ADAPTIVE USE OF ITERATIVE METHODS IN INTERIOR POINT METHODS FOR LINEAR PROGRAMMING

WEICHUNG WANG* AND DIANNE P. O'LEARY†

November 21, 1995

Abstract. In this work we devise efficient algorithms for finding the search directions for interior point methods applied to linear programming problems. There are two innovations. The first is the use of updating of preconditioners computed for previous barrier parameters. The second is an adaptive automated procedure for determining whether to use a direct or iterative solver, whether to reinitialize or update the preconditioner, and how many updates to apply. These decisions are based on predictions of the cost of using the different solvers to determine the next search direction, given costs in determining earlier directions. These ideas are tested by applying a modified version of the OB1-R code of Lustig, Marsten, and Shanno to a variety of problems from the NETLIB and other collections. If a direct method is appropriate for the problem, then our procedure chooses it, but when an iterative procedure is helpful, substantial gains in efficiency can be obtained.

1. Introduction. Interior point algorithms are now widely used to solve linear programming problems

$$\begin{array}{ll} \textit{minimize} & c^T x \\ \textit{subject to} & Ax = b, \\ & x \geq 0, \end{array} \tag{1}$$

where c, x are real n-vectors, b is a real m-vector, and A is a real $m \times n$ matrix of rank m, with $m \leq n$. These methods typically solve a sequence of logarithmic barrier subproblems with the barrier parameter decreasing to zero. Newton's method is applied to solve the first order optimality conditions corresponding to each of the logarithmic barrier subproblems. The bulk of the work in such algorithms is the determination of a search direction for each iteration.

Gonzaga [17] and Wright [32] surveyed interior point methods, and many computational issues are addressed by Lustig, Marsten, and Shanno [24]. Therefore, in this section we focus only on the linear systems arising in interior point methods. For definiteness, we consider the primal-dual formulation of interior point methods, but the linear algebra of primal formulations and dual formulations is similar.

The search direction is usually determined by solving either the reduced KKT (Karush-Kuhn-Tucker) system,

$$\begin{pmatrix} -X^{-1}Z & A^T \\ A & 0 \end{pmatrix} \begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix} = \begin{pmatrix} r_b + Ze - \mu X^{-1} e \\ r_p \end{pmatrix}, \tag{2}$$

or the normal equations, formed by eliminating Δx from this system. Defining $r_b = b - Ax$, $r_p = c - A^T y - z$, and $D^2 = Z^{-1}X$, we obtain

$$(AD^2A^T)\Delta y = AD^2(r_b + Ze - \mu x^{-1} e) + r_p. \tag{3}$$

Here z is the vector of dual slack variables, μ is the barrier parameter, and X and Z are diagonal matrices containing x and z (respectively) on their main diagonals.

* Applied Mathematics Program, University of Maryland, College Park, MD 20742 (weichung@cs.umd.edu)

† Department of Computer Science and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742 (oleary@cs.umd.edu). This work was supported by the National Science Foundation under Grant CCR 95-03126.

Once Δy is determined from the normal equations, Δx may be easily computed from

$$-(X^{-1}Z)\Delta x + A^T \Delta y = r_b + Ze - \mu X^{-1}e. \tag{4}$$

Comparing the normal equations (3) and the KKT system (2), we observe that the matrix for the normal equations is positive definite and symmetric, has smaller size ($m \times m$), and may be more dense. In contrast, the KKT matrix is symmetric indefinite and usually more sparse.

One nice feature of these problems is that only D and the right hand side of the system change from iteration to iteration. Thus, the sparsity structure of the problem remains the same, in contrast to the linear systems arising in the simplex algorithm which differ by exchanges of columns of A.

The roots of interior point algorithms date back to the algorithms of Fiacco and McCormick [9], but ever since interior point algorithms first gained prominence in 1984 [18], researchers have given attention to speeding up the iteration time through efficient solution of the linear system. Direct methods that rely on sparse matrix factorizations have been the most popular approaches (e.g., [23], [31]), although iterative methods for solving linear systems have also received a fair amount of attention.

Karmarkar and Ramakrishnan reported computational results of Karmarkar's dual projection algorithm using a preconditioned conjugate gradient solver [19]. An incomplete Cholesky factorization of the matrix AD^2A^T was computed for one interior point iteration and then used as a preconditioner over several subsequent iterations. In their experiments, Cholesky factorization was performed on average every 2 to 3 iterations. Mehrotra used preconditioned conjugate gradients to solve the normal equations in a dual affine scaling interior point algorithm [25]. He addressed issues such as the stopping criterion and the stability of the implementation. At each interior point iteration, an incomplete Cholesky factor was computed and used as the preconditioner. Carpenter and Shanno used a diagonal preconditioner for a conjugate gradient solver for the normal equations in an interior point method for quadratic and linear programs [3]. They also considered recomputing the preconditioner every other iteration. Portugal, Resende, Veiga, and Júdice introduced a truncated primal-infeasible dual-feasible interior point method, focusing on network flow problems [29]. The preconditioned conjugate gradient algorithm was used to solve the normal equations. They initially used the diagonal of the matrix AD^2A^T as a preconditioner and replaced it by spanning tree preconditioners in later iterations. Mehrotra and Wang [26] used an incomplete Cholesky factor of AD^2A^T as a preconditioner for conjugate gradients in a dual interior point method for network flow problems. Gill, Murray, Saunders, Tomlin, and Wright established the equivalence between Karmarkar's projected method and their projected Newton barrier method [13]. They used LSQR [28], preconditioned by an approximation to AD^2A^T, to find the search directions. Goldfarb and Mehrotra developed a relaxed version of Karmarkar's method that allows inexact projection [15]. They applied CGLS [28] to determine the search direction. Nash and Sofer investigated the choice of a preconditioner in the positive definite system Z^TGZ where Z is rectangular and G is general symmetrici [27].

Chin and Vannelli [5] solved a reduced KKT system using the preconditioned conjugate gradient algorithm and Bi-CGSTAB with incomplete factorization. In a different paper [4] they used an incomplete factorization as a preconditioner for the normal equations (3). Freund and Jarre [10] employed a symmetric variant of the quasi-minimal residual (QMR) method to solve the KKT systems. They suggested using indefinite SSOR preconditioners to accelerate the convergence.

The use of iterative methods has so far produced limited success. The obstacles to the use of these methods are considerable.

- Over the course of the interior point iterations, the requirements on accuracy change greatly; approximate solutions can be allowed early in the iterations but can cause the algorithm to fail when the iterates are near the boundary.
- The matrix D changes quite rapidly and becomes highly ill-conditioned in the final iterations.

For these reasons, it is difficult to find a preconditioning strategy that produces good performance of iterative methods over the entire course of the interior point iteration.

In this paper we develop an adaptive algorithm that changes strategy over the course of the interior point iteration. It determines dynamically whether the preconditioner should be held constant, updated, or recomputed, and it switches to a direct method when it predicts that an iterative method will be too expensive. In our experiments, we use a preconditioned conjugate gradient iteration on the linear system involving the matrix ADA^T, but our ideas could be extended to iterations involving the KKT formulation as well.

In the next section, we discuss the characteristics of direct and iterative methods and present several preconditioners. Section 3 focuses on our algorithm for the adaptive choice of direct vs. iterative methods and the adaptive choice of a preconditioner. Numerical results obtained from a modified version of the OB1-R code of Lustig, Marsten, and Shanno [23] are presented in § 4. Final comments are made in § 5.

2. The linear system solvers. Either direct or iterative methods may be used for determining the search directions, the most expensive part of an interior point algorithm. In this section, we focus on the solution of the normal equations (3). This discussion sets the goals to be accomplished in designing an efficient algorithm.

We will assume that the columns of A have been permuted using standard techniques in order to improve sparsity in the Cholesky factor of AD^2A^T (e.g., [8], [22]).

2.1. Direct solvers: Cholesky factorization. Most existing linear programming interior point methods solve the normal equations by direct methods. The careful implementation OB1-R of Lustig, Marsten, and Shanno (LMS) [23] is representative of these methods, and the iterative methods will be compared with this implementation.

To solve equation (3), the LMS implementation computes a sparse Cholesky factorization of the matrix $K = AD^2A^T$ as LPL^T, where L is a unit lower triangular matrix and P is a diagonal matrix. Forward and backward substitution are then applied to compute the search direction Δy_k. The LMS algorithm then checks whether $A\,\Delta x_k$ is close enough to the artificial variables $(b - Ax_k)$. If not, iterative refinement using the factored matrix LPL^T is employed repeatedly until the one-norm of the difference is sufficiently small. To deal with the dense columns in A, the LMS algorithm adopts the method suggested by Choi, Monma, and Shanno [6].

There are three main disadvantages to direct methods such as that in OB1-R. First, the methods will fail if the matrix $K = AD^2A^T$ is very ill-conditioned. If the computed solution is not sufficiently accurate, then iterative refinement produces a sequence of approximations

$$\Delta y^{j+1} = \Delta y^j + (LPL^T)^{-1}(r_n - K\Delta y^j), \tag{5}$$

where r_n is the right-hand side of equation (3). This iteration will only converge if the spectral radius of the matrix $(I - (LPL^T)^{-1}K)$ is less than one. If K is ill-conditioned, however, this condition may not be satisfied due to inaccuracy in the computed factorization, and refinement may fail to converge. Such a situation can occur when the primal and dual variables are near to the optimal solution, since then the matrix D is quite ill-conditioned. The iteration can also be affected by ill-conditioning in A.

Another potential problem of direct methods is fill-in. Although the dense columns of A can be treated separately, the remaining Cholesky factor may still be rather dense. This might be caused by difficulty in detecting "dense" columns or by the nature of the problem. For example, network problems solved by linear programming may lead to a Cholesky factor that is much more dense than AD^2A^T even though A has no dense columns; see, for example, Table 6.

Lastly, the LMS algorithm forms and factors the matrix $K = AD^2A^T$each time μ is changed. This procedure may be expensive in time, especially when the problem size is large. If $m \ll n$, the resulting matrix K may be small and easy to factor, but forming it can still be costly.

2.2. Iterative solvers: preconditioned conjugate gradients. A variety of iterative methods can be used to solve the normal equations or the KKT system. For definiteness, we focus on the preconditioned conjugate gradient method for solving equation (3). In this method, a sequence of approximate solutions are computed that converge to the true solution. The work during each iteration involves one product of K with a vector, one solution of a linear system involving the preconditioner, and several vector operations. More details about the method can be found in [16].

The conjugate gradient method preconditioned by the Cholesky factorization of K has somewhat better stability than the direct solver since the method does not require that the spectral radius of the iteration matrix be less than one. Thus, convergence can be achieved even if the factorization is quite inaccurate.

The storage requirement for the preconditioned conjugate gradient method is quite low, amounting to a few vectors of length m. Although a matrix-vector multiplication $Kv = (AD^2A^T)v$ is required at each iteration, we may compute Kv as $(A(D^2(A^Tv)))$ and thus need only to store the nonzeros of A and the diagonal of D rather than the matrix AD^2A^T, which can be quite dense. The preconditioner should also be chosen to conserve storage.

Since accuracy requirements for the search direction in the beginning phase of the interior point algorithm are quite low, only a few conjugate gradient iterations are required. As the primal and dual variables approach the optimal solution, the convergence tolerance must be tightened and more iterations are needed.

The crucial issue in the preconditioned conjugate gradient algorithm is to find a preconditioner for each step of the interior point method. A good preconditioner may dramatically accelerate the convergence rate and gain great computational savings. We consider some strategies for choosing the preconditioners in the next subsection.

2.3. The preconditioners. Convergence of the conjugate gradient iteration will be rapid if the preconditioned matrix has either a small condition number or great clustering of eigenvalues [16, Chap. 10]. We discuss five strategies for preconditioning. The first two are based on complete factorizations of the matrix K. The others factor a sparse portion of K or update a previous preconditioner. Our experiments are based on the first and fifth, but our techniques apply to all of them.

Preconditioner 1 : Cholesky factorization. We can use the Cholesky factorization of the matrix, computed as in the LMS algorithm, as the preconditioner. Effectively, then, we replace their iterative refinement algorithm by the preconditioned conjugate gradient iteration. In exact arithmetic, this preconditioned conjugate gradient iteration will converge to the solution in one iteration. If K is ill-conditioned, however, then the computed Cholesky factorization might be inaccurate and more iterations will be required. Such ill-conditioning is inevitable in the end stages of the interior point method.

An alternative to computing the Cholesky factorization on every interior point iteration is to use the preconditioner computed for one fixed value of the barrier parameter μ for several values of μ [3] [19]. This reduces the computational work in forming the factorization.

Preconditioner 2 : QR decomposition. Rather than computing the Cholesky factors of the matrix $K = AD^2A^T$, we may simply compute the $n \times m$ matrix DA^T and factor it as QR, where Q is a $n \times m$ matrix with orthonormal columns and R is a $m \times m$ upper triangular matrix. We then have a preconditioner R^TR. Note that this preconditioner is mathematically identical to the Cholesky preconditioner since $R = P^{\frac{1}{2}}L^T$, but computationally we may be able to compute a much more accurate factor because we avoid the loss of precision inherent in forming AD^2A^T. The time for computing this preconditioner, however, is usually larger than that for computing the Cholesky factors of K, and except on very ill-conditioned problems, this approach cannot be recommended.

Preconditioner 3 : Cholesky factorization for sparse part only. If the coefficient matrix A contains some dense columns, the LMS algorithm partitions A as $[A_S, A_D]$ where A_S and A_D contain the sparse and dense columns, respectively. The system involving the matrix A is then solved by using the partial factor and the Sherman-Morrison-Woodbury formula [16, Chap. 2].

Similarly, we let D_S^2 denote the diagonal matrix containing only the elements corresponding to A_S. A preconditioner M_S can then be defined by $L_SP_SL_S^T$, where $A_SD_S^2A_S^T = L_SP_SL_S^T$. Using this preconditioner, the conjugate gradient solver converges in $(k+1)$ steps provided A_D contains k columns. To see this, we denote $G_S = A_SD_S^2A_S^T$ and $G_D = A_DD_D^2A_D^T$ and then obtain $K = AD^2A^T = G_S + G_D$. Since the preconditioner $M_S = G_S$, the resulting preconditioned matrix is

$$G_S^{-1}A = G_S^{-1}(G_S + G_D) = I + (G_S^{-1}G_D) \tag{6}$$

and thus $(G_S^{-1}A)$ is the identity plus a rank k matrix and has at most $(k+1)$ distinct eigenvalues. A standard theorem for the preconditioned conjugate gradient method guarantees termination (in exact arithmetic) in at most $(k+1)$ steps [16, Chap. 10].

Preconditioner 4 : Incomplete factorization. The preconditioner can be calculated by using incomplete Cholesky factorization [8], [16], an approximation to the exact Cholesky factorization determined by neglecting small elements in the triangular matrix [30] or by discarding elements that do not fit a preassigned sparsity pattern [5]. An incomplete QR factorization could also be determined.

Preconditioner 5 : Updated Cholesky factorization. Rather than discarding one of these preconditioners or keeping it fixed when μ changes, we can try to update it by a small-rank change, since the normal equations matrix is a continuous function of μ. Let $\hat{D}$ be the current diagonal matrix and D be the one for which we have a factorization $AD^2A^T = LPL^T$. Define $\Delta D = \hat{D}^2 - D^2$ and let a_i be the i-th column of matrix A. Since

$$A\hat{D}^2A^T = AD^2A^T + A\,\Delta DA^T \;=\; LPL^T + \sum_{i=1}^{n} \Delta d_{ii}a_ia_i^T, \tag{7}$$

we may obtain an improved preconditioner $\hat{L}\hat{P}\hat{L}^T$ by applying a rank-α update to LPL^T, where $\alpha \leq n$. This update may be computed as in [1] and [7]. If α is big enough to include most of the large magnitude terms in the summation, then we have factored a matrix that differs from $A\hat{D}^2A^T$ by a matrix of rank $n-\alpha$. This difference matrix can be expressed as a matrix of small norm plus one of small rank, and we can hope for rapid convergence of the conjugate gradient iteration.

Given these five families of preconditioners, we turn our attention to criteria for deciding to keep or update the current preconditioner.

3. The algorithms. Based on the discussion in previous sections, we suggest combining the use of direct and iterative methods within the interior point algorithm. We present this algorithm in two parts. Algorithm 3.1 shows how the solvers fit within the interior point iteration. Algorithm 3.2 gives a more detailed description of the iterative solver and its preconditioner.

3.1. The interior point algorithm with adaptive solver. Our interior point algorithm, Algorithm 3.1, chooses the initial variables, the step sizes, the barrier parameter, and convergence criteria following standard strategies [23]. The linear equation solver, however, has been modified to improve efficiency.

In the first iteration of the algorithm, the normal equations (3) are solved directly by factoring $K = AD^2A^T = LPL^T$. Starting from the second iteration, the algorithm uses preconditioned conjugate gradients. The preconditioner for each iteration is determined by factoring the current matrix K or by updating the current preconditioner. This *"factor-update cycle"* will be continued up to the *"end-game,"* entered when the relative duality gap is small enough.

In the end-game, the iterates are close to the optimal solution and accuracy requirements are high. The elements in matrix D vary significantly and make the matrix $K = AD^2A^T$ very ill-conditioned. The Cholesky factorization of K may not generate a good preconditioner, even if stable methods such as [12] are used. For all of these reasons, a direct method is used to determine the final search directions.

We also switch to a direct method when OB1-R computes a Cholesky factorization with a zero on the diagonal. This contingency could be avoided by using a modified Cholesky factor; see, for example, [14, Chap. 4].

3.2. The adaptive conjugate gradient solver. We now focus on the details of the implementation of the preconditioned conjugate gradient solver in Algorithm 3.1. We make decisions regarding refactorization or update of the preconditioner based on the actual cost incurred in determining previous search directions, as measured in seconds by a system timing program:

`drct_cost`	=	the cost of factoring and solving the system directly;
`updt_cost`	=	the cost of each rank-one update;
`pcgi_cost`	=	the cost of each conjugate gradient iteration.

Algorithm 3.1. Interior point algorithm with adaptive solver

```
Initialize k ← 1; μ_0 > 0; x_0, y_0, z_0 > 0; Endgame ← False; UseDirect ← False.

while (not convergent)
    if [ (k > 1) and (Endgame = False) and (UseDirect = False) ] then
        | Solve using PCG. (See Algorithm 3.2 for details.)
        | Determine the preconditioner.
        | if (the diagonal of the preconditioner is singular) then
        |    UseDirect ← True
        | else
        |    Iterate the PCG method.
        | end if
    end if

    if [ (k = 1) or (Endgame = True) or (UseDirect = True) ] then
        | Solve using direct solver.
        | Form the matrix AD^2A^T.
        | Factor AD^2A^T = LPL^T.
        | Solve the normal equations using LPL^T,
        |       applying iterative refinement if necessary.
        | Compute drct_cost as the elapsed time of the direct solver.
    end if

    | Update the primal and dual variables.
    | Compute x_{k+1} ← x_k + α_p Δx; y_{k+1} ← y_k + α_d Δy; z_{k+1} ← z_k + α_d Δz.

    | Check for end-game.
    | if (the relative duality gap is small) then (Endgame ← True)

    Choose μ_{k+1} < μ_k.
    Set k ← k + 1.
end while
```

(For simplicity, we neglect the fact that updates and downdates have slightly different costs.) We initialize each of these estimates to zero, but after the first few iterations of the interior point method, we have accurate estimates of each. In order to reduce the effects of variability from the timer output, though, we suggest that these estimates continue to be updated over many iterations.

ALGORITHM 3.2. **The preconditioned conjugate gradient solver**

```
                         Solve using PCG.

  ┌─────────────────────── Determine the preconditioner. ───────────────────────┐
  │ updt_nmbr ← the number of rank-one updates to be performed.                  │
  │ pred_cost ← predicted cost of updating the preconditioner                    │
  │                   and iterating PCG                                          │
  │ if [ (prev_cost > .8 × drct_cost) or (drct_cost < pred_cost) ] then          │
  │       Form the matrix AD^2A^T.                                               │
  │       Factor AD^2A^T to get the preconditioner.                              │
  │ else                                                                         │
  │       Perform updt_nmbr rank-one updates to get the new preconditioner.      │
  │ end if                                                                       │
  └──────────────────────────────────────────────────────────────────────────────┘

  if (the diagonal of the preconditioner is singular) then
     UseDirect ← True
  else
    ┌──────────────────── Iterate the PCG method. ────────────────────┐
    │ pcg_itn ← 0                                                      │
    │ while (not convergent)                                           │
    │       Execute a PCG iteration.                                   │
    │       pcg_itn ← (pcg_itn + 1)                                    │
    │       if (pcg_itn > max_pcg_itn) then                            │
    │             if (this happens twice) then ( UseDirect ← True )    │
    │             Factor (AD^2A^T) to reinitialize the preconditioner. │
    │             Restart the PCG iteration.                           │
    │       end if                                                     │
    │ end while                                                        │
    └──────────────────────────────────────────────────────────────────┘
  end if
```

3.2.1. Determining the preconditioner. First we determine whether to update the current preconditioner or refactor the matrix AD^2A^T to obtain a new preconditioner. This decision is based on the approximate cost of the preceding iteration, including the cost of any updates that were made to the preconditioner. This cost is

$$\texttt{prev\_cost} = (\texttt{updt\_cost} \times \texttt{updt\_nmbr}) + (\texttt{pcgi\_cost} \times \texttt{pcgi\_nmbr}) + (\mathit{overhead}),$$

where updt_nmbr is the number of updates that were performed and pcgi_nmbr is the number of pcg iterations. The overhead includes operations such as initializing the solution to zeros, computing the norm of the right-hand side, deciding on the number of rank-one updates, etc.

- If the cost of determining the previous search direction was high, we reinitialize the preconditioner by factoring the current matrix $K = AD^2A^T$. We take this action when the cost of previous iteration exceeds 80% of the cost of direct solution:

$$\texttt{prev\_cost} > .8 \times \texttt{drct\_cost}.$$

- If the cost of the previous iteration was not that high, then we base our decision on a *prediction* of the cost of the current iteration, refactoring if the predicted cost is greater than the cost of the direct method.
  Our prediction method is simple and requires only a few arithmetic operations. We fit a straight line to the number of iterations required to determine two preceding search directions. We choose the previous number, and the latest other one that gives a line with positive slope, and use this line to predict the number of iterations, `predi_nmbr`, required to determine the current search direction. If the solver refactored on the previous iteration, or if we cannot obtain a positive slope with data since the last refactorization, then our predicted number of iterations is one more than the number taken last time, `predi_nmbr` = `pcgi_nmbr`+1.
  Given this predicted number of iterations, our predicted cost for computing the search direction, neglecting overhead, is

  $$\texttt{pred\_cost} = (\texttt{updt\_cost} \times \texttt{updt\_nmbr}) + (\texttt{pcgi\_cost} \times \texttt{predi\_nmbr}).$$

  If this cost is less than `drct_cost`, then the preconditioner is obtained by updating the previous one. Otherwise it is obtained by factoring $K = AD^2A^T$.

3.2.2. The adaptive updating strategy. We adopt the strategy discussed in § 2.3 in Preconditioner 5: we update the Cholesky factors using the `updt_nbmr`= α "largest" outer product matrices as determined by $|\Delta d_{ii}|$. (We could have used $|\Delta d_{ii}|\|a_i\|^2$ instead.)

We change the number of Cholesky updates adaptively over the course of the algorithm in order to improve efficiency. The number is increased if the previous search direction took many iterations, and decreased if it took a very small number.

Two parameters `sml` < `lrg` are initially set to 20 and 30 respectively. The parameter `sml` denotes a number of conjugate gradient iterations that takes time much less than `drct_cost`, while `lrg` denotes a number that requires a more substantial fraction of `drct_cost`. After timing data is available, we set

$$\texttt{lrg} = 0.15 \times \frac{\texttt{drct\_cost}}{\texttt{pcgi\_cost}}; \qquad \texttt{sml} = 0.12 \times \frac{\texttt{drct\_cost}}{\texttt{pcgi\_cost}}.$$

To decide the number of rank-one updates, `updt_nmbr`, to be performed, let `pcgi_slope` be the slope of the line connecting last two `pcgi_nmbr`s.

$$\text{The } \texttt{updt\_nmbr} \text{ is } \begin{cases} \text{increased,} & \text{if } \texttt{lrg} \le \texttt{pcgi\_nmbr} \text{ and } \texttt{pcgi\_slope} > 0, \\ \text{decreased,} & \text{if } \texttt{pcgi\_nmbr} \le \texttt{sml} \text{ and } \texttt{pcgi\_slope} < 0, \\ \text{unchanged,} & \text{otherwise.} \end{cases}$$

Increases or decreases in `updt_nmbr` are proportional to the `pcgi_slope`:

$$\text{(to increase)} \quad \texttt{updt\_nmbr} = \texttt{updt\_nmbr} \times \max(1.2,\ \frac{\texttt{pcgi\_slope}}{8.0}),$$

$$\text{(to decrease)} \quad \texttt{updt\_nmbr} = \texttt{updt\_nmbr} \times \min(0.9,\ \frac{8.0}{|\texttt{pcgi\_slope}|}).$$

3.2.3. Iterating the PCG method. After computing the preconditioner, we solve the normal equations using the preconditioned conjugate gradient method. We start from an initial guess of zero, and iterate until the computed residual norm is less than a parameter ε_{pcg} times the norm of the right-hand side. We choose the parameter ε_{pcg} adaptively:

$$\varepsilon_{pcg} = \begin{cases} 10^{-8}, & \text{if } \texttt{relgap} > 10^{-2}; \\ 10^{-8} \times (\texttt{relgap})^{\frac{1}{2}}, & \text{otherwise,} \end{cases}$$

where `relgap` is the relative duality gap for the previous value of μ. This is similar to the stopping criterion in [26].

If the preconditioned conjugate gradient iteration number exceeds the maximum number of iterations allowed, then the current preconditioner is abandoned and a new preconditioner is determined by Cholesky factorization. If this happens twice, the iterative method is not suitable and we switch to a direct method. Unfortunately, the preconditioned conjugate gradient iteration might be stopped just before convergence, thereby making the refactoring wasteful, but we consider such a safeguard bounding the number of iterations to be important.

The maximum number of iterations is set to the number that produces a cost of 1.2 times the cost of a direct method:

$$\texttt{max\_pcg\_itn} = 1.2 \times \frac{\texttt{drct\_cost}}{\texttt{pcgi\_cost}}.$$

To sum up, our algorithm solves the normal equations directly to determine the first search direction, uses a preconditioned conjugate gradient method starting from the second search direction, and switches back to the direct method for the final search directions. The preconditioned conjugate gradient solver solves the normal equations by first choosing and computing a preconditioner. The algorithm automatically sets all parameters expected to influence performance, based on actual time performance of the components of the algorithm.

4. Numerical results. We modified the code OB1-R to adaptively choose the linear system solver, and we performed numerical experiments comparing the results of this modified version of OB1-R to the standard OB1-R code, dated December 1989.

Both OB1-R and the adaptive algorithm are coded in FORTRAN and use double precision arithmetic. Our experiments were performed on a SUN SPARCstation 20 with 64 megabytes of main memory, running SunOS Release 4.1.3. The FORTRAN optimization level was set to `-O3`. We report CPU time in seconds, omitting the time taken by the preprocessor HPREP since it is the same for both codes.

Before comparing the two codes, we illustrate the behavior of the adaptive algorithm on a large problem, `pds-10` (with artificial variables) whose problem characteristics are given in Table 3. Figure 1 shows the number of iterations needed by the preconditioned conjugate gradient method for the μ values chosen by OB1-R. Conjugate gradients are used for μ_2 through μ_{118}, and then the algorithm chooses to switch to direct solution because it detects a zero on the diagonal of the preconditioner. The horizontal line at 169 marks the maximum number of conjugate gradient iterations allowed (i.e. `max_pcg_itn`). The two dashed lines at 21 and 16 indicate `lrg` and `sml`, respectively. The Cholesky factorization is recomputed 25 times, marked by $\oplus$ in the figure. This is a savings of 92 factorizations compared to the OB1-R algorithm. In between refactorizations, the number of conjugate gradient iterations generally grows, more quickly for later values of μ than for earlier ones.

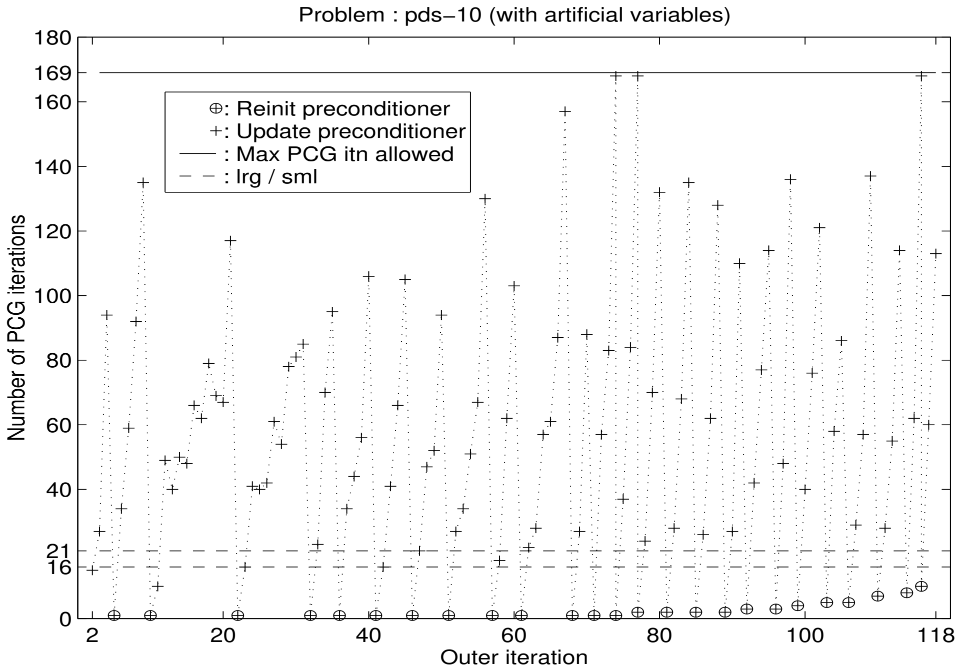

FIG. 1. *Number of PCG iterations for the adaptive algorithm*

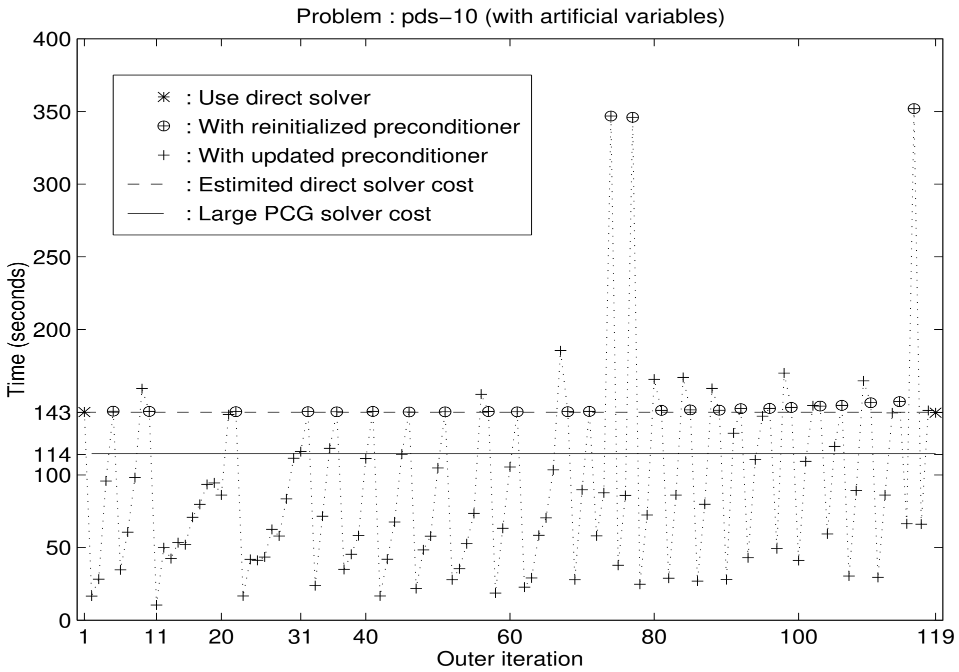

FIG. 2. *Timing performance for the adaptive algorithm*

Figure 2 displays the time taken by each of these linear system solves. The dashed line is `drct_cost`, the estimated direct solver cost based on its performance for the first value of μ. The solid line marks 0.8 times `drct_cost`.

We highlight the following observations from the figures.

- The adaptive algorithm produces significant savings in the beginning stage, especially from the 11th to the 31st value of μ.
- The frequency of reinitializing the preconditioner grows as μ is decreased.
- The preconditioners obtained from refactoring the matrix AD^2A^T are unsuitable in the later stage.
- The adaptive algorithm succeeds in keeping the cost at or better than the direct cost on all iterations but three. On those, the predicted number of iterations is too low.

We now report computational results on various types of linear programming problems.

4.1. The NETLIB problems. We first consider the NETLIB collection of test problems [11]. For small problems, forming and factoring the matrices is rather inexpensive, so the adaptive algorithm chooses the direct method and its performance is similar to OB1-R. Therefore, we select a few typical smaller problems but focus on the large problems in the NETLIB collection, those containing more than than $25,000$ nonzero entries in the coefficient matrix A. We present results for all of these problems except `fit2p`, which has a large dense AD^2A^T matrix and cannot be solved in a reasonable time on our workstation by either OB1-R or the adaptive algorithm.

Table 1 summarizes the problem characteristics. The numbers of rows, columns, and nonzeros indicated in the table refer to the output from the OB1-R preprocessor HPREP and may be different from the data in [11]. The tabulated number of nonzero elements of AA^T and L count only the lower sub-diagonal part of AA^T and L. The density of AA^T and L is computed as the ratio of the number of nonzeros to the number of elements in the lower sub-diagonal portions of the matrices.

Table 2 shows the computational results on the NETLIB problems, comparing the number of μ values needed by the interior point method, the relative duality gap in the final answer, and the CPU time required by OB1-R and the adaptive algorithm. The last column is the difference between the OB1-R and the adaptive times. A positive difference means the adaptive algorithm is faster.

Table 2 shows that both algorithms attain a small relative duality gap except on the problem `greenbea`, which is well-known to be difficult for interior point methods [31]. The algorithms take the same number of μ values and achieve similar duality gaps except on the problem `d6cube`. On this problem the adaptive algorithm takes one additional iteration, achieves a duality gap 3 orders of magnitude smaller, and is faster. On this problem the adaptive algorithm terminated successfully, while OB1-R stopped because the duality gap was increasing.

If the total time for solution is small (i.e., 5 minutes or less), then the performance of the two algorithms is similar. On more costly problems such as `dfl001`, `maros-r7`, and `pilot87`, the adaptive method is faster.

Note that in `dfl001` we keep the artificial variables, the slack variables of the equality constraints, to prevent rank deficiency. Without them, both algorithms terminate unsuccessfully in a few iterations because of the extreme ill-conditioning of the computed matrix AD^2A^T

4.2. The "Kennington" problems. There is a set of larger test problems in the NETLIB site, the "Kennington" problems used by Carolan, Hill, Kennington,

Smaller NETLIB problems							
Problem	LP size and nonzeros			Nonzeros		Density	
Name	Rows	Columns	Nonzeros	AA^T	L	AA^T	L
maros	845	1443	9614	11409	24839	.03	.07
scfxm3	990	1371	7777	8749	13520	.02	.03
seba	515	1028	4352	51400	53748	.39	.41
ship12l	1042	5427	16170	10673	11137	.02	.02
vtp.base	198	203	909	1575	2121	.08	.11

Larger NETLIB problems							
Problem	LP size and nonzeros			Nonzeros		Density	
Name	Rows	Columns	Nonzeros	AA^T	L	AA^T	L
80bau3b	2237	9799	21002	9972	40895	.00	.02
d2q06c	2171	5167	32417	26991	165676	.01	.07
d6cube	404	6184	37704	13054	54445	.16	.67
degen3	1503	1818	24646	50178	119403	.04	.11
dfl001	6071	12230	35632	38098	1634257	.00	.09
fit2d	25	10500	129018	296	299	.99	1.00
greenbea	2389	5405	30877	33791	81914	.01	.03
greenbeb	2389	5405	30882	33766	80503	.01	.03
maros-r7	3136	9408	144848	330472	1195107	.07	.24
pilot	1441	3652	43167	59540	193137	.06	.19
pilot87	2030	4883	73152	115951	421194	.06	.20
stocfor3	16675	15695	64875	103360	206731	.00	.00
truss	1000	8806	27836	12561	52509	.03	.11
wood1p	244	2594	70215	18046	18082	.61	.61
woodw	1098	8405	37474	20421	47657	.03	.08

TABLE 1
Statistics for the larger test problems from NETLIB.

Smaller NETLIB problems							
	IPM ite.		Rel. dual gap		Time		
Problem	OB1-R	Adp	OB1-R	Adp	OB1-R	Adp	Diff
maros	45	45	.11e-08	.11e-08	11.17	11.27	-0.10
scfxm3	39	39	.21e-08	.21e-08	4.25	4.80	-0.55
seba	30	20	.15e-08	.15e-09	43.05	40.60	2.45
ship12l	26	26	.53e-08	.53e-08	4.15	5.33	-1.18
vtp.base	26	26	.33e-08	.33e-08	0.45	0.53	-0.08

Larger NETLIB problems							
	IPM ite.		Rel. dual gap		Time		
Problem	OB1-R	Adp	OB1-R	Adp	OB1-R	Adp	Diff
80bau3b	78	78	.44e-08	.44e-08	46.15	48.32	-2.17
d2q06c	55	55	.25e-08	.25e-08	257.13	253.25	3.88
d6cube	77	78	.67e-06	.18e-09	113.90	100.52	13.38
degen3	30	30	.16e-09	.16e-09	66.22	65.57	0.65
dfl001	98	98	.27e-06	.27e-06	19844.37	16644.35	3200.02
fit2d	54	54	.21e-08	.21e-08	46.80	47.85	-1.05
greenbea	52	52	-.62e-04	-.62e-04	52.03	54.30	-2.27
greenbeb	74	74	.80e-09	.82e-09	69.15	72.12	-2.97
maros-r7	29	29	.31e-09	.71e-09	1952.93	1414.20	538.73
pilot	77	77	.71e-08	.70e-08	485.08	441.42	43.66
pilot87	82	82	.94e-08	.81e-08	1948.82	1584.77	364.05
stocfor3	87	87	.70e-09	.70e-09	142.22	157.28	-15.06
truss	30	30	.80e-09	.80e-09	19.55	22.22	-2.67
wood1p	18	18	.35e-08	.27e-08	12.95	13.77	-0.82
woodw	37	37	.27e-08	.27e-08	25.30	27.65	-2.35

TABLE 2
Computational results for the larger test problems from NETLIB.

Niemi, and Wichmann [2]. We report results on problems with between $25,000$ and $370,000$ nonzero elements in the coefficient matrix A. These results suggest that the adaptive algorithm would work well on the bigger problems that we omitted from testing.

Table 3 gives the statistics of these problems, and Table 4 gives the results. If we allow OB1-R to eliminate artificial variables, then the performance of the adaptive algorithm is very similar to OB1-R; in all of these problems except `osa-07` and `osa-14`, the adaptive algorithm discovers zeros in the diagonal matrix of the Cholesky factor for the second μ value and therefore switches to the direct solver.

If we keep all artificial variables in order to guarantee full row rank, then the cost of OB1-R generally increases, but the adaptive algorithm performs quite well on the costly problems. The cost of the adaptive algorithm keeping artificial variables is less than that of OB1-R on these problems, even when OB1-R discards these variables and works on a smaller problem.

4.3. Network problems. Minimum cost flow network problems may be solved using linear programming algorithms (although it is generally more efficient to use a network algorithm like [20]). We test our algorithm on this class of problems because the matrix AA^T and its resulting Cholesky factor tend to be much more dense than the original coefficient matrix A, even if there is no dense column in A. Forming and factoring the matrix AD^2A^T is thus quite expensive.

We generated minimum cost flow network problems using NETGEN, developed by Klingman, Napier, and Stutz [21]. Table 5 gives the parameters we used, except for the number of nodes and arcs. Table 6 summarizes the characteristics of the resulting linear programming problems. To prevent rank deficiency, we let the adaptive algorithm keep the artificial variables, but we let OB1-R discard the artificial variables to improve performance. The computational results are reported in Table 7.

The size of AD^2A^T for the first four problems is either 999 or 1000, and the adaptive algorithm achieves significant improvement over OB1-R as the density of L increases. If we compare problems with similar density in L but different sizes (for example, problems `net0104` and `net0416`) we find that the adaptive algorithm does increasingly well as the problem size increases.

5. Conclusion. We have presented an adaptive automated procedure for determining whether to use a direct or iterative solver, whether to reinitialize or update the preconditioner, and how many updates to apply, and demonstrated that it can enhance performance of interior point algorithms on large sparse problems.

Our preconditioning strategy is based on recomputing or updating the previous preconditioner.

Our numerical tests were performed using the OB1-R code, but it is relatively easy to implement this idea in other codes by adding three pieces:

1. a mechanism to determine whether a direct or iterative solver should be used;
2. a routine that performs updating and downdating of an existing matrix factorization (Cholesky, QR, etc.); and
3. an iterative solver, such as a preconditioned conjugate gradient method.

Further improvements could be made in the algorithm. Deeper understanding of effective termination criteria for the iterative method may further enhance the efficiency of the algorithm. A block implementation of the matrix updating and downdating would reduce overhead. Finally, parameters such as `max_pcg_itn`, `lrg`, `sml`, and `updt_nmbr` might be tuned to particular problem classes.

Kennington problems							
Problem	LP size and nonzeros			Nonzeros		Density	
Name	Rows	Columns	Nonzeros	AA^T	L	AA^T	L
cre-b	7240	72447	256095	194579	940374	.01	.04
cre-d	6476	69980	242646	181670	853300	.01	.04
ken-11	14694	21349	49058	33880	118869	.00	.00
ken-13	28632	42659	97246	66586	315642	.00	.00
osa-07	1118	23949	143694	52466	54783	.08	.09
osa-14	2337	52460	314760	113843	116160	.04	.04
pds-06	9881	28655	62524	39061	582158	.00	.01
pds-10	16558	48763	106436	66550	1674872	.00	.01

TABLE 3
Statistics for the Kennington problems.

Kennington problems without artificial variables							
	IPM ite.		Rel. dual gap		Time		
Problem	OB1-R	Adp	OB1-R	Adp	OB1-R	Adp	Diff
cre-b	91	91	-.16e-07	-.16e-07	5020.10	4872.30	147.80
cre-d	92	92	-.69e-08	-.69e-08	3872.00	3761.92	110.08
ken-11	33	33	.16e-09	.16e-08	53.28	51.93	1.35
ken-13	51	51	.43e-08	.43e-08	244.95	240.90	4.05
osa-07	53	53	.11e-05	.18e-05	80.68	88.83	-8.15
osa-14	55	55	-.81e-06	-.81e-06	191.52	218.75	-27.23
pds-06	102	102	.48e-09	.48e-09	2817.62	2781.30	36.32
pds-10	128	128	.19e-08	.19e-08	19650.00	18718.57	931.43

Kennington problems keeping artificial variables							
	IPM ite.		Rel. dual gap		Time		
Problem	OB1-R	Adp	OB1-R	Adp	OB1-R	Adp	Diff
cre-b	102	102	-.99e-09	-.10e-08	5365.32	4472.75	892.60
cre-d	103	103	-.60e-08	-.60e-08	4415.48	3698.33	717.45
ken-11	44	44	.21e-08	.21e-08	73.70	94.42	-20.72
ken-13	48	47	-.60e-09	.67e-08	238.77	274.68	-35.91
osa-07	53	53	.11e-05	.18e-05	78.43	88.32	-9.89
osa-14	55	55	-.81e-06	-.81e-06	187.63	220.68	-33.05
pds-06	117	116	.50e-08	.64e-08	3216.82	2554.35	662.47
pds-10	131	131	.69e-08	.64e-08	19978.53	13546.32	6432.21

TABLE 4
Results for the Kennington problems.

Random no. seed	13502460		Transshipment source	0
Number of sources	4		Transshipment sink	0
Number of sinks	4		% of arcs with max cost	5
Min cost of arcs	1		% of capacitated arcs	60
Max cost of arcs	400		Min upper bnd for cap. arcs	1
Total supply	800000		Max upper bnd for cap. arcs	10000

TABLE 5
NETGEN parameters used for generating network problems.

Minimal cost flow network problems							
	LP size & nonzeros			Nonzeros		Density	
Problem	Row/Node	Col/Arc	Nzros	AA^T	L	AA^T	L
NET0102	999	2000	3999	1997	40014	.00	.08
NET0104	1000	4000	8000	3991	115628	.01	.23
NET0108	1000	8000	16000	7964	206560	.02	.41
NET0116	1000	16000	32000	15873	279678	.03	.56
NET0408	4000	8000	16000	7996	552366	.00	.07
NET0416	4000	16000	32000	15989	1762394	.00	.22

TABLE 6
Statistics for the network problems.

Minimal cost flow network problems							
	IPM ite.		Rel. dual gap		Time		
Problem	OB1-R	Adp	OB1-R	Adp	OB1-R	Adp	Diff
NET0102	43	40	.68e-10	.30e-09	34.02	32.58	1.44
NET0104	41	41	.30e-09	.63e-08	171.23	135.52	35.71
NET0108	43	43	.76e-08	.75e-08	461.05	335.58	125.47
NET0116	58	59	.44e-09	.77e-09	1005.77	718.75	287.02
NET0408	43	42	.42e-09	.17e-09	2099.43	1371.17	728.26
NET0416	53	53	.31e-09	.50e-09	16674.13	9265.85	7408.28

TABLE 7
Computational results for the network problems.

REFERENCES

[1] Richard Bartels and Linda Kaufman. Cholesky factor updating techniques for rank 2 matrix modifications. *SIAM Journal on Matrix Analysis and Applications*, 10(4):557–592, October 1989.

[2] W. Carolan, J. Hill, J. Kennington, S. Niemi, and S. Wichmann. An empirical evaluation of the KORBX algorithms for military airlift applications. *Operations Research*, 38(2):240–248, 1990.

[3] Tamra J. Carpenter and David F. Shanno. An interior point method for quadratic programs based on conjugate projected gradients. *Computational Optimization and Applications*, 2:5–28, 1993.

[4] P. Chin and A. Vannelli. Computational methods for an LP model of the placement problem. Technical Report UWE&CE-94-02, Department of Electrical and Computer Engineering, University of Waterloo, November 1994.

[5] P. Chin and A. Vannelli. Iterative methods for the augmented equations in large-scale linear programming. Technical Report UWE&CE-94-01, Department of Electrical and Computer Engineering, University of Waterloo, October 1994.

[6] In Chan Choi, Clyde L. Monma, and David F. Shanno. Further development of a primal-dual interior point method. *ORSA Journal on Computing*, 2(4):304–311, 1990.

[7] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. *LINPACK User's Guide*. SIAM, Philadelphia, 1979.

[8] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. Clarendon Press, Oxford, 1986.

[9] A. V. Fiacco and G. P. McCormick. *Nonlinear Programming : Sequential Unconstrained Minimization Techniques*. John Wiley & Sons, New York, 1968. Reprint : Volume 4 of *SIAM Classics in Applied Mathematics*, SIAM Publications, Philadelphia, PA 19104–2688, USA, 1990.

[10] Roland W. Freund and Florian Jarre. A QMR-based interior-point algorithm for solving linear programs. Technical report, AT&T Bell Laboratories and Institut für Angewandte Mathematik und Statistik, 1995.

[11] D. M. Gay. Electronic mail distribution of linear programming test problems. *Mathematical Programming Soc. COAL Newsletter*, 1985.

[12] Philip E. Gill and Walter Murray. Newton-type methods for unconstrained and linearly constrained optimization. *Mathematical Programming*, 7:311–350, 1974.

[13] Philip E. Gill, Walter Murray, Michael A. Saunders, J. A. Tomlin, and Margaret H. Wright. On projected Newton barrier methods for linear programming and an equivalence to Karmarkar's projective method. *Mathematical Programming*, 36:183–209, 1986.

[14] Philip E. Gill, Walter Murray, and Margaret H. Wright. *Practical Optimization*. Academic Press, 1981.

[15] D. Goldfarb and S. Mehrotra. A relaxed version of Karmarkar's method. *Mathematical Programming*, 40(3):289–315, 1988.

[16] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, second edition, 1989.

[17] Clovis C. Gonzaga. Path-following methods for linear programming. *SIAM Review*, 34(2):167–224, June 1992.

[18] N. K. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4:373–395, 1984.

[19] N. K. Karmarkar and K. G. Ramakrishnan. Computational results of an interior point algorithm for large scale linear programming. *Mathematical Programming*, 52:555–586, 1991.

[20] J. L. Kennington and R. V. Helgason. *Algorithms for network programming*. John Wiley and Sons, New York, NY, 1980.

[21] D. Klingman, A. Napier, and J. Stutz. NETGEN: A program for generating large scale capacitated assignment, transportation, and minimum cost flow network problems. *Management Science*, 20(5):814–821, January 1974.

[22] J. Liu. Modification of the minimum-degree algorithm by multiple elimination. *ACM Transactions on Mathematical Software*, 11:141–153, 1985.

[23] Irvin J. Lustig, Roy E. Marsten, and David F. Shanno. Computational experience with a primal-dual interior point method for linear programming. *Linear Algebra and Its Application*, 152:191–222, 1991.

[24] Irvin J. Lustig, Roy E. Marsten, and David F. Shanno. Interior point methods for linear programming: Computational state of the art. *ORSA Journal on Computing*, 6(1):1–14, Winter 1994.

[25] Sanjay Mehrotra. Implementation of affine scaling methods: Approximate solutions of systems of linear equations using preconditioned conjugate gradient methods. *ORSA Journal on Computing*, 4(2):103–118, 1992.

[26] Sanjay Mehrotra and Jen-Shan Wang. Conjugate gradient based implementation of interior point methods for network flow problems. Technical Report 95-70.1, Department of Industrial Engineering and Management Sciences, Northwestern University, Evanston, IL 60208-3119, U.S.A., October 1995.

[27] Stephen G. Nash and Ariela Sofer. Preconditioning of reduced matrices. Technical Report Report 93-01, Department of Operations Research and Engineering, George Mason University, Fairfax, VA 22030, February 1993.

[28] C. C. Paige and M. A. Saunders. LSQR : An algorithm for sparse linear equations and sparse least squares. *ACM Transactions on Mathematical Software*, 8:43–71, 1982.

[29] L. F. Portugal, M. G. C. Resende, G. Veiga, and J. J. Júdice. A truncated primal-infeasible dual-feasible network interior point method. November 1994.

[30] Youcef Saad. *SPARSKIT : A Basic Tool Kit for Sparse Matrix Computations*, 1994. Version 2 is located in an anonymous ftp area in `ftp.cs.umn.edu` within directory `/dept/sparse`.

[31] Robert J. Vanderbei. LOQO : An interior point code for quadratic programming. Program in Statistics and Operations Research, Princeton University. `rvdb@princeton.edu`, 1995.

[32] M. H. Wright. Interior methods for constrained optimization. In A. Iserles, editor, *Acta Numerica 1992*, pages 341–407. Cambridge University Press, New York, USA, 1992.