position upwards.

Step 5. Store the modified row (initially row p, now row r) back to the sparse row- and columnwise representations.



c) Upper Hessenberg matrix after eliminating the bump elements on row p



d) The upper triangular matrix generated after row permutation

(4.3)

The basis factorization $L^{-1}B = RUR'$ is used at each iteration in the FTRAN and BTRAN operations. Mathematically, the following operations are needed:

FTRAN:
- compute $g = L^{-1}a_q$, (FTRANL)
- solve $RUR'y = g$ for y; (FTRANU)

BTRAN:
- solve $RU'R'z = f$ for z, (BTRANU)
- compute $\pi = L^{-1'}z$. (BTRANL)

Since FTRAN and BTRAN are the most time-consuming components in a simplex code, special attention has to be devoted to their implementation. One key factor is to exploit the sparsity of the incoming vector a_q and the pricing input vector f. We next discuss the impact of storage schemes in determining the expense of FTRAN and BTRAN.

In FTRANL, the spike g is computed by multiplying the incoming vector a_q with the L inverse. L^{-1} is represented as a consequence of elementary transformations, which are created columnwise in the LU factorization and row-wise in the LU update (as column- and row-wise eta vectors). The eta vectors have to be applied in the sequence they were generated. In FTRANL, each columnwise eta vector affects only one component in a_q, whereas a row-wise elimination vector usually changes the value of several components of a_q. Thus, a whole column can be skipped if the corresponding element of a_q is equal to zero – row-wise etas do not have this advantage.

In our representation, U is stored in sparse form; possible storage schemes are row- and columnwise storage with a permutation vector representing R. Since U is an upper triangular matrix, the resulting vector y is generated in FTRANU in a certain order: the elements are created backwards in the permuted order, starting with the last element and ending with the first one. Since one column of U affects only one element of y and g, whole columns can be skipped in the columnwise representation if the corresponding element of g is equal to zero. Again, this advantage is not given with the row-wise storage. This implies that a columnwise storage of U is favourable in FTRAN. To our knowledge, this was first observed by Fourer [7].

In BTRAN, a linear system in B' is solved in two stages. First, in BTRANU the matrix U' is used to give the intermediate vector z in permuted order. If U is stored row-wise, rows can be skipped if the corresponding element in f is equal to zero. If a columnwise storage is used, this is not possible. Thus, in BTRAN it is favourable to have a row-wise access to U. In the second stage, in BTRANL $L^{-1'}$ is used to compute the π vector. In BTRANL, we can exploit the sparsity of f by skipping row-wise etas – this can not be done with columnwise etas.

To optimize both the FTRAN and BTRAN operations, we maintain a *row- and columnwise representation* of the U matrix. During FTRAN, we exploit the sparsity of the incoming column a_q using the columnwise stored U. During BTRAN in phase 2, we solve $B'z = e_p$ and update $\pi = \pi - d_q z$, where the reduced costs d_q and the pivot index p are determined at the previous iteration. The advantage is that e_p has just one nonzero. The row-wise representation of U is used for BTRAN.

5. Implementation aspects of the new update

Several versions of the LU update were implemented, which improved the initial speed of the code significantly. The main goal was to minimize search in inner loops as well as unnecessary scanning of elements during the update by using sophisticated new data structures and algorithmic improvements.

The LU update in MOPS is implemented with three routines:

- XMODLU: removes the leaving column and adds the entering column in U; acts as a driver routine for the other tasks.
- XELIMN: expands row p: eliminates elements in positions $(p, p), \ldots, (p, r-1)$ of the upper Hessenberg matrix.
- XRBACK: stores the updated row back to the sparse representation.

Since the factored matrix is stored in a sparse form, access and update operations are much more complex than if it is stored dense. Our central problem was how to implement a fast scheme which simultaneously updates both the row- and columnwise representation of U. A fast update could be realized by introducing two pointer arrays: for each element in one representation (row- or columnwise), the pointer gives its position in the other representation. We are well aware that this solution needs more main memory than previously published schemes. However, this is justified, since main memory has become cheap and there is enough memory even on PCs to solve large-scale problems with MOPS. For the same reason, we store indices and pointers always in 32 bits.

Three arrays with double precision real numbers (64 bits) are reserved for the numerical values of L and U: one for U stored row-wise, one for U stored columnwise, and one for L. Row and column indices are stored in two integer arrays together with both representations. There are integer arrays for row and column counts and the starting positions of rows and columns of U.

Two pointer arrays are used to store for each row file element its index in the column file, and vice versa. It would be possible to store the numerical elements of U only once, say columnwise, and keep an index pattern of the row-wise U with the pointer arrays. Using the pointers, one could have a row-wise access to U. However, this is not optimal on machines with cache memory, since there is poor data locality which results in slow access to the rows of U.

By using the double pointer technique, we could eliminate much of the searching which initially slowed down the LU update. Virtually all the rest of searching in inner loop was eliminated by the following techniques:

- Maintaining a stack of row indices of nonzero entries of the spike vector $L^{-1}a_q$, to avoid scanning a work region of dimension m in XMODLU, when the new vector is embedded in U.
- Maintaining a stack of column indices of nonzero entries on the pivot row in XELIMN; this is used in XRBACK to avoid scanning a whole work region of dimension m.
- Using double pointers to find an element in the column file when its position in the row file is known, and vice versa.

The stack of row indices of the incoming vector is built in FTRAN. Initially, the row indices of the nonzeros in the incoming column a_q are placed in the stack. When a nonzero element of the transformed vector is computed in FTRAN, we check whether its position is already in the stack. If not, it is added to the stack. In XMODLU the stack is scanned, and the corresponding elements of the incoming vector are placed in U. A similar scheme is used in XELIMN and XRBACK.

6. Numerical results

The numerical results are based on some of the largest and most difficult LP problems from the NETLIB test set. In addition, we used some LP models from other real-life applications.

The computing environment was an IBM PS/2 PC with an i80486 processor (25 MHz) and 16 MB main memory. MOPS was compiled with the NDP FORTRAN compiler V3.0 and linked with the PharLap linker V4.0. The operating system was MS-DOS 5.0. In addition, in table 3 we present some numerical results with 25VF47 on some other machines. Among them is a Siemens Nixdorf mainframe H-120-F (BS2000), which is a fast scalar machine. The PS/2-486 is about 5% slower than some PCs with the same processor.

Table 1 shows the names of the (NETLIB) LP problems and their dimensions. In addition, numerical results with MOPS V1.4 are shown using the standard strategies (scaling, LP preprocessing, default tolerances, crash start). Under the heading "phase I" are the number of LP iterations to become feasible. Under the heading "total" are the total number of LP iterations (phase 1 and phase 2). An LP iteration is either a pivot or a bound switch. The CP times in seconds on the PC are reported in the last column in table 1. This includes optimizing the unscaled problem with the presumably optimal basis from the scaled optimization, and various safety checks once a problem has been solved. They do not include the time for converting the mps data.

Table 2 shows the CP time in seconds spent in the various computational kernels for solving the LP problems. The times for the LU update are in all cases lower than the time spent in selecting the pivot row! Also, the time spent in the LU factorization is typically very small except on problems where there is excessive

Table 1

LP test models and MOPS run times.
(All times are in seconds on a PS/2 (25 MHz) with MSDOS.)

Problem	cons.	vars.	nonzeros	phase I	total	Time (sec)
NETLIB LP models						
scsd8	397	2750	8584	333	973	21
25fv47	821	1571	10400	720	3033	124
degen3	1503	1818	24646	2010	4070	362
stocfor3	16675	15695	64875	3805	9772	2764
80bau3b	2262	9799	21002	1984	7879	379
greenbeb	2392	5405	30877	1217	4113	323
fit2d	25	10500	129018	1533	12584	1081
fit2p	3000	13525	50284	6531	14029	2073
truss	1000	8806	27836	1993	9528	511
d2q06c	2171	5167	32417	988	9506	1829
pilots	1441	3652	43167	3161	5709	1579
pilot87	2030	4883	73152	3502	7556	5102
Other real-life LP models						
P1	4481	10958	29840	0	1955	144
P2	3525	9625	69323	2864	15771	2366
P3	5563	6181	39597	1279	3229	423
P4	2356	11004	119360	13592	33733	7413

Table 2

CP time (sec) for computational kernels.
(All times are in seconds on a PS/2 80486 (25 MHz) with MS-DOS.)

Problem	FTRAN	BTRAN	Update	FACTOR	PRICE	CHUZR
scsd8	5	4	1	1	7	2
25fv47	43	31	9	7	17	16
degen3	89	149	22	24	42	35
80bau3b	76	72	19	11	167	32
greenbeb	86	101	22	21	51	41
fit2p	839	136	85	50	577	384
pilots	469	581	59	287	93	87
P1	37	33	12	5	42	14
P2	689	701	94	167	481	232
P3	133	122	31	26	41	68
P4	1725	2051	233	533	2511	356

Table 3

CP times (sec) to solve 25FV47 on various machines.

Computing environment	Time (sec)
Siemens H-120-F BS2000, FORTRAN-77	13
HP-730 HP-UX, f77	16
IBM 3090-J, VM/CMS, ESA VS-FORTRAN	21
i80486, 25 MHz MS-DOS, NDP FORTRAN	124

Table 4

Comparison of MOPS V1.4 to OSL R2 on some NETLIB models.
(All times are in seconds on a PS/2 80486 (25 MHz) with MS-DOS.)

Model	*m*	*n*	OSL		MOPS	
			iters	time	iters	time
25fv47	821	1571	2436	184	3033	124
truss	1000	8806	13028	1377	9528	511
degen3	1503	1818	3864	479	4070	362
80bau3b	2262	9799	6435	848	7879	379
greenbeb	2392	5405	3964	562	4113	323
fit2d	25	10500	9832	1171	12584	1081
fit2p	3000	13525	13459	4335	14029	2073
pilots	1441	3652	5104	1711	5709	1579
d2q06c	2171	5167	8558	1627	9506	1829
stocfor3	16675	15695	14889	8024	9772	2764
pilot87	2030	4883	7858	4885	7556	5102
Grand total				25203		16127

fill-in such as pilots and problem P4. The reason is that the other kernels have to be executed at each iteration, whereas on nearly all problems we perform 100 iterations on the average before factorizing.

Different LP solvers produce different pivot sequences and reinvert at different points. It is, therefore, difficult to have a direct comparison between updating methods. A comparison with another high-speed LP optimizer in the same computing environment and the same development tools is probably the best one can do.

Table 4 shows a comparison of MOPS to OSL release 2 (IBM's Optimization Subroutine Library) [6] on an MS-DOS PC with an i486 processor (25 MHz). Both OSL and MOPS were produced with the same Microway NDP-FORTRAN compiler and linked with the PharLap Linker V4.0. The time was measured in seconds (rounded). OSL was run with the recommended default pricing strategy (approximate Devex, idevexm = 1). OSL runs with steepest edge pricing (idevexm = 2) produced much lower iteration counts, but took more time in total. These results should be interpreted with great care, since minor changes in tolerances and other strategies (scaling, crash, preprocessing, pricing) can produce different results.

However, it seems to be evident (at least in this computing environment) that MOPS is very competitive in speed on those NETLIB problems. Since OSL frequently performs fewer iterations, this indicates that our sparse matrix kernels, in particular the LU update, are faster. Pilot87 seems at first glance a counter example. However, for that problem factorization frequency and threshold pivoting tolerance (100 and 0.1 in MOPS) have a large impact on the solution time since each LU factorization takes very long. Different settings produce vastly different running times for both codes.

7. Conclusions

Judging speed and numerical stability, it seems that the sparse matrix kernels in MOPS for maintaining a sparse LU factorization are very competitive with the best other known simplex codes on scalar machines. Compared to the LU update and FTRAN of Reid, we are much faster without any apparent disadvantage in numerical stability. Since these kernels are in the innermost loop of the simplex method, we do not believe that a significantly more expensive update can be competitive. One can always compute a fresh factorization if there are too many nonzeros or if there are numerical stabilities. Even with the most unstable LP problems, there are only a few factorizations in MOPS based on numerical grounds. It is our believe that further performance improvements can only be achieved by better pivot selection to reduce the number of iterations.

References

[1] R. Bartels and G. Golub, The simplex method of linear programming using LU decomposition, Commun. ACM 12(1969)266-268.

[2] M. Benichou, J.N. Gauthier, G. Hentges and G. Ribière, The efficient solution of large scale linear programming problems. Some algorithmic techniques and computational results, Math. Progr. 13(1977)280-322.

[3] R.K. Brayton, F.G. Gustavson and R.A. Willoughby, Some results on sparse matrices, Math. Comp. 24(1970)937-954.

[4] I.S. Duff, A.M. Erisman and J.K. Reid, *Direct Methods for Sparse Matrices* (Oxford University Press, Oxford, 1986).

[5] J. Forrest and J. Tomlin, Updating the triangular factors of the basis to maintain sparsity in the product form simplex method, Math. Progr. 2(1972)263–278.
[6] IBM, Introducing the Optimization Subroutine Library Release 2, Publication No. GC23-0517-03.
[7] R. Fourer, Solving staircase linear programming problems by the simplex method, 1: Inversion, Math. Progr. 23(1982)274–313.
[8] J.K. Reid, A sparsity exploiting variant of the Bartels–Golub decomposition for linear programming bases, Math. Progr. 24(1982)55–69.
[9] U. Suhl, MOPS – Mathematical OPtimization System, Institut für Wirtschaftsinformatik, FU-Berlin (1992), to appear in Eur. J. Oper. Res., Software Tools for Mathematical Programming.
[10] J.A. Tomlin, An accuracy test for updating triangular factors, Math. Progr. Study 4(1975)142–145.
[11] U. Suhl and L. Suhl, Computing sparse LU-factorizations for large-scale linear programming bases, ORSA J. Comput. 2(1990)325–335.