Speeding-up linear programming using fast matrix multiplication
(extended abstract)

Pravin M. Vaidya
AT&T Bell Laboratories,
Murray Hill, NJ 07974

Abstract: We present an algorithm for solving linear programming problems which requires $O((m+n)^{1.5}nL)$ arithmetic operations in the worst case where m is the number of constraints, and n is the number of variables. This improves on the best known time complexity for linear programming by about $\sqrt{n}$. A key ingredient in obtaining the speed-up is a proper combination and balancing of precomputation of certain matrices via fast matrix multiplication and low rank incremental updating of inverses of other matrices. Specializing our algorithm to problems such as minimum cost flow, flow with losses and gains, and multicommodity flow leads to algorithms whose time complexity closely matches or is better than the time complexity of the best known algorithms for these problems.

1. Introduction

We study the linear programming problem

$$\max\ c^T x$$

$$\text{s.t.}\quad Ax \geq b$$

where $A \in R^{m \times n}$, $b \in R^m$, and $c \in R^n$. We assume that the given polytope $\{ x : Ax \geq b \}$ is bounded and has a non-zero interior. As the polytope is bounded we can assume that $m \geq n$, and that the columns of A are linearly independent. Let $\det_{\max}$ denote the largest absolute value of the determinant of any square submatrix of $\begin{bmatrix} c^T & 0 \\ A & b \end{bmatrix}$. We assume that each number in the input is a rational number, and let ρ denote the least common multiple of the denominators of all the numbers in the input. The parameter L is defined as

$$L = \log_2(1+\det_{\max}) + \log_2\rho + \log_2(m+n)\ .$$

Note that the coordinates of a vertex of the given polytope are rational numbers with numerators and a common denominator bounded by $2^{O(L)}$.

A polynomial time algorithm for the linear programming problem was first presented by Khachian [11] using the ellipsoid method. In [10] Karmarkar presented an interior point algorithm that required $O((m^{1.5}n^2 + m^2 n)L)$ arithmetic operations, each operation being performed to a precision of $O(L)$ bits. Using the ideas in [10, 16], in [22] the worst case time complexity of linear programming was reduced to $O((mn^2 + m^{1.5}n)L)$ arithmetic operations, each operation being peformed to a precision of $O(L)$ bits. An algorithm requiring $O(m^3 L)$ operations was also independently developed in [6]. In this paper we shall give an algorithm for solving linear programming problems that requires $O(m^{1.5}nL)$ arithmetic operations, and it is adequate to perform each operation to a precision of $O(L)$ bits. Thus our algorithm is asymptotically faster than the one in [22] for $m \leq n^4$. (Typically, m and n are of the same order.) Furthermore, specialization of our algorithm to problems such as minimum cost network flow, generalized flow, and multicommodity flow leads to algorithms whose time complexity either closely matches or is better than that of previously known algorithms.

In the algorithms described in [6, 22], the bottleneck is the number of operations required to maintain the inverse of a matrix that approximates the Hessian of a barrier (potential) function. We use three key ideas in our algorithm to reduce the number of operations required for maintaining the inverse of this matrix; this reduction leads to a better time complexity for linear programming. Suppose M denotes this matrix. The first idea is to maintain an expression for the inverse of M rather than maintaining the inverse explicitly; the expression involves a constant number of additions, subtractions and multiplications of matrices that have been explicitly computed. Such an expression for M^{-1} suffices because the only computation in which M^{-1} is involved consists of multiplying M^{-1} by a vector. The second idea is to divide the maintainence of the expression for M^{-1} into two parts: *precomputations* that use fast matrix multiplication, and *incremental updates*. The $O(\sqrt{m}L)$ iterations in the algorithm are divided into periods, each period consisting of r consecutive iterations. The precomputations are performed at the beginning of each period, and consist of explicitly recomputing M^{-1} and certain related matrices using fast matrix multiplication. During a period the expression for M^{-1} is incrementally updated using the precomputed matrices. The third idea is to balance the number of operations for the precomputations and the number of operations for the incremental updates; the period size r is chosen to achieve this balance. A combination of the three ideas allows us to prove a bound of $O(m^{1.5}nL)$ on the number arithmetic operations performed by our algorithm.

For problems such as minimum cost flow, generalized flow, and multicommodity flow, our algorithm can be sped up even further. Such problems are of the form $\{ \max w^T z \mid Gz = 0,\ Hz \geq b \}$ where H is block diagonal and G has at most a constant number of nonzeros per column. The extra speedup for network problems is obtained by exploiting the sparsity of G and the block diagonal structure of H. For these problems we let V and E denote the number of nodes (vertices) and arcs (edges) in the associated network. The bounds for

CH2806-8/89/0000/0332/$01.00 © 1989 IEEE

minimum cost flow, generalized flow and multicommodity flow are summarized below.

Minimum Cost Flow. We are given a network with a source and a sink, and a lower and an upper bound and a cost for each edge. A flow satisfies lower and upper bound constraints and conservation constraints at all vertices except the source and the sink. The value of the flow is the amount flowing into the sink and the cost of a flow is the sum over all edges of the cost of the edge times the flow through the edge. The problem is to find a flow of a fixed value that has the minimum cost among all flows of that value. We assume that the costs and the upper/lower bounds are integers whose magnitudes are less than γ. A detailed formulation of the problem may be found in [7, 12, 15]. In this case $L = O(\log_2(V\gamma)$, and the total number of arithmetic operations in our algorithm is bounded by $O(V^2\sqrt{E}\log(V\gamma))$. Our bound is within a logarithmic factor of the best known bound for this problem for dense networks [4]. (For minimum cost flow the best known bound for dense networks is $O(V^3\log(V\gamma_1))$ operations where γ_1 is the largest magnitude of any edge cost [4].)

Minimum Cost Generalized Flow. This problem is the same as the minimum cost flow problem except that each edge has a gain (loss) factor [see 3, 12]. Hence the flow going into an edge may not equal the flow coming out of an edge; the outgoing flow is given by the gain (loss) factor of the edge times the incoming flow. The cost of a flow is the sum over all edges of the cost of the edge times the flow coming into the edge. We assume that the gain (loss) factors, the costs and the upper/lower bounds are rationals with the absolute value of numerator and denominator bounded by γ. Then $L = O(E\log(V\gamma))$ and the total number of arithmetic operations in our algorithm is bounded by $O(V^2E^{1.5}\log(V\gamma))$. This bound is better than the one in [3] by $\sqrt{E}$ and better than one in [9] by $\sqrt{V}$; the bound in [9] is currently the best known and hence we obtain an improvement of $\sqrt{V}$.

Minimum cost multicommodity flow. We are given a network with s source-sink pairs, a capacity for each edge, and a cost per unit of flow of each commodity through each edge. The sum of the flows of all the commodities through an edge cannot exceed the capacity of the edge. For each i, the flow of the *ith* commodity satisfies conservation constraints at all vertices except the *ith* source and sink; the value of the flow of the *ith* commodity is the amount flowing into the *ith* sink. The cost of a flow is defined in the natural way; the value of a flow specifies the flow value for each commodity in each edge. The problem is to find a flow of a fixed value that has the minimum cost among all flows of that value. For a detailed formulation of the problem, see [7, 12]. In this case the bound on the total number of arithmetic operations is $O(s^{2.5}V^2\sqrt{EL})$. This improves on the best known bound [9] for this problem by $\sqrt{sV}$.

At this point we note that a bound of $O(m^{2.5+\nu^2}L)$ arithmetic operations for linear programming is claimed in [13] for $m = O(n)$, where $m^{2+\nu}$ is the number of arithmetic operations required to multiply two $m\times m$ matrices. The bound in [13] is claimed using a combination of fast matrix multiplication and the method of conjugate gradients [5]. However, [13] does not give a proof of this claimed bound; no explicit algorithm that might possibly achieve the bound is given either. Futhermore, [13] also does not address the issue of the precision of arithmetic operations. Even though the claimed arithmetic complexity of the algorithm in [13] is better than that of the ones in [6, 22], the bit complexity may be worse since the method of conjugate gradients requires very high precision in the worst case. Our bound of $O(m^{1.5}nL)$ on the number of arithmetic operations is better than the one claimed in [13] by at least $O(m^{\nu^2})$. (The best known upper bound for ν is about 0.38). In addition it suffices to perform arithmetic operations to a precision of $O(L)$ bits in our algorithm; as a result we obtain an improvement in the arithmetic complexity as well as the bit complexity of linear programming. Moreover, the worst case time complexity of specializations of our algorithm for network problems closely matches or is better than the time complexity of the best known algorithms for these problems.

It is worth noting that our algorithm does not use the fastest known matrix multiplication algorithm; an algorithm that can multiply two $n\times n$ matrices in $O(n^{2.4})$ operations suffices. (The best known matrix multiplication algorithm requires $O(n^{2.38})$ operations [see 2].) Furthermore, using a faster algorithm for matrix multiplication does not lead to a better upper bound on the number of operations required for solving linear programming problems. This is because the bottleneck in our algorithm is the computation of the gradient of the potential function $\sum_{i=1}^{m}\ln(a_i^Tx - b_i) + m\ln(c^Tx - \beta)$ which requires $O(mn)$ operations per iteration and a total of $O(m^{1.5}nL)$ operations.

In section 2 we give an overview of a generic path following algorithm for linear programming and in section 3 we discuss how to perform the computations in the algorithm to reduce the worst case time complexity to $O(m^{1.5}nL)$ operations. In the full paper we shall describe how to further speed up our algorithm for problems of the form $\{\max w^Tz \mid Gz = 0,\ Hz \geq b\}$ where H is block diagonal and G has at most a constant number of nonzeros per column; problems such as minimum cost flow, generalized flow and multicommodity flow are of this form. The full paper will also address the issue of the precision requirement of arithmetic operations.

2. A generic path following algorithm

Let P be the given polytope

$$P = \{ x : Ax \geq b \}$$

and let β^{max} be the maximum value of the objective function $c^T x$ over P. Let a_i^T denote the *ith* row of the constraint matrix A. Let β be a scalar parameter. The center $\omega(\beta)$ of the system of linear inequalities $\{ Ax \geq b,\ c^T x \geq \beta \}$ is the unique point that maximizes the strictly concave potential function

$$F(x, \beta) = \sum_{i=1}^{m} \ln(a_i^T x - b_i) + m \ln(c^T x - \beta)$$

over the interior of the polytope $\{x : Ax \geq b,\ c^T x \geq \beta \}$.

We shall give a brief overview of the linear programming algorithm that follows the path of centers. A complete description may be found in [16, 22]. As the parameter β continuously varies from $-\infty$ to β^{max}, the center $\omega(\beta)$ moves along a continuous trajectory and the limit point of this trajectory as β tends to β^{max} is a point that maximizes $c^T x$ over the polytope P. The linear programming algorithm generates a strictly increasing sequence of parameters $\beta^0, \beta^1, \cdots, \beta^k, \cdots$ such that $\beta^{max} - \beta^k \leq (1 - \frac{\hat{\alpha}}{\sqrt{m}})(\beta^{max} - \beta^{k-1})$ where $\hat{\alpha}$ is a small constant. The algorithm also generates a sequence of points $x^0, x^1, \cdots, x^k, \cdots$ such that x^k is a good approximation to the center $\omega(\beta^k)$; specifically

$$F(\omega(\beta^k), \beta^k) - F(x^k, \beta^k) \leq 0.04 .$$

The sequence $x^0, x^1, \cdots, x^k, \cdots$ geometrically converges to a point that optimizes $c^T x$ over the polytope P.

We shall now describe our algorithm, it is a modification of the one in [22]. The algorithm starts with a β^0 such that $\beta^{max} - \beta^0 \leq 2^{O(L)}$. The issue of obtaining a suitable starting point is addressed in [16, 22]. At the beginning of the *kth* iteration we have a parameter β^{k-1}, and a feasible point x^{k-1} such that $c^T x^{k-1} > \beta^{k-1}$, and

$$F(\omega(\beta^{k-1}), \beta^{k-1}) - F(x^{k-1}, \beta^{k-1}) \leq 0.04 .$$

We also have a diagonal matrix D such that the *ith* diagonal entry D_{ii} satisfies the condition

$$\frac{1}{1.1\,(a_i^T x^{k-1} - b_i)^2} \leq D_{ii} \leq \frac{1.1}{(a_i^T x^{k-1} - b_i)^2}, \quad i = 1, \cdots, m.$$

Let r be a positive integer parameter. During the *kth* iteration we perform the following computations in sequence.

1. $\beta^k := \beta^{k-1} + \frac{1}{30\sqrt{m}} (c^T x^{k-1} - \beta^{k-1})$.
2. Let η^k be the gradient of $F(x, \beta^k)$ evaluated at x^{k-1}. Determine a direction ξ^k as

$$\xi^k := \left[A^T D A + \frac{m}{(c^T x^{k-1} - \beta^k)^2} cc^T \right]^{-1} \eta^k .$$

3. Compute a scalar $t^k > 0$ such that

$$0.018 \leq (t^k)^2 (\xi^k)^T \left(A^T D A + \frac{m}{(c^T x^{k-1} - \beta^k)^2} cc^T \right) \xi^k \leq 0.0196 .$$

4. If $F(x^{k-1} + t^k \xi^k, \beta^k) > F(x^{k-1}, \beta^k)$ then $x^k := x^{k-1} + t^k \xi^k$ else $x^k := x^{k-1}$.
5. For each i, $1 \leq i \leq m$,

   if $D_{ii} \notin [\frac{1}{1.1\,(a_i^T x^k - b_i)^2}, \frac{1.1}{(a_i^T x^k - b_i)^2}]$

   then $D_{ii} := \frac{1}{(a_i^T x^k - b_i)^2}$.
6. If the iteration number k is a multiple of r then

   for each i, $1 \leq i \leq m$, $D_{ii} := \frac{1}{(a_i^T x^k - b_i)^2}$.

The number of iterations in the above algorithm does not depend on the choice of r, only the computational effort per iteration is affected by the choice of r. The above algorithm halts when $c^T x^k - \beta^k \leq 2^{-\alpha' L}$, for some suitably large positive constant α', and an exact optimum may then be found as described in [16]. The following Lemma is proved in [22].

Iterations Lemma. If $m \geq 16$ then $\beta^{max} - \beta^k \leq (1 - \frac{0.4}{30\sqrt{m}})(\beta^{max} - \beta^{k-1})$ and the algorithm halts in $O(\sqrt{m} L)$ iterations. ■

It is worth noting that the scheme for updating D in the above algorithm is quite different from the one in [6, 22]. The choice of the parameter r will depend on the structure and the sparsity of the constraint matrix in the given linear program. The next Lemma bounds the number of updates to D in Step 5 of the algorithm between successive resettings in Step 6, and will be useful in proving the desired bound on the number of arithmetic operations performed by the algorithm.

Update Lemma. Between successive resettings of D in Step 6, the total number of modifications to elements of D in Step 5 is $O(r^2)$, if $r \leq \sqrt{m}$. ■

A proof of the Update Lemma will be given in the full paper.

3. Bounding the number of arithmetic operations

We shall show that the algorithm in section 2 can be implemented so that the total number of arithmetic operations performed is $O(m^{1.5} nL)$. During an iteration we perform the following computations.

1. Solve a system of linear equations to determine the direction ξ^k.
2. Compute the gradient η^k of the potential $F(x, \beta^k)$ and also compute the scalar t^k.

We shall maintain an expression for $(A^T D A)^{-1}$ involving a

constant number of multiplications, additions and subtractions of matrices, each matrix containing $O(mn)$ entries. Once such an expression for $(A^TDA)^{-1}$ is available, the remaining computations such as determining the gradient η^k, the scalar t^k and the direction ξ^k may be performed in $O(mn)$ operations. We shall show that a suitable expression for $(A^TDA)^{-1}$ can be maintained at the average cost of $O(mn)$ operations per iteration. Since the number of iterations is $O(\sqrt{m}L)$ this leads to a bound of $O(m^{1.5}nL)$ on the total number arithmetic operations performed by the algorithm.

The maintainence of $(A^TDA)^{-1}$ consists of two kinds of computations: the precomputation that is performed when D is reset i.e. whenever the iteration number k is a multiple of r, and the incremental updates between successive resettings of D. In section 3.1 we shall show that the average number of arithmetic operations per iteration for the precomputations is $O(\frac{mn^{1.4}}{r})$. In section 3.2 we shall show that the average number of arithmetic operations per iteration for the incremental updates is $O(r^5 + nr^{2.4})$. Choosing $r = n^{0.4}$ balances the number of operations for the precomputations and the number of operations for the incremental updates, and gives a bound of $O(mn)$ on the average number of operations per iteration for maintaining $(A^TDA)^{-1}$.

3.1. Precomputations

The precomputations are performed whenever D is reset i.e. whenever the iteration number k is a multiple of r. The matrices $(A^TDA)^{-1}$ and $(A^TDA)^{-1}A^T$ are recomputed and stored for use in the next r iterations. We utilize fast matrix multiplication to obtain these matrices quickly. It is well-known that using fast matrix multiplication the product of two $n\times n$ matrices and the inverse of a non-singular $n\times n$ matrix can each be computed in $O(n^{2.4})$ arithmetic operations [2, 14]. Computing $(A^TDA)^{-1}$ involves multiplying an $n\times m$ matrix and an $m\times n$ matrix ($m \geq n$), and then inverting an $n\times n$ matrix; moreover, computing the product of an $n\times m$ matrix and an $m\times n$ matrix can be reduced to computing $\lceil \frac{m}{n} \rceil$ products of two $n\times n$ matrices followed by $\lceil \frac{m}{n} \rceil$ additions of $n\times n$ matrices. Furthermore, once $(A^TDA)^{-1}$ is available, evaluating $(A^TDA)^{-1}A^T$ requires multiplying an $n\times n$ matrix and an $n\times m$ matrix which can be also reduced to computing $\lceil \frac{m}{n} \rceil$ products of pairs of $n\times n$ matrices. Thus computing both the matrices requires $O(\frac{m}{n} n^{2.4}) = O(mn^{1.4})$ operations. Since the precomputation is performed only once every r iterations, the average number of arithmetic operations per iteration for the precomputation is $O(\frac{mn^{1.4}}{r})$.

3.2. Incremental updates

The incremental updates to $(A^TDA)^{-1}$ are performed between successive resettings of D in Step 6 of the algorithm i.e. between successive precomputations. The goal is to incrementally maintain an expression for $(A^TDA)^{-1}$ which involves a constant number of additions, subtractions and multiplications of matrices, each matrix containing $O(mn)$ entries. We shall show that the average number of operations per iteration for incrementally updating such an expression for $(A^TDA)^{-1}$ is $O(r^5 + nr^{2.4})$. The strategy for incremental updating will be the same for the period between any two successive resettings of D; so we shall focus on the period between the k_0^{th} and the $(k_0+r)^{th}$ iteration where k_0 is some multiple of r.

Suppose an update to D consists of adding σ to the ith diagonal element of D, and suppose D' denotes the matrix D after this update. Then we may write $A^TD'A$ as

$$A^TD'A = A^TDA + \sigma\, a_i a_i^T$$

where a_i is the ith column of A^T. Thus an update to D leads to a rank one correction to A^TDA. By the Update Lemma there are $O(r^2)$ such rank one corrections during the period.

Let B be the matrix A^TDA at the beginning of the period under consideration. After q updates to D during the period, A^TDA may be expressed as

$$A^TDA = B + U\Delta U^T = B + UV^T = B + \sum_{i=1}^{q} u_i v_i^T$$

where Δ is a diagonal matrix, $V = U\Delta$, and u_i, v_i denote the ith column of U, V respectively. Notice that $u_i v_i^T$ is the rank one correction to A^TDA resulting from the ith update to D during the period. Also, note that each column of U is also a column of A^T, and that by the Update Lemma, U and V have $O(r^2)$ columns.

Using the Sherman-Morrison-Woodbury formula [5, 18], $(A^TDA)^{-1}$ may be expressed as

$$(A^TDA)^{-1} = (B + UV^T)^{-1}$$
$$= B^{-1} - B^{-1}U(I + V^TB^{-1}U)^{-1}V^TB^{-1}.$$

Note that B^{-1} is available because of the precomputation at the beginning of the period. Thus to maintain the above expression for $(A^TDA)^{-1}$ its suffices to maintain U, V, and $(I + V^TB^{-1}U)^{-1}$. We shall prove the following Lemmas about maintaining these matrices.

Lemma 3.2.1. Given U and V, the average number of operations per iteration for maintaining $V^TB^{-1}U$ is $O(nr^{2.4})$.

Lemma 3.2.2. Given $V^TB^{-1}U$, the average number of operations per iteration for maintaining $(I + V^TB^{-1}U)^{-1}$ is $O(r^5)$.

U and V can be maintained at the average cost of $O(nr)$ operations per iteration, since the average number of columns added to U and V per iteration is $O(r)$. So from Lemmas 3.2.1

and 3.2.2 we can conclude that the average number of operations per iteration for maintaining the above expression for $(A^TDA)^{-1}$ is $O(r^5+nr^{2.4})$. The matrices B^{-1}, U, V, and $(I + V^TB^{-1}U)^{-1}$, have $O(n^2)$, $O(nr^2)$, $O(nr^2)$, and $O(r^4)$, entries respectively; hence for our choice of $r = n^{0.4}$ each of these matrices has $O(n^2) = O(mn)$ entries. Therefore each matrix in the above expression for $(A^TDA)^{-1}$ has $O(mn)$ entries as desired.

We shall now prove Lemmas 3.2.2 and 3.2.1.

Proof of Lemma 3.2.2. Note that $V^TB^{-1}U$ has $O(r^2)$ rows and columns. An update to D leads to the addition of a column to U and V and a row and a column to $V^TB^{-1}U$. Suppose that the row and the column that get added to $V^TB^{-1}U$ as a result of the update to D are available. Then the change in $(I + V^TB^{-1}U)^{-1}$ resulting from the update to D may be computed in $O(r^4)$ additional operations using the formula

$$\begin{pmatrix} M & \mathbf{u} \\ \mathbf{v}^T & d \end{pmatrix}^{-1} = \frac{1}{\lambda}\begin{pmatrix} \lambda M^{-1} + M^{-1}\mathbf{u}\mathbf{v}^TM^{-1} & -M^{-1}\mathbf{u} \\ -\mathbf{v}^TM^{-1} & 1 \end{pmatrix}$$

where $\lambda = d - \mathbf{v}^TM^{-1}\mathbf{u}$, and M, $\mathbf{u}$, $\mathbf{v}$, d, are $\theta\times\theta$, $\theta\times1$, $\theta\times1$, 1×1, matrices respectively. By the Update Lemma, the average number of updates to D per iteration is $O(r)$. So the average number of operations per iteration for computing the changes in $(I + V^TB^{-1}U)^{-1}$ given the changes in $V^TB^{-1}U$ is $O(r^3)$. ■

Proof of Lemma 3.2.1. Note that $V = U\Delta$, that $V^TB^{-1}U = \Delta U^TB^{-1}U$, and that Δ is a diagonal matrix. Given $U^TB^{-1}U$ and Δ, $V^TB^{-1}U$ is computable in $O(r^4)$ operations since $U^TB^{-1}U$ has $O(r^2)$ rows and columns. The average cost per iteration for maintaining Δ is $O(r)$ operations, since the average number of updates to D per iteration is $O(r)$. So to prove Lemma 3.2.1 it suffices to show that given U we can update $U^TB^{-1}U$ at the average cost of $O(nr^{2.4})$ operations per iteration.

An update to D leads to the addition of a column to U and a row and a column to $U^TB^{-1}U$. The number of updates to D over the period is $O(r^2)$, and so the number of columns added to U over the period is also $O(r^2)$. During an iteration the columns that are added to U are batched together in batches of size r, and when a batch is added to U we compute the change in $U^TB^{-1}U$ resulting from the addition of this batch. Note that during each iteration only the last batch may contain less than r columns. Thus the total number of batches during the period is $O(r)$, and the average number of batches per iteration is $O(1)$. From Lemma 3.2.3 below it follows that when a batch of r columns is added to U the resulting changes in $U^TB^{-1}U$ can be computed in $O(nr^{2.4})$ operations. This gives a bound $O(nr^{2.4})$ on the average number of operations per iteration for updating $U^TB^{-1}U$ and Lemma 3.2.1 then follows. ■

Lemma 3.2.3. The changes in $U^TB^{-1}U$ resulting from the addition of a batch of r columns to U can be computed in $O(nr^{2.4})$ operations.

Proof of Lemma 3.2.3. Let $U = [\,U_1,\, U_2]$ where U_2 is the batch of r columns that is being added and U_1 is the matrix U before the addition of this batch. Then $U^TB^{-1}U$ may be expressed as

$$U^TB^{-1}U = \begin{pmatrix} U_1^TB^{-1}U_1 & U_1^TB^{-1}U_2 \\ U_2^TB^{-1}U_1 & U_2^TB^{-1}U_2 \end{pmatrix}.$$

Note that $B^{-1}A^T$ is available because of the precomputation at the beginning of the period. Since each column of U is also a column of A^T and $B^{-1}A^T$ is available, each column of $B^{-1}U$ can be obtained in $O(n)$ operations. Thus $B^{-1}U_1$ and $B^{-1}U_2$ can be obtained in $O(nr^2)$ operations since the number of columns in U is $O(r^2)$. So to prove Lemma 3.2.3 it suffices to show that given $B^{-1}U_1$, $B^{-1}U_2$, and U_2, we can compute $U_2^TB^{-1}U_1$, $U_1^TB^{-1}U_2$, and $U_2^TB^{-1}U_2$ in $O(nr^{2.4})$ operations.

Note that $B^{-1}U_1$, $B^{-1}U_2$, and U_2, are $n\times\theta$, $n\times r$, and $n\times r$, matrices respectively where $\theta = O(r^2)$. Computing $U_2^TB^{-1}U_1$ requires computing the product of U_2^T and $B^{-1}U_1$ i.e. computing the product of an $r\times n$ matrix and and an $n\times\theta$ matrix where $\theta = O(r^2)$; for $r \le n$ this can be reduced to computing $O(\frac{n\theta}{r^2}) = O(n)$ products of two $r\times r$ matrices followed by $O(n)$ additions of two $r\times r$ matrices. (For our choice of $r = n^{0.4}$, $r \le n$.) Using fast matrix multiplication the product of two $r\times r$ matrices may be computed in $O(r^{2.4})$ operations, and hence $U_2^TB^{-1}U_1$ can be obtained in $O(n\,r^{2.4})$ operations. Similarly, it is easily seen $U_2^TB^{-1}U_2$ may be obtained in $O(\frac{n}{r}\,r^{2.4}) = O(n\,r^{1.4})$ operations. Moreover, since B is symmetric, $U_1^TB^{-1}U_2 = (U_2^TB^{-1}U_1)^T$. Thus given $B^{-1}U_1$, $B^{-1}U_2$, and U_2, we can compute $U_2^TB^{-1}U_1$, $U_1^TB^{-1}U_2$, and $U_2^TB^{-1}U_2$ in $O(nr^{2.4})$ operations. ■

References

1. D. A. Bayer, and J. C. Lagarias, The non-linear geometry of Linear Programming I. Affine and Projective scaling trajectories, *Trans. Amer. Math. Soc.*, (to appear).
2. D. Coppersmith and S. Winograd, Matrix multiplication via arithmetic progressions, 19*th Annual ACM Symp. Theory of Computing*, 1987, pp. 1-6.
3. A. Goldberg, S. Plotkin, and E. Tardos, Combinatorial Algorithms for the generalized circulation problem, *Proc. 29th Annual IEEE Symp. Foundations of Computer Science*, 1988, pp.432-443.
4. A. Goldberg and R. E. Tarjan, Solving minimum cost flow problems by successive approximation, 19*th Annual ACM Symp. Theory of Computing*, 1987, pp. 7-18.
5. G. H. Golub and C. F. Van Loan, *Matrix Computations*, The John Hopkins University Press, Baltimore, MD, 1983.

6. C. C. Gonzaga, An algorithm for solving linear programming problems in $O(n^3L)$ operations, Memorandum UCB/ERLM87/10, Electronics Research Laboratory, University of California, Berkeley, 1987.

7. T. C. Hu, *Integer Programming* and *Network Flows*, Addison-Wesley, Reading, MA, 1969.

8. S. Kapoor and P. M. Vaidya, Fast algorithms for convex quadratic programming and multicommodity flows, *Proceedings 18th Annual Symp. Theory of Computing*, May 1986, 147-159.

9. S. Kapoor and P. M. Vaidya, Speeding up Karmarkar's algorithm for multicommodity flows, to appear, Mathematical Programming.

10. N. Karmarkar, A new polynomial time algorithm for linear programming, *Combinatorica*, 4 (1984), 373-395.

11. L. G. Khachian, Polynomial algorithms in linear programming, *Zhurnal Vychislitelnoi Mathematiki i Matematicheskoi Fiziki*, 20 (1980), 53-72.

12. E. Lawler, *Combinatorial Optimization*: *Networks* and *Matroids*, Holt Rinehart and Winston, New York, 1976.

13. Ju. E. Nesterov, and A. S. Nemirovsky, Self-Concordant functions and polynomial-time methods for convex programming, (summary, Moscow, July 1988), Handout at the 13*th* International Symposium on Mathematical Programming, Tokyo, August 1988.

14. V. Pan, *How to multiply matrices faster*, Lecture notes in Computer Science, Springer-Verlag Berlin Heidelberg, 1984.

15. C. Papadimitriou, and K. Steiglitz, *Combinatorial Optimization* : *Algorithms* and *Complexity* (Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1982).

16. J. Renegar, A polynomial-time algorithm, based on Newton's method, for linear programming, *Mathematical Programming*, 40 (1988), 59-93.

17. G. Zoutendijk, *Mathematical Programming Methods* (North-Holland, New York, 1976).

18. G. W. Stewart, *Introduction to matrix computations* (Academic Press, Inc., New York, 1973).

19. J. H. Wilkinson, *The Algebraic Eigenvalue Problem* (Oxford University Press (Clarendon), London and New York, 1965).

20. J. Edmonds, Systems of distinct representatives and linear algebra, *Journal of Research of the National Bureau of Standards*, 71B (1967), 241-245.

21. F. R. Gantmacher, *Matrix Theory*, vol. 1 (Chelsea, London, 1959), Chapter 2.

22. P. M. Vaidya, An algorithm for linear programming which requires $O(((m+n)n^2 + (m+n)^{1.5}n)\,L)$ arithmetic operations, *Proceedings 19th Annual ACM Symposium Theory of Computing* (May 1987), 29-38, to appear *Mathematical Programming*.