The "inexact" minimum local fill–in ordering algorithm

Csaba Mészáros
Computer and Automation Research Institute, Hungarian Academy of Sciences

The success of the implementation of the normal equations approach of interior point methods (IPM) for linear programming depends on the quality of its *analysis* phase, i.e. reordering for sparsity. The goal of this analysis is to find a permutation matrix P such that the Cholesky factor of $PAD^2A^TP^T$ is the sparsest possible. In practice, heuristics are used to solve this problem because finding an optimal permutation is an NP-complete problem. Two such heuristics, namely the *minimum degree* and the *minimum local fill–in* orderings are particularly useful in the context of IPM implementations. In this paper a parametric set of symbolic orderings is presented, which connects these two major approaches. It will be shown that in the "neighborhood" of the minimum degree ordering a practically efficient method exist. Implementation details will be discussed as well, and on a demonstrative set of linear programming test problems the performance of the new method will be compared with Sparspak's GENQMD subroutine which was for a long time public accesible from NETLIB, and with the minimum local fill–in ordering implementation of CPLEX version 4.0.

Categories and Subject Descriptors: G.4 [**Mathematics of Computing**]: MATHEMATICAL SOFTWARE—*Algorithm analysis*

General Terms: Algorithms

Additional Key Words and Phrases: ordering algorithms, interior point methods

1. INTRODUCTION

During the past years issues around efficient implementations of interior point methods (IPM) for linear programming has gained extraordinary interest. This subject is so complex that it allows improvements at many places: a significant amount of effort was put in to decrease the number of necessary iterations [Mehrotra 1991; Gondzio 1994], to make the numerical computations faster by exploiting the hardware environment better [Rothberg and Gupta 1991; Lustig et al. 1992; Esmond and Peyton 1993; Mészáros 1996], or to recognize and handle special structures [Todd 1988; Maros and Mészáros 1995].

In this paper the following linear programming problem is considered

$$\begin{array}{rl} \min & c^Tx, \\ \text{subject to} & Ax = b, \\ & x \geq 0, \end{array} \tag{1}$$

where A is an $m \times n$ matrix, c and x are n vectors, and b is an m vector. Without loss of generality we may assume that $rank(A) = m$. Some recent papers [Gondzio and

Address: H-1518 Budapest, P.O. Box 63. Hungary

Terlaky 1995; Andersen et al. 1996] give a survey of the developments achieved with the different algorithmic variants of interior point methods of linear programming. In all these IPMs the most costly operation is to determine the search direction. For most problems an efficient way to compute it is to solve the so–called *system of normal equations*

$$AD^2A^T\Delta y = h, \tag{2}$$

where D is an $n \times n$ diagonal matrix with positive diagonal values, and h is an m vector. In every iteration the diagonal matrix D is updated, and system (2) is to be solved once or twice with different right–hand sides to compute the next iterate. The big advantage of the normal equation approach is that it deals with a system of linear equations with a positive definite coefficient matrix. As such, the Cholesky factorization can be used with a great success. Since theoretically all pivots are positive, one can pay full attention to the sparsity issues in the symbolic factorization phase. The symbolic factorization is made once in advance, i.e., the pivot agenda is the same during iterations. The pivot order of the Cholesky factorization has a very strong influence on the number of fill–in and, therefore, on the storage requirement and the speed of factorizations. In the symbolic phase we try to find a good row permutation of A that minimizes the number of nonzeros in the Cholesky factors of AA^T. Since finding the "optimal" permutation is an NP complete problem [Yannakakis 1981], heuristics are used in practice.

The *minimum degree* ordering [George and Liu 1981] is one of the most important and popular methods. The implementation of this ordering algorithm is a well–developed area, the modern implementations are extremely powerful in the sense of speed and storage requirements. The most important aspect of this ordering is the connection with graph theory, and the technique of the implicit representation of the elimination, which makes compact storage and efficient implementation possible. For further details on the above, the reader is referred to the excellent summary in [George and Liu 1989]. Most of implementations of interior point methods are equipped with this ordering as default.

The other frequently used algorithm is the so–called *minimum local ill–in* ordering [Duff et al. 1986]. It is included in some of the implementations as an alternative to the minimum degree ordering. This method computes the number of fills produced by each pivot candidate, and chooses that one where this number is minimal for the next pivot. Generally, the minimum local fill–in algorithm produces sparser factorization but at higher initial cost to obtain the ordering, because the analysis in the pivot search process is very expensive [Lustig et al. 1992].

The choice between the ordering methods depends on the ordering time and the sparsity of the ordering which influence the speed of numerical computation. During the past years by the help of the new numerical implementation technologies (e.g. using supernodes with loop unrolling) the speed of the numerical Cholesky factorization is increased. The higher order techniques decreased the number of necessary iterations dramatically and, therefore, the number of necessary numerical Cholesky factorizations. This leads to pushing the minimum local fill–in ordering into the background, because the high initial cost is recompensed only at few problems.

In the paper another efficient technique is proposed to determine the pivot order. The method first selects a set of attractive pivot candidates and, in the next step, from this smaller set chooses the pivot that generates the minimal predicted fill–in. Therefore the new method is much less expensive than the "exact" version of the minimum local fill–in ordering, but it preserves the efficiency in the sense of fills.

The paper is organized as follows: In Section 2 some terminology and notation is introduced followed by the description of the minimum degree ordering (see [George and Liu 1989]). These help to make the implementation faster and more efficient. The pivot searching method and the investigation of the most important pivot search parameter are described in Section 3. Section 4 contains comments on the efficient implementation techniques. Numerical results are shown in Section 5 on a set of linear programming test problems from NETLIB [Gay 1985]. The implementation of the new ordering method is compared with the minimum degree subroutine GENQMD [George and Liu 1981], and with the minimum local fill–in algorithm implemented in CPLEX 4.0. Section 5 contains some remarks and conclusions.

2. FOUNDATION

For presenting ideas we need the general framework of the symbolic factorization process on symmetric systems. In contrast to most implementations of the minimum degree ordering, our approach is based on the explicit representation of the elimination. As the first step to determine the ordering permutation, the sparsity pattern of matrix AA^T is to be computed and stored. The sparsity structure of AA^T can be represented as a boolean $m \times m$ matrix M:

$$m_{ij} = \begin{cases} true & \text{if position } (ij) \text{ is filled in } AA^T \\ false & \text{if position } (ij) \text{ is not filled in } AA^T \end{cases} . \tag{3}$$

Because M is of order m, the elimination process includes $m-1$ steps. Let I^s denote the set of row/column indices that were not pivot until the stage s, and M^s the active submatrix, i.e., the part of the matrix that has not yet been factorized. With this convention let $M^0 = M$. At each step we choose the pivot position from the diagonal of the current active submatrix, which preserve the symmetric structure. At the s-th step of the elimination we choose a pivot index $p \in I^s$. Then, $I^{s+1} = I^s - \{p\}$, and the elements of the active submatrix at stage $s+1$ can be computed as

$$m_{ij}^{s+1} = \begin{cases} m_{ij}^s & \text{if } m_{jp}^s = false \\ m_{ij}^s \text{ OR } m_{ip}^s & \text{if } m_{jp}^s = true \end{cases} \quad \text{for } (i,j) \in I^{s+1} \times I^{s+1}, \tag{4}$$

where OR is the conventional "logical or" function. The sequence of the pivot indices $\{p\}_1^m$ define the ordering permutation. To follow the graph theory conventions up to the end we denote the possible pivot indices $i \in I^s$ as *nodes*, and we call the nodes i and j *adjacent*, if $m_{ij}^s = true$. The matrix M^s is also called an *adjacency structure*. Because the rows/columns and nodes are associated to each other, further on we do not distinguish them. This way, under the sparsity structure of a node, we mean the structure of the corresponding row or column, and the active submatrix is referred to as active nodes.

2.1 Indistinguishable nodes

Both in the pivot searching and elimination processes such nodes (rows/columns) play an important role which share the same nonzero pattern. They are referred to as *indistinguishable* nodes. In other words, the nodes $i,j \in I^s$ are denoted indistinguishable if

$$m^s_{ik} = m^s_{jk} \quad \text{for } k \in I^s. \tag{5}$$

Clearly, the indistinguishable nodes i and j behave in the same way, i.e. if one of them is transformed by some other node, the other is also transformed in a similar a way. Therefore, their indistinguishable property remains unchanged. The other advantage of using indistinguishable nodes is in the pivot searching process. By this, only one member of the indistinguishable nodes should be checked, the transformation effect of the others do not differs from the representative.

The advantages of the indistinguishable nodes can be exploited with the so–called *supernodal* technique, i.e., we store, update and check by pivot searching only one representative of the indistinguishable nodes. This representative is called a supernode [George and Liu 1989]. The administration of the other indistinguishable nodes is possible in the following way: We set up a linked list, which holds the indices of the indistinguishable nodes. Each list is linked to their supernode. When choosing the supernode as pivot, the nodes in the corresponding linked list are inserted into the pivot sequence as well.

By this technique we can represent the active submatrix in a more compact form, i.e., we can remove all rows/columns that are indistinguishable from some supernode. As an example, we consider the sparsity pattern on the left–hand of Figure 1. The nodes (1), (5) and (6) are indistinguishable. The right–hand of the figure shows the compact representation of the matrix.

$$\begin{pmatrix} 1 & & \otimes & & \otimes & \otimes \\ & 2 & & \otimes & & \\ \otimes & & 3 & \otimes & \otimes & \otimes \\ & \otimes & \otimes & 4 & & \\ \otimes & & \otimes & & 5 & \otimes \\ \otimes & & \otimes & & \otimes & 6 \end{pmatrix} \qquad \begin{pmatrix} 1 & & \otimes & \\ & 2 & & \otimes \\ \otimes & & 3 & \otimes \\ & \otimes & \otimes & 4 \end{pmatrix}$$

Fig. 1. The full matrix structure and its compact representation

Later on, we will need the numbers of indistinguishable nodes represented by one node. We refer to this number as a *supernodal degree*. We will denote the number of nodes found indistinguishable from node i for all $i \in I^s$ by sd_i. We note that $sd_i \geq 1$ because each node is indistinguishable from itself. The indistinguishable nodes can be detected after building matrix M and during the elimination process. By this, if nodes i and j are found indistinguishable, the row and column j are removed from the matrix representation, j and the list representing indistinguishable nodes of j are merged in to the list of node i and sd_i is updated as $sd_i = sd_i + sd_j$.

2.2 True degree

Originally, the basic ideas of the symmetric elimination come from the unsymmetric case. For the first time, the well–known Markowitz idea [Markowitz 1957] was applied by Tinney and Walker for symmetric factorization [Tinney and Walker 1967]. The *true degree* is the synonym of the row and column counts of the unsymmertic case, i.e., the true degree of a node is the number of nodes adjacent to it. We will denote the true degree of the node $i \in I^s$ with td_i. This value was the basis of the original minimum degree ordering [George and Liu 1989]. With our notations the true degree of node $i \in I^s$ can be computed as

$$td_i = \sum_{(j \in I^s,\ m^s_{ji}=true)} sd_j \quad \text{for } i \in I^s. \tag{6}$$

Then the Markowitz count can be defined as $(td_i - 1)^2$ for all $i \in I^s$, which reflects the number of arithmetical operations when pivoting on node i, and gives an overestimation on the number of fill–in.

2.3 External degree

By the *external degree* of a node, we mean the number of nodes adjacent to it which are not indistinguishable from the given node itself. The external degree of node i, denoted by ed_i, can be computed as:

$$ed_i = td_i - sd_i \quad \text{for } i \in I^s. \tag{7}$$

In [Liu 1985], Liu suggests the use of the external degree in the minimum degree algorithm instead of the true degree. The motivation comes from the fact that $(ed_i)^2$ estimates the number of fill–in than $(td_i - 1)^2$ better. In Section 4 we compare the use of the true degree and external degree in our new algorithm as well.

2.4 Mass elimination

This technique is also originated from the minimum degree algorithm [George and McIntyre 1978]. After the elimination of node p in the elimination process at step s, all adjacent nodes of p can be eliminated, the true degree of which is less than the true degree of p, without generating new fill–in. In other words, the following subset $J \subseteq I^{s+1}$of the active nodes can be eliminated:

$$J = \left\{ j \in I^{s+1},\ m_{jp} = true \mid td_j < \alpha \right\}, \tag{8}$$

where α is the true degree of node p in the elimination step s. Clearly,

$$\alpha - 1 \leq \min_{j \in I^{s+1},\ m_{jp}=true} td_j \tag{9}$$

because the adjacent nodes of p inherit the sparsity structure of p. If the equation holds, then the sparsity pattern of node j was a subset of the sparsity pattern of node p, and therefore, the elimination of node j can be performed without fill–in after the elimination of node p.

3. THE NEW PIVOT SEARCHING ALGORITHM

The new ordering method is basically related to the minimum local fill–in heuristic. At the s-th stage of the elimination that heuristic selects that node as next pivot which introduces the least amount of fill–in. This criterion has been known for a long time. Actually, Markowitz suggested this strategy, but rejected to use it due to its computational cost [Markowitz 1957]. One difficulty with the minimum local fill–in algorithm is that the process which computes the exact number of fill–in is costly. Noteworthy, it is not necessary to recompute the fill–in produced by all active nodes at each stage of the elimination, because each elimination step makes changes in the sparsity pattern to a small part of the active submatrix. Therefore, the update of the number of caused fill–in is necessary only on the nodes which were influenced by the last elimination step [Duff et al. 1986]. However, this and the existing ideas do not give enough improvement in the performance, and for this reason, the minimum local fill–in ordering was not considered, in general, as an alternative to the minimum degree ordering in the IPM implementations. In the paper the *inexact* variant of the minimum local fill–in algorithm is introduced. By this method there is no guarantee that the node with the minimum local fill–in property can be found at each pivot step, but we reach a good approximation on the minimum local fill–in with considerably less effort, which makes the method very powerful in the practice.

3.1 Motivation

First, we go back to the unsymmetric Gaussian elimination. Here the Markowitz criterion is the general principle to the pivot selection. Typically, the number of pivot candidates is considerably large: all nonzero entries in the active submatrix (satisfying certain stability criteria in the case of the numerical elimination) are candidates for pivoting. Therefore, choosing one with the smallest Markowitz count is very expensive. To allow improvements in the speed of the pivot searching, another pivot choice principle is used: from among the elements which satisfy the numerical criteria, we want to choose a pivot with *small* Markowitz count [Suhl and Suhl 1990]. A number of various techniques were developed to exploit this scheme. Zlatev proposed to search for pivot in the first few rows in increasing order of row counts [Zlatev 1980], Suhl suggested an algorithm which searches among few rows and columns with a small row or column count [Suhl and Suhl 1990]. Our method can be regarded as the application of the above two ideas to the symmetric symbolical factorization. We will compute the caused fill–in of the first p nodes in the increasing order of the true or external degree, and we will choose one of them which introduces the least amount of fill–in.

3.2 The algorithm

In this section we present the algorithm Find_Pivot in "Fortran 77"–like pseudo–code. Similarly to the algorithm described in [Suhl and Suhl 1990], we use linked lists to access to the nodes with the same true or external degree. Two cases are discussed. In the first case the true degree is used to set up the list, and to process the searching. In the second case the nodes are linked based on their external degree plus one. Thus the key, used for listing the nodes, is between 1 and m in both cases. To store and maintain the linked lists we need two integer working arrays of size m, **degpnt** and **deglst**. In $degpnt(j)$ we hold the entry point into the sublist of nodes with true degree j (or with external degree $j-1$, respectively); the array **deglst** is used for holding the sublists represented by the successors.

As noted above, it is not necessary to recompute the number of fill–in for all pivot candidates. Therefore we introduce the use of the integer working array **fillin** of size m to control this and to make an access to the previously computed values as follows:

$$fillin(j) = \begin{cases} \text{number of fill–in produced by node } j, \\ -1, \text{ if the number of fill–in is to be recompute} \end{cases} \quad \text{for } j \in I^s. \qquad (10)$$

Let p denote the search parameter. The input parameters of the procedure are m, p, **degpnt**, **deglst** and **fillin**. The output parameter is $pivot$, which holds the selected pivot position, or 0 if no other pivots are available. The function Compute_Fill predicts the number of fill–in of a node.

```
procedure Find_Pivot(m,p,pivot,degpnt,deglst,fillin)
 1.   fill = m^2 + 1;
 2.   k = 1;
 3.   deg = 1;
 4.   pp = 0;
 5.   pivot = 0;
 6.   j = degpnt(deg);
 7.   if j > 0 then
 8.      if fillin(j) < 0 let fillin(j) =Compute_Fill(j,...);
 9.      if fillin(j) ≥ fill goto 13.
10.      fill = fillin(j);
11.      pivot = j;
12.      if fill = 0 goto 16
13.      j = deglst(j);
14.      pp = pp + 1;
15.      if pp > p goto 20.
16.      goto 7.
17.   endif
18.   deg = deg + 1;
19.   if deg ≤ m goto 6.
20.   STOP
End
```

The procedure Find_Pivot stops the search if all active nodes were checked (step 19), or the number of nodes examined exceeds the value of the parameter p (step 15), or a node is found which produces no fill–in (step 12). As a result, a node is obtained which produces the minimal amount of fill–in from the set of the first $p+1$ nodes with a small true or external degree. If more than one node exist with minimal fill–in, the procedure chooses one with the smallest true or external degree.

3.3 Experiments with the pivot search parameter

The most important parameter of the above algorithm is p, which balances between the cost and the efficiency of the pivot searching. When increasing p, we search longer, and probably we gain a better pivot. For example, the setting $p = 0$ gives the minimum degree principle, and $p = m$ the exact minimum local fill–in rule. Naturally, no unique value exists for p which makes the best performance on all problems. Finding a *good* value is subject of extensive testing. Because testing like this is very sensitive, for the computing environment here a standalone IBM PC 486/DX2 50 Mhz computer was used rather than a networked multitasking Unix workstation, where the execution of a program can be influenced by many factors. In our experiments the Watcom Fortran Compiler 77^{32} version 9.5 was used with the /OX option.

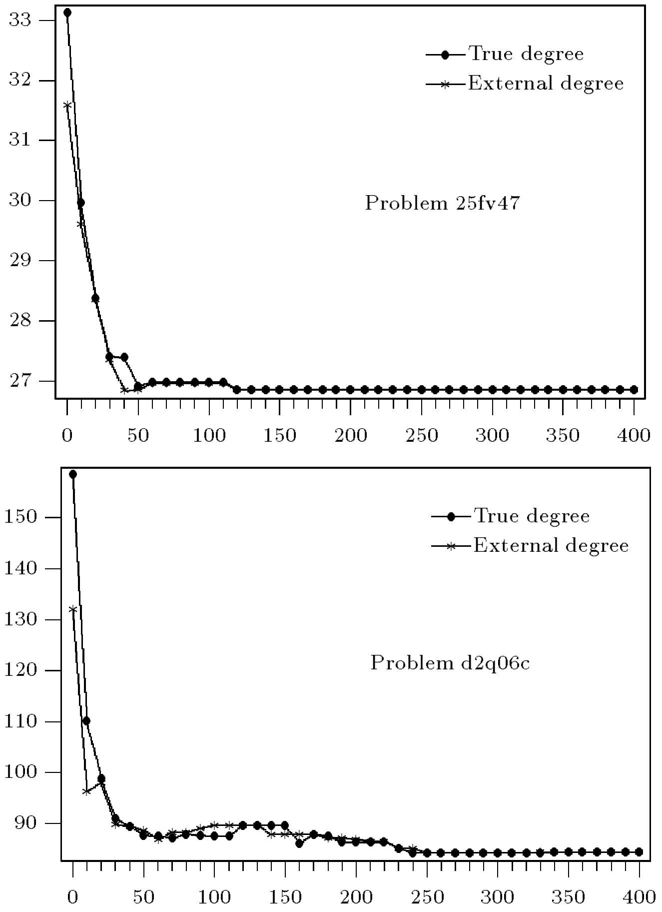

Fig. 2. Number of nonzeros in the factorization (in thousand)

First, two test problems, 25fv47 and d2q06c from the NETLIB test set are chosen for an extensive study on the effect of the search parameter p. Problem 25fv47 behaves like most problems from our point of view, that is, the minimum local fill–in heuristic produces perceptible sparser factorization of AA^T (27167 nonzeros in the lower triangular) than minimum degree algorithm (33572 nonzeros in the lower triangular part).

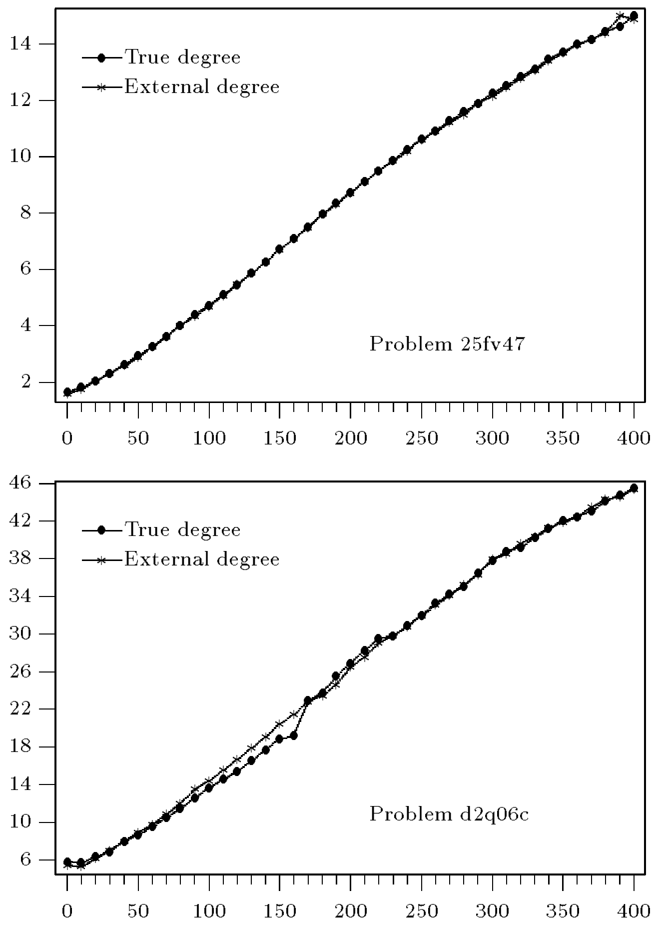

Fig. 3. Ordering times (in seconds)

The other test problem is d2q06c. On this problem the performance of the two ordering methods is significantly different, the minimum degree ordering generates 154397, the minimum local fill–in ordering 92032 nonzeros in the triangular factors of AA^T. In the first experiment p varies between 0 and 400 with step 10. Both the true degree and external degree variant of the algorithm are run. Figure 2 shows the number of nonzeros in the factorization in thousands in the case of problems 25fv47 and d2q06c respectively, as it is plotted against the value of p. From 0 to 30 the results show significant improvement in the number of fill–in, which slows down between 30 and 50. As it can be seen, if $p > 50$, the longer search for a better pivot does not make any remarkable improvement. In both problems the time spent on obtaining the ordering (reported on Figure 3), grows nearly linearly with the value of p. These results indicates tat we should concentrate on the values $p \in [0, 50]$.

In the second experiment p is increased from 0 to 50 by step 1. Figure 4 shows the behaviour of fill–in depending on the value of p at problems 25fv47 and d2q06c. In both problems the external degree version tends to produce better factorization,

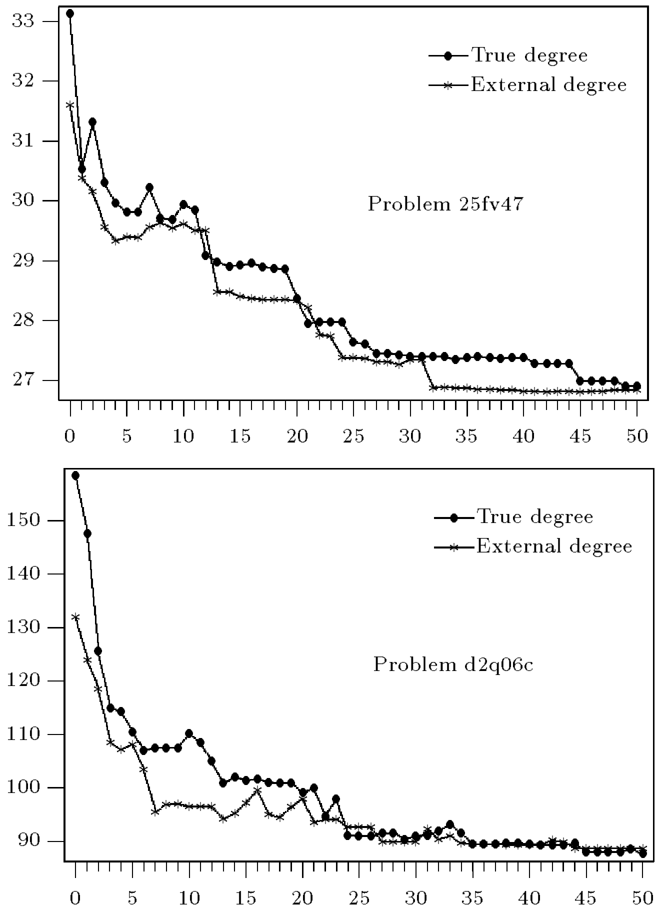

Fig. 4. Number of nonzeros in the factorization (in thousand)

but in the later stages (after about $p = 45$ and $p = 25$ at the problems 25fv47 and d2q06c, respectively) the difference becomes insignificant. The ordering time reported on the left side of Figure 5 is in accordance with the previous experiences. A more dramatic effect is displayed in the case of problem d2q06c. Here, the ordering time is decreased in the first stage as a result of the large decrease in the number of fill–in.

The heuristical assumption, namely that the node which produces few fill–in must have small degree, is confirmed by the experiments. The other advantageous feature of our pivot searching algorithm is that it computes the predicted fill–in on nodes with small degree only. This is very important because the complexity of this process depends on the degree of the nodes.

The presented experiments show that in general the increase of p results in a nearly uniform increase in the ordering time, but the quality of the ordering remains unchanged after a certain threshold. Thus, there is no reason for using a value larger than 35. Due to the better performance, the use of the external degree version of

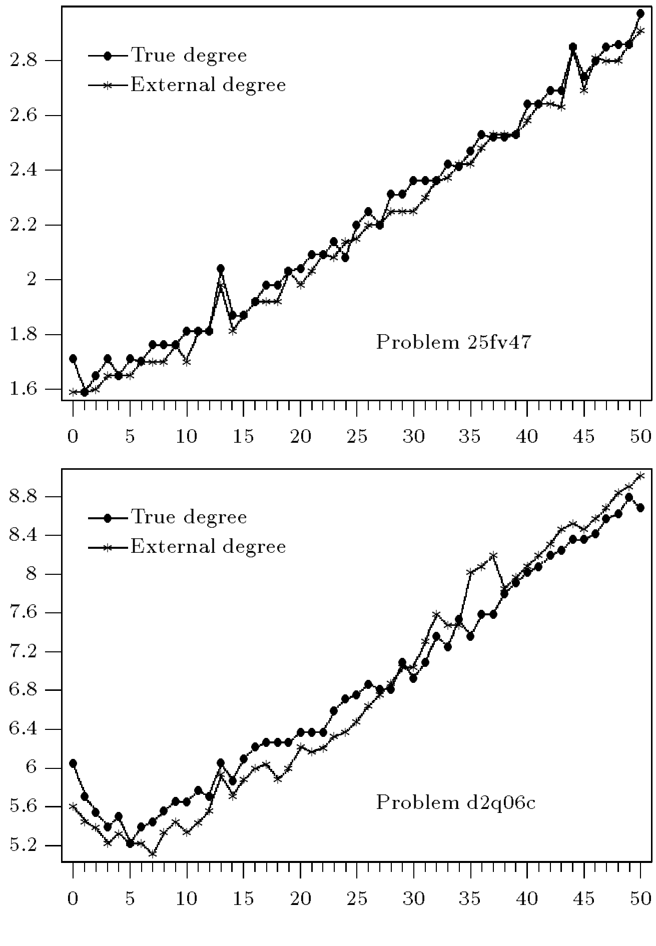

Fig. 5. Ordering times (in seconds)

the new algorithm is suggested. Based on solving many problems with different settings, it seems that $p = 25$ is a good value in practice. The solver BPMPD [Mészáros 1996] is equipped with this value as default, but the possibility kept open to change it for the user.

4. IMPLEMENTATION

The first step of the implementation does the initialization. This includes the computation of the sparsity structure of the initial matrix $M = AA^T$, the search on it for indistinguishable nodes, furthermore the set–up of linked lists and arrays used in the elimination loop. The elimination loop begins with pivot searching. Then, update (4) is performed on the active submatrix. This process is very similar to the unsymmetric case. This part of the implementation is based on the ideas described in [Suhl and Suhl 1990] with modifications to exploit the specialities of the symmetric and symbolic factorization. Before returning to pivot search, the implementation searches indistinguishable nodes caused by the last transformation and indicates the requirement of the recomputation of fill–in for certain nodes. The global scheme of the algorithm as implemented in BPMPD is displayed in Figure 6.

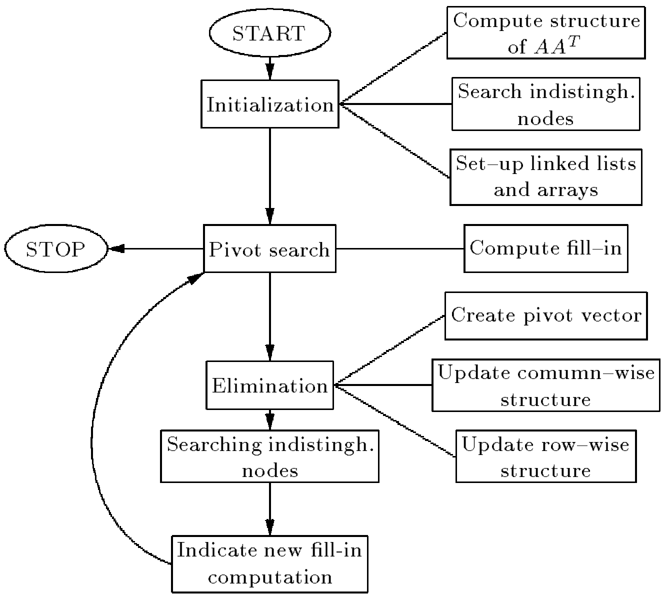

Fig. 6. The outline of the implementation

Since M, the active submatrix at each elimination step is symmetric, its lower triangular part is stored only. The off–diagonal nonzeros of the lower triangular part of the active submatrix are stored column–wise as a sequence of sparse vectors. A pointer array **pntc** is used for pointing to the starting position of columns in the column file **rowidx** which contains the row indices. The array **ccol** holds the number of entries of each column. For efficient access, the row–wise representation is also maintained. Similarly to the column- data structure, the pointer array **pntr** points to the column file **colidx** and array **crow** holds the number of entries of each column. Due to the symmetric property of the active submatrix, the row–wise storage scheme can be interpreted as the columns of the upper triangular part of the active submatrix. Additionally to the compact representation of the active submatrix, some integer arrays of size m are necessary in our new algorithm. They are: **degpnt**, **deglst**, **fillin** (as introduced in Section 3.2), **trudeg** for holding the true degree of nodes (6), **supdeg** for holding the supernodal degree of nodes and **indlst** for holding the lists of indistinguishable nodes, as mentioned in Section 2.1. Additionally, the array **permut** is used for storing the pivot sequence. This array is the output of the algorithm. The other occasional temporary working arrays used by the different fragments of our implementation are referred in to their places.

Many parts of the implementation are not discussed here in details, such as the preparation of the sparsity pattern of AA^T, using and updating linked lists, and some simple procedures and functions. We believe that they have limited interest for the experienced reader only.

It is to be noted that the memory requirement of the implementation is significantly larger than that of the minimum degree algorithm. But, in context of the interior point methods, it is not a decisive issue. The matrix ordering is only a preparation phase of interior point implementations, therefore all working arrays used by the main algorithm are free at this stage. Since higher order techniques and efficient numerical factorization methods require considerable amount of memory, usually no extra memory array are needed for the ordering, just the memory what is used in a later phase for other purpose is used here.

4.1 Predicting fill–in

One of the most important step of the new algorithm is to compute the fill–in caused by a certain node. We note again that this expensive process should be executed on few nodes in each pivot search only. The following function, presented in "Fortran 77"–like pseudo–code, computes the number of fill–in produced by node j. It uses two integer working arrays of size m, **work** and **mark**, in additional to the input parameters.

```
function Compute_Fill(j,m,pntc,ccol,rowidx,pntr,crow,colidx,supdeg,trudeg)
   fill = 0;
   q = 0;
   do o = pntc(j);
      pntc(j) + ccol(j) - 1;
      q = q + 1;
      work(q) = rowidx(o);
      mark(rowidx(o)) = supdeg(rowidx(o));
      fill = fill - (supdeg(rowidx(o)) * (supdeg(rowidx(o)) - 1))/2;
   enddo
   do o = pntr(j), pntr(j) + crow(j) - 1
      q = q + 1;
      work(q) = colidx(o);
      mark(colidx(o)) = supdeg(colidx(o));
      fill = fill - (supdeg(colidx(o)) * (supdeg(colidx(o)) - 1))/2;
   enddo
   fill = fill + ((trudeg(j) - supdeg(j)) * (trudeg(j) - supdeg(j) - 1))/2;
   do p = 1, q
      fl = 0;
      do oo = pntc(work(p)), pntc(work(p)) + ccol(work(p)) - 1
         fl = fl + mark(rowidx(oo));
      enddo
      fill = fill - supdeg(work(p)) * fl;
   enddo
   do o = 1, q
      mark(work(o)) = 0
   enddo
Return (fill)
```

It is to be noted that the complexity of the above process depends on the number of entries of the compact representation of node j only. It is very important on large problems where the number of nonzeros is usually independent of problem size. Even on denser problems the supernodal representation technique reduces the number of entries, therefore, the complexity may depend on a significantly smaller number than the true degree.

4.2 Elimination

After computing the pivot index *pivot*, we need the transformation (4) on the active submatrix. In our implementation this elimination process consists of three parts. In the first part the pivot column is extracted (and deleted) from the row– and column–wise representation, and stored in the array **pivcol** as an ordered sparse vector. Variable *pivlen* is used for holding the number of nonzero entries of the pivot node. While the entries of the pivot column are deleted from the column–wise representation, a pointer array **cfill** is initialized to mark the first position after each column.

The next part of this process performs the update on the nonzero structure of the column–wise representation. An additional working array **mark** is used in the process which marks the nonzero positions of the pivot column in a full vector. Many details are ignored such as: checking the available space for the fill in the column file, moving columns in the column–file, etc., readers interested in the full development are referred to [Suhl and Suhl 1990]. The result of the procedure Eliminate is the column–wise representation of the updated matrix. For each transformed column, we obtain its column–wise storage in the form showed in Figure 7.

```
procedure Eliminate(pivot,pivlen,pivcol,pntc,ccol,rowidx,pntr,crow,colidx,cfill)
do i = 1, pivlen
   mark(pivcol(i)) = 1;
enddo
do p = 1, pivlen
   i = pivcol(p);
   mark(i) = 0;
   pnt1 = pntc(i);
   pnt2 = cfill(i) − 1;
   do k = pnt1, pnt2
      if (mark(rowidx(k)) ≠ 0) let mark(rowidx(k)) = 0;
   enddo
   do k = p + 1, pivlen
      if (mark(pivcol(k)) = 0)then
         mark(pivcol(k)) = 1;
      else
         pnt2 = pnt2 + 1, rowidx(pnt2) = rowidx(k);
      endif
   enddo
   ccol(i) = pnt2 − pnt1 + 1;
enddo
End
```

The last part of this process updates the row–wise structure. By this, the information of the fill–in is used via the array **cfill**. Two passes are necessary through the new entries, by the first we count the number of the fills in each row, then we reserve space for the new entries, and in the second pass we place the new entries in the row–file. Simultaneously, **crow, trudeg** are to be updated.

$$
\begin{array}{ccccccccccc}
\text{cpnt(j)} & & & & & & \text{cfill(j)} & & & & \text{cpnt(j)+ccol(j)-1} \\
\downarrow & & & & & & \downarrow & & & & \downarrow \\
\underbrace{\text{entry} \;\; \text{entry} \;\; \text{entry} \;\; \text{entry} \;\; \text{entry} \;\; \text{entry}}_{old\ entries} & & & & & & \underbrace{\text{entry} \;\; \text{entry} \;\; \text{entry} \;\; \text{entry} \;\; \text{entry}}_{new\ entries\ (fill-in)} & & & &
\end{array}
$$

Fig. 7. Representation of the transformed columns after Eliminate

4.3 Searching indistinguishable nodes

By the elimination process, the pivot node passes its sparsity structure to their adjacent nodes. This unioning of sparsity structures tends to create new indistinguishable nodes. To find them, we compare nodes $pivcol(j)$ and $pivcol(k)$ for all $1 \leq j < k \leq pivlen$. By the comparison, first the true degree of nodes are compared. If these are equal, we start the comparison of the sparsity structures. By the first occurring difference, we break the process. If the structures are identical, we remove node j from both the column– and row–wise representation, and update **supdeg** and **indlst** as described in Section 2.1.

In our implementation we assume that the new indistinguishable nodes must be adjacent to the pivot node. A more rare possibility for arising indistinguishable nodes is shown in Figure 8.

$$
\begin{pmatrix}
\mathbf{1} & & \otimes \\
& 2 & \otimes \\
\otimes & \otimes & 3
\end{pmatrix}
$$

Fig. 8. Possibility for arising new indistinguishable nodes

When pivoting on node 1, only node 3 is transformed, but after the transformation nodes 2 and 3 become indistinguishable. We reject the use of a process that searches also for such situations because of its high computational cost. Our experience on real–life LP problems is that this has no influence on the performance of the new algorithm.

The other place for detecting indistinguishable nodes is at the beginning of the elimination, after building the matrix AA^T. By this, nodes i and j are compared for all $1 \leq i < j \leq m$ in the previous manner.

4.4 Indicating the recomputation of fill–in

As discussed previously, to reduce the effort of pivot searching, we set–up the array **fillin** which holds the number of fills produced by the nodes (10). At the beginning of the algorithm, the array **fillin** is filled with -1. During the pivot search, the fill–in produced by some nodes are computed and the result is stored in the corresponding position of **fillin**. After the elimination, all positions of **fillin**, corresponding to nodes that are adjacent to nodes containing new entries are set to -1 to indicate the recomputation of the fill–in.

5. NUMERICAL RESULTS

The aim of this paper is to show that the well–established minimum degree ordering can be improved by using the ideas of the minimum local fill–in heuristic. The only way of verification is computational testing. Due to the rapidly changing hardware technologies, the comparison of two algorithms is a highly complex task. In the present case, the performance of the hardware in terms of the ratio of executing integer and floating point operations plays significant role, therefore, on different architectures the conclusion can be different. In the numerical investigations three different platforms were used (IBM PC 486, IBM PC Pentium, Sun Sparc-2000) because the tests were performed at different institutes. It is to be noted that except of a speed factor, the behaviour of these platforms is nearly equal.

Table 1. Comparison of minimum degree (MDO) and "inexact" minimum local fill–in (MFO) orderings

Name	Analysis time		Nonzeros in L		Flops in thousand	
	MDO	MFO	MDO	MFO	MDO	MFO
25fv47	1.35	1.09	33572	27379	1337	831
80bau3b	3.14	1.83	39821	33977	1341	890
bnl2	2.47	1.67	64476	55759	4655	3376
cycle	2.59	2.58	51160	39796	1674	911
d2q06c	5.84	3.54	154397	92492	16249	4923
degen3	72.43	9.89	119241	115749	7930	7441
dfl001	194.23	338.80	1619564	1396096	683586	506614
greenbea	7.93	1.75	47669	45110	941	826
nesm	0.64	1.16	20594	19720	591	535
perold	1.21	0.85	23192	19229	885	596
pilot87	83.07	74.20	464167	389356	106260	75749
scfxm3	0.39	0.28	12723	10257	181	127
sctap3	0.83	0.58	14769	14432	224	208
shell	0.13	0.16	3505	3301	33	28
ship12l	0.78	0.23	4920	4896	39	38
truss	1.57	1.79	64575	51593	2784	1689
tuff	0.91	0.17	6806	6238	160	129
wood1p	2.36	0.86	11473	11473	1087	1087
osa-30	1957.81	31.11	110768	110768	2533	2533
osa-60	11692.44	109.82	255666	255666	5635	5635

To demonstrate the advantages of the new ordering scheme to the minimum degree ordering described in [George and Liu 1989], we shall compare their performance on a subset of large scale linear programming problems from the NETLIB collection [Gay 1985]. Table 1 collects the results of this comparison. Abbreviations MDO and MFO in it denote the minimum degree ordering [George and Liu 1989] and the "inexact" minimum local fill–in ordering, described in this paper, respectively.

The first three columns of Table 1 contain the problem names and the times (in seconds on a SUN Sparc-2000 workstation) of the analysis phase for the two orderings considered. The analysis time includes the setup for the ordering (i.e.

building a representation of AA^T), the ordering time, and the time for building the nonzero patterns of the Cholesky factors. For both algorithms, the ordering time is the dominating factor. The following columns contain the number of nonzeros in the Cholesky factors produced by the two orderings, the number of flops (in thousands) needed to compute the factorization, including the flops required by the computation of AA^T.

Table 2. Performance comparison of BPMPD with the minimum degree (MDO) and "inexact" minimum local fill–in (MFO) orderings

Problem	Time for one fact.		Iterations		Solution time	
Name	MDO	MFO	MDO	MFO	MDO	MFO
25fv47	0.351	0.280	25	24	15.73	13.68
80bau3b	0.470	0.386	45	45	85.46	75.67
bnl2	1.033	0.809	31	30	50.31	41.71
cycle	0.608	0.363	23	23	29.11	22.87
d2q06c	2.903	1.481	30	30	129.13	79.45
degen3	2.619	2.956	17	18	83.08	80.00
dfl001	195.112	149.829	43	43	9145.73	7410.33
greenbea	0.947	0.461	31	31	59.76	45.30
nesm	0.200	0.220	28	30	16.07	18.99
perold	0.201	0.176	37	37	16.01	15.52
pilot87	21.338	20.934	41	44	1016.67	1116.50
scfxm3	0.090	0.070	18	18	5.37	5.04
sctap3	0.112	0.104	13	14	5.35	5.81
shell	0.020	0.020	21	21	3.69	3.92
ship12l	0.050	0.051	14	14	6.17	6.27
truss	0.631	0.634	16	16	25.93	26.97
tuff	0.050	0.050	17	17	2.25	2.18
wood1p	0.477	0.482	29	29	26.22	26.20
osa-30	4.651	4.580	31	33	755.17	697.45
osa-60	11.718	11.248	38	32	2749.28	1601.54

It is to be noted that the largest difference in the symbolic time occurs on problem osa-60 where the new method is more than 100 times faster than GENQMD. The reason of it is the special behaviour of the problem, namely, by a very large size few fill–in take place. For this problem the implicit representation of the elimination seems to be disastrous. The example of the reverse effect is the problem dfl001 where very heavy fill–in occurs. On problems like this, GENQMD works faster due to the better performance of the implicit representation. Considering the fill–in the results show the advantages of the new method.

Table 2 shows the comparison of the performance of the two ordering methods in the solver BPMPD which is based on the primal–dual interior point algorithm. Figures given include the average time spent to one numerical factorization, the number of iterations to 8-digit optimality and the total solution time. All timing results were obtained on a SUN Sparc-2000 workstation and are given in seconds.

The next experiment was to compare the barrier solver CPLEX 4.0 with its minimum local fill–in ordering option and BPMPD which uses the "inexact" minimum

Table 3. Comparison of CPLEX and BPMPD

Problem Name	Symbolic time CPLEX	Symbolic time BPMPD	Nonzeros in L CPLEX	Nonzeros in L BPMPD	Solution time CPLEX	Solution time BPMPD
25fv47	1.65	0.77	42017	38648	8.78	7.85
80bau3b	2.75	1.42	62734	57450	24.55	36.03
BNL2	8.35	1.49	93330	87019	48.88	37.41
cycle	3.46	1.21	79780	73376	19.94	14.00
d2q06c	9.11	2.41	130039	128816	47.40	44.22
degen3	27.40	6.70	156941	140496	67.83	35.60
dfl001	3265.00	204.49	1504318	1405086	7532.93	4239.92
greenbea	6.32	2.42	112249	99947	53.88	28.28
nesm	1.21	0.83	36269	32822	15.11	10.43
perold	1.21	0.66	31309	29137	10.38	9.22
pilot87	261.00	42.29	496945	457743	841.18	546.94
scfxm3	0.38	0.22	22170	19734	3.79	3.24
sctap3	1.05	0.33	28470	25096	4.23	3.68
shell	0.11	0.16	7308	6407	1.98	2.09
ship12l	0.49	0.33	29523	27307	5.27	6.31
truss	3.02	1.15	90423	79429	17.20	15.55
tuff	0.38	0.16	13530	12367	2.37	1.81
wood1p	2.58	0.83	90853	88297	18.90	16.97
woodw	4.50	1.93	87669	81994	20.98	22.41
Total	3600	270	3115877	2891171	8476	5082

local fill–in ordering [1]. Both of the solvers were ran on an IBM Pentium P-90 machine with 32 Mbyte of memory with disabled presolve feature. Table 3 contains the results in the following order: the symbolic time, the number of nonzeros in the factorization and the total solution time to obtain a solution of a 8-digit accuracy. All timing results are given in seconds, and represent the pure algorithmic time, without the input of the MPS file and the time spent on producing the output.

The results presented in Table 3 show that the new ordering algorithm is much faster than the "exact" minimum local fill–in ordering and competitive with it regarding the fill–in.

6. CONCLUSIONS AND RECOMMENDATIONS

In this paper the "inexact" variant of the minimum local fill–in ordering was presented. We discussed the necessary theoretical and programming techniques which make this ordering very powerful in the practice. Computational tests proved that the new ordering method is a competitive alternative to the commonly used minimum degree and minimum local fill–in orderings.

The author's opinion is that a significant improvement in the inexact minimum fill–in ordering can probably be the use of another key, different from the external degree, to select the nodes for pivot searching. There is more possibility to set–up indicators which give more accurate information about the expected fill–in and hopefully will make it possible to increase the speed of pivot searching.

[1] This experiment was performed at the Department of Mathematics and Statistics, Brunel University, London, UK

REFERENCES

ANDERSEN, E., GONDZIO, J., MÉSZÁROS, CS., AND XU, X. 1996. Implementation of interior point methods for large scale linear programming. In T. TERLAKY Ed., *Interior point methods of mathematical programming*. Kluwer Academic Publisher, 189–252.

DUFF, I., ERISMAN, A., AND REID, J. 1986. *Direct methods for sparse matrices*. Oxford University Press, New York.

ESMOND, N. AND PEYTON, B. 1993. A supernodal Cholesky factorization algorithm for shared-memory multiprocessors. *SIAM J. on Scientific Computing 2*, 14, 761–769.

GAY, D. M. 1985. Electronic mail distribution of linear programming test problems. *COAL Newsletter 13*, 10–12.

GEORGE, A. AND LIU, J. 1981. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, Englewood Cliffs, NJ.

GEORGE, A. AND LIU, J. 1989. The evolution of the minimum degree ordering algorithm. *SIAM Rev. 31*, 1–19.

GEORGE, A. AND MCINTYRE, D. 1978. On the application of the minimum degree ordering to finite elements systems. *SIAM J. Numer. Anal. 15*, 90–111.

GONDZIO, J. 1994. Multiple centrality corrections in a primal-dual method for linear programming. Technical Report 1994.20 (Nov.), Logilab, HEC Geneva, Section of Management Studies, University of Geneva. Revised May 1995, to appear in Computational Optimization and Applications.

GONDZIO, J. AND TERLAKY, T. 1995. A computational view of interior point methods for linear programming. In J. BEASLEY Ed., *Advances in linear and integer programming*. Oxford, England: Oxford University Press, 103-144.

LIU, J. 1985. Modification of the minimum-degree algorithm by multiple elimination. *ACM Trans. Math. Software 11*, 2, 141–153.

LUSTIG, I. J., MARSTEN, R. E., AND SHANNO, D. F. 1992. The interaction of algorithms and architectures for interior point methods. In P. M. PARDALOS Ed., *Advances in optimization and parallel computing*, pp. 190–205. Elsevier Sciences Publishers B.V.

MARKOWITZ, H. M. 1957. The elimination form of the inverse and its application to linear programming. *Management Sci. 3*, 255–269.

MAROS, I. AND MÉSZÁROS, CS. 1995. The role of the augmented system in interior point methods. Technical Report TR/06/95, Brunel University, Department of Mathematics and Statistics, London.

MEHROTRA, S. 1991. Handling free variables in interior methods. Technical Report 91-06 (March), Department of Industrial Engineering and Managment Sciences, Northwestern University, Evanston, USA.

MÉSZÁROS, CS. 1996. Fast Cholesky factorization for interior point methods of linear programming. *Computers & Mathematics with Applications 31*, 4/5, 49–54.

ROTHBERG, E. AND GUPTA, A. 1991. Efficient sparse matrix factorization on high-performance workstations – exploiting the memory hierarchy. *ACM Trans. Math. Software 17*, 3, 313–334.

SUHL, U. H. AND SUHL, L. M. 1990. Computing sparse LU factorizations for large-scale linear programming bases. *ORSA J. on Comput. 2*, 4, 325–335.

TINNEY, W. AND WALKER, J. 1967. Direct solution of sparse network equations by optimally ordered triangular factorization. In *Proceedings of IEEE*, Volume 55, pp. 1801–1809.

TODD, M. 1988. Exploiting special structure in Karmarkar's linear programming algorithm. *Mathematical Programming 41*, 81–103.

YANNAKAKIS, M. 1981. Computing the minimum fill–in is NP–complete. *SIAM J. Algebraic Discrete Methods 2*, 77–79.

ZLATEV, Z. 1980. On some pivotal strategies in Gaussian elimimination by sparse techniques. *SIAM J. on Numerical Analysis 17*, 1, 18–30.